Le modèle flot de données appliqué à la synthèse haut-niveau pour le traitement d'images sur caméra intelligente à base de FPGA. Application aux systèmes d'apprentissage supervisés

Cédric Bourrasset

▶ **To cite this version:**

Cédric Bourrasset. Le modèle flot de données appliqué à la synthèse haut-niveau pour le traitement d'images sur caméra intelligente à base de FPGA. Application aux systèmes d'apprentissage supervisés. Autre. Université Blaise Pascal - Clermont-Ferrand II, 2016. Français. NNT : 2016CLF22673 . tel-01280468v2

HAL Id: tel-01280468

https://theses.hal.science/tel-01280468v2

Submitted on 13 Jun 2016

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

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

N° d'ordre : 2676

EDSPIC : 743

UNIVERSITÉ BLAISE PASCAL - CLERMONT II

ÉCOLE DOCTORALE : SCIENCES POUR L'INGÉNIEUR

THÈSE

présentée par

Cédric BOURRASSET

Pour obtenir le grade de :

DOCTEUR DE L'UNIVERSITÉ BLAISE PASCAL

Spécialité : Électronique et Architecture des Systèmes

Titre de la thèse :

Le modèle flot de données appliqué à la synthèse haut-niveau pour le traitement d'images sur caméra intelligente à base de FPGA. Application aux systèmes d'apprentissage supervisés

Thèse soutenue le 9 février 2016 devant le jury composé de :

Président	M.Jean-Pierre Dérutin
Directeur de Thèse	M.Jocelyn Sérot
Co-Directeur de Thèse	M.François Berry
Rapporteurs	M.Vincent Charvillat
	M.Julien Dubois
Examinateurs	M.Paolo Pagano
	M.Maxime Pelcat

Cédric BOURRASSET : *Le modèle flot de données appliqué à la synthèse haut-niveau pour le traitement d'images sur caméra intelligente à base de FPGA. Application aux systèmes d'apprentissage supervisés*, © 1er mars 2016

Résumé

La synthèse de haut niveau (High Level Synthesis (HLS)) est un domaine de recherche qui vise à automatiser le passage de la description d'un algorithme à une représentation au niveau registre de celui-ci en vue de son implantation sur un circuit numérique. Si le problème reste à ce jour largement ouvert pour des algorithmes quelconques, des solutions ont commencé à voir le jour au sein de domaines spécifiques. C'est notamment le cas dans le domaine du traitement d'images où l'utilisation du modèle flot de données offre un bon compromis entre expressivité et efficacité.

C'est ce que nous cherchons à démontrer dans cette thèse, qui traite de l'applicabilité du modèle flot de données au problème de la synthèse haut niveau à travers deux exemples d'implantation d'applications de vision complexes sur FPGA. Les applications, issues du domaine de l'apprentissage supervisé sont un système de classification à bases de machines à vecteurs supports (SVM) et un système de reconnaissance exploitant un réseau de neurones convolutionnels (CNN). Dans les deux cas, on étudie les problématiques posées par la reformulation, au sein du modèle flot de données, des structures de données et algorithmes associés ainsi que l'impact de cette reformulation sur l'efficacité des implémentations résultantes. Les expérimentations sont menées avec CAPH, un outil de HLS exploitant le modèle flot de données.

Mot clés : Synthèse de haut niveau, Flot de données, FPGA, Traitement d'images, Systèmes d'apprentissage

Abstract

High-level synthesis is a field of research that aims to automate the transformation from an high-level algorithmic description to a register level representation for its implementation on a digital circuit. Most of existing tools based on imperative languages try to provide a general solution to any type of existing algorithm. This approach can be inefficient in some applications where the algorithm description relies on a different paradigm from the hardware execution model. This major drawback can be figured out by the use of specific langages, named Domain Specific Language (DSL). Applied to the image processing field, the dataflow model appears as a good compromise between the expressiveness of the algorithm description and the final implementation efficiency.

This thesis address the use of the dataflow programming model as response to high-level synthesis problematics for image processing algorithms on FPGA. To demonstrate the effectiveness of the proposed method but also to put forth the algorithmic reformulation effort to be made by the developer, an ambitious class of applications was chosen : supervised machine learning systems. It will be addressed in particular two algorithms, a classification system based on Support Vector Machine and a convolutional neural network. Experiments will be made with the CAPH langage, a specific HLS tool based on the dataflow programming model.

Keywords : High-Level Synthesis, Dataflow, FPGA, Image Processing, Machine Learning, Deep Learning

Á mes amours,
Muriel et Evan

Remerciements

Ce travail n'aurait pas pu aboutir sans la participation, les conseils et les relectures de mes deux encadrants, Pr Jocelyn Sérot et Dr François Berry, qui m'ont accompagné tout au long de ces travaux.

Je tiens également à remercier les personnes avec qui j'ai travaillé, en particulier, Dr Jean-Charles Quinton pour ses idées, sa disponibilité et pour ses emails détaillés. Merci également à Dr Claudio Salvadori pour sa bonne humeur et Pr Paolo Pagano pour ses conseils. Merci aussi à Dr Maxime Pelcat pour les discussions sur les modèles flot de données et pour avoir écrit du code VHDL qui va moins vite que du code CAPH. Merci à Sebastien Caux pour le support technique.

Une mention spéciale pour Luca Maggiani, avec qui j'ai partagé de nombreux moments, que ce soit pour écrire des articles, partir en conférence, préparer des démos ou bien simplement râler.

Merci à Sylvain Bayle pour avoir relu et corrigé ce document. Merci aussi à mes amis et ma famille pour leur support pendant ces trois années.

Enfin, comme il faut toujours remercier les gens qui nous paient, merci Labex IMOBS3.

Table des matières

Table des figures

Liste des tableaux

Acronymes

AE Auto-encoders. 30

ALM Adaptive Logic Modules. 8–12, 99, 129, 134, 136, 137, 139, 140, 150

ANN Artificial Neural Networks. 57, 115

ASIC Application-specific integrated circuit. 7, 19

AUC Area Under Curve. 58, 102, 104

CLB Configurable Logic Block. 11

CNN Convolutional Neural Networks. 3–5, 14, 29, 30, 53, 57, 58, 109, 112, 115, 120, 122, 126, 151, 153, 154

DAE Denoising Auto-encoders. 30

DAG Direct Acyclic Graph. 22

DBM Deep Boltzmann Machines. 30

DBN Deep Belief Network. 30

DET Detection Error Tradeoff. 58

DNF Dynamic Neural Field. 103, 104, 191

DPN Dataflow Process Network. 22, 23, 25

DSL Domain Specific Language. 3, 5, 24, 25, 28

DSP Digital Signal Processor. 20, 99, 129, 134, 136, 142–144, 147

FIFO First-In First-Out. 12, 15, 19, 21–25, 61, 85, 90, 93, 95, 97, 130, 133, 137–140, 142, 143, 145, 147, 151, 183

FPGA Field Programmable Gate Array. 2–5, 7, 8, 10–15, 17–21, 23–25, 41, 59, 64, 65, 68, 82, 91, 95, 97, 109, 129, 131, 137, 139, 143, 145, 149, 150, 152–154

FPR False Positive Rate. 58, 80, 104

GLOH Gradient location and Orientation Histogram. 35

GMM Gaussian Mixture Model. 29, 30

HAL Hardware Abstraction Layer. 24

HDL Hardware Design Language. 3, 9, 13, 14, 18, 19, 27, 59

HLS High Level Synthesis. 3, 5, 18–20, 23, 59, 97, 130

HMM Hidden Markov Models. 30

HOF Histogram of Optical Flow. 40, 41

HOG Histogram of Oriented Gradients. 4, 11, 13, 31, 37–43, 51, 52, 59–61, 63–65, 67–69, 71, 72, 74, 75, 77, 79, 99, 107, 148, 149, 153, 154

ISP Image Signal Processor. 7

KPN Kahn Process Network. 22

LAB Logic Array Blocks. 11–13
LBP Local Binary Pattern. 31, 39–41, 108, 154
LE Logic Element. 8, 9, 14, 99
LLVM Low Level Virtual Machine. 24, 25
LOO Leave-One-Out Error. 31, 49, 50, 80
LSS Local Selft-Similarities. 40
LUT Look-up Tables. 8–10, 13–15, 23, 64, 95, 96, 150

MAC Multiply–Accumulate operation. 12
MF Multi-Features. 40
ML Machine Learning. 29
MLAB Memory Logic Array Blocks. 12
MLP MultiLayer Perceptron. 53

OpenCL Open Computing Language. 20
ORCC Open RVC-CAL Compiler. 24

PCA Principal component analysis. 35
PCA-SIFT Principal Component Analysis and Scale Invariant Feature Transform. 35

RBM Restricted Boltzmann Machines. 30
ROC Receiver Operating Characteristic. 58, 84, 90
ROI Region Of Interest. 50
RTL Register Transfer Level. 2, 13, 14, 19, 26

SC Smart Camera. 1
SDF Synchronous Data Flow. 22, 23
SIFT Scale Invariant Feature Transform. 33, 35–37, 40, 41
SIMD Single Instruction Multiple Datas. 20
SRAM Static Random Access Memory. 12, 91, 136, 139, 140, 145, 151
SURF Speed Up Robust Features. 36, 37, 40, 41
SVM Support Vector Machine. 3–5, 29, 30, 40, 41, 43, 45–51, 53, 57–59, 61, 77, 83, 99, 100, 103, 107, 109, 128, 149, 153, 154

TPR True Positive Rate. 58, 80, 104, 128

1

Contexte et introduction

1.1 Les réseaux de caméras intelligentes pour la mobilité

Les systèmes de vision multi-caméras opèrent souvent de manière centralisée dans laquelle les données sont envoyées à une unité de traitement centrale. Ce noeud central va utiliser les données émises par chaque caméra, brutes ou compressées, afin d'en extraire des informations de haut niveau, par exemple la position d'une personne dans une certaine zone. Une telle approche atteint rapidement ses limites lorsque le nombre de caméras augmente. Les capacités de calcul de l'unité de traitement et les bandes passantes du réseau deviennent alors des facteurs limitants. Ceci est d'autant plus vrai lorsque les systèmes déployés intègrent des contraintes de traitement en temps réel sur des flux vidéos de résolution importante avec un rafraîchissement d'image élevé. Dans ce contexte, aucune technologie réseau existante n'est actuellement capable d'assurer un trafic de données suffisant pour assurer des spécifications de traitement temps réel durs.

Cette problématique a conduit à l'émergence de systèmes de vision distribués. L'objectif est de décentraliser les calculs directement sur chaque caméra (communément appelé noeud) du système. Les noeuds, en plus d'assurer l'acquisition des images, doivent aussi effectuer des traitements afin de délivrer une information de plus haut niveau au système. Ces caméras sont communément appelés *caméras intelligentes* ou Smart Camera (SC) et intègrent des unités de calculs (processeur, DSP , FPGA, ...). Cette approche permet de contourner le goulot d'étranglement dû à la centralisation de l'unité de traitement et aux contraintes de la bande passante réseau mais en introduit de nouvelles. Ces nouvelles contraintes induites sont en relation avec des domaines de recherche variés, notamment l'étude des transmission sans fils (RF, protocoles réseaux sans fils) mais aussi le traitement d'images embarqué, la géométrie (autocalibration des réseaux de caméras) ou encore l'algorithmique distribuée.

Les travaux décrits dans ce mémoire se placent dans le contexte des réseaux de caméras intelligentes pour la mobilité. Prenons comme exemple le cas de la figure 1.1 où le véhicule A est capable d'effectuer la détection d'obstacles (piétons, motos,...) dans son champ visuel. Le véhicule B n'est pas en mesure de détecter ce piéton dû à une occlusion. Si le véhicule A transmet l'information de la position du piéton à l'aide d'un réseau sans fils, le véhicule B peut être informé d'un danger potentiel et prendre les mesures nécessaires afin d'éviter un accident. On peut facilement imaginer de nombreuses applications où chaque véhicule est capable d'informer les autres véhicules connectés de dangers potentiels.

1.2 Le challenge des systèmes de vision embarquée

Les problématiques de la vision embarquée sont celles du traitement d'image en général, auxquelles s'ajoutent celles liées au respect d'un ensemble de contraintes liées à l'implanta-

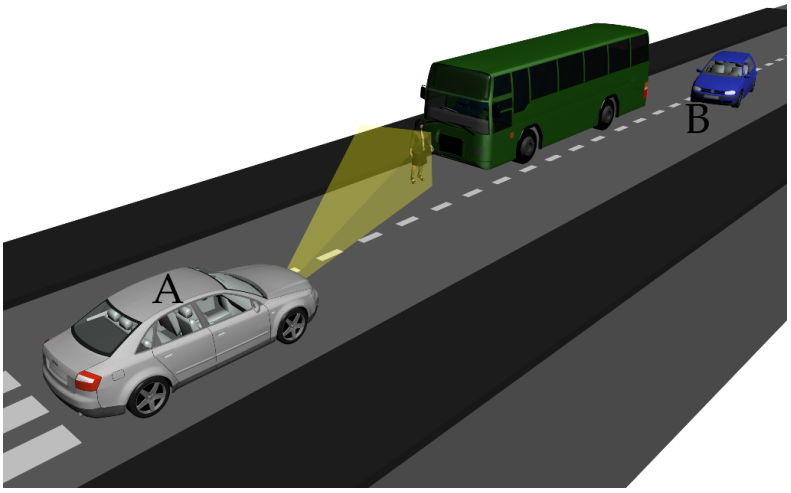


Fig. 1.1: Contexte applicatif de la thèse, le réseau de capteur intelligent pour la mobilité.

tion physique en environnement embarqué : complexité, empreinte mémoire,... Alors que les avancées théoriques sur les algorithmes ont pour objectif d'améliorer la qualité des traitements, souvent au détriment de la complexité algorithmique, leur mise en œuvre implique de profondes analyses et adaptations afin de les rendre exécutables en temps réel sur des cibles matérielles. En fonction du type d'unité utilisée, l'adaptation peut aller d'un simple portage sur processeur standard [1] à une complète reformulation de l'algorithme telle qu'une représentation Register Transfer Level (RTL) de l'algorithme initial. Bien évidemment, un compromis doit être fait entre le temps de développement et les performances finales.

De plus, un flux vidéo brut (même compressé) correspond à un débit de données difficile à transmettre sur certains médiums de communication. Un des intérêts majeurs des caméras intelligentes est d'extraire des informations **pertinentes** du champ de vision tout en réduisant le flux de données. On entend par informations **pertinentes** des informations telles que des descripteurs d'images (Harris, Saillance) qui servent de données d'entrée pour des algorithmes de navigation et de localisation ou bien des informations de plus haut-niveau telles que la présence d'objets dans le champ de vision.

Afin de répondre à ces contraintes, plusieurs architectures de système de vision embarquée ont été définies et réalisées. Un état de l'art de ces architectures est disponible dans [Bir15]. On ne s'intéressera qu'aux architectures intégrant des circuits reconfigurables (Field Programmable Gate Array (FPGA)).

1.3 Les FPGAs pour le traitement vidéo

De nombreux systèmes de traitement vidéo utilisent des circuits FPGAs notamment grâce à leur support naturel pour le traitement sur le flot de données [2], leur granularité fine pour les opérations pixelliques ou bien leur capacité de parallélisation et leur reconfigurabilité intéressante pour le prototypage d'algorithme.

Cependant, la précision des opérations peut parfois être un facteur limitant. Les opérations flottantes sont fortement déconseillées en raison des ressources qu'elles engendrent. La représentation en virgule fixe et l'arithmétique d'intervalle sont couramment utilisées, ce qui peut causer des problèmes sur des cas où la précision des calculs est primordiale (convergence d'un algorithme d'optimisation par exemple). De plus, la contrainte majeure lorsque

1. Passage d'une formulation mathématique vers un code impératif (C++)
2. Le pipeline d'opérations est immédiat, ce qui permet une parallélisation des tâches de manière naturelle

l'on a recourt à des cibles FPGAs se situe sur la méthode de programmation utilisée, basée sur des langages de description matérielle (HDL). Cette approche nécessite de profondes connaissances en architecture de systèmes numériques, ce qui limite depuis longtemps l'utilisation des FPGAs aux personnes issues de différents milieux.

Afin de résoudre les problèmes de programmabilité des FPGAs, de nombreuses recherches ont été entreprises sur des systèmes de synthèse matérielle automatique à partir de langages de plus haut niveau. Ces systèmes sont communément appelés High Level Synthesis (HLS). La plupart des outils, industriels ou universitaires, proposent des conversions directes depuis un langage de haut-niveau (C/C++) vers un code HDL. Cette approche a été peu efficace pendant les premières décennies de recherche dans ce domaine pour plusieurs raisons [MS09]. La principale est la divergence sémantique existant entre le modèle de programmation utilisé pour la description de l'application et le modèle d'exécution de la cible finale. L'outil de synthèse doit donc faire une conversion d'un modèle vers un autre, ce qui complexifie l'analyse de l'application et dégrade les performances au final. Un problème couramment rencontré est l'incompatibilité entre certaines constructions du langage de description (langage C/C++ par exemple) qui ne sont pas supportées par les outils HLS simplement car il n'existe pas de transcription en langage matériel de ces constructions, typiquement les allocations dynamiques qui n'existent pas sur un FPGA. Ceci implique donc une réécriture fréquente du code original afin d'être accepté par le HLS. En pratique, la réécriture du code pour qu'il devienne compatible avec l'outil HLS ne peut se faire sans un minimum de connaissance du matériel et des modèles d'exécution. Une autre cause d'inefficacité des HLS provient de leur incapacité à complètement extraire le parallélisme automatiquement à partir d'une description séquentielle d'un algorithme. Ces problématiques sont complexes à gérer dans le cas général car il peut exister différents niveaux de parallélisme au sein d'une même application.

1.4 Les méthodologies de programmation

Une solution au problème sus-cité consiste à renoncer au modèle de programmation impératif des langages C/C++ et à intégrer au langage source lui-même certains aspects liés aux architectures visées. C'est l'approche dite Domain Specific Language (DSL). Le choix d'un modèle de programmation adapté permet alors de satisfaire à la fois aux exigences de l'abstraction des programmes tout en assurant l'efficacité d'implémentation.

Pour des raisons que l'on détaillera plus loin, le modèle flot de données, utilisé par exemple par les DSL tels que CAPH [SBB14] ou CAL [LMW+08], est particulièrement bien adapté à la programmation de circuits de type FPGA. Ce point a d'ailleurs été souligné par Najjar [NLG99] qui, afin de maîtriser la complexité de la programmation des FPGAs, proposa le modèle de calcul flot de données. En effet, ce dernier offre deux caractéristiques à savoir que toutes les données sont des valeurs et toutes les opérations sont des fonctionnelles. Malheureusement ces travaux n'ont été que peu appliqués dû au manque de maturité de la technologie FPGA qui à cette époque ne pouvait pas supporter des designs complexes.

1.5 Contributions

Une des motivations de ce travail consiste à démontrer que moyennant un travail de reformulation suivant le modèle de type flot de données, il est alors possible de traduire automatiquement l'algorithme reformulé en une description matérielle synthétisable. De manière plus précise, cette thèse a pour objectif de quantifier l'effort de reformulation vers la modélisation flot de données.

Pour cela, nous avons axé notre travail sur l'outil CAPH [SBB14, Ser15] qui est un langage de synthèse haut-niveau(HLS) permettant de traduire une description flot de données en un code VHDL synthétisable. Les systèmes matériels automatiquement générés sont alors capables d'exécuter des algorithmes sur un flot vidéo en temps réel.

La cible algorithmique choisie pour cette démonstration concerne les systèmes d'apprentissage. C'est un domaine complexe reflétant différents types de "figure algorithmique" et nécessitant une importante puissance de calcul permise par les architectures reconfigurables.

Les deux algorithmes choisis sont d'une part un système à base de Support Vector Machine (SVM) utilisant comme données d'entrée des descripteurs d'images de type Histogram of Oriented Gradients (HOG) et d'autre part un réseau de neurones convolutionnels CNN.

Le premier algorithme a été choisi pour plusieurs raisons :

- Il correspond à l'état de l'art des systèmes d'apprentissages déployés sur une architecture FPGA. La première implémentation publiée d'un tel système est récente [HSH+13].
- La formulation initiale de l'algorithme intègre une forte dépendance de données. L'effort de formulation à effectuer pour extraire le parallélisme de l'application constitue un objectif en soi. De plus, les opérations arithmétiques sont complexes à mettre en œuvre avec le modèle choisi.
- Enfin même si de nouvelles méthodes ont été proposées depuis les travaux de Dalal [DT05], cet algorithme reste une référence pour l'évaluation des performances de nouvelles méthodes [BOH+14].

Le second algorithme, basé sur un réseau de neurones convolutionnels, admet trois autres motivations :

- Ce type de réseau procure une meilleure fiabilité des résultats dans de nombreux domaines de la vision telles que la détection d'objets ou la segmentation d'images[Far13].
- Dans ce type d'algorithme, l'effort de formulation est relativement faible ; en effet les traitements sont "naturellement" parallélisables et il sera intéressant de voir comment de tels algorithmes sont bien adaptés au traitement flot de données.
- La plupart des techniques récentes d'apprentissage de la communauté de la vision sont basées sur ces structures.

1.6 Plan du manuscrit

Le manuscrit est articulé en trois parties. La première partie regroupe les chapitres 2 et 3, introduisant respectivement les notions liées au développement d'algorithmes sur FPGA et aux systèmes d'apprentissage. La seconde partie regroupant les chapitres 4 et 5 présentent les deux études de cas choisis. Enfin, la dernière partie traitera des problématiques liées à la reformulation algorithme et à la transcription sur des architectures matérielles à l'aide de CAPH.

- **Chapitre 2 : Les méthodologies d'implémentation d'algorithmes sur FPGA**
  Le premier chapitre introduira l'architecture générale des FPGAs, le flot de conception classique ainsi que les problématiques liées à la synthèse haut-niveau. Ensuite, le modèle flot de données sera introduit, ainsi que ses avantages pour le prototypage rapide d'applications. Enfin, le chapitre se concluera par l'introduction du langage CAPH.
- **Chapitre 3 : Les systèmes d'apprentissage**.
  Après quelques définitions générales sur les systèmes d'apprentissage, une attention particulière sera prêtée aux deux algorithmes qui seront reformulés suivant le modèle flot de données dans les chapitres 4 et 5. Le premier système à base de Support Vector Machine (SVM) est couramment utilisé en traitement d'images mais n'a été que très récemment implémenté en pratique sur des flux vidéos de par sa complexité algorithmique. L'objectif de cette formulation est de traiter du cas pratique de la détection de personnes. La seconde partie de ce chapitre sera consacrée aux Convolutional Neural Networks (CNN), un système d'apprentissage très étudié actuellement et dont les caractéristiques sont intéressantes dans le cadre de nos travaux de par l'efficacité que peut apporter une architecture matérielle à base de FPGA.

- **Chapitre 4 :Reformulation flot de données et implantation : application aux algorithmes HOG-SVM**

  Dans ce chapitre, nous allons voir comment reformuler une description de type HOG associé à un classifieur de type SVM suivant la méthodologie proposée dans le chapitre 2.

  La suite du chapitre porte sur l'implémentation de la formulation proposée avec le langage CAPH. Les impacts de chaque étape de la formulation sur la précision de détection mais aussi sur l'implantation matérielle (ressources, fréquences,...) seront les principaux résultats de ce chapitre.

  Nous verrons les limitations du système reformulé et tenterons d'apporter les réponses afin de dépasser ces limitations et ainsi améliorer les performances du système final. Nous aborderons en particulier la mise en œuvre d'un système de filtrage des résultats bruts.

- **Chapitre 5 :Reformulation flot de données et implantation : application aux réseaux de neurones convolutionnels (CNN)**

  Dans le chapitre 5, un second système d'apprentissage à base de réseaux convolutionnels (CNNs) sera abordé. Cette méthode d'apprentissage est très utilisée de nos jours dans l'apprentissage profond (*Deep Learning*). et de récents travaux ont montré que les CNNs pouvaient également servir d'amélioration au système étudié dans le chapitre 4 [GDDM14].

  Ce chapitre a donc pour objectif de proposer une seconde étude de cas afin de démontrer que le modèle flot de données n'est pas limité à la seule étude de cas du chapitre 4.

- **Chapitre 6 : Problématiques de transcription du modèle flot de données vers les systèmes matériels**

  Le chapitre 6 présentera les problématiques majeures de la transcription du modèle flot de données vers les systèmes matériels. Pour cela, nous utiliserons l'outil CAPH et les mécanismes sous-jacents comme cadre d'expérimentations.

  Il résultera de ces travaux une série de recommandations sur l'impact de la formulation des acteurs et des réseaux sur les performances finales. Une autre part concerne certaines optimisations possibles sur les modèles de FSM utilisés lors de la génération automatique du code VHDL par l'outil CAPH afin d'améliorer les transcriptions automatiques des acteurs flot de données vers leur implémentation matérielle.

  Ces recommandations ont pour objectifs de fournir au développeur des informations sur la manière d'aborder la formulation flot de données mais aussi d'ouvrir des pistes de réflexion sur les futurs travaux à entreprendre afin d'améliorer la génération automatique de l'outil CAPH.

2

Méthodologies d'implémentation d'algorithmes sur FPGA

Bien que les circuits intégrés dédiés (Application-specific integrated circuit (ASIC) ou Image Signal Processor (ISP)) puissent être des solutions efficaces en terme de puissance de calcul ou d'efficacité énergétique, il a été choisi de s'intéresser aux architectures FPGA pour leur grande flexibilité de programmation, idéale pour le prototypage d'algorithmes, afin de répondre aux problématiques de traitement d'images embarqué.

Après avoir présenté en détail les architectures Field Programmable Gate Array (FPGA), les notions de synthèse matérielle seront introduites ainsi que les méthodes de programmation classique (langage HDL). Dans un second temps, les problématiques introduites par l'utilisation des langages HDL seront développées. Ensuite, les concepts de la synthèse haut-niveau seront présentés ainsi que les différents outils existants. Enfin, ce chapitre sera conclu par l'introduction du modèle flot de données pour la programmation des FPGAs, avec la présentation de l'approche et des outils utilisés dans ces travaux.

2.1 FPGAs

Depuis leur création dans les années 1980 par la société Xilinx, les FPGAs fournissent des ressources et des fréquences de fonctionnement croissantes, permettant ainsi leur utilisation sur des applications toujours plus complexes. L'architecture d'un FPGA simplifié, illustré sur la Fig. 2.1, est à minima composée de trois éléments :

- **Les blocs logiques** constituent le cœur du FPGA. Ces cellules sont constituées d'éléments logiques programmables disposés sous forme matricielle, comme le montre la Fig. 2.1. Chaque bloc logique est identique aux autres et peut être relié à ses voisins par le biais d'un réseau d'interconnexion.
- **Le réseau d'interconnexion des blocs** relie les éléments logiques entre eux afin de réaliser des fonctions complexes. Au sein d'un FPGA, les interconnexions sont souvent hiérarchisées, où chaque niveau de hiérarchie supporte une fréquence de transmission différente.
- **Les entrées/sorties configurables** sont des cellules permettant d'interfacer le FPGA avec l'environnement extérieur. Chaque bloc d'entrée/sortie contrôle une broche du composant et peut être défini en entrée, en sortie, en signal bidirectionnel ou être inutilisé (haute impédance). [1]

1. Le fonctionnement des entrées/sorties ne sera pas détaillé car leur connaissance n'est pas fondamentale pour l'interprétation des résultats donnés dans la suite du manuscrit.

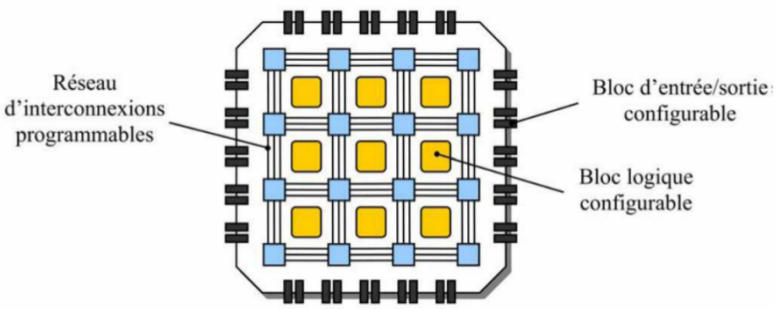
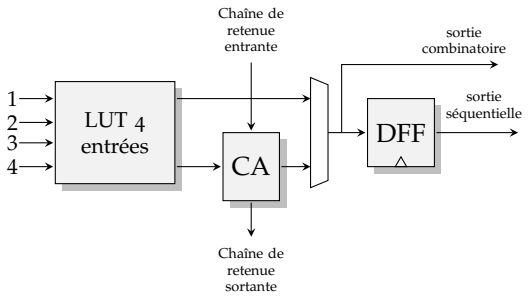


Fig. 2.1: Schéma simplifié de l'architecture d'un FPGA, source [Bir15]

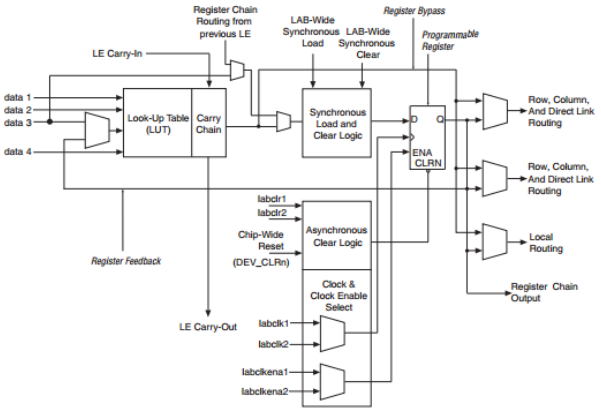
2.1.1 Blocs logiques

La fonction de base d'un bloc logique est de fournir les éléments de calculs et de stockage. Un schéma simplifié de bloc logique est illustré sur la Fig. 2.2a. Un bloc élémentaire est composé d'une partie combinatoire (LUT et Carry chain) et d'un élément de mémorisation (bascule D Flip-Flop (DFF)) pour la logique séquentielle. Une LUT à N entrées sert à implémenter des équations logiques, et se comporte comme une mémoire à 2^N emplacements. Afin de réaliser une fonction combinatoire, la table de vérité correspondante à l'équation souhaitée est chargée dans la mémoire.

La Fig. 2.2b illustre un exemple de bloc logique du Cyclone III d'Altera, appelé un Logic Element (LE) [Altb]. Chaque LE dispose de six entrées : quatre provenant du réseau d'interconnexion, une provenant de la chaîne retenue, et une de la chaîne de registre. Les signaux de contrôles du registre [2] proviennent également du réseau d'interconnexion. Il existe deux modes de fonctionnement possibles pour une cellule logique de ce type : le mode normal utilisé pour les fonctions combinatoires ou séquentielles et le mode arithmétique utilisé pour implémenter des opérations tels que des additions, des accumulations, des comptages ou bien des comparaisons.



(a) Schéma simplifié d'un bloc logique



(b) Bloc logique du cyclone III [Altb]

Fig. 2.2: Bloc logique d'un FPGA

Toutefois, les éléments logiques présentés sur la Fig. 2.2 ont évolué au profit de structures offrant une plus grande flexibilité, permettant d'optimiser le placement de la logique sur le FPGA. La société Altera a introduit un nouveau type de bloc élémentaire, appelé Adaptive Logic Modules (ALM) (illustré sur la Fig. 2.3).

2. clock, enable, clear, load

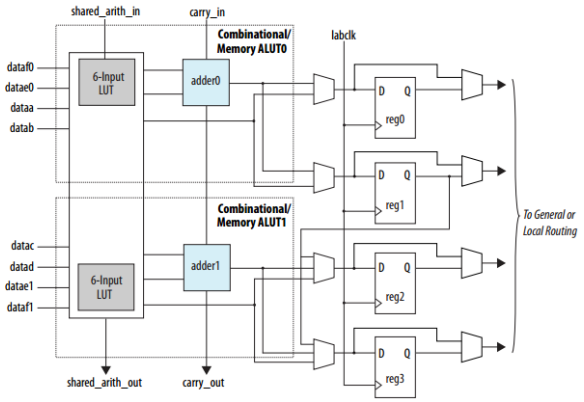


Fig. 2.3: Représentation haut-niveau d'un ALM (Cyclone V). Ce schéma ne montre pas toutes les connexions internes entre les composants qui permettent de changer de mode de fonctionnement mais seulement les connexions de principe entre les modules.

Chaque ALM comporte quatre registres et les modes de fonctionnement ont été étendus par rapport aux LE. La liste suivante énumère les différents modes de fonctionnement :

Mode normal Dans le mode normal, il est possible d'implémenter deux fonctions de quatre entrées ou une seule fonction de six entrées dans un ALM. Jusqu'à huit entrées provenant de l'extérieur du bloc (interconnexion) peuvent être utilisées en tant qu'entrée de la logique combinatoire.

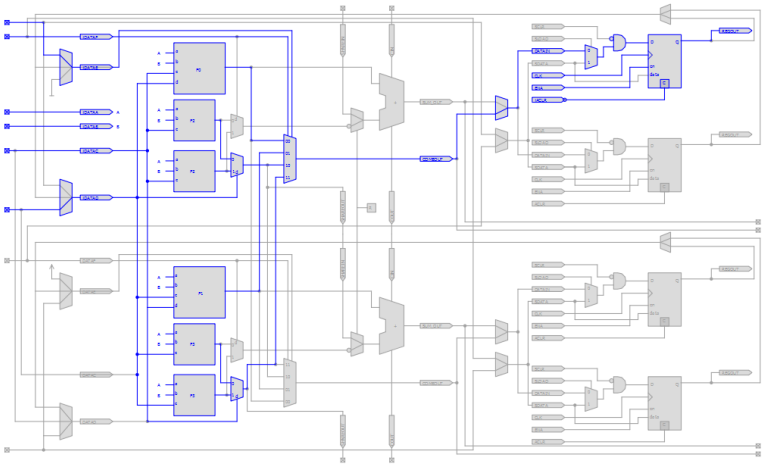


Fig. 2.4: ALM en mode normal. Sur cet exemple, un seul registre de sortie est utilisé.

Mode LUT étendue Dans le mode LUT étendue, la LUT peut être connectée à sept entrées mais la sortie n'est pas synchronisable par un registre en sortie. Ce mode est couramment utilisé pour les structures *if-else* des langages de description HDL. La huitième entrée de l'ALM non utilisée est potentiellement connectable à un registre de sortie, permettant une optimisation en surface du compilateur (technique de *register packing*). Cette optimisation regroupe dans le même ALM la fonction purement combinatoire de la LUT et la mémorisation d'un signal provenant de l'extérieur du bloc (non dépendant de la sortie de la LUT).

Mode arithmétique Un ALM en mode arithmétique utilise deux LUTs quatre entrées avec deux additionneurs complets. Chaque additionneur dédié permet aux LUT d'effectuer des pré-additions logiques. De plus, chaque additionneur dédié peut utiliser le résultat de l'autre additionneur avant de commander la sortie. La chaine de retenue[3] fournit des propagations de retenue rapide entre deux blocs d'additions.

3. Carry chain

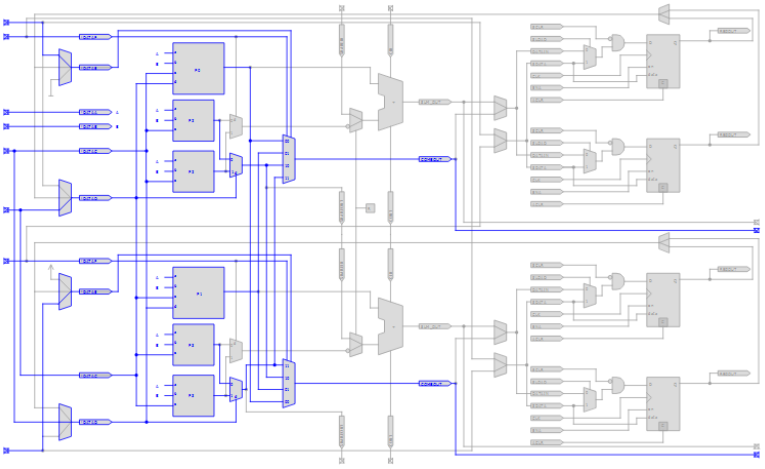


Fig. 2.5: ALM en mode LUT étendue. La fonction exécutée est purement combinatoire ici. Un registre est disponible pour la mémorisation d'une donnée provenant de l'extérieur du bloc.

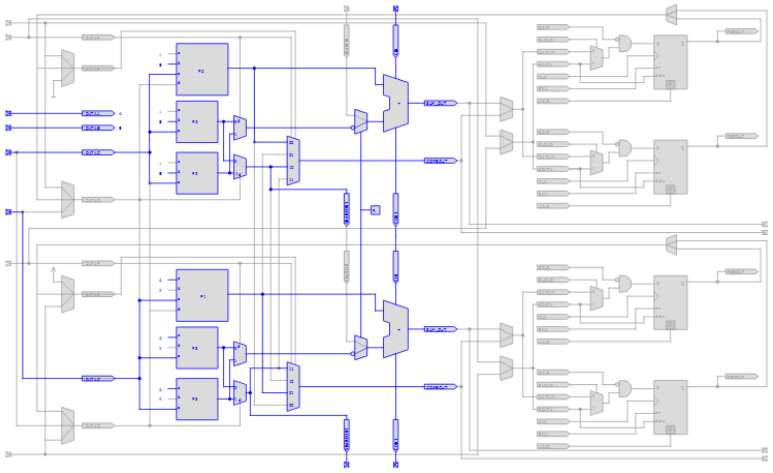


Fig. 2.6: ALM en mode arithmétique

Mode arithmétique partagé Dans ce mode, l'ALM est configuré avec quatre LUTs quatre entrées. Chaque LUT calcule la somme des trois entrées ou la retenue des trois entrées. Le calcul de la retenue de sortie est transmis au prochain bloc additionneur via une connexion particulière, appelé la chaine arithmétique partagée.

Maintenant que l'architecture des blocs logiques a été présentée, le réseau d'interconnexion permettant de construire des applications va être exposé.

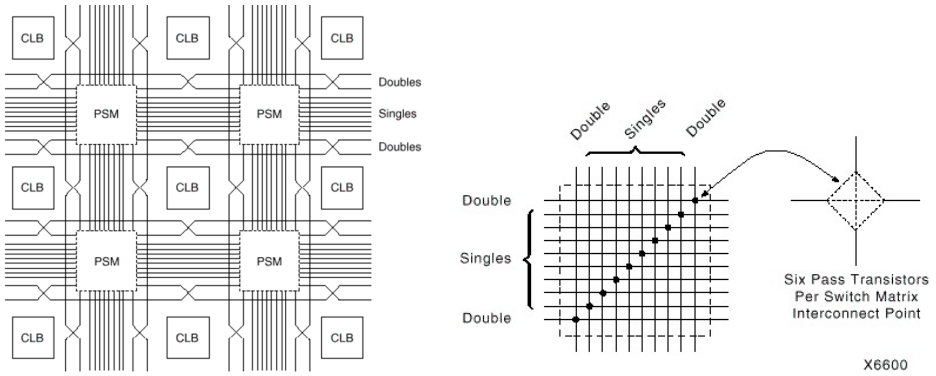
2.1.2 Interconnexion des blocs

La topologie du réseau d'interconnexion diverge d'un FPGA à l'autre, bien que les topologies récentes utilisent une représentation hiérarchique comprenant plusieurs niveaux d'interconnexion. Il existe différents types de connexions possibles, illustrés sur la Fig. 2.7 :

- **Les interconnexions directes** sont des liaisons entre blocs logiques ou entre un bloc logique et une broche d'entrée/sortie.
- **Les lignes** parcourent toute la longueur et la largeur du FPGA. Ces lignes peuvent être de différent type selon leur dimension (exemple : *single-length*, *double-length* ou *longlines* pour un FPGA Xilinx *XC*4000). Les lignes courtes offrent plus de flexibilité et un routage rapide entre des blocs adjacents, mais traversent forcément toutes les matrices d'interconnexion. Lorsque les connexions sont éloignées, le passage par les matrices d'interconnexion ralentit la propagation des signaux dans le FPGA, limitant la

fréquence maximale de fonctionnement de l'application. L'utilisation des lignes longues (traversant moins de matrices d'interconnexions) permet d'obtenir une propagation des signaux plus rapides entre deux blocs éloignés.

- **Les matrices d'interconnexion** permettent de relier un bloc à n'importe quel autre bloc du FPGA. Ces systèmes sont des aiguilleurs configurables situés à chaque intersection et raccordent les lignes entre elles selon diverses configurations (Fig. 2.7b).



(a) Interconnexion d'un FPGA *(b)* Illustration d'une matrice d'interconnexion (PSM)

Fig. 2.7: Interconnexion interne d'un FPGA

Toutefois, les connexions entre les blocs sont un facteur primordial pour les performances des implantations FPGAs. Plus la connexion entre deux blocs sera longue, plus le temps de transmission d'une donnée sera également long, faisant chuter la fréquence maximale de fonctionnement. Ceci est dû au modèle équivalent des lignes (Réseau RC [HWY+09]), illustré sur la Fig. 2.8. Le placement des éléments sur le FPGA est donc un élément crucial pour les performances.

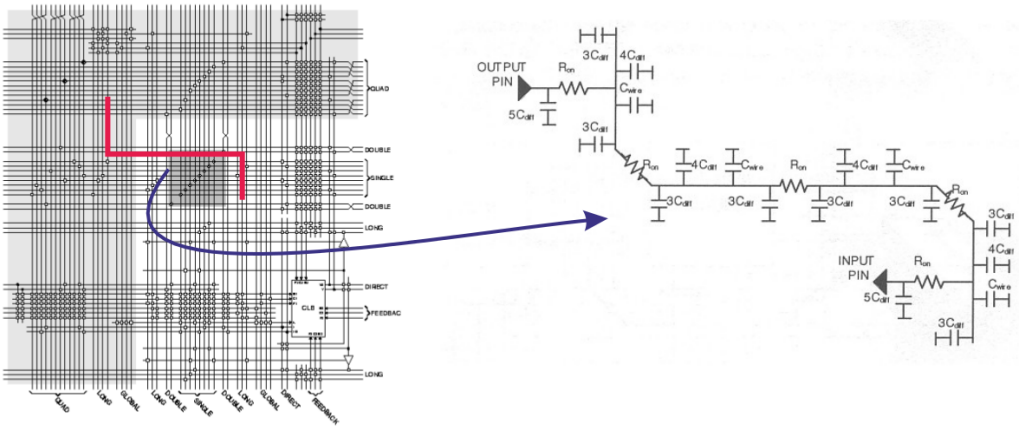


Fig. 2.8: Exemple de routage dans un FPGA avec le modèle équivalent, source [GR12]

Avec l'évolution des FPGAs qui intègrent de plus en plus d'éléments logiques, les systèmes d'interconnexions ont été hiérarchisés afin d'améliorer les performances en fréquence des applications. Les éléments logiques sont désormais regroupés dans des entités plus importantes, appelées Logic Array Blocks chez Altera et Configurable Logic Block (CLB) chez Xilinx. Les éléments au sein d'un même LAB ont des communications privilégiées, afin de construire des fonctions sans passer par le réseau d'interconnexion, et ainsi obtenir de meilleurs fréquences de fonctionnement. Un schéma d'un LAB de Cyclone V est donnée sur la Fig. 2.9.

Chaque LAB d'un Cyclone V regroupe dix éléments logiques (ALM). Il contient aussi des chaînes de propagation retenus (*carry chains*) pour le transfert du résultat d'un calcul arithmétique et des chaînes de registres (*register chains*) pour le transfert d'une sortie séquentielle

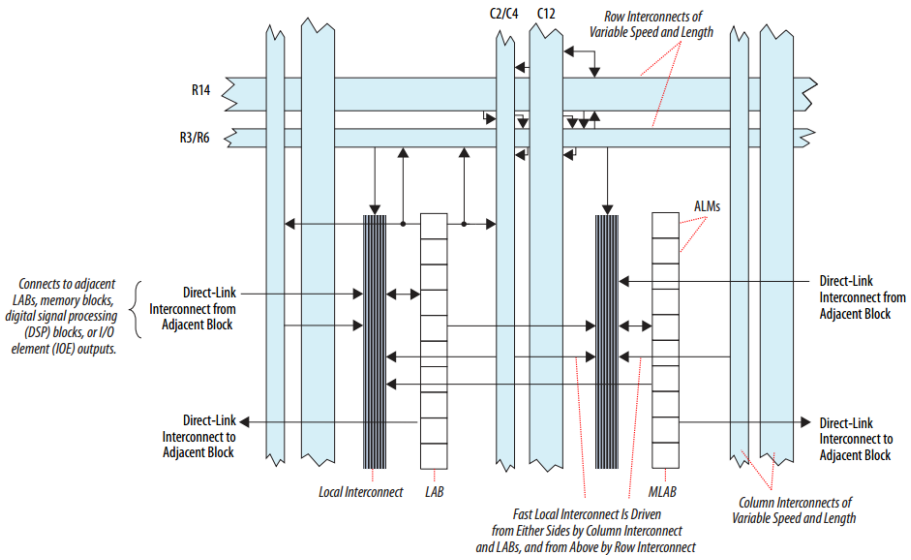


Fig. 2.9: LAB Cyclone V [Altc]

d'un élément logique à l'autre au sein du même LAB. Des liens de communications directs (*Directs Links* sur la Fig. 2.9) servent aussi à piloter le LAB à partir d'éléments extérieurs tels que les LABs adjacents, les blocs mémoires SRAM ou les unités multiplieurs DSP. Chaque LAB peut piloter jusqu'à trente ALM à travers les liens directs.

2.1.3 Blocs Additionnels

Dans le cas des architectures récentes tel que le Zynq de Xilinx [zyn], des éléments supplémentaires sont disponibles au sein du FPGA tels que :

- Les blocs mémoires synchrones SRAM double-port *dédiés*, idéals pour les mémorisations importantes telles que des FIFOs. Les blocs fournissent deux ports indépendants en entrée et en sortie, partageant le même espace mémoire. Suivant les versions des composants, un Zynq peut fournir entre 60 et 465 blocs mémoires, chacun supportant $36Kb$. Chaque port peut être configuré en fonction de la dynamique des données d'entrées, gérant automatiquement l'alignement des données en mémoire ($32K \times 1bit, 8K \times 4bits$,...).
- Les blocs mémoires *distribués* peuvent être implantés n'importe où dans le FPGA, à contrario des blocs SRAM dédiés qui ont une position fixe. Ces blocs sont crées automatiquement à partir d'un ensemble de blocs logiques regroupés au sein d'un même LAB. Ce type de mémoire est idéal pour les petites mémorisations (registres à décalages, petites FIFOs). A titre d'exemple, une mémoire distribuée MLAB chez Altera fournit 640 bits de mémoires. De plus, il est possible d'allouer jusqu'à 25% de la logique disponible en tant que MLAB, procurant jusqu'à $1.7Mbits$ de mémoire supplémentaire dans un Cyclone V.
- Les blocs DSP sont des unités arithmétiques optimisées en performance et en consommation pour effectuer des opérations de multiplications/accumulations (MAC). Différentes précisions sont possibles (9, 18, 27 bits pour les opérandes). Il est également possible de faire des calculs directement sur des nombres complexes.
- Le *Hardcore* qui remplace les systèmes Co-Design à base de softcore (NIOS, MicroBlaze). Aujourd'hui les deux principaux concurrents (Altera et Xilinx) utilisent des processeurs ARM connus pour leur faible consommation. L'intégration des cœurs de processeurs ouvre de nombreuses possibilités en terme de flexibilité, de reconfiguration (le hardcore peut reconfigurer le FPGA dynamiquement) ou de communication avec le monde extérieur. Ces systèmes intègrent également de nombreux contrôleurs matériels dédiés[4].

4. Voici quelques exemples de contrôleurs intégrés : Ethernet, USB OTG, NAND Flash, QSPI, SD, MMC, SPI, I2C, UART, CAN, SDRAM, DDRAM

Le support de cartes SD et de mémoires DDR permettent la mise en œuvre de systèmes d'exploitation légers (Micro OS Linux), simplifiant d'une part considérablement la mise en œuvre des communications avec l'extérieur (couches réseau), et d'autre part, améliorant la portabilité des applications de Co-Design plus efficacement (serveur SFTP, compilateur standard, etc,...).

Après avoir abordé l'architecture des FPGAs, leur flot de conception va être présenté.

2.2 Flot de conception FPGA

Afin de passer d'un code source, décrit dans un langage HDL, à l'exécution de ce code sur une cible FPGA, il convient de respecter un certain flot de conception, illustré sur la Fig. 2.10, et dont les étapes vont être détaillées par la suite.

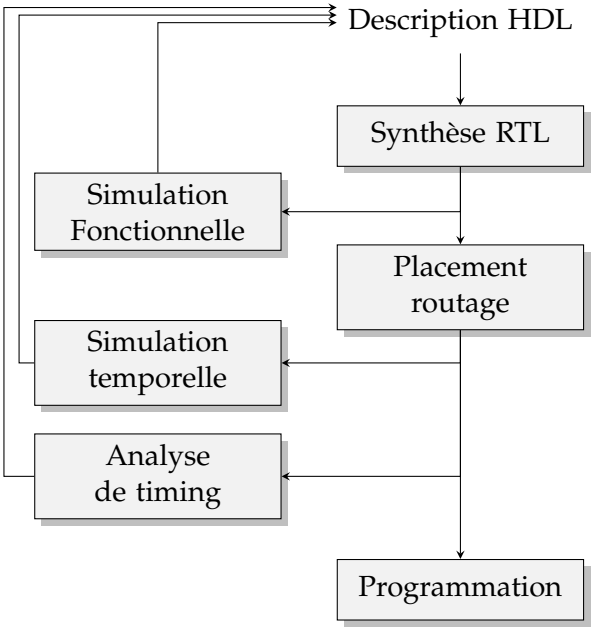


Fig. 2.10: Flot de conception FPGA

1. **Synthèse Register Transfer Level (RTL).**
   La première étape, appelé synthèse, consiste à passer de la transcription HDL à une représentation dite Register Transfer Level (RTL). Tout d'abord, l'outil d'analyse et de synthèse crée une base de données du projet intégrant tous les fichiers sources. A partir de cette base, la complétude et la consistance du projet est testée, en particulier en vérifiant les connexions entre les modules et les erreurs syntaxiques.
   Le synthétiseur infère en premier les éléments de base tels que les bascules, les LUTs et les machines d'états. Ensuite, il extrait du code source des macros correspondantes à des comportements ou des opérations spécifiques (mémoire RAM, registres à décalages, multiplexeurs, additions/soustractions/multiplications), qui seront remplacées par la suite par des composants du constructeur. Il est d'ailleurs souvent conseillé par les fabricants de FPGA d'utiliser directement leurs composants dans le code source, ou de respecter une certaine façon d'écrire le code source, afin de guider le synthétiseur dans son analyse et de permettre d'obtenir des performances optimales. Toutefois, le code devient dépendant d'un constructeur, voire d'une cible particulière. L'outil effectue ensuite une optimisation du design en minimisant le nombre de portes et la représentation de chaque machine d'états, supprimant la logique redondante ou inutile. A ce niveau, des contraintes de temps associées au projet sont prises en compte . Enfin, un regroupement des ressources combinatoires et séquentielles est effectué afin de minimiser le nombres de blocs logiques utilisés suivant la technologie utilisée.

Après le processus de synthèse, une étape de simulation fonctionnelle est possible afin de vérifier que le comportement du système répond aux fonctionnalités attendues.

2. **Placement routage**. L'étape suivante est subdivisée en deux parties. La première transforme la représentation RTL en une représentation niveau portes logiques, prenant en compte les caractéristiques du FPGA ciblé. La seconde étape effectue le placement de la logique obtenue sur le FPGA souhaité.
   Une étude de la fréquence du système est ensuite effectuée afin d'obtenir les performances maximales du système. Une dernière étape de simulation dite post-routage est possible. Cette étape prend en considération les caractéristiques du FPGA et le placement effectué afin de vérifier que les temps de propagation dans le FPGA ne contrarient pas le bon fonctionnement du système. A la fin de ce processus, une analyse précise des temps de propagation entre les éléments du FPGA permet de fournir à l'utilisateur une information sur la fréquence maximale de fonctionnement de son application.
3. **Génération du *bitstream* et programmation du FPGA**.
   La dernière étape transforme la représentation obtenue en un fichier de configuration (*bitstream*) qui sert à programmer les connexions internes et des blocs logiques du FPGA.

Le flot de conception décrit sur la Fig. 2.10 est désormais complètement automatisé par les outils des fournisseurs de FPGA qui à partir d'une description HDL RTL produisent les fichiers de programmation du circuit.

2.2.1 Exemples de processus

Afin d'illustrer le flot de conception complet, considérons quelques exemples. Le premier exemple décrit, purement combinatoire, correspond à un multiplexeur 2 vers 1. Le listing 2.1 correspond à la description VHDL de ce composant.

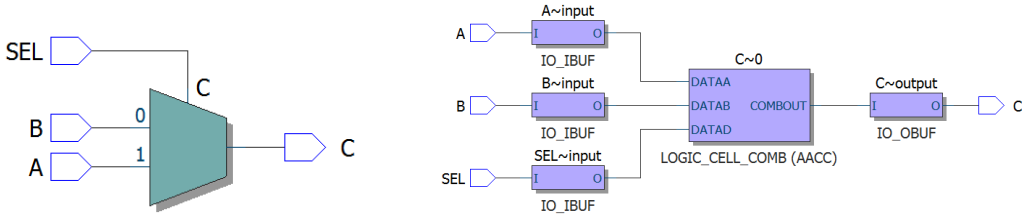
Listing 2.1: Code VHDL d'un multiplexeur 2 vers 1

```
entity mux2 is
  port (A: in std_logic;
        B: in std_logic;
        SEL: in std_logic;
        C: out std_logic);
end mux2;

architecture comb of mux2 is
begin
        C <= A when (SEL = '1') else B;
end comb;
```

Un multiplexeur est un élément de base présent dans tous les designs numériques, une macro est donc associée à cette opération dans le synthétiseur. La Fig. 2.11a illustre la transcription RTL du code VHDL faite par le synthétiseur. La Fig. 2.11b correspond à la représentation du multiplexeur après l'étape de placement routage sur le FPGA. Tout d'abord, les entrées et les sorties du système sont automatiquement associées à des blocs entrées/sorties du FPGA (IO_BUF). Ensuite, le multiplexeur de la Fig. 2.11a a été remplacé par un seul élément logique LE (LOGIC_CELL_COMB (AACC)) dont les entrées (DATAA, DATAB, DATAD) sont respectivement connectées aux IOs (A, B, SEL) et la sortie combinatoire (COMBOUT) à la sortie C.

La Fig. 2.12 montre l'allocation des éléments sur le FPGA. La première figure à gauche représente le FPGA Cyclone III complet, où un LAB est alloué (encerclé sur la figure) ainsi que les quatre entrées/sorties nécessaires. Le LAB alloué (contenant 16 éléments logiques) est occupé avec un seul élément logique (encerclé) où est implanté le multiplexeur (schéma de droite).

A partir de l'équation du multiplexeur, il est possible de retrouver la valeur qui sera chargée dans la LUT au moment de la programmation. Cette valeur, donnée par l'outil, est



(a) Représentation RTL *(b)* Représentation post-routage

Fig. 2.11: Représentation RTL et post-routage du multiplexeur 2 vers 1. Le code VHDL du listing. 2.1 après synthèse se traduit par un multiplexeur. Après routage, les entrées/sorties (signaux A, B, C et SEL) sont remplacées par des composants entrées/sorties. Le multiplexeur est instancié dans un seul bloc logique.

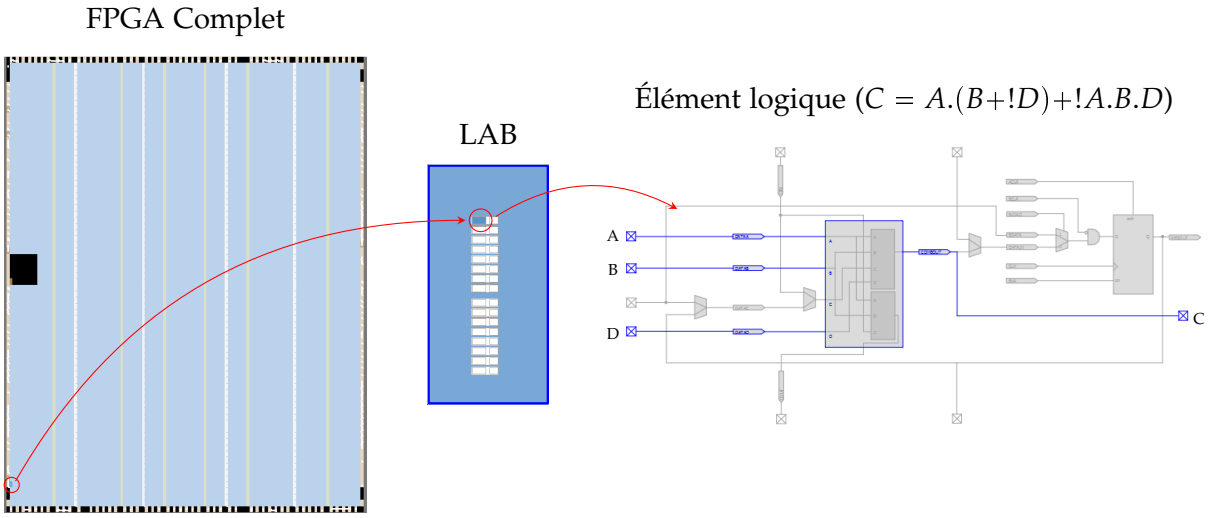


Fig. 2.12: Implantation finale sur le FPGA du multiplexeur 2 vers 1.

dans cet exemple égale à $CCAA$ (hexadécimal). La table de vérité permettant de retrouver la valeur qui sera programmée dans la LUT est illustrée sur la Fig. 2.13.

d	c	b	a	out	
0	0	0	0	0	C
1	0	0	0	0	
0	1	0	0	1	
1	1	0	0	1	
0	0	1	0	0	C
1	0	1	0	0	
0	1	1	0	1	
1	1	1	0	1	
0	0	0	1	0	A
1	0	0	1	1	
0	1	0	1	0	
1	1	0	1	1	
0	0	1	1	0	A
1	0	1	1	1	
0	1	1	1	0	
1	1	1	1	1	

Fig. 2.13: Table de vérité de l'équation du multiplexeur

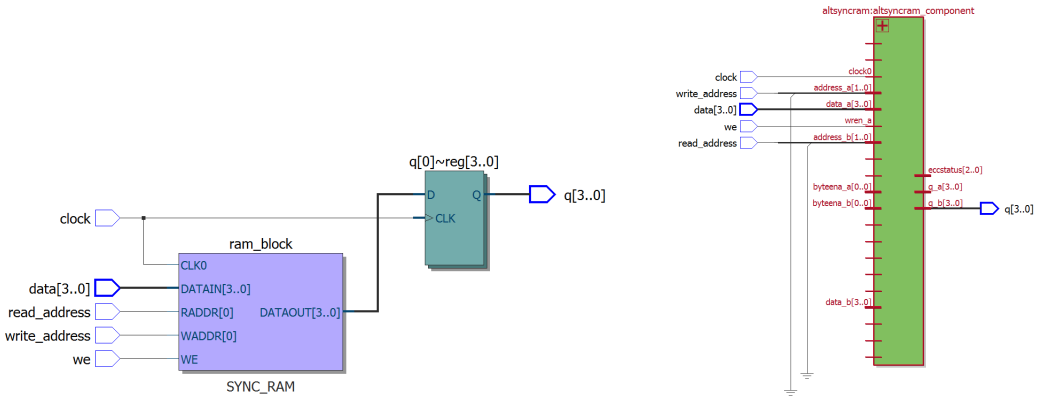
Un second exemple basé sur un processus séquentiel correspondant à une mémoire synchrone est présenté. Deux manières de spécifier le comportement de la mémoire sont décrites sur le listing 2.2. La première architecture (rtl_hdl) est une description en langage VHDL dont le synthétiseur transcrit le code source en un bloc mémoire SRAM. Cette représentation est la plus générique, grâce à l'utilisation stricte de constructions permises par le langage VHDL,

procurant le même résultat sur plusieurs outils différents (Xilinx, Altera,...). La seconde architecture (altera_comp) instancie un composant dédié fourni par le fabricant et spécialement paramétré pour la cible finale. Cette deuxième manière de faire est optimale en terme de ressources mais dépendante du FPGA choisi (ici Cyclone III Altera).

Listing 2.2: Code VHDL d'une mémoire SRAM

```
ENTITY memoire IS
 generic ( depth: integer := 2; size: integer := 4);
        PORT (
        clock: IN STD_LOGIC;                                    -- horloge
        data: IN STD_LOGIC_VECTOR (size-1 DOWNTO 0);            -- donnée d'entrée
        write_address: IN INTEGER RANGE 0 to depth-1;           -- adresse d'écriture
        read_address: IN INTEGER RANGE 0 to depth-1;            -- adresse de lecture
        we: IN STD_LOGIC;                                       -- signal de contrôle d'écriture
        q: OUT STD_LOGIC_VECTOR (size-1 DOWNTO 0));             -- sortie de la mém
END memoire;

-- Première méthode: Code VHDL
ARCHITECTURE rtl_hdl OF memoire IS
-- déclaraton d'un tableau de la longueur de la mémoire
TYPE MEM IS ARRAY(0 TO depth-1) OF STD_LOGIC_VECTOR(size-1 DOWNTO 0);
SIGNAL ram_block: MEM;
BEGIN
        PROCESS (clock)
        BEGIN
                IF (clock'event AND clock = '1') THEN       -- sur chaque front montant
                        IF (we = '1') THEN                      -- si validation écriture
                                ram_block(write_address) <= data; -- on écrit la donnée ds mem
                        END IF;
                        q <= ram_block(read_address);    -- lecture synchrone
                END IF;
        END PROCESS;
END rtl_hdl;

-- Seconde méthode: Composant constructeur
ARCHITECTURE altera_comp OF memoire IS
BEGIN
        altsyncram_component : altsyncram
        GENERIC MAP (
                intended_device_family => "CycloneIII",
                lpm_type => "altsyncram",
                numwords_a => depth,
                widthad_a => depth,
                width_a => size)
        PORT MAP (
                address_a => std_logic_VECTOR(to_unsigned(write_address,depth)),
                clocko => clock,
                data_a => data,
                wren_a => we,
                address_b => std_logic_VECTOR(to_unsigned(read_address,depth)),
                q_b => q);
END altera_comp;
```

La fig 2.14 illustre les deux résultats de synthèse.



(a) Schéma RTL - Architecture rtl_hdl

(b) Schéma RTL - Architecture altera_comp

Fig. 2.14: Représentations RTL des deux approches courantes pour l'instanciation d'une mémoire SRAM par un synthétiseur.

Sur la Fig.2.14a, un registre de sortie est ajouté (non représenté sur la Fig.2.14b où le registre fait partie du composant fabricant). Ce registre découle de l'affectation de la sortie q dans un processus synchrone. Ce schéma est nécessaire car les composants mémoires nécessitent une écriture et une lecture synchrone pour être utilisés. Malgré cette légère différence au niveau de la représentation RTL, les deux descriptions du composant aboutissent au même résultat final une fois le placement-routage effectué, à savoir l'instanciation d'une mémoire SRAM double-port, illustré sur la Fig. 2.15.

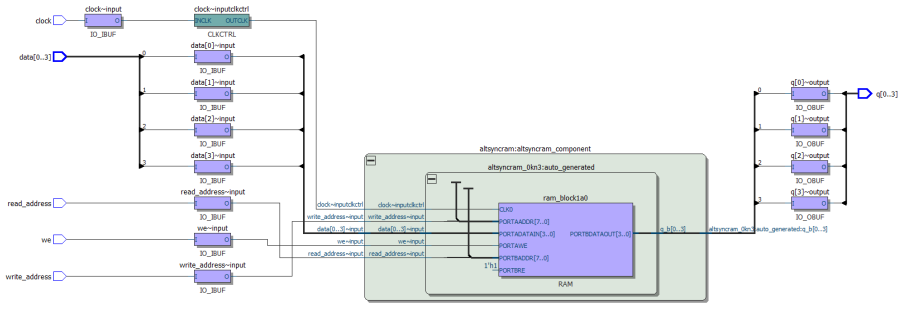


Fig. 2.15: Représentations post-routage de la mémoire SRAM

La Fig. 2.16 illustre l'implantation finale de la mémoire sur le FPGA.

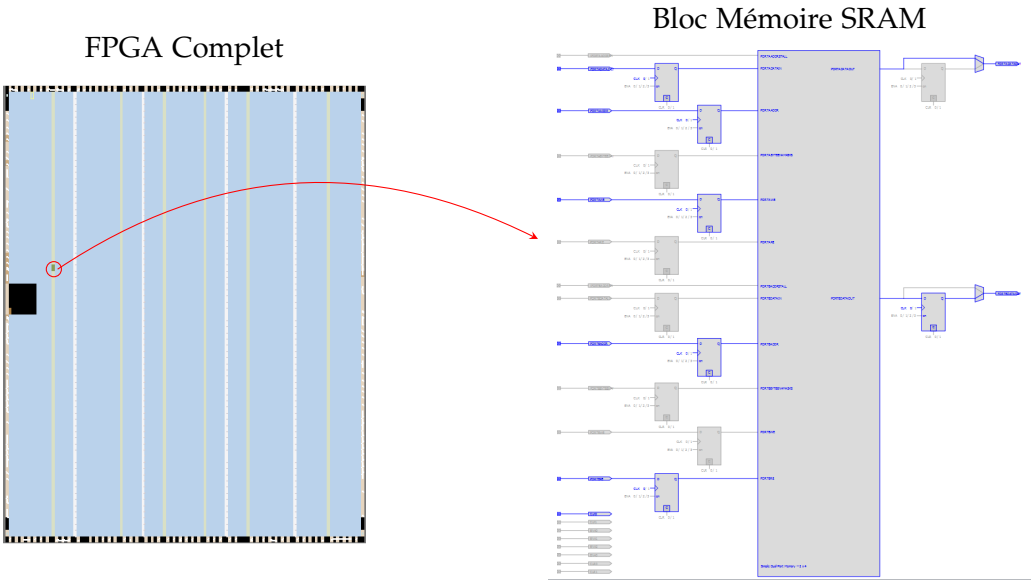


Fig. 2.16: Implantation finale sur le FPGA d'une mémoire SRAM.

Avec un exemple tel que celui de la mémoire SRAM, il a été montré que plusieurs approches sont possibles pour la description d'une application. Bien que le processus de synthèse ait abouti au même résultat dans cet exemple, il existe de nombreux cas où la convergence des résultats après le placement-routage n'est pas assurée. Même si le synthétiseur aboutit à l'instanciation d'un même composant, sa configuration sera souvent différente. Prenons à titre d'exemple un composant de division. L'utilisation de l'opérateur arithmétique VHDL standard fournira une vitesse de fonctionnement 3 fois inférieure (sur un Cyclone III) à l'utilisation d'un composant spécifique, car ce dernier pourra être paramétré en nombre d'étages de pipeline (impossible à faire avec l'utilisation d'un simple opérateur).

Le choix du type de description de l'application ainsi que l'ensemble des paramètres de l'outil de synthèse influeront sur les performances finales d'une application. L'approche avec les langages HDL a toutefois l'avantage de fournir les meilleurs résultats en terme d'implémentation (ressources et vitesse) mais nécessite de la part du développeur de bonnes connaissances en conception numérique afin de pouvoir transcrire l'algorithme original en une description HDL synthétisable. Avec des algorithmes complexes, cette tâche peut devenir ardue et prendre un temps de mise au point considérable, ce qui a souvent restreint l'utilisation des FPGAs.

2.3 Les outils de synthèse haut-niveau (HLS) pour FPGA

En réponse aux problèmes de temps et de complexité de développement induits par l'usage des HDLs, de nombreux travaux ont été voués au développement d'outils de synthèse de haut niveau High Level Synthesis (HLS). Cette deuxième approche est illustrée sur la Fig. 2.17. Ces outils ont pour objectif de faciliter l'accès à l'utilisation des FPGAs par des personnes non expertes en conception numérique. Une grande partie des HLS existants est basée sur des langages impératifs comme le C/C++, proposant aux utilisateurs des modèles de programmation avec lesquels ils ont l'habitude de raisonner.

2.3.1 Les HLS C-Likes

La plupart des outils de synthèse de haut-niveau sont basés sur des langages standard tels que le ANSI C/C++ et SystemC. De nombreux outils basés sur les langages impératifs ont été proposés, en voici une liste non exhaustive :

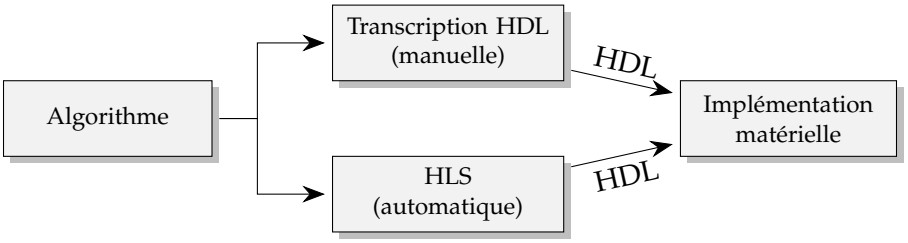


Fig. 2.17: Méthodes possibles pour le portage d'algorithmes sur une cible matérielle

- **Handle-C** [han] est un sur-ensemble du langage C apparu en 1996 à l'université d'Oxford. Le langage rajoute la possibilité de définir des sections parallèles (mot clé **par**) et construit des canaux de communication (FIFOs) entre chaque unité. Il supporte également les types de variables et signaux avec des précisions au niveau bit. Handle-C est désormais supporté par la société Mentor Graphics.
- **Mitrion-C** [mit] est également un HLS C-like proposé par Mitrionics. Le code est converti en un Mitrion Virtual Processor (MVP), un coeur de processeur implanté sur un FPGA.
- **Carte-C** [Poz05] utilise une approche structurelle avec la définition d'une librairie pré-synthétisée de modules matériels. Le développeur peut utiliser ces modules ou bien en définir de nouveaux (mais développés en langage HDL)
- **Stream-C** [GSAK00] a été proposé par le laboratoire de Los Alamos. Le langage est basé sur le modèle de programmation *Communicating Sequential Process* (CSP) de Hoare [Hoa78] et cible des applications opérant sur des flux de données.
- **Single-Assignment C** (SA-C) [NBD+03] est une variante du langage C utilisée pour automatiquement compiler un algorithme séquentiel vers un système reconfigurable FPGA. Pour cela, cette variante du C rajoute des restrictions au langage initial (suppression des pointeurs, des appels récursifs et des boucles *while*).
- **SPARK** [GDGN03] transforme également du C en VHDL. Ce HLS a été conçu afin d'étudier les impacts des transformations et des heuristiques appliqués à la production de code VHDL à partir de code C.
- **Mobius** [uAS07] est également basé sur le modèle CSP, comme Stream-C. Le compilateur extrait certaines primitives dont des équivalences sur des circuit électroniques sont connus.
- **Impulse-C** [imp] supporté par Impulse Accelerated Technologies, permettant l'écriture d'applications avec l'aide d'une librairie de fonctions décrivant des processus parallèles. Les communications entre processus sont basées sur un modèle de flot de données. L'utilisateur peut interfacer ces propres fonctions matérielles au sein de la librairie.
- **Catapult** [GT10] est certainement un des outils industriels les plus répandus. Projet initié en 2004 par la société Mentor Graphics et maintenant proposé par la Calypto Design Systems, Catapult supporte du C++ ou du SystemC comme langage de description et génère du code RTL pour FPGA ou ASIC. Les dernières avancées de cet outil sont principalement sur l'intégration de méthodes de vérification du code généré, primordiales dans le déploiement d'applications industrielles.
- **Vivado Xilinx HLS** [viv] fait partie de la suite d'outil du fabricant Xilinx. L'outil est basé sur un environnement de développement (EDI) Eclipse et utilise des directives de précompilation afin de générer le code RTL. Un ensemble de composants avancés (FFT, Factorisation de Cholesky's, noyau SVD) est fourni. L'EDI intègre également la compilation et la simulation RTL du composant. Lors du processus de transcription, l'outil génère les représentations en SystemC, VHDL et Verilog. Il est possible d'exporter directement un composant sous forme de bloc IP depuis le HLS, facilement intégrables dans un projet.

Tous ces outils ont pour caractéristique commune de dériver du même langage, souvent en rajoutant des restrictions et des spécifications similaires. Toutefois, les modèles de programmation impératifs, conçus pour être exécutés sur des architectures à base de processeurs, et le modèle d'exécution d'un FPGA diffèrent trop pour pouvoir passer de l'un à l'autre de manière automatique et efficace. En voici quelques raisons :

- L'extraction automatique du parallélisme du code original est une tâche complexe à mettre en œuvre. Même si les compilateurs récents sont capables de paralléliser certaines boucles de codes séquentiels comme sur les DSPs, il n'en résulte que des opérations SIMD. Dans le contexte d'une programmation FPGA, chaque partie de l'application travaille de manière totalement concurrente, ce qui est à l'opposé des concepts même des langages utilisés par les HLS basés sur des langages impératifs. En pratique, certains HLS vont avoir recours à des annotations (via des directives de précompilation) pour accompagner le compilateur dans l'analyse des sections parallélisables.
- Certaines constructions explicitant des concepts clés du design numérique comme la précision au niveau bit, la notion d'**horloge**, de synchronisation ou de hiérarchie entre modules manquent. Parmi ces concepts, la notion d'horloge est certainement une des plus importantes. La notion de temps est implicite dans un programme impératif. Le développeur spécifie une succession d'opérations logiques ou arithmétiques et le compilateur s'occupe entièrement du séquencement des instructions, quitte à réorganiser certaines instructions pour optimiser l'exécution. Dans le monde matériel, le temps au contraire est explicite, avec pour unité de base le cycle d'horloge.
- Réciproquement, certaines constructions des langages impératifs n'ont pas d'équivalent en matériel. La notion de l'allocation dynamique de mémoire qui est couramment utilisée dans les codes impératifs n'existe pas en description matérielle. Ceci implique que l'algorithme de base doit forcément être réécrit selon certaines règles. Ces règles sont fortement contraignantes pour les développeurs de logiciels qui ne sont pas familiers des approches matérielles.

Afin de limiter l'expansion des dérivées des langages ANSI C/C++, des industriels ont décidé de se regrouper autour d'un consortium pour la définition d'un standard de programmation commun à tous les types d'unités de calculs possibles, OpenCL.

2.3.2 OpenCL

Open Computing Language (OpenCL) [ope] est le premier standard open-source, sous licence gratuite, unifiant le modèle de programmation pour l'accélération d'algorithmes sur des systèmes hétérogènes. Il est issu d'un consortium d'universitaires et d'industriels (Khronos) visant à proposer un environnement de travail commun, dans le but d'éviter ainsi la multiplication des sur-ensembles du même langage.

OpenCL utilise également un sur-ensemble du standard ANSI C pour le développement de code mais propose un cadre plus général que la programmation FPGA, supportant l'exécution d'une multitude d'architectures (processeurs, GPUs ou encore DSPs). Les fonctionnalités d'OpenCL intègrent une interface de programmation (API) pour la communication entre une unité hôte (généralement un processeur) et un ou plusieurs **OpenCL Devices**, souvent via une liaison PCI Express. Un **OpenCL device** est divisé en une ou plusieurs unités de calculs (**Compute units** (CUs)) , elles-mêmes subdivisées en un ou plusieurs éléments de calculs (**processing elements** (PEs)).

La définition d'un programme OpenCL est séparée en deux parties : les noyaux (*kernels*) qui s'exécutent sur un ou plusieurs **OpenCL devices** et un programme hôte, définissant le contexte et l'exécution des *kernels*, et gérant les transferts de données. Lorsque qu'un *kernel* est soumis par le processeur hôte, une collection d'exécutions parallèles de ce noyau peut être invoquée sur la cible et chaque instance *(work-item)* exécutera le même *kernel* sur des données différentes. L'utilisation d'un identifiant par instance permet de séparer le chemin d'exécution

au sein du même noyau. Il est également possible d'exécuter différent noyaux en parallèle, permettant un parallélisme de tâche.

A ce jour, il est difficile de savoir si OpenCL se révèlera comme le standard incontournable de la HLS des prochaines années. La société Altera propose actuellement le support d'OpenCL pour certain de ces circuits mais sa mise en œuvre intègre de nombreuses couches d'abstraction, consommatrice de ressources.

2.4 Le modèle flot de données

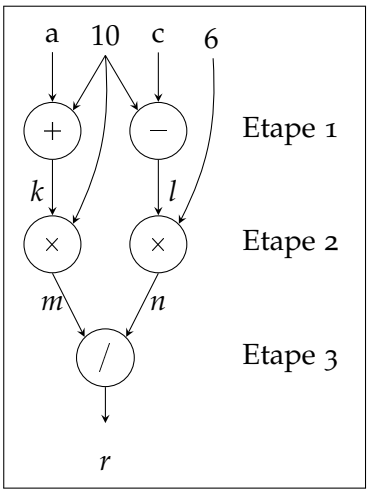
Dans le cas du traitement d'images pour le FPGA, les différences entre le modèle de programmation impératif et le modèle d'exécution d'un FPGA sont trop importantes pour obtenir une correspondance automatique et efficace. Une solution possible est de choisir un modèle de calcul pour la description des applications plus adapté à ce domaine, en particulier le modèle **flot de données**.

Le modèle flot de données est apparu du MIT en 1966 avec les travaux de Sutherland [Sut66] et a été formalisé par la suite par Dennis [Den74]. Le but de leurs travaux était de créer une architecture de calcul parallèle avec un langage de programmation associé dans laquelle des fragments de programmes pouvaient s'exécuter simultanément. Les architectures flot de données étaient capables de meilleures performances que les architectures de processeurs classiques, limitées par le goulot d'étranglement que constituent les échanges entre le processeur et la mémoire stockant le programme et les données [Bac78, AC86].

Les programmes flot de données sont décrits sous la forme un réseau d'unités de calcul, appelés **processus** ou **acteurs**, qui s'échangent des jetons (**tokens**) à travers des liens de communication unidirectionnels de type FIFO. La notion de temps est transparente dans un programme flot de données, où seule la notion de causalité est importante. Au sein d'un acteur, les exécutions sont déclenchées dès lors que les jetons nécessaires sont disponibles en entrée. Le comportement d'un acteur est défini par un ensemble de **règles d'activation** (*firing rules*). Une règle d'activation est définie par une combinaison des jetons disponibles en entrée et spécifie les jetons à produire en sortie. Lorsqu'une règle s'active, les jetons d'entrées sont consommés et les jetons en sortie produits. Un ensemble de règles définit alors la totalité du comportement d'un acteur.

Etape 1	$k = a + 10$
Etape 2	$l = 10 - c$
Etape 3	$m = k * 10$
Etape 4	$n = l * 6$
Etape 5	$r = m/n$

(a) Modèle Von Neumann



(b) Modèle flot de données

Fig. 2.18: Comparaison du modèle impératif et du modèle flot de données

Une comparaison entre le modèle flot de données et le modèle Von Neumann est donnée dans la figure 2.18 sur un exemple simple. Avec le modèle von Neumann l'exemple s'exécute en cinq étapes. Avec la formulation suivant le modèle flot de données, ce code ne requiert que

trois étapes. Les étapes 1 et 2, opérant sur des données indépendantes, peuvent être exécutées en parallèle, tout comme les étapes 3 et 4. Il est important de garder à l'esprit que chaque acteur peut s'exécuter de manière concurrente, différence avec un unique cœur de processeur qui ne peut exécuter qu'une opération à la fois même si les données sont indépendantes[5]. Une autre différence entre les deux modèles est l'absence de variable. Sur la Fig. 2.18b, les variables k, l, m, n de la Fig. 2.18a sont traduites par les liens de communications entre acteurs.

Suite aux propositions de Dennis, de nombreux travaux ont visé à formaliser la sémantique des modèles de calcul de type flot de données. Dans [LP95], en particulier, Lee et Parks ont défini le modèle **Dataflow Process Network** (DPN), inspiré des travaux de Dennis [Den74] et de Kahn [Kah74]. Un DPN est défini par un graphe orienté dans lequel les sommets sont des acteurs et les arcs des canaux de communications de type FIFO, comme le modèle de Dennis. La divergence entre les deux modèles se fait au niveau du comportement des acteurs. Dennis définit les acteurs comme des opérateurs logiques ou arithmétiques alors que la sémantique de Lee est plus générale, intégrant la notion de processus associé à un ensemble de règles d'activations (notion issue des processus de Kahn [Kah74]). Plus formellement, Le modèle DPN peut être défini comme un graphe orienté fini, caractérisé par $G =< A,\ F >$ où :

- A est l'ensemble des **sommets** du graphe. Chaque sommet représente un acteur du graphe consommant un ou plusieurs flots de jetons d'entrées et produisant un ou plusieurs flots de jetons de sortie. Un tel acteur $a \in A$ est défini par un t-uplet $a =< P^{in}_{data},\ P^{out}_{data},\ R,\ Rate >$ où :
  - P^{in}_{data} et P^{out}_{data} sont respectivement les ensembles des ports d'entrées et de sorties de l'acteur.
  - R est l'ensemble des règles d'activations d'un acteur. Une règle d'activation $R_i \in R$ est activée lorsque les conditions sur les entrées associées à la règle sont satisfaites.
  - $Rate : (R,\ P^{in}_{data},\ P^{out}_{data}) \rightarrow \mathbb{N}$ associe un nombre de jetons consommés ou produits sur un port de l'acteur, pour chaque activation de chaque règle R_i de l'acteur.
- $F \subseteq A \times A$ est l'ensemble des arcs du graphe G. Chaque arc est un lien de communication unidirectionnel de type First-In First-Out (FIFO) de capacité supposée infinie pour le transfert des jetons entre acteurs.

Le modèle DPN est à la base de nombreux modèles flot de données existants à ce jour. La sémantique du modèle définit le comportement *extérieur* des acteurs : les règles d'activation spécifient seulement les conditions sur les entrées associées à chaque règle et les données produites par leur activation. La description du comportement *interne* de l'acteur ne fait pas partie du modèle de calcul. Cette spécificité fait que depuis la formalisation de Lee, de nombreuses variantes basées sur les DPN sont apparues dans la littérature, où chacune offre un compromis différent entre les capacités de modéliser des comportements dynamiques et de prédictibilité du système. Les variations dans les modèles se situent principalement sur la sémantique des règles d'activation. On retrouve comme modèles les plus courants :

- **Dynamic Data Flow (DDF)** [BDT13] est un modèle de calcul dans lequel les acteurs peuvent avoir accès à un jeton sans le consommer, créant des lectures non bloquantes (interdites dans les KPN). Cette propriété forte rend les DDF non déterministes, complexifiant leur étude [LP95]. Le langage de programmation CAL est basé sur ce modèle.
- **Synchronous Data Flow (SDF)** [LM+87] est certainement la spécialisation statique du DPN la plus utilisée pour sa simplicité et sa prédictibilité. Chaque acteur SDF consomme et produit un nombre fixe de jetons à chaque activation. Ce modèle est très populaire car il permet de vérifier qu'un graphe est *déterministe* et *ordonnançable*, permettant notamment une détermination statique (à la compilation) de la taille des canaux de communication.

5. Avec les extensions SIMD, les cœurs de processeurs sont en mesure d'exécuter des opérations vectorielles mais ceci se limite à du parallélisme de données avec un nombre limité de données parallèles

- **Single-Rate SDF** est une spécialisation du modèle SDF dans lequel les taux de production et de consommations de tous les acteurs sont égaux. Il est possible de convertir un graphe SDF en **Single-Rate SDF** [SB09]. Ce modèle peut également être converti en un Direct Acyclic Graph (DAG) directement ordonnançable sur des architectures de processeurs à mémoire partagée.
- **Cyclo Static Dataflow (CSDF)** [BELP96]. Le modèle cyclo-statique (CSDF) étend le modèle SDF en définissant des motifs fixes de production/consommation. Par exemple avec un pattern (1,2), un acteur peut produire alternativement un ou deux jetons lors de son activation.
- **Parameterized SDF (PSDF)** [BB01] est une extension du modèle SDF qui permet de définir des rapports production/consommation relatifs à des paramètres qui peuvent être réévalués dynamiquement à certains points de reconfiguration.

D'autre part, des propriétés telles que l'*ordonnançabilité* ou la *consistence* peuvent être utilisées pour caractériser les applications et les modèles entre eux. Un graphe flot de données est *ordonnançable* s'il est possible de trouver une séquence d'activation d'acteurs qui satisfassent l'intégralité des règles d'inférences définies dans le graphe. En fonction des spécifications du modèle, l'ordonnançabilité peut être vérifiée à la compilation ou à l'exécution. Dans certains modèles dynamiques, des inter-blocages (deadlock) sont parfois possibles. Un graphe sera *consistant* si son exécution ne causera pas une accumulation infinie de jetons dans une ou plusieurs FIFOs. En théorie, les FIFOs d'un DPN sont supposées illimitées mais les ressources mémoires sont limitées en pratique, nécessitant un dimensionnement précis des FIFOs. Un graphe sera donc considéré *inconsistant* s'il peut générer des débordements dans les FIFOs.

D'autres propriétés sur les modèles permettent de comparer les modèles entre eux. Il est complexe de dresser une liste exhaustive de ces propriétés de part la multitude existant dans la littérature. Desnos propose une liste intéressante des propriétés utilisables pour comparer les modèles flot de données dans [Des14].

2.5 Application du modèle flot de données aux systèmes matériels pour le prototypage rapide

Malgré les performances que pouvaient procurer les premières machines flot de de données dans les années 1980-1990, elles ont été abandonnées à cause de leur coût de fabrication prohibitif [W+94]. A cette époque, les circuits électroniques configurables ne fournissaient pas suffisamment de capacité d'implémentations pour supporter des applications réelles. Aujourd'hui avec les récentes avancées dans les procédés de gravure, les FPGAs offrent des matrices de logiques importantes, ouvrant l'utilisation de ce type de circuit à de nouvelles applications. Ces avancées relancent aujourd'hui l'intérêt du modèle flot de données pour la programmation des FPGAs. Cet intérêt a d'ailleurs été de nouveau soulevé dès la fin des années 1990 où le modèle flot de données a été relancé par Najjar, qui dans [NLG99], avait déjà souligné les correspondances entre le modèle de programmation flot de données et le modèle d'exécution des FPGAs. Les principales correspondances soulignées par Najjar sont les suivantes :

- Les données sont représentées par des valeurs sur des arcs et non des adresses mémoires. Il en résulte que l'ordre des opérations dépend seulement de l'ordre d'arrivée des données sur les liens et non de dépendances introduites par l'utilisation d'un espace mémoire. En terme d'exécution sur un FPGA, ce type de communication se traduit par une connexion de type FIFO, dont l'utilisation est courante pour la synchronisation entre blocs de traitement.
- Les opérations sont purement fonctionnelles. Les sorties d'un acteur dépendent seulement de ses entrées, ce qui sémantiquement correspond aux opérations combinatoires effectuées au sein d'un bloc logique FPGA. (l'équation de sortie d'une LUT ne dépend que de ses entrées). Les opérations d'un acteur flot de données sont ainsi facilement synthétisables sur un FPGA.

- Toutes les formes de parallélisme (données, contrôle, flux) sont exprimables via un graphe flot de données. Le modèle d'exécution d'un FPGA, basé sur des unités concurrentes, permet de supporter ces différents niveaux naturellement.

Vis à vis des problématiques de la HLS, le recours à un modèle de programmation similaire au modèle d'exécution de la cible matérielle évite de concevoir un système de reformulation inter-modèle (évitant ainsi de retomber sur les problématiques de la HLS sus-citées). Dans le cadre du traitement d'images, de précédents travaux ont démontré que le modèle flot de données pouvaient être utilisé pour exploiter des unités de calcul parallèles dédiés au traitement d'images temps réels [SQZ93, SQZ95]. De fait, plusieurs langages et outils se proposant de faciliter la programmation des FPGAs en s'appuyant sur le modèle flot de données ont été proposés. Tous ces langages peuvent être vus comme des Domain Specific Language (DSL), autrement dit des langages orientés domaine où la notion de domaine recouvre ici à la fois le modèle de programmation et la classe d'applications visées. On citera par exemple :

- **StreamIT** [str] est un langage de programmation et un compilateur développés par le MIT, spécialement conçu pour les applications flot de données. Un programme est représenté par un graphe où les noeuds, appelés *filtres*, encapsulent le traitement et les arcs sont des canaux de communications de type FIFO. StreamIt est basé sur le modèle SDF. Chaque filtre est une fonction qui sera exécutée lorsque ses données d'entrées seront disponibles sur les FIFOs du filtre. StreamIt peut produire des applications sur un ensemble d'architectures (processeurs mono ou multi-coeurs, clusters) Le modèle d'exécution ne fait aucune hypothèse sur les spécifications de l'architecture matérielle. Un back-end particulier par architecture ajoutera certains restrictions sur les conditions d'activation des règles en fonction de l'architecture ciblée. Par exemple, un processeur monocoeur ne peut exécuter qu'une seule règle à la fois alors qu'un processeur multicoeur pourra activer plusieurs règles simultanément (à condition qu'il n'y ait pas de dépendances de données entre les règles). StreamIt sert de langage d'entrée à l'outil Optimus [HKM+08] qui génère un code Verilog à partir d'une description StreamIt. Les filtres sont synthétisés à partir de squelettes de traitement déjà existants.
- **Canals** [DEY+09] est un langage de modélisation d'application flots de données avec un compilateur associé. Le langage est basé sur le concept de noeuds (*kernels*) et de réseaux. Un noeud (ou un noyau) est une unité de calcul décrite de manière impérative, consommant un taux fixe de jetons en entrée et produisant un nombre d'éléments fixes en sortie. Un noyau peut également observer les valeurs sur ces entrées sans les consommer. Des noyaux spéciaux permettent de distribuer et de regrouper la distribution des données Afin de former une application, un réseau est défini par l'utilisateur. Le réseau regroupe l'instanciation des noyaux ainsi que des connections entre noyaux

  L'ordonnancement est un point critique des applications flot de données sur des architectures processeurs. Canals donne la possibilité à l'utilisateur de définir son propre algorithme d'ordonnancement, afin d'obtenir les meilleurs performances possibles. Le compilateur Canals génère tout d'abord un premier un modèle comportemental, puis un modèle d'architecture depuis le programme. A partir de cela,un modèle de l'implémentation est crée. Ce modèle d'implémentation sert de point d'entrée à des générateurs de codes spécifiques (C, C++, Cell processor ou FPGA Altera) qui rajouteront une Hardware Abstraction Layer (HAL) contenant les mécanismes de communication avec la cible finale.
- **RVC-CAL** [BEJ+11]. RVC-CAL est un sous-ensemble du langage orienté acteur CAL, initié par le projet Ptolemy [EJ03] de l'université de Berkeley. Un acteur CAL définit des ports d'entrées et de sorties, des variables contenant un état interne à l'acteur et un certain nombre de règles de transitions appelés *actions*. Chaque acteur exécute une action à la fois, en fonction des conditions associées à chacune. Chaque exécution consiste à consommer les jetons, produire les jetons de sortie et modifier l'état interne de l'acteur. RVC-CAL ajoute certaines restrictions sur les données et les opérateurs d'une part et enlève certaines possibilités sur la description comportementale des acteurs d'autre part.

Une des motivations pour l'émergence de ce sous-ensemble vient de la difficulté d'implémenter un générateur de code efficace pour l'ensemble des constructions possibles permises par CAL. RVC-CAL est le nouveau standard MPEG Reconfigurable Video Coding (RVC) choisi par l'organisation ISO/IEC. Actuellement une série d'outils relatifs à RVC-CAL sont disponibles à la communauté sous le nom Open RVC-CAL Compiler (ORCC) [YLJ+13]. ORCC permet de générer du code ANSI C, du Java et du Low Level Virtual Machine (LLVM) pour des unités processeurs ou du VHDL/Verilog pour les architectures à base de FPGAs à partir des descriptions RVC-CAL et de l'outil Xronos.

- **Synflow** est un environnement de développement basé sur le langage Cx (extension du C) permettant la génération de code VHDL et exploitant les caractéristiques du modèle DPN [LP95]. L'utilisateur définit un ensemble de tâches. Chaque tâche est un processus séquentiel définissant deux fonctions particulières *setup* et *loop*. La fonction *setup* correspond à l'initialisation de la tâche et la fonction *loop* est une boucle infinie spécifiant le comportement de la tâche. Le compilateur associe à chaque tâche Cx une entité matérielle indépendante. Le code séquentiel est transformé en un ensemble de règles d'activations s'exécutant en un seul cycle d'horloge dans le processus matériel généré. Les canaux de communication sont représentés par des *ports* au niveau du langage Cx et sont transcrits matériellement par des FIFOs. Un type de port connecté en entrée et en sortie influence le comportement des règles d'activations (lectures/écritures bloquantes ou non).
- **CAPH** est un langage dédié à la description flot de données d'application opérant sur des flux de données et à leur implémentation sur cibles FPGA. Ce langage, qui a été spécifiquement utilisé dans cette thèse, est décrit dans la section suivante.

2.6 Le langage CAPH

CAPH est un DSL pour la description et l'implémentation d'applications flot de données sur circuits reconfigurables. CAPH est basé sur un modèle flot de données hérité des DPN. Une application est décrite comme un réseau d'acteurs indépendants échangeant des jetons à travers des canaux de communications de type FIFO. Le modèle d'exécution d'un programme CAPH est similaire à celui dit DDF (tel que défini dans [BDT13] par exemple) à ceci près que le système de *pattern-matching* des règles d'activations de CAPH est plus riche, supportant un type algébrique de données mais ne supportant pas le *multi-tokens* (activation d'une règle qui dépend d'une séquence).

2.6.1 Le flot de conception

La chaîne de développement CAPH est présenté en figure 2.19. L'outil est séparé en deux parties : le front-end, et les back-end. Le *front-end* intègre un simulateur basé sur un interpréteur déduit de la sémantique formelle du langage. Ce simulateur permet de faire une première vérification de l'algorithme que l'on souhaite implémenter. Il vérifie la conformité syntaxique du code et la cohérence des types entre acteurs. Le *front-end* est également doté d'un compilateur qui va générer une représentation intermédiaire de l'application. Cette représentation est utilisée pour la génération du code dans les deux *back-ends*.

Le *back-end SystemC* est à la fois utilisé pour accélérer les temps de simulation dans le cas de larges applications et pour le dimensionnement des canaux de communications. Comme le modèle de CAPH n'est pas statique et que les canaux de communications ne sont pas de taille infinie en réalité, le dimensionnement des FIFOs doit être effectué par un profilage du réseau d'acteurs en fonction de stimuli d'entrée.

Le *back-end VHDL* génère une transcription VHDL RTL du réseau d'acteurs pour la synthèse matérielle. Ce *back-end* utilise les informations générées par le profilage des FIFOs lors de la simulation SystemC afin de dimensionner chaque canal de communication au mieux.

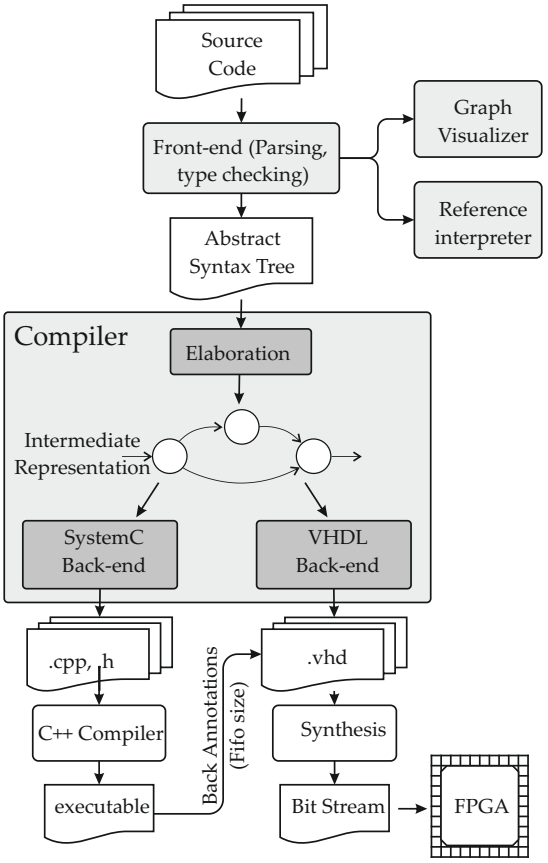


Fig. 2.19: Flot de conception CAPH

2.6.2 Description des acteurs

Le comportement de chaque acteur en CAPH est spécifié par un ensemble de règles d'activation. Chaque règle comporte un ensemble de *motifs (patterns)* (pat_i) impliquant les entrées et/ou les variables locales et un ensemble d'expressions (exp_i) décrivant les modifications sur les sorties et/ou les variables locales lors de l'activation de cette règle. Chaque règle est de la forme :

$$\mid \quad (qual_1 : pat_1, \cdots, qual_n : pat_n) \rightarrow (qual'_1 : exp_1, \cdots, qual'_n : exp_n)$$

Les jetons circulant sur les canaux de communication et manipulés par les acteurs peuvent être soit par exemple des jetons de données (valeur d'un pixel par exemple) ou de contrôle (un délimiteur). La distinction entre les jetons de données et de contrôle est assurée par le système de type du langage. Par exemple, le type `dc` est défini comme suit :

```
type t dc = SoS | EoS | Data of t
```

permet de coder des listes ou des images. Ici `SoS` (début de structure), `EoS` (fin de structure) et `Data` sont des *constructeurs* encodant respectivement les jetons de contrôle et les jetons de données. Le paramètre t dénote une variable de type [6].

Comme exemple, prenons l'acteur `suml` décrit sur le listing. 2.3 [7]. Cet acteur calcule la somme des valeurs d'une liste. Appliqué au flux `< 1 2 3 > < 4 5 6 >`, par exemple, il produit le résultat `6, 15`. Pour cela, il utilise deux variables locales : un accumulateur `s` et une variable d'état `st`. La variable `st` indique si l'acteur est en train de traiter une liste ou d'en attendre une nouvelle. La première règle peut se lire : lorsque l'on a attend une liste (`st=S0`)

6. Le système de typage de CAPH est similaire à ceux équipant les langages de programmation fonctionnelles modernes comme Haskell or ML. Il supporte le polymorphisme paramétré et les fonctions d'ordre supérieur.

7. Il est possible d'utiliser une notation abrégée pour le type `dc`, où `SoS` est équivalent à `'<`, `EoS` à `'>` et `Data of t` à `'t`.

et que l'on lit en entrée le jeton de début de liste (`i='<`), alors on initialise l'accumulateur (`s:=0`) et on commence à calculer (`st=S1`). La seconde règle dit : lorsque l'on calcule (`st=S1`), et que la valeur en entrée est une donnée (`a='v`), alors on met à jour l'accumulateur (`s:=s+v`). La dernière règle est activée lorsque le jeton d'entrée correspond à la fin de liste (`i='>`) alors le résultat de l'accumulateur est écrit sur la sortie `c`.

```
actor suml
   in (a: signed<8> dc)
  out (c: signed<16>)

var st: {So,S1}=So
var s : signed<16>

rules
  (st:So, a:'<) -> (st:S1, s:0)
| (st:S1, a:'v) -> (st:S1, s:s+v)
| (st:S1, a:'>) -> (st:So, c:s)
```

Listing 2.3: Acteur calculant la somme d'une liste en CAPH

Le schéma RTL obtenu après synthèse du code VHDL produit par le compilateur CAPH à partir de la description de l'acteur `suml` du listing 2.3 est illustré sur la Fig. 2.20. Les deux variables internes sont transcrites par des registres. La logique utilisée permet de vérifier la correspondance entre l'entrée et les types possibles (données ou délimiteurs), mais aussi de mettre à jour la variable d'états et de piloter la sortie.

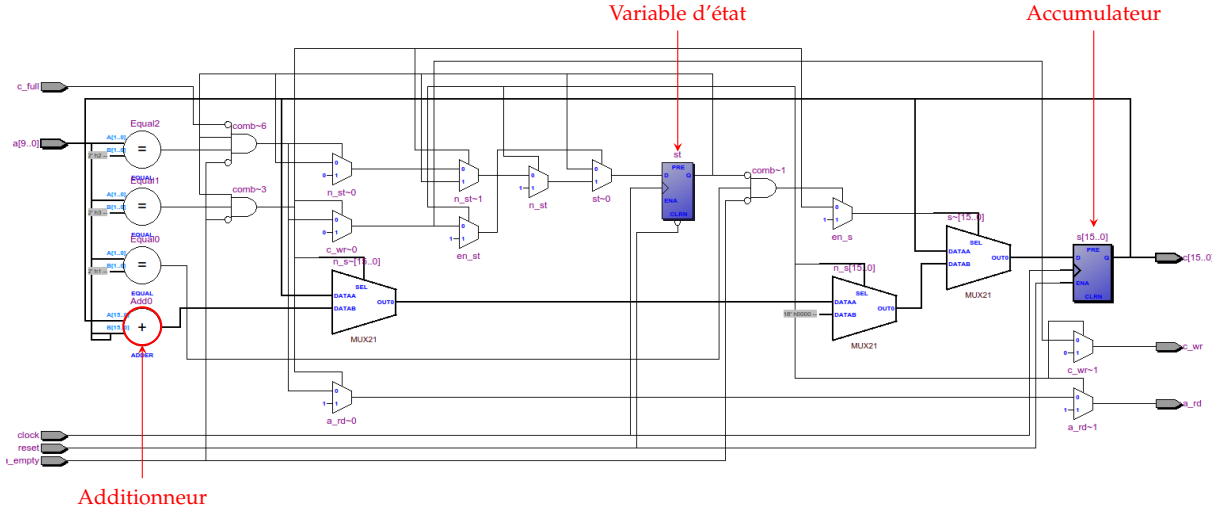


Fig. 2.20: Schéma RTL généré par le compilateur CAPH à partir de la description CAPH de l'acteur `suml`

Ce style de description correspond naturellement à une exécution flot de données pour le traitement d'images dans laquelle les pixels sont traités au fur et à mesure de leur arrivée dans le réseau d'acteurs. De plus, les signaux de contrôles et les données sont des parties intégrantes de la structure des jetons circulant entre les acteurs. Cette structuration permet de s'affranchir d'un système de contrôle global ou de synchronisation entre acteurs, permettant une production de code HDL efficace.

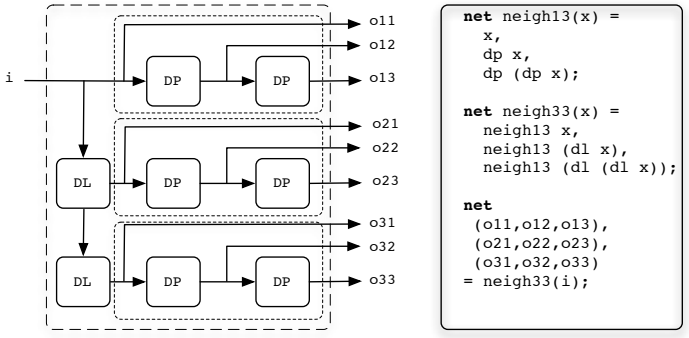
2.6.3 Description des réseaux

CAPH utilise un langage de description de réseau purement textuel pour la description d'un graphe flot de données. Ce langage, purement fonctionnel, décrit des graphes par l'ap-

plication de fonctions de connexions (*wiring functions*). Ce concept est introduit sur la Fig. 2.21 où un graphe d'acteurs (à gauche) est décrit par le programme CAPH (droite).
Ici, deux fonctions sont définies : `neigh13` et `neigh33`.

La première fonction prend en entrée une connexion et produit un ensemble de connexions, représentant le voisinage 1×3 du flux d'entrée obtenu par l'application de deux acteurs `dp` (retard pixel). La seconde fonction prend une connexion et retourne un ensemble de neuf connexions représentant le voisinage 3×3 du flux d'entrée, obtenu par l'application de la fonction `neigh13` et de deux acteurs `dl` (retard ligne).

Les sous graphes résultants de l'application des fonctions `neigh13` et `neigh33` sont respectivement délimitées avec des courts et longs traits sur la Fig. 2.21.




```
net neigh13(x) =
  x,
  dp x,
  dp (dp x);

net neigh33(x) =
  neigh13 x,
  neigh13 (dl x),
  neigh13 (dl (dl x));

net
 (o11,o12,o13),
 (o21,o22,o23),
 (o31,o32,o33)
= neigh33(i);
```

Fig. 2.21: Exemple de description d'un réseau d'acteurs en CAPH

2.7 Conclusion

En réponse aux problématiques de temps et de complexité de développement d'applications sur des architectures FPGA, la communauté scientifique ainsi que les industriels proposent depuis plusieurs années différents outils. Les objectifs sont toujours les mêmes : permettre un développement rapide des applications en utilisant un langage le plus abstrait possible, et rendre ainsi accessible cette technologie performante à des personnes non spécialistes.

Les DSL font partie des solutions actuelles permettant d'obtenir un bon compromis entre l'expressivité d'une application et les performances finales. L'approche proposée dans ces travaux se concentre sur l'utilisation du modèle flot de données afin d'exprimer des algorithmes complexes comme ceux utilisés pour la classification d'objets dans des images. La transcription des algorithmes reformulés suivant le modèle flot de données vers une implémentation matérielle se fera à partir de l'outil CAPH.

3

Systèmes d'apprentissage

L'apprentissage (ou Machine Learning (ML)) est apparu dans les années 1950 avec pour objectif de traiter des questions de l'intelligence artificielle. Différentes méthodes ont été exploitées, comme l'approche basée sur des réseaux de neurones (Perceptron), les méthodes statistiques (GLM) ou bien les méthodes probabilistes (SVM). Depuis les années 1990, l'apprentissage est devenu un champ de recherche à part entière ayant pour but de résoudre des problèmes spécifiques tels que la reconnaissance d'objets. Il existe de nombreux algorithmes d'apprentissage pouvant être classés suivant différents critères. La classification proposée ici se base sur la méthode d'apprentissage utilisée, à savoir :

- L'apprentissage *supervisé* : l'apprentissage est réalisé à partir de nombreux exemples annotés. Chaque exemple est un couple constitué d'un objet d'entrée, typiquement un vecteur décrivant l'objet et une classe associée. L'algorithme d'apprentissage analyse les données d'apprentissage pour construire un modèle permettant de séparer les classes entre elles. Ensuite, la fonction résultant de la phase d'apprentissage est appliquée à chaque nouvel exemple à classifier. Les méthodes les plus répandues sont les machines à vecteurs de support (SVM), le Boosting [FS+96] et les réseaux de neurones convolutionnels [LBBH98].
- L'apprentissage *non supervisé*, aussi appelé *clustering*, n'utilise aucune annotation sur les données d'entrée. Les algorithmes d'apprentissage cherchent à modéliser la distribution des données. Une fonction de décision est ensuite déduite de ce modèle. Les méthodes les plus connues sont les chaînes de Markov (HMM [BP66]), les réseaux bayésiens [Pea14] et les mixtures de gaussiennes (Gaussian Mixture Model (GMM) [Ras99]).
- L'apprentissage *semi-supervisé* qui mélange les deux premières catégories évoquées. Dans ce type d'algorithme, les données d'entrée sont un mélange d'exemples annotés et non annotés.
- L'apprentissage par *renforcement* fait référence à une classe de problèmes d'apprentissage automatique. Considérons l'exemple d'un agent autonome qui prend des décisions en fonction de son état courant et des données d'entrée qui sont injectées par son environnement. En fonction des retours de l'agent aux données d'entrée, l'environnement procure à l'agent des récompenses/punitions. L'agent cherche un comportement décisionnel optimal en maximisant la somme des récompenses au cours du temps. Ces méthodes sont principalement utilisées dans le contrôle de robots (algorithme Q-learning [WD92]).

Le tableau. 3.1 montre une répartition des algorithmes d'apprentissage les plus répandus suivant leur méthode d'apprentissage.

Dans cette thèse, nous allons nous concentrer sur deux méthodes d'apprentissage supervisées particulières, les Support Vector Machine (SVM) et les Convolutional Neural Networks (CNN). Il est à noter que les CNNs ont récemment montré de bonnes capacités à fonctionner

Non supervisé	Supervisé
Gaussian Mixture Model [Ras99]	Boosting [FS+96]
Auto-encoders [WH86]	Perceptron [Ros58]
Restricted Boltzmann Machines [SMH07]	Support Vector Machine [CV95]
Sparse Coding [MPS+09]	DecisionTree [FB97]
Deep Belief Network [HOT06]	Neural Net [Lip87]
BayesNP [Chi96]	RNN [MJ01]
Denoising Auto-encoders [VLL+10]	Convolutional Neural Networks [LBBH98]
Deep Boltzmann Machines [SH09]	Pi-Sigma Network [SG91]
Hidden Markov Models [BP66]	Random Forest [Bre01]

Tab. 3.1: Liste non exhaustive des algorithmes d'apprentissage

également avec des données non annotées [JKRL09, LGRN09]. Ces deux méthodes sont abordées selon une approche *discriminatoire*, c'est-à-dire que si l'on considère une entrée x dont on veut connaitre l'étiquette y, on essaie d'apprendre la probabilité conditionnelle $P(y|x)$. Cette approche est généralement plus performante que les approches dites *génératives* [1] lorsque la base d'apprentissage est conséquente [NJW+02].

3.1 Généralités sur l'apprentissage supervisé

L'apprentissage supervisé a souvent été utilisé avec succès dans le domaine de la détection d'objets. Parmi les exemples, on retrouve la détection de voitures [ZL09], de piétons [DT05] ou encore de visages [VJS03]. La totalité de ces systèmes reposent sur deux étapes fondamentales : une étape d'apprentissage hors ligne et une étape de classification en ligne.

3.1.1 L'apprentissage hors ligne

La phase d'apprentissage hors ligne, illustrée sur la figure 3.1, se déroule de la façon suivante :

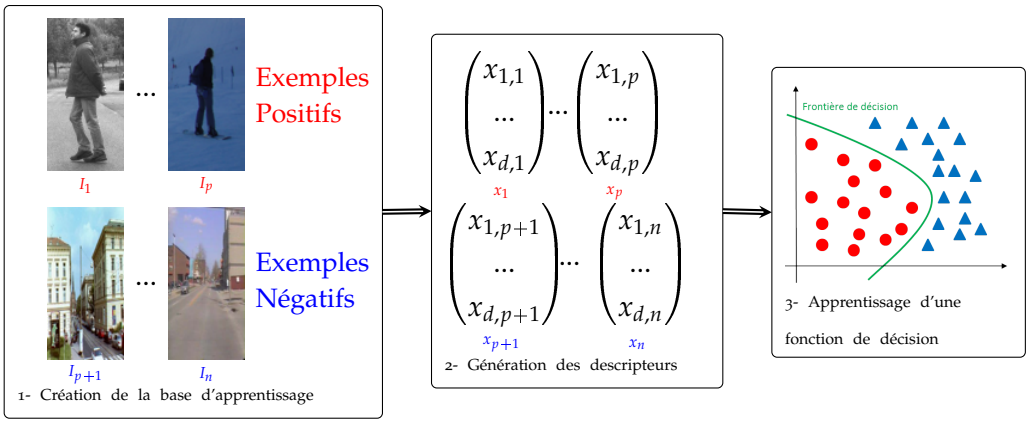


Fig. 3.1: Différentes étapes de l'apprentissage hors ligne.

1. **Création de la base d'apprentissage**.
   Cette base est constituée de deux sous ensembles. Le premier, regroupant les exemples positifs, est composé d'images de l'objet à détecter. Le second, regroupant les exemples négatifs est composé de tout ce qui n'est pas l'objet à détecter. La qualité des sous-ensembles a un impact direct sur l'efficacité de l'apprentissage. La base d'apprentissage

1. Approches qui apprennent une probabilité conjointe $P(x, y)$ au lieu d'une probabilité conditionnelle

se doit d'être la plus exhaustive possible. Dans le cas des exemples positifs, cette exhaustivité est approchée en collectant plusieurs milliers d'images de l'objet à détecter. Le sous-ensemble des exemples négatifs est plus délicat à composer. On peut utiliser une technique de *bootstrap* qui consiste à générer des exemples négatifs pertinents en réalisant plusieurs apprentissages. Il existe des méthodes dites de validation croisée (*cross validation*) afin de quantifier la qualité des sous-ensembles. On utilisera notamment la validation croisée à k-parts et la validation *Leave-One-Out* (LOO) [Sha93].

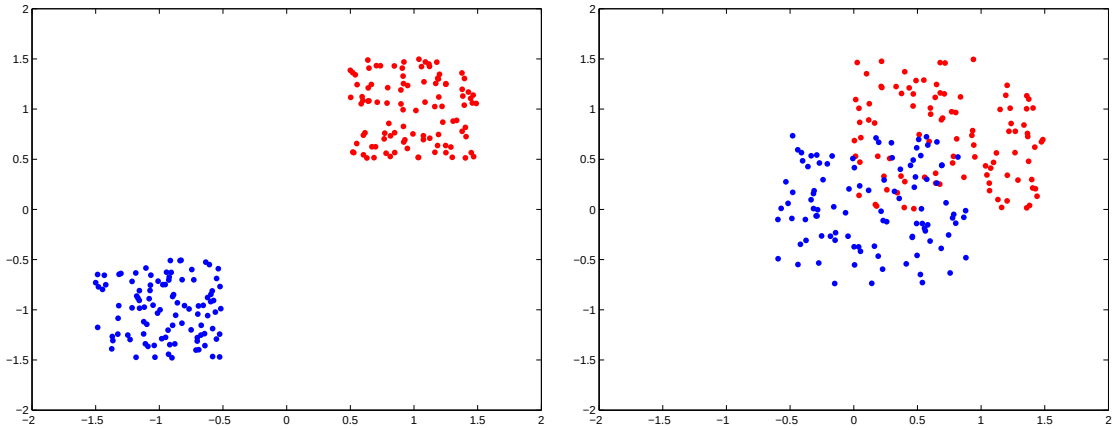
2. **Génération des descripteurs**.
   Il est en théorie possible d'utiliser l'image brute comme entrée du système d'apprentissage. Toutefois cette approche comporte plusieurs inconvénients. Tout d'abord, l'algorithme d'apprentissage risque d'avoir des difficultés à déterminer une fonction de décision à partir de ces données du fait de la grande variabilité entre les exemples d'une même classe, surtout pour les exemples négatifs. Ensuite, dans le cas où l'algorithme parvient quand même à apprendre une fonction de décision, celle-ci risque d'être lente lors de la phase de classification car la taille du descripteur va être égale au nombre de pixels de l'image. Plus la dimension du vecteur descripteur est importante, plus la phase de classification nécessitera de ressources de calculs.

   L'approche couramment utilisée consiste donc à extraire un descripteur à chaque image, puis à appliquer l'algorithme d'apprentissage sur l'ensemble des descripteurs de la base d'apprentissage. L'objectif de cette approche est de réduire la variabilité intra-classe tout en maximisant la variabilité inter-classe. Dans l'espace des descripteurs (Fig. 3.2), chaque sous ensemble forme un nuage de points. On parle de variabilité intra-classe pour quantifier la répartition spatiale des exemples d'une même classe dans l'espace de description. Si le nuage est compact, on parle d'une variabilité intra-classe faible alors qu'un nuage étendu est associé à une variabilité intra-classe forte. De même, La variabilité inter-classe quantifie la répartition spatiale des deux nuages de points. Si les deux nuages sont disjoints, la variabilité inter-classe est forte. À l'opposé, plus les nuages se recouvrent et plus la variabilité inter-classe diminue. Le choix du type de descripteurs et le type d'objets à détecter auront un impact important sur la séparabilité des nuages de points. Parmi les descripteurs les plus utilisés, on trouve les ondelettes de Haar [POP98], les Local Binary Pattern (LBP) [AHP06] ou encore les histogrammes de gradients orientés [DT05] (HOG) . Notons par ailleurs que l'utilisation de descripteurs permet également de garantir des invariances à certaines caractéristiques des images comme l'illumination par exemple. Enfin, il est souhaitable que le descripteur soit de dimension réduite afin de réduire le coût de calcul de la fonction de décision.

3. **Apprentissage d'une fonction de décision**.
   Un algorithme d'apprentissage est appliqué sur l'ensemble des descripteurs. Chaque élément de cet ensemble est un couple constitué d'un vecteur de descripteurs et d'une étiquette qui représente la classe de l'élément associé. Généralement, dans le cas de classification binaire, l'étiquette 1 va représenter la classe de l'objet et l'étiquette -1 représente la classe non-objet.

   La description des exemples d'apprentissage permet d'obtenir deux nuages de points dans l'espace des descripteurs. Chacun de ces points est associé à un label (1 ou -1). Le but d'un algorithme d'apprentissage, reposant sur une approche discriminatoire, est alors de construire une fonction de régression permettant de relier les étiquettes aux vecteurs de descripteurs. Si on note $\{(x_i, y_i)\}_{i=1,\dots,N}$ un ensemble d'apprentissage à N éléments, où x_i est un des vecteurs descripteur et y_i l'étiquette associée, alors le but de l'algorithme d'apprentissage est de construire une fonction f telle que $y_i = f(x_i) \quad \forall i \in \{1, .., n\}$. En pratique, il est généralement impossible d'avoir $y_i = f(x_i) \quad \forall i$. On cherche donc la fonction f qui vérifie $y_i = f(x_i)$ pour le plus grand nombre d'exemples d'apprentissage. Le nombre d'exemples ne vérifiant pas $y_i = f(x_i)$ permet alors de définir l'erreur d'apprentissage.

   Une fois la fonction de décision définie, on s'intéresse à sa capacité de généralisation,



(a) variabilité intra-classe est faible et inter-classe forte

(b) variabilité intra-classe est forte et inter-classe faible

Fig. 3.2: (a) La variabilité intra-classe est faible alors que la variabilité inter-classe est forte, les deux nuages de points sont simple à séparer. (b) La variabilité intra-classe est forte alors que la variabilité inter-classe est faible, le problème n'est pas complètement séparable dans l'espace de descripteurs choisi.

c'est-à-dire sa capacité à prédire des étiquettes correctes sur des exemples inconnus (non utilisés pendant l'apprentissage). Pour cela, on utilise un ensemble de tests contenant des exemples non utilisés lors de la phase d'apprentissage mais dont on connait la classe. On applique ensuite f sur ces exemples et on compare les étiquettes prédites avec les étiquettes attendues. On peut ainsi calculer une erreur de classification. Si cette erreur est élevée, cela reflète un des deux phénomènes propres à l'apprentissage artificiel : le sous-apprentissage et le sur-apprentissage. Deux causes peuvent expliquer le sous-apprentissage. D'une part, la base d'apprentissage n'est pas assez exhaustive et ne reflète pas assez la réalité. D'autre part, les classes positives et négatives sont trop difficiles à discerner. Ce dernier point se traduit par des nuages de points se chevauchant dans l'espace des descripteurs. Le sur-apprentissage signifie que la fonction apprise f est trop spécifique aux données d'apprentissage. Le sur-apprentissage peut être évité en limitant la complexité de la fonction de décision. Par exemple, il peut être préférable de chercher f sous la forme d'une fonction linéaire plutôt que sous la forme d'une fonction non-linéaire.

3.1.2 La classification en ligne

Le processus de classification en ligne est illustré sur la figure 3.3. Lorsqu'on souhaite classifier une image I inconnue, on commence tout d'abord par décrire cette image. On obtient donc un vecteur de descripteurs x sur lequel on applique la fonction f pour obtenir son étiquette $y = f(x)$. Graphiquement, l'étiquette prédite y dépend de la position du point x dans l'espace des descripteurs. Si ce point se trouve dans la zone de classe positive ou sur la frontière de décision, alors l'étiquette sera 1. À l'opposé, si le point se trouve dans la zone de la classe négative, alors l'étiquette sera -1.

Dans le cas de la classification illustré sur la Fig. 3.3, le calcul du descripteur ainsi que l'application de la fonction de classification f sont effectuées sur une "imagette" d'entrée. Les systèmes de détection qui nous intéressent dans ces travaux ont eux pour objectif de détecter l'ensemble des occurrences de l'objet recherché sur une image de grande dimension (et non de classifier des "imagettes" isolées). Pour cela, le calcul des descripteurs se fait sur l'ensemble de l'image et la fonction de classification est appliquée sur un ensemble de **fenêtres de détection**. Chaque fenêtre, de dimension égale à l'objet recherché, est déplacée dans l'image à évaluer afin de déterminer tous les occurrences de l'objet recherché. Cette technique de fenêtrage glissant sera détaillée dans la suite.

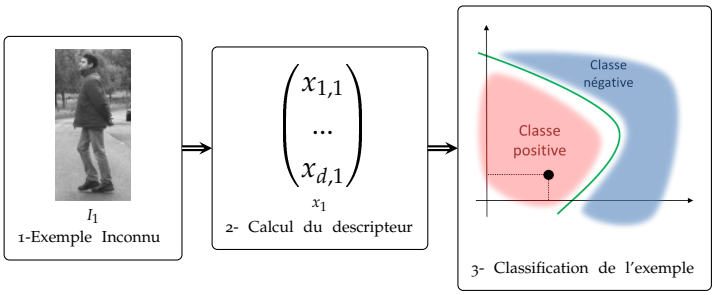


Fig. 3.3: Les deux étapes de la classification en ligne. Pour déterminer l'étiquette d'une image, on commence par calculer son vecteur de descripteurs. Puis, on applique la fonction de décision sur ce vecteur

Comme introduit à la Sec. 3.1.1, le choix du type de descripteur aura un impact sur la qualité du système d'apprentissage. La prochaine section présente les principaux descripteurs existants pour la détection de piétons.

3.2 Les descripteurs existants pour la détection

Comme introduit à la Sec. 3.1, un descripteur d'images pour la classification doit être le plus discriminant entre les deux classes possibles dans l'espace de description. Ce descripteur doit aussi permettre au système de classification d'être le plus robuste possible vis à vis des distorsions géométriques comme les mises à l'échelle, les rotations, les changements de point de vue, les variations d'illumination ou bien le bruit. Le coût de calcul associé à l'élaboration du descripteur est également un facteur important car le calcul du descripteur est effectué à la fois par l'étape d'apprentissage hors-ligne mais surtout par la phase de classification en ligne qui est soumise à des contraintes de performances fortes. Le choix du descripteur aura donc un rôle clé dans les performances du système de classification. De nombreux descripteurs d'images et comparatifs entre descripteurs existent dans la littérature, sur des domaines d'application différents comme la détection, la localisation, la navigation ou l'appariement d'objets entre images. Dans la suite de cette section, une liste non exhaustive de descripteurs pouvant être utilisés pour la détection de personnes est faite. Les descripteurs potentiels pour la détection de piétons ont été classifiés en trois catégories : ceux fondés sur le gradient, sur la texture et la fusion de descripteurs.

3.2.1 Les approches basées gradient

.

- **Scale Invariant Feature Transform (SIFT)**. Le descripteur SIFT proposé par Lowe [Low04] est certainement un des descripteurs les plus utilisés en pratique de par sa robustesse et son invariance aux translations, mises à l'échelle et rotations. Souvent couplé avec un algorithme d'appariement (ou *features matching*), il permet de faire de la reconnaissance d'objets entre deux images ou de la localisation de robots. Le descripteur est basé sur des histogrammes locaux d'orientation de gradients autour de points d'intérêt, ce qui nécessite quatre étapes :

  1. Détection de maximum et minimum locaux à travers un ensemble d'échelles. Cet ensemble d'échelles, connu sous le nom d'*espace d'échelles* (*scale-space*), est le résultat de la convolution de l'image d'entrée $I(x,y)$ avec un ensemble de gaussiennes $G(x,y,\sigma)$. Il est défini par la fonction $L(x,y,\sigma)$ tel que :

  $$L(x,y,\sigma) = G(x,y,\sigma) * I(x,y) \tag{3.1}$$

  avec $*$ l'opérateur de convolution et $G(x,y,\sigma)$ une gaussienne d'écart type σ. Il résulte de cette opération un ensemble d'images dans l'espace des échelles. Cet

espace est subdivisée en s octaves (où chaque octave est séparée d'un facteur σ multiple de 2), comme illustré sur la Fig. 3.4. Afin de détecter les maxima et minima locaux dans l'espace d'échelles, Lowe propose l'utilisation d'une différence de gaussienne, notée $D(x, y, \sigma)$, calculée par :

$$D(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma) \tag{3.2}$$

où k est une constante séparant deux images dans l'espace d'échelles (typiquement $2^{1/n}$ où n est le nombre d'images dans une octave). Chaque point est comparé avec son voisinage direct dans l'image et ses neuf voisins dans l'échelle supérieure et inférieure. Le point est sélectionné à la seule condition d'être le maximum ou le minimum de tous ses voisins.

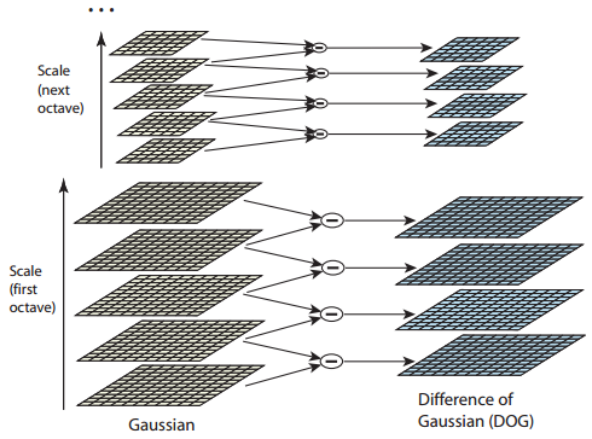


Fig. 3.4: Pour chaque octave de l'espace des échelles, l'image d'entrée est convoluée avec différentes gaussiennes afin de produire un ensemble d'images (colonne de gauche). Ensuite les images gaussiennes adjacentes sont soustraites pour produire les différences de gaussiennes (colonne de droite). Pour chaque octave, l'image est sous échantillonnée d'un facteur 2 et le processus est répété.

2. Localisation des points d'intérêts. Un fois qu'un point d'intérêt a été détecté grâce à la première étape, l'étape suivante consiste à le localiser précisément dans l'image. Cette localisation se fait par une méthode d'interpolation qui fait correspondre une fonction quadratique $3D$ au dit point d'intérêt. Cette fonction est issue du développement de Taylor au second ordre de la fonction d'échelle $D(x, y, \sigma)$, exprimé à la position du point d'intérêt tel que :

$$D(\mathbf{x}) = D + \frac{\partial D^T}{\partial \mathbf{x}}\mathbf{x} + \frac{1}{2}\mathbf{x}^T\frac{\partial^2 D}{\partial \mathbf{x}^2}\mathbf{x} \tag{3.3}$$

où D et ses dérivées sont évaluées au point d'intérêt et $\mathbf{x} = (x\ y\ \sigma)$ la distance du point d'intérêt à l'origine. L'estimée, $\hat{\mathbf{x}}$, se déduit de la dérivée à l'origine de la fonction D (Eq. 3.3), donnée par :

$$\hat{\mathbf{x}} = \left(-\frac{\partial^2 D}{\partial \mathbf{x}^2}\right)^{-1}\frac{\partial D}{\partial \mathbf{x}} \tag{3.4}$$

Une étape de filtrage est appliquée à l'ensemble des points d'intérêts car la différence de gaussiennes peut donner une réponse forte le long des contours mais avec une localisation instable face au bruit. Les courbures sont calculées par la matrice Hessienne 2×2, $\mathbf{H}$, calculée à la position et à l'échelle du point d'intérêt :

$$\mathbf{H} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix} \tag{3.5}$$

où les dérivées sont estimées par la différence des points d'intérêts voisins. Les valeurs propres de la matrice **H** sont directement liées aux courbures de D. Si le déterminant de la matrice est négatif, cela signifie que les courbures ont des signes différents autour du point, et le point sera rejeté.

3. L'assignation des orientations. En affectant une orientation relative à chaque point d'intérêt basé sur les propriétés locales, le descripteur qui sera lié à ce point sera décrit de manière relative à sa propre orientation, le rendant invariant aux rotations de l'image. L'échelle à laquelle le point d'intérêt appartient est utilisée pour sélectionner l'image d'entrée lissée L avec l'échelle correspondante, afin de rendre le calcul invariant au changement d'échelle. Pour chaque image $L(x, y)$ à cette échelle, l'amplitude du gradient $m(x, y)$ et son orientation $\theta(x, y)$ sont calculés par :

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2} \tag{3.6}$$

$$\theta(x,y) = \tan^{-1}((L(x,y+1) - L(x,y-1))/(L(x+1,y) - L(x-1,y))) \tag{3.7}$$

Des histogrammes d'orientation des gradients sont calculés pour chaque zone au voisinage de chaque point d'intérêt. Un histogramme est composé de 36 classes couvrant les 360 degrés de l'espace des orientations possibles. Chaque échantillon ajouté à l'histogramme est pondéré par l'amplitude du gradient et une fenêtre gaussienne circulaire (d'écart type $1.5 \times \sigma$ où *sigma* est l'écart type de gaussienne associée à l'échelle du point d'intérêt). La valeur maximale de l'histogramme détectée ainsi que chaque orientation supérieure à 80% de cette valeur maximale sont utilisées pour créer un point d'intérêt associée à la dite orientation. Si les valeurs sont égales, un ensemble de points est créé à la même place et échelle mais associée à une orientation différente. Selon Lowe, environ 15% des points sont assignés à des orientations multiples, contribuant à la stabilité des phases d'appariement. Une dernière phase d'interpolation (basée sur une parabole) sert à améliorer la précision de l'orientation maximale.

4. Description des points d'intérêt. La figure 3.5 illustre la création du descripteur à partir des orientations de gradients. La zone autour du point d'intérêt est divisée en 4×4 cellules (2×2 cellules Fig. 3.5). Le descripteur SIFT pour un point d'intérêt est l'agrégation de tous les histogrammes d'orientations précédemment calculés. Finalement, une dernière étape de normalisation est effectuée afin de rendre le descripteur moins sensible aux changements d'illumination.

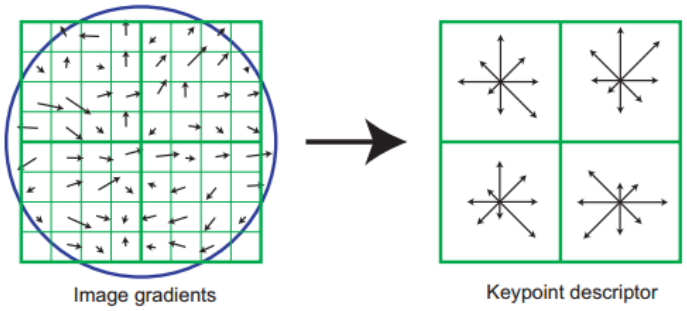


Fig. 3.5: Création du descripteur SIFT pour un point d'intérêt à partir des gradients dans le voisinage du point d'intérêt. Une fenêtre gaussienne circulaire, représentée par le cercle, est appliquée. Enfin, les histogrammes d'orientation des gradients calculés pour chaque sous région sont aggrégés afin de former le descripteur final.(Extrait et adapté de [Low04])

Différentes améliorations ont été proposées depuis la formulation originale de Lowe. On peut citer par exemple le PCA-SIFT [KS04], qui utilise l'analyse en composantes principales (PCA) afin de réduire la dimension du descripteur et d'accélérer les phases d'appariements. Cependant, il a été prouvé par Mikolajczyk dans [MS05] que le PCA-SIFT réduisait la qualité du descripteur. Mikolajczyk a donc proposé une autre variante,

le Gradient location and Orientation Histogram (GLOH), qui a la même dimension que le PCA-SIFT tout en gardant le même niveau de discrimination qu'un SIFT normal. Malheureusement, le GLOH a un coût algorithmique plus important au final. Même si le descripteur SIFT est performant et certainement un des descripteurs les plus utilisés, son élaboration engendre des coûts de calculs assez importants et la haute dimensionnalité du descripteur reste un inconvénient notamment pour les phases d'appariement.

- **Speed Up Robust Features (SURF).** Le descripteur SURF introduit par Bay dans [BETVGo8] propose une alternative au SIFT, en réduisant les coûts de calcul lié à l'élaboration du descripteur mais en gardant les invariances aux rotations et mises à l'échelle. De la même façon que le SIFT, un descripteur SURF est associé à un point d'intérêt de l'image. La détection des points d'intérêt est basée sur les matrices hessiennes et les images intégrales [Poro5]. Pour chaque pixel $\mathbf{x} = (x, y)$ de l'image I, la matrice hessienne en $\mathbf{x}$ à l'échelle σ est calculée par :

$$\mathbf{H} = \begin{bmatrix} L_{xx} & L_{xy} \\ L_{xy} & L_{yy} \end{bmatrix} \tag{3.8}$$

où

$$L_{xx} = \frac{\partial^2 G(\sigma)}{\partial x^2} * I, \qquad L_{xy} = \frac{\partial^2 G(\sigma)}{\partial x \partial y} * I, \qquad L_{yy} = \frac{\partial^2 G(\sigma)}{\partial y^2} * I$$

$G(\sigma)$ est la fonction gaussienne d'écart type σ et $*$ l'opérateur de convolution Le calcul de la matrice $\mathbf{H}$ est approximé en pratique par un ensemble de convolutions discrètes, comme illustré par la figure 3.6. La détermination de l'espace des échelles est

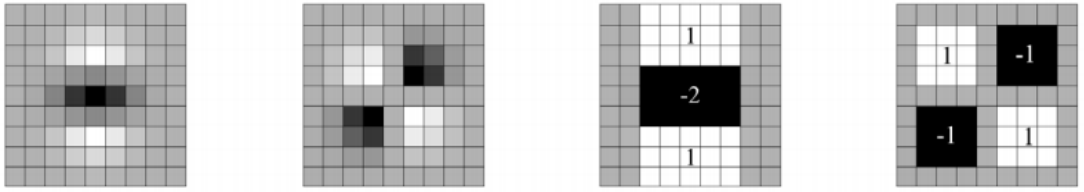


Fig. 3.6: De gauche à droite. Respectivement les dérivées d'ordre d'ordre deux en y et xy discrétisées et les deux approximations faites dans le calcul du SURF (les région grises sont égales à zéro). Extrait de [BETVGo8]

légèrement différente par rapport à la classique pyramide gaussienne utilisée dans le SIFT. Au lieu d'appliquer itérativement un filtrage gaussien d'écart type différent et un sous-échantillonnage pour chaque échelle, Bay propose d'augmenter la dimension des filtres utilisés dans l'approximation de la matrice hessienne (Fig. 3.6), ainsi que le pas de sous-échantillonnage. L'avantage majeur de cette technique est que la détermination de toutes les échelles peut se faire de manière parallèle alors qu'une pyramide gaussienne se construit de manière itérative. Afin de déterminer les points d'intérêts, le score de chaque pixel correspond au déterminant de la matrice hessienne approximée ($\mathbf{H}_{approx}$) calculée en ce point par la formule :

$$det(\mathbf{H}_{approx}) = D_{xx}D_{yy} - (0.9D_{xy})^2 \tag{3.9}$$

Afin d'être invariant aux rotations, une transformée des ondelettes de Haar suivant x et y est calculée pour chaque point d'intérêt dans un voisinage circulaire égal à six fois le facteur d'échelle s du point. Les ondelettes sont pondérées avec une fonction gaussienne d'écart type 2.5 fois le facteur d'échelle, centré sur le point d'intérêt. L'orientation dominante est estimée en fonction des réponses des ondelettes dans différentes directions (décalées de $\pi/3$).

La création du descripteur se fait par une première phase qui détermine une région rectangulaire centrée sur le point d'intérêt et orientée suivant l'orientation dominante précédemment calculée. Cette région est ensuite séparé en 4×4 sous-régions sur lesquelles

sont encore calculées les transformations des ondelettes de Haar en x et y, respectivement notées d_x et d_y (orientation relative au point d'intérêt). Pour chaque sous-région, les réponses des ondelettes sont accumulées et vont former une partie du descripteur final. Ensuite, se rajoute la somme des valeurs absolues des réponses, ce qui donne un vecteur, noté $\mathbf{v}$, formé de quatre composantes $\mathbf{v} = (\sum dx, \sum dy, \sum |dx|, \sum |dy|)$, pour chaque sous région. Le descripteur final est formé de ce vecteur $\mathbf{v}$ pour chaque région. Une dernière étape de normalisation est effectuée afin de rendre le descripteur invariant aux changements de contraste. Conjointement avec la proposition de Bay dans [BETVG08], une version dégradée appelée USURF, non invariante aux rotations d'images mais encore plus rapide à calculer, a également été proposée.

- **Histogram of Oriented Gradients**. Inspiré du SIFT, les descripteurs HOG ont été introduits par Dalal et Triggs [DT05]. Le descripteur HOG n'utilise pas de point d'intérêt mais calcule la distribution des orientations des gradients sur une zone fixe. L'idée est de caractériser l'apparence et la forme d'un objet par la distribution des intensités et des orientations locales des gradients, mais sans localisation précise de ces gradients dans l'espace (différence fondamentale avec le SIFT et le SURF). En pratique, la zone à caractériser est divisée en régions rectangulaires appelées *cellules*. Pour chaque cellule, les histogrammes locaux des orientations de gradients issues de chaque pixel appartenant à la cellule sont accumulés. Afin de rendre le descripteur robuste aux variations d'illuminations et aux ombres, les cellules sont regroupées en entités de plus grandes dimensions, appelées *blocs*. Le descripteur associé à un bloc est l'agrégation des histogrammes de chaque cellule composant ce bloc. Le descripteur HOG final, correspondant à la zone de détection où l'objet est cherché, est l'agrégation de tous les descripteurs de chaque bloc composant cette zone. Ce descripteur a été proposé initialement pour la détection de piétons même s'il a été utilisé pour la détection d'objets par la suite [MGE+11]. L'ensemble des opérations est présenté sur la figure 3.7.

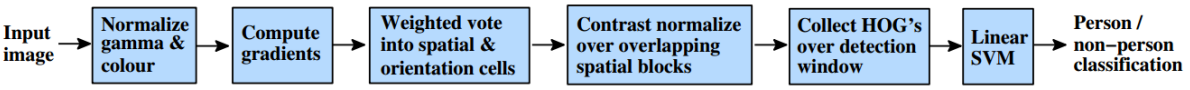


Fig. 3.7: Présentation de l'extraction des descripteurs HOG (Extrait de [DT05])

L'élaboration du descripteur HOG sera détaillée plus précisément dans la section 3.4.

- **Le descripteur Pseudo-Haar**. Ce descripteur a été largement popularisé par Viola et Jones dans [VJ01] pour la détection de visages. Il est inspiré des ondelettes de Haar, calculant des différences de sommes de pixels dans des zones rectangulaires. Ce descripteur a l'avantage d'être extrêmement rapide à calculer à l'aide des images intégrales. La Fig. 3.8 illustre des exemples de schémas utilisés par Viola et Jones. Dans chaque zone rectangulaire illustrée, la somme des pixels de la zone sombre est soustraite à celles de la zone claire. Ces schémas ont été enrichis par la suite dans [LM02].

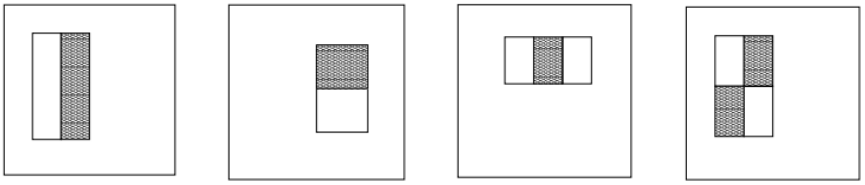
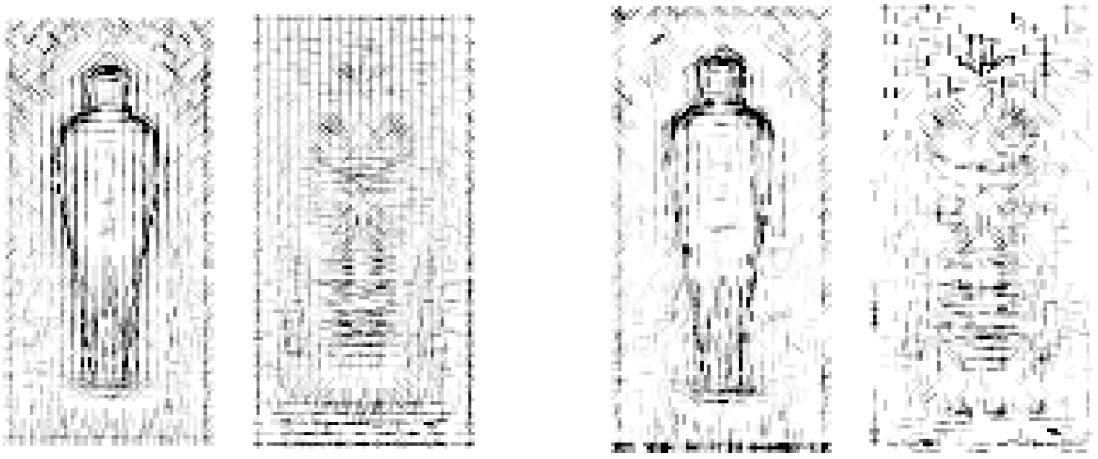


Fig. 3.8: Exemple de caractéristiques pseudo-Haar

- **Shapelet/Edgelet**. Ces deux approches similaires, respectivement proposées par Sabzmeydani [SM07] et [WN05], créent un descripteur à partir d'un algorithme d'apprentissage supervisé (AdaBoost [VJ01]). Dans le cas du *Shapelet*, la première étape consiste à

extraire des descripteurs de bas-niveau, qui sont les valeurs absolues des réponses des gradients sur quatre orientations distinctes. De la même manière que pour le HOG, ce premier calcul s'effectue sur une fenêtre de détection. Les amplitudes des réponses sont moyennées avec un filtre 5x5 pour réduire l'influence des déplacements de la fenêtre de détection. Les informations extraites servent à construire un descripteur de plus haut niveau, appelé *Shapelet*. Un *Shapelet* est une combinaison pondérée des descripteurs bas-niveau définis sur une sous-région w_i de la fenêtre de détection initiale. Pour chaque itération de l'algorithme *AdaBoost* utilisé, un *Shapelet* sélectionné aléatoirement est utilisé comme un classifieur faible qui sera ajouté au classifieur final pour la sous région w_i. Une illustration de *Shapelet* sur une fenêtre de détection entière (agrégation de sous-régions w_i) est donnée sur la figure 3.9a sur deux classes (piétons et non piétons). Une fois que les *shapelets* ont été obtenus, une seconde phase d'apprentissage AdaBoost est effectuée. L'espace de description est cette fois constitué de la combinaison des *shapelets* pour la fenêtre de détection complète. Un exemple de descripteurs finaux est montré sur la figure 3.9b.



(a) *(b)*

Fig. 3.9: Illustration des Shapelets. La figure de gauche (a) représente les shapelets selectionnés par chaque sous-région sur deux images entières (piéton et non piéton). La figure de droite (b) représente les shapelets utilisés dans le descripteur final après la seconde phase d'apprentissage pour les mêmes images. (Extrait de [SM07])

Le paramètre le plus important pour les *Shapelets* concerne la dimension de la sous-région w_i. Cette dimension impacte directement la performance du descripteur et les auteurs conseillent la fusion de *Shapelet* de trois tailles différentes, ce qui va nécessiter trois apprentissages au lieu d'un pour la première phase de cette approche, complexifiant de manière significative la mise en œuvre. Cependant le descripteur final serait selon les auteurs de meilleure qualité que celui fourni par la version courante du HOG.

- **Adaptive Contour Features** (ACF), proposé par Gao [GAL09]. Ce descripteur est basé sur un espace de représentation particulier, connu sous le nom de *Granular Space* [HALL06]. Cet espace contient un ensemble de *granules*, où chaque *granule* est en fait une fenêtre carrée, définie par le triplet $G = (x, y, s)$ où (x, y) est la position et s l'échelle (taille de la fenêtre). A partir de cette définition, Gao a rajouté la notion d'orientation. Les descripteurs ACF caractérisent les contours de chaque élément de l'image par un calcul d'amplitude et d'orientation de gradient (similaire au HOG). Les valeurs d'amplitude et d'orientation de gradient sont accumulées pour chaque granule. Les granules sont ensuite agrégées suivant différentes techniques possibles en fonction d'un algorithme d'apprentissage spécifique pour construire les descripteurs ACF.
- **Covariances**. Il existe plusieurs approches basées sur les co-variances de descripteurs. La principale formulation vient de Tuzel *et al* [TPM08] qui ont introduit la covariance

de descripteurs, couplée avec un système de Boosting (LogitBoost). Soit une image I et F l'espace de description de dimension $W \times H \times d$ extrait de l'image I. On définit une fonction Φ, appelé *fonction de mapping*, qui peut dépendre de plusieurs paramètres (intensités, couleur, gradients,etc...) telle que :

$$F(x,y) = \Phi(I,x,y) \tag{3.10}$$

Soit la région rectangulaire $R \subset F$ et $\{\mathbf{z}_i\}_{i=1..S}$ les S descripteurs de dimension d appartenant à R, la matrice de covariance des descripteurs, de dimension $d \times d$ est calculée par :

$$\mathbf{C}_R = \frac{1}{S-1}\sum_{i=1}^{S}(\mathbf{z}_i - \mu)(\mathbf{z}_i - \mu)^T \tag{3.11}$$

où μ correspond à la moyenne des points. Les éléments diagonaux de la matrice de covariance représentent la variance de chaque descripteur et les éléments non-diagonaux les corrélations entre descripteurs. Dans le cas de la détection de piétons, Tuzel *et al* ont proposé pour fonction Φ, la fonction suivante :

$$\Phi = \left[\begin{array}{cccccccc} x & y & |I_x| & |I_y| & \sqrt{I_x^2 + I_y^2} & |I_{xx}| & |I_{yy}| & \arctan\dfrac{|I_x|}{|I_y|} \end{array} \right]^T \tag{3.12}$$

où x et y sont les positions des pixels, I_x, I_{xx} les dérivées premières et secondes des intensités et le dernier terme l'orientation du contour. Avec cette fonction Φ, la dimension du descripteur est de $d = 8$ pour chaque point (x,y). La matrice de covariance pour une région sera de dimension 8×8, et grâce la symétrie de cette matrice, le descripteur final est réduit à 36 dimensions.

3.2.2 Les approches basées texture

.

- **Les ondelettes de Gabor** [Gab46] sont un descripteur de texture courant. Les filtres de Gabor peuvent être paramétrés en fréquence et orientation, ce qui permet d'obtenir des informations à des résolutions spatiales et fréquentielles différentes, important pour la classification et la segmentation de texture. Ce descripteur été utilisé par Lin [LLZ09] pour la reconnaissance d'objet.
- **Local Binary Pattern (LBP)**. Introduits dans [OPH96], les LBP sont invariants aux variations des niveaux de gris dans les textures et peuvent être utilisés pour discriminer un large spectre de textures efficacement. Les LBP ont également l'avantage d'être simple à calculer, opèrent au niveau du pixel sur des niveaux de gris, en décrivant le voisinage direct (3×3) avec une représentation binaire. Le descripteur est défini par :

$$LBP = \sum_{p=1}^{8} s(g_p - g_0)2^p \tag{3.13}$$

où g_0 correspond au niveau de gris du pixel associé au descripteur et les g_p correspondent aux pixels voisins. La fonction $s(x)$ retourne 1 si la valeur x est positive et 0 sinon. La formulation initiale du descripteur a plusieurs inconvénients : la taille du voisinage est limité à un voisinage 3×3, la distance des pixels au centre n'est pas constante et la dimension du descripteur est très importante (3 fois celle du HOG). Une évolution proposée dans [OPM02], introduit les Circular LBP avec un voisinage circulaire symétrique dont le rayon et le nombre de voisins sont paramétrables. Cette méthode rend le descripteur invariant aux rotations mais nécessite une étape d'interpolation pour la détermination des valeurs des voisins dont les coordonnées ne correspondent pas à un pixel existant.

De nombreuses variantes du LBP existent à ce jour telles que le Multi-Block LBP (MB-LBP) [ZCX+07], le Local Ternary Patterns (LTP) [TT10] ou plus récemment le Co-Occurrence of Adjacent Local Binary (CoA-LBP) [NOF12]. Un état de l'art exhaustif des descripteurs LBP existants peut être trouvé dans [Dos14]. Appliquée à la classification, chaque variante offre un espace de description plus ou moins adapté à une classe d'objet. On retrouve des applications de détection de visage à partir de C-LBP [AHP06], ou plus récemment à l'aide de Robust Local Binary Pattern (RLBP) [DPR+15]. Le CS-LBP [ZSHH10] a été utilisé pour la détection de piétons.

- **Weber Local Descriptor** (WLD). Introduit par Chen [CSH+10], ce descripteur est inspiré de la loi de Weber (domaine psychologique), qui stipule que le changement d'un stimuli (visuel, auditif, ...) attirant l'attention d'un être humain est exprimable avec un ratio constant par rapport à l'intensité du stimuli original. Motivé par cette loi, Chen a défini un descripteur fonction de deux composantes : *une différentielle excitatrice*, calculée à partir de l'intensité des pixels du voisinage du pixel courant et une information d'orientation, calculée par l'histogramme des orientations dominantes du gradient au pixel courant.

3.2.3 La fusion de descripteurs (Multi-Features (MF))

.

Les approches modernes en vision utilisent la complémentarité que peut apporter différents descripteurs afin d'améliorer les performances globales des systèmes de détection. L'idée est que plus le descripteur combine d'informations, plus il pourra être performant. En contrepartie, plus le descripteur combine d'informations, plus son implémentation va requérir de ressources et de temps de calcul. Dans cette catégorie, il existe :

- **Descripteur Histogram of Oriented Gradients (HOG)/Histogram of Optical Flow (HOF)** introduit par Laptev dans [LMSR08] utilise une combinaison de descripteurs HOG et HOF sur des points d'intérêts extraits du domaine spatio-temporel comme présenté dans [Lap05]. La classification se fait à l'aide d'une SVM non linéaire (noyau Gaussien multi-canal).
- **HOG-LBP** proposé par Wang [WHY09] qui permet la gestion des occlusions partielles et qui rend le descripteur HOG plus robuste aux changements de textures. La fusion des deux descripteurs se fait simplement par la simple agrégation des deux descripteurs avant de faire l'apprentissage, comme illustré sur la Fig. 3.10. Des travaux simi-

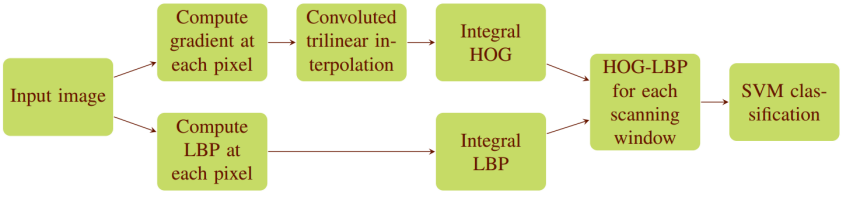


Fig. 3.10: HOG-LBP (Méthode de fusions des descripteurs HOG-LBP (Extrait de [WHY09])

laires ont étendus le descripteur HOG avec différentes variantes du LBP. Par exemple, Zhang [ZHYT11] utilise le Local Structured LBP et remplace la SVM par des modèles déformables (DPBM [FGMR10]).

- Yao dans [YPW+15] a évalué différentes combinaisons de descripteurs et leur impact sur la classification. Les descripteurs utilisés sont : SIFT, SURF, HOG, Haar, LBP et Local Selft-Similarities (LSS). Chaque combinaison testée est l'agrégation de deux descripteurs parmi la liste disponible, qui est ensuite donnée à un système de classification par SVM. Le descripteur Local Selft-Similarities (LSS) [SI07] n'a pas été introduit précédemment car il est originalement dédié à l'appariement d'objets dans des séquences vidéos, et

non à la détection d'objets dans des images. Les meilleurs combinaisons obtenues sont **HOG-LBP** et **HOG-LSS**.

3.3 Le choix du descripteur HOG

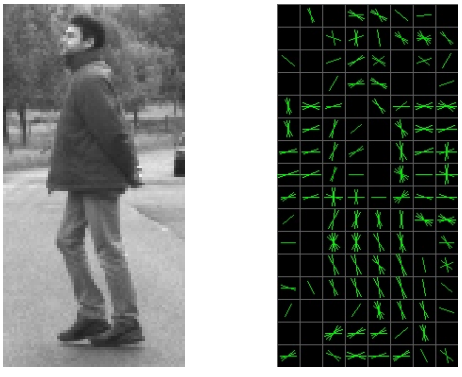
L'état de l'art des descripteurs possibles pour la détection de piétons montre qu'il existe de nombreuses manières d'aborder le problème. La multitude de méthodes et de descripteurs - illustrée par la revue non exhaustive faite dans la section précédente - vient du fait que le problème n'est toujours pas résolu à ce jour après une dizaine d'années de recherche sur le sujet, ce qui incite la communauté à continuer de proposer de nouvelles méthodes pour en améliorer les performances.

Les approches SIFT ou bien SURF ont été initialement proposées pour l'extraction de points d'intérêts dans le but de faire de l'appariement entre deux images. Même si le SIFT est un excellent descripteur de par ses propriété d'invariances notamment, la multitude des transformations, la complexité des structures, la dimension du descripteur et les calculs nécessaires à son élaboration sont des inconvénients pour son implémentation. Les descripteurs basés sur des informations temporelles (descripteurs à base de Histogram of Optical Flow) peuvent être également problématique. Bien qu'il soit possible de les reformuler suivant le modèle flot de données, ces algorithmes nécessitent forcément la mémorisation d'une image, qui risque de poser des problèmes lors de leur implémentation sur une cible FPGA où les quantités de mémoires sont souvent limitées.

Restent dès lors à ce niveau, le descripteur HOG, les LBP (une version étendue comme le CS-LBP) et les covariances. Les approches les plus récentes, procurant les meilleurs résultats, sont basées sur la fusion de descripteurs existants. La plupart intègrent le descripteur HOG. Nous avons donc choisi d'explorer ce descripteur car il a par ailleurs montré de bonnes capacités à détecter des piétons lorsqu'il est couplé à un système d'apprentissage de type SVM [DT05, DWSP09] . Il pourrait aussi être couplé avec un autre descripteur (par example un LBP) dans le futur afin d'améliorer la fiabilité de détection.

3.4 Considérations détaillées sur le descripteur HOG

Introduit par Dalal et Triggs dans [DT05], le descripteur HOG est fondé sur le calcul des histogrammes locaux de chaque orientation des gradients normalisés d'une image sur une grille dense. Un exemple de descripteur HOG est illustré sur la figure 3.11. La Fig 3.11b schématise l'orientation du gradient la plus représentée dans l'histogramme calculé sur la case correspondante. On retrouve bien, visuellement, la silhouette du piéton.



(a) *(b)*

Fig. 3.11: (a) Une image de piéton extraite de la base de données Daimler. (b) Le descripteur HOG associé. Dans chaque case on a fait figurer les orientations maximales relevées dans l'histogramme des orientations des gradients calculé sur cette case.

Une représentation sous une forme flot de données du calcul du HOG au plut haut niveau est donnée sur la Fig. 3.12. Les pixels venant du capteur d'image arrivent par la gauche, traversent les différents blocs pour sortir sous forme d'une flot de descripteurs. Nous avons volontairement supprimé de l'algorithme original décrit dans [DT05] deux étapes car leur impact sur la qualité de l'espace de description est négligeable. La première est la normalisation des couleurs et la correction du gamma, qui n'ont quasiment aucun effet sur le bon fonctionnement, comme expliqué dans [DT05]. La seconde est celle du filtrage effectué sur les gradients calculés. Les expériences de Dalal et Triggs visant à quantifier l'impact de différents masques de filtrage ont en effet montré que cette étape avait pour effet de réduire la qualité du descripteur final pour la plupart des filtres testés. Au final, restent trois étapes : **Extraction des gradients**, **histogramme**, et **normalisation**, comme illustré sur la Fig. 3.12.

Fig. 3.12: Etapes de l'extraction de descripteur HOG

La **première** étape calcule les dérivées partielles G_x et G_y dans les directions x et y pour chaque pixel (x, y) de l'image I.

$$G_x(x,y) = \frac{\partial I}{\partial x} = I(x+1,y) - I(x-1,y) \tag{3.14}$$

$$G_y(x,y) = \frac{\partial I}{\partial y} = I(x,y+1) - I(x,y-1) \tag{3.15}$$

L'amplitude du gradient et l'orientation associée à chaque pixel (x, y) sont alors obtenues par les formules suivantes :

$$m(x,y) = \sqrt{G_x(x,y)^2 + G_y(x,y)^2} \tag{3.16}$$

$$\theta(x,y) = \arctan\left(\frac{G_y(x,y)}{G_x(x,y)}\right) \tag{3.17}$$

La **seconde** étape, illustrée sur la Fig. 3.13, est la plus complexe à élaborer . Une fois les gradients calculés, l'espace des orientations possibles $(0 \ldots 180^o)$ est discrétisée en N_b intervalles (aussi appelés *bins*) égaux[2]. L'image des gradients est alors découpée en régions spatiales rectangulaires (ou circulaires) appelé *cellules*. Au sein de chaque *cellule*, un histogramme est construit. L'histogramme est constitué de N_b classes représentant les N_b intervalles d'orientations possibles et l'amplitude du gradient est accumulée pour chacune des classes. Les sommes des amplitudes des gradients sont considérées suffisantes en pratique pour la détermination de la valeur de l'histogramme bien qu'en théorie des fonctions de vote plus complexes puissent êtres utilisées (voir [DT05] pour plus de détails). Dans le but de limiter les problèmes d'aliasing, les valeurs des histogrammes peuvent également être interpolées tri-linéairement en orientation et position.

2. Le signe du gradient n'a pas d'impact sur la qualité de détection pour un piéton selon [DT05], ce qui permet de réduire l'espace des orientations à la demi-circonférence supérieure du cercle.

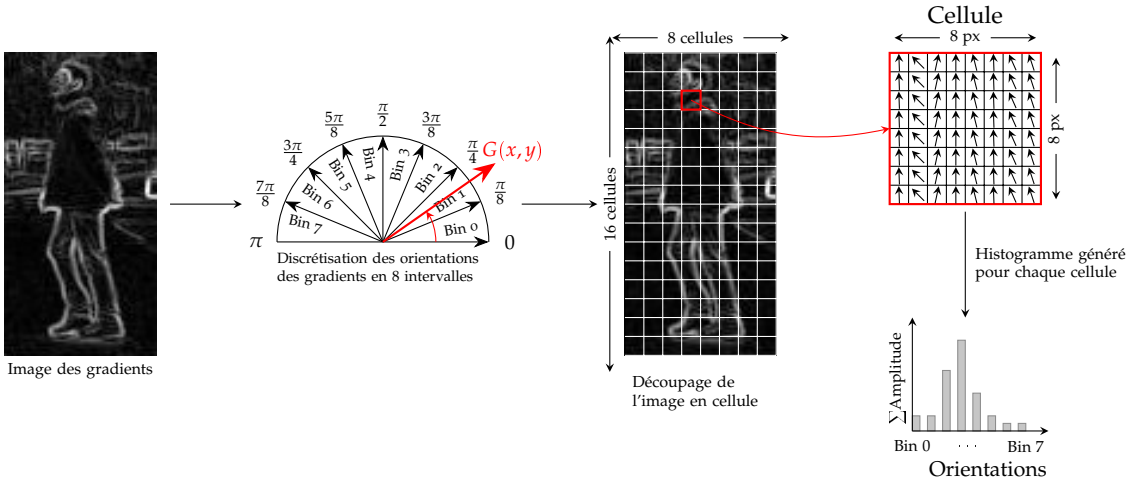


Fig. 3.13: Calcul de l'histogramme des orientations de gradients

La **troisième** étape est une normalisation des histogrammes afin de rendre le descripteur moins sensible aux changements d'illumination et de contraste. Dalal [DT05] a quantifié l'impact de différentes méthodes de regroupement de cellules sur la qualité du descripteur final. Le schéma le plus utilisé consiste à regrouper les cellules en entités plus larges, appelées blocs, où un bloc est typiquement composé de quatre cellules adjacentes, comme illustré sur la Fig. 3.14. Chaque descripteur généré sur chaque bloc est ensuite normalisé. Différentes opérations de normalisation peuvent être utilisées chacune induisant des performances finales différentes. La formulation proposée utilise une normalisation de type $L1$, dont le choix sera argumenté dans la suite. L'opération de normalisation est appliquée sur chaque bloc indépendamment mais les blocs se recouvrent.

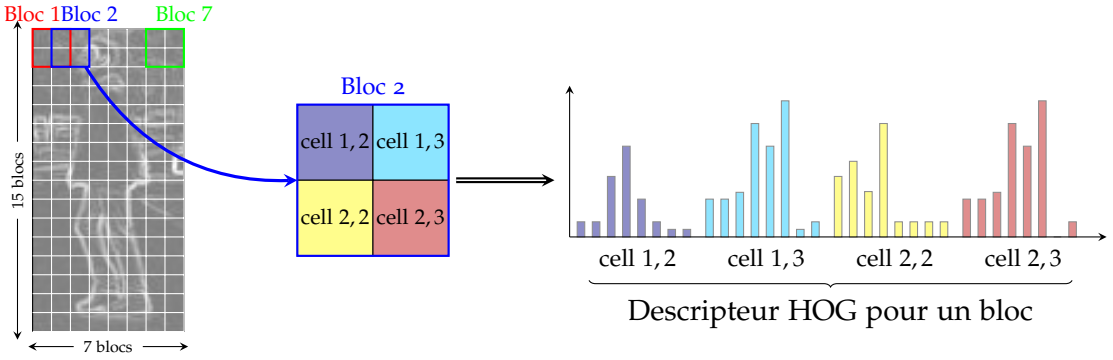


Fig. 3.14: Regroupement des cellules en blocs

Considérant une image constituée de $a \times b$ cellules avec le schéma de regroupement de blocs de la Fig. 3.14 ($a = 8$, $b = 16$), le nombre total de descripteurs HOG généré sera égal au nombre de blocs de l'image soit $a - 1 \times b - 1$ pour l'ensemble de l'image.

Maintenant que le choix du descripteur a été fait, la prochaine section s'intéresse au fonctionnement de la seconde partie du système de détection, à savoir un système de classification par Support Vector Machine (SVM) opérant une fenêtre glissante.

3.5 Apprentissage supervisé à base de Machines à vecteurs support (SVM)

Les machines à vecteurs support (SVM) sont des méthodes d'apprentissage discriminatoires relativement récentes dans le domaine de l'apprentissage supervisé, introduites par Vapnik et al [CV95]. La notion fondamentalement introduite par les SVM est celle d'**hyperplan de marge maximale**. Vapnik propose un moyen de créer des classifieurs non-linéaires à partir de séparateurs linéaires et de l'utilisation de *noyaux*. Prenons le cas simple d'une classification avec deux classes, illustré sur la figure 3.15.

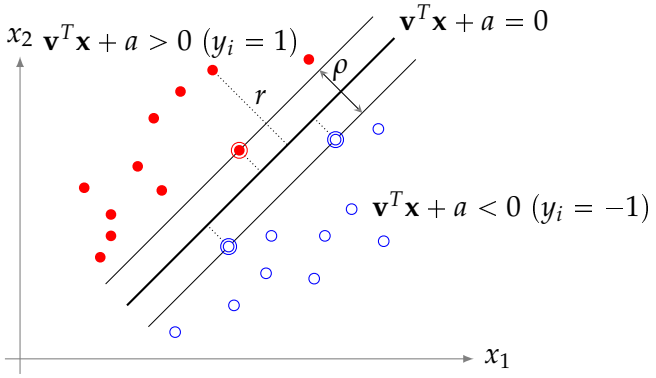


Fig. 3.15: Illustration du principe des SVMs sur une classification avec deux classes et des données séparables.

Soit un ensemble de descripteurs et leurs étiquettes associées $\{(\mathbf{x_i}, y_i)\}_{i=1...n}$, $\mathbf{x_i} \in \mathbb{R}^d$ ($d = 2$ sur la Fig. 3.15) et $y_i \in \{-1, 1\}$ séparables par un hyperplan de marge ρ. Pour chaque exemple d'apprentissage $(\mathbf{x_i}, y_i)$, on a l'inégalité suivante :

$$y_i(\mathbf{v}^T\mathbf{x_i} + a) \geqslant \rho/2 \tag{3.18}$$

où $(\mathbf{v}, a)$ est le couple de paramètres associés à la frontière de décision $f(x) = \mathbf{v}^T\mathbf{x} + a = 0$, $\mathbf{v} \in \mathbb{R}^p$ (normale à l'hyperplan) et $a \in \mathbb{R}$ (distance de l'hyperplan à l'origine). La première étape de la classification consiste à effectuer les changements de variables : $\mathbf{w} = \frac{\mathbf{v}}{m||\mathbf{v}||}$ et $b = \frac{a}{m||\mathbf{v}||}$ où m est la distance minimale entre un échantillon $\mathbf{x_i}$ et la frontière de décision. Cette étape permet de garantir l'unicité de la solution car la frontière définie par le couple $(\mathbf{v}, a)$ peut être également l'être pour tous les multiples $(k\mathbf{v}, ka)$ où $\forall k \in \mathbb{R}$.

La distance r qui sépare chaque point de l'hyperplan recherché est définie par sa projection orthogonale sur l'hyperplan canonique[3] :

$$r = \frac{y_i(\mathbf{w}^T\mathbf{x_i} + b)}{\|\mathbf{w}\|} \tag{3.19}$$

Les points dont la distance r satisfait l'égalité de l'équation 3.18 sont appelés *vecteurs support* (points encerclés sur la figure) et notés $\mathbf{x_s}$. Pour chaque vecteur support, on obtient sa distance à l'hyperplan par l'équation :

$$r = \frac{y_s(\mathbf{w}^T\mathbf{x_s} + b)}{\|\mathbf{w}\|} = \frac{1}{\|\mathbf{w}\|} \tag{3.20}$$

On cherche donc à déterminer les paramètres $\mathbf{w}$ et b tels que la marge de l'hyperplan qui sépare le mieux les deux classes dans l'espace des descripteurs soit maximale, ce qui revient à résoudre le problème d'optimisation quadratique suivant :

3. L'hyperplan canonique est l'hyperplan après une normalisation de l'équation 3.18

$$\begin{cases} \max\limits_{\mathbf{w},b} & \dfrac{1}{\|\mathbf{w}\|} \\ \text{avec } y_i(\mathbf{w}^T\mathbf{x_i} + b) \geqslant 1 & i = 1, n \end{cases}$$

Classiquement, ce problème est souvent reformulé en minimisant $||w||^2$ au lieu de maximiser l'inverse de la norme, ce qui donne le problème suivant :

$$\begin{cases} \min\limits_{\mathbf{w},b} & \dfrac{1}{2}\|\mathbf{w}\|^2 \\ \text{avec } y_i(\mathbf{w}^T\mathbf{x_i} + b) \geqslant 1 & i = 1, n \end{cases} \tag{3.21}$$

La résolution du problème 3.21 (le lecteur peut se référer aux travaux de Vapnik dans [Vap82] pour la démonstration) donne :

$$\mathbf{w} = \sum_{i \in A} \alpha_i y_i \mathbf{x}_i, \qquad \alpha_i \geqslant 0 \tag{3.22}$$

où A est l'ensemble des indices vecteurs supports[4], α les multiplicateurs de Lagrange associés[5] et y la classe du point $\mathbf{x}_i$. La valeur du biais b est égale aux multiplicateurs de Lagrange de la contrainte d'égalité à l'optimum. Une fois que $\mathbf{w}$ et b sont déterminés, la fonction f de classification qui sera appliquée aux nouvelles entrées $\mathbf{x}$ est définie par :

$$f(\mathbf{x}) = signe\left(\sum_{i \in A} \alpha_i y_i \mathbf{x}_i^T \mathbf{x} + b\right), \qquad \alpha_i \geqslant 0 \tag{3.23}$$

On remarque que $\mathbf{w}$ n'apparaît pas explicitement dans l'Eq. 3.23 et que la fonction de classification se réduit à un produit scalaire entre le vecteur d'entrée $\mathbf{x}$ et les vecteurs supports $\mathbf{x_i}$. Le signe de $f(x)$ indique la classe correspondante au vecteur d'entrée.

Malheureusement, il n'existe que peu de cas dans le domaine du traitement d'images où les données sont linéairement séparables comme illustré sur la Fig. 3.15. Un des intérêts des systèmes SVM réside dans leur capacité à converger et à produire une fonction de décision même dans le cas où les données d'entrée ne sont pas séparables. C'est d'ailleurs cette spécificité qui a fait le succès des systèmes SVMs par rapport au perceptron. Toutefois, la fonction de décision obtenue engendre forcément des erreurs de classification. Dans le cas où l'on souhaite plus précisément séparer les données, la seconde possibilité offerte par les SVMs consiste en l'utilisation de noyaux. Les noyaux vont permettre de projeter l'espace de description dans un espace de dimension supérieure et dans lequel on espère que le problème deviendra complètement séparable. Cependant, l'utilisation d'un noyau engendre un cout de calcul supplémentaire dans l'élaboration de la fonction de décision.

3.5.1 Le cas des données non séparables

Afin de rendre possible la convergence de l'algorithme d'optimisation dans le cas où les données d'entrées ne sont pas séparables, Vapnik a introduit la notion de *marge souple* en utilisant des variables d'écart positives (ξ_i sur la figure 3.16) associées à chacune des observations $(\mathbf{x_i}, y_i)$. Dans le cas où le point vérifie la contrainte de marge $y_i(\mathbf{w}^T\mathbf{x_i} + b) \geqslant 1$, la variable d'écart sera nulle, ce qui donne comme fonction de coût :

$$\xi_i = \max(0, 1 - y_i(\mathbf{w}^T\mathbf{x_i} + b)) \tag{3.24}$$

Le problème d'optimisation dans le cas de données non séparables consiste désormais à maximiser la marge séparant les deux classes tout en minimisant le nombre d'erreurs,

4. Les vecteurs supports sont déterminés par le processus d'optimisation
5. Les multiplicateurs de Lagrange apparaissent lors de la formulation duale du problème quadratique [CV95]

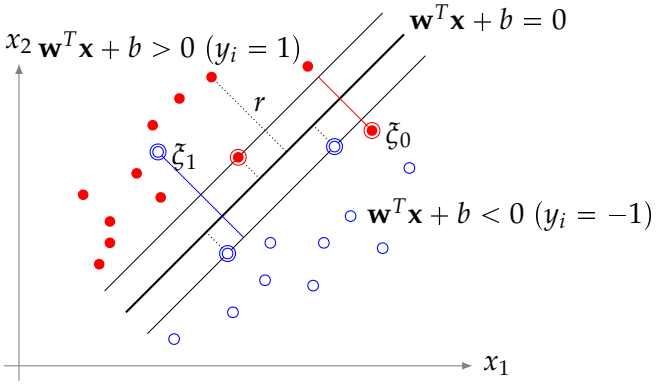


Fig. 3.16: Illustration du principe des SVMs sur une classification binaire avec des données non séparables.

plus précisément le nombre de vecteurs qui sont du mauvais côté de l'hyperplan obtenu. Analytiquement, ceci se fait en intégrant une contrainte sur les variables d'écart positives dans le problème d'optimisation, qui se formule désormais de la manière suivante :

$$\begin{cases} \min\limits_{\mathbf{w},b,\xi} \begin{cases} \dfrac{1}{2}\|\mathbf{w}\|^2 \\ \dfrac{1}{d}\sum\limits_{i=1}^{n}\xi_i^d, \quad (d \text{ fixé arbitrairement, typiquement 2}) \end{cases} \\ \text{avec } y_i(\mathbf{w}^T\mathbf{x_i}+b) \geqslant 1-\xi_i \qquad i=1,n \\ \qquad \xi_i \geqslant 0 \qquad i=1,n \end{cases} \tag{3.25}$$

La résolution du problème ne peut s'effectuer que grâce à l'utilisation d'un terme d'équilibrage, également appelé facteur de régularisation, $C > 0$. Ce terme d'équilibrage est un moyen de contrôler le compromis entre l'importance de maximiser la marge et celle de coller au mieux aux données d'apprentissage. L'introduction des variables d'écart positives n'a pas d'impact sur la formulation de $\mathbf{w}$ (Eq. 3.22) mais seulement sur les valeurs des multiplicateurs de Lagrange qui seront bornés supérieurement par le terme d'équilibrage.

$$\mathbf{w} = \sum_{i\in A}\alpha_i y_i \mathbf{x}_i, \qquad 0 \leqslant \alpha_i \leqslant C \tag{3.26}$$

L'expression de la fonction de décision reste identique à celle de l'Eq.3.23.

$$f(\mathbf{x}) = signe\left(\sum_{i\in A}\alpha_i y_i \mathbf{x}_i^T\mathbf{x}+b\right), \qquad 0 \leqslant \alpha_i \leqslant C \tag{3.27}$$

3.5.2 Le cas non linéaire : les noyaux

Nous venons de voir qu'une SVM est capable de produire une fonction de décision dans le cas où quelques erreurs de classification peuvent être autorisées. Cependant, il existe également des cas où une séparation linéaire est impossible (ou ne produit aucun résultat satisfaisant). Si on prend l'exemple de la Fig. 3.17, la fonction de décision obtenue sera de mauvaise qualité avec un séparateur linéaire, peu importe le nombre d'erreurs que l'on accepte. Le problème illustré sur la Fig. 3.17 est simplement non séparable par une fonction linéaire. La solution existante dans ce cas de figure consiste en l'utilisation de **noyaux**.

Un noyau projette l'espace de description original dans un espace de dimension supérieure ou égale sur lequel les données transformées deviendront séparables. La table 3.2 montre quelques exemples de noyaux couramment utilisés. Les noyaux sont classés généralement en deux types, les radiaux qui dépendent d'une distance (le carré de la distance euclidienne pour le noyau gaussien) et les projectifs qui sont définis à partir d'un produit

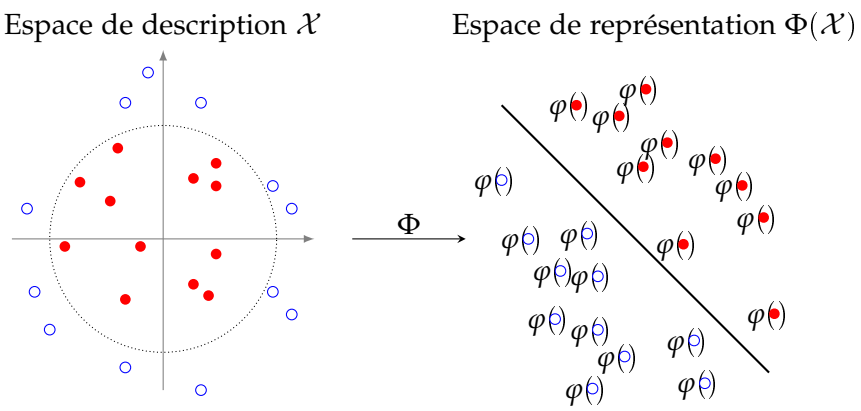


Fig. 3.17: Exemple d'un séparateur SVM non linéaire. L'application Φ transforme l'espace des descripteurs en un nouvel espace dans lequel les données sont séparables. Dans ce cas, les espaces sont de dimension égale.

scalaire (comme le noyau polynomial). Toutes les fonctions respectant le théorème de Mercer [Mer09] peuvent être utilisées comme noyaux.

Nom	Noyau $K(\mathbf{x}_i, \mathbf{x}_j)$	Type
Polynomial	$(\mathbf{x}_i^T \mathbf{x}_j)^p$	Projectif
Affine	$(\sigma + \mathbf{x}_i^T \mathbf{x}_j)^p$	Projectif
Gaussien (RBF)	$\exp\left(-\frac{\|\mathbf{x}_i - \mathbf{x}_j\|^2}{2\sigma^2}\right)$	Radial

Tab. 3.2: Exemples de noyaux de SVMs pour l'espace $\mathcal{X} = \mathbb{R}^p$

Dans le cas général, la fonction de décision d'une SVM est définie comme suit. Soit $\{(\mathbf{x_i}, y_i)_{i=1\ldots n}\}$ un ensemble de descripteurs avec $\mathbf{x_i} \in \mathcal{X}$ et $y_i \in \{-1, 1\}$. La fonction de classification est exprimée par :

$$f(\mathbf{x}) = signe\left(\sum_{i \in \mathcal{A}} y_i \alpha_i K(\mathbf{x_i}, \mathbf{x}) + b\right), 0 \leqslant \alpha_i \leqslant C \tag{3.28}$$

où $\mathcal{A}$ est l'ensemble des vecteurs supports, $K(\mathbf{x_i}, \mathbf{x})$ représente la fonction noyau et C le terme d'équilibrage. Le cas linéaire que nous avons vu dans l'Eq. 3.27 devient alors un cas trivial découlant de l'Eq.3.28 dans lequel la fonction noyau $K(\mathbf{x}_i, \mathbf{x}_j)$ est la fonction identité ($K(\mathbf{x}_i, \mathbf{x}_j) = \varphi(\mathbf{x}_i)^T \varphi(\mathbf{x}_j)$ avec $\varphi(\mathbf{x}) = \mathbf{x}$). L'utilisation des noyaux est cependant à considérer dans son ensemble car même s'ils permettent d'améliorer la qualité de séparation d'un problème, ils engendrent également un surcout calculatoire non négligeable car la fonction K doit être calculée pour chaque vecteur d'entrée.

3.5.3 Extension aux SVMs multiclasses

Par essence, les SVM sont définies pour traiter le problème de la séparation entre deux classes. Le passage des SVMs biclasses aux SVMs multiclasses est possible par l'intermédiaire de méthodes de décomposition qui traitent multiclasses comme une combinaison de problèmes biclasses, par dichotomie. Pour un descriptif complet de la théorie des SVMs multiclasses, on peut se référer à Guermeur qui expose le contexte théorique et applicatif des SVMs multiclasses [Gue07]. Il existe trois méthodes de décomposition :

1. L'approche un contre tous. On utilise un classifieur binaire par catégorie. Pour chaque exemple d'entrée on applique l'ensemble des classifieurs et on utilise le principe du

"winner takes all" pour déterminer la classe. Cette technique a cependant montré certaines limites [LZ05] où, dans certains cas, il est impossible de séparer une classe contre l'ensemble des classes possibles.

2. L'approche un contre un [Fri96, HT98]. La méthode consiste à utiliser un classifieur par couple de catégories. Le classifieur indicé (k,l) distingue l'exemple d'entrée de la catégorie k à celle de l. La détermination s'effectue en appliquant C_Q^2 classifieurs et la décision finale s'obtient souvent en effectuant un vote majoritaire.
3. Méthode fondée sur des graphes de décision, nommé DAGSVM [Pol86]. Cette méthode s'appuie sur un graphe de décision orienté sans cycle DDAG où chaque noeud correspond à une liste de classes. La SVM correspondante effectue une décision entre les deux catégories de la liste.

3.5.4 Evaluation de l'apprentissage d'une SVM

Choisir les bons paramètres est une étape essentielle lors de la mise en œuvre d'un système SVM. Ce choix ne peut se faire sans une méthode permettant d'évaluer ses conséquences sur les performances du système complet. Il existe plusieurs moyens de mesure de performance d'un système SVM [CVBM02, DKP03]. Si l'on suppose un modèle avec un ensemble de paramètres inconnus et un jeu de données d'apprentissage sur lequel il est possible d'obtenir un modèle, le processus d'apprentissage optimise les paramètres afin que le modèle se rapproche le plus possible des données d'apprentissage. Cependant, si on extrait un échantillon de la base d'apprentissage, il arrive que le modèle obtenu ne soit plus autant en adéquation avec cet élément. Ce problème, appelé *sur-apprentissage* peut survenir lorsque la base de données est incomplète ou que le nombre de paramètres du modèle est trop important.

Afin de pallier le sus-cité, des méthodes de validation dites croisées sont communément utilisées. Le but de la validation croisée est d'estimer le niveau de correspondance entre le modèle et les données utilisées lors de la phase d'apprentissage. Un état de l'art des procédures de validation croisée dans le cadre général est disponible dans [AC10]. Dans le cas de nos travaux, on utilisera la validation croisée à k-parts, la validation *Leave-One Out* et la borne Xi-Alpha de Joachims [Joa02].

3.5.4.1 Validation croisée à k-parts

La validation croisée à k-parts (*k-fold cross-validation*) est une technique très employée pour évaluer la qualité d'un espace de description et éviter les problèmes de sur-apprentissage (*overfitting*). La base d'apprentissage composée de n échantillons est sous-échantillonnée aléatoirement en k sous ensembles de taille égale. Sur les k ensembles, un sous ensemble est retenu comme base de test et les $k-1$ autres ensembles sont utilisés comme base d'apprentissage. Le processus de validation croisée est répété k fois en changeant de sous ensemble de test à chaque fois. Les k résultats de classification obtenus sur l'ensemble de test sont moyennés pour produire une estimation finale des performances du système.

En pratique, le choix du nombre de sous-ensembles va dépendre de la dimension du jeu de données. Pour les bases de données importantes, seulement 3 sous-ensembles peuvent être suffisants pour obtenir des résultats significatifs. Le choix le plus commun est d'utiliser $k = 10$, signifiant 10 apprentissages successifs avec 10% de la base utilisée pour le test. Plus le nombre de sous-ensembles est important, plus le temps d'apprentissage est long mais plus les estimations de performances seront précises.

3.5.4.2 Validation croisée Leave-One Out

Dans le cas de la validation $k = n$, la validation croisée à k parts revient à une validation dite Leave-One-Out Error (LOO). Dans un cas plus général, le terme de *Leave-p-out* est également utilisé, supposant p observations comme jeu de données de validation et $n-p$ comme base d'apprentissage. La validation croisée requiert d'apprendre et de tester C_p^n fois

où n est la dimension de la base d'apprentissage. Cette technique, même si elle fournit des résultats assez précis, peut prendre un temps considérable lorsque la base d'apprentissage est importante. A titre d'exemple, nous utiliserons dans ces travaux la librarie d'apprentissage SVMLight [Joa99] sur lequel nous pourrons faire de la validation L-p-out avec $p = 10$ sur une base d'apprentissage de 15000 images, le calcul de cet estimateur nécessite environ 1500 apprentissages successifs, nécessitant une quarantaine d'heures de calcul sur un processeur de bureau [6].

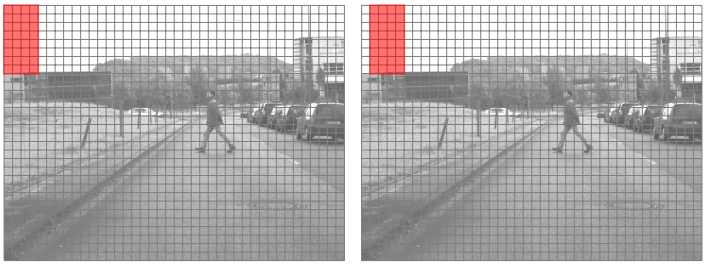
3.5.4.3 La borne Xi-Alpha

Nous utiliserons également l'estimateur Xi-Alpha, introduit par Joachims [Joa02] et disponible dans la librairie d'apprentissage SVMLight [Joa99], qui procure une borne supérieure de l'erreur LOO à partir des solutions du problème d'optimisation de la SVM. L'avantage de cette méthode est qu'elle fournit une information simple à partir des résultats obtenus sur l'ensemble d'apprentissage complet, évitant les apprentissages successifs nécessaires au calcul de l'estimateur LOO. Il existe cependant d'autres estimateurs de la qualité d'un apprentissage, évalués et comparés par Duan [DKP03].

3.6 Classification par technique de fenêtrage glissant

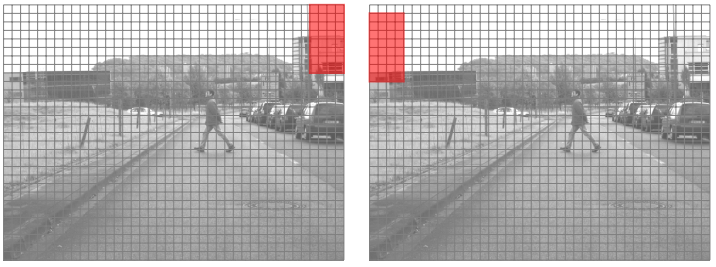
Comme on l'a vu (Eq. 3.28), la fonction de classification s'applique sur un vecteur d'entrée $\mathbf{x}$. Pour les applications qui nous intéressent, ce vecteur correspond au descripteur calculé sur une sous-région de l'image originale. Cette région est communément appelée **fenêtre de détection** (équivalente à l'*imagette* de la Fig. 3.3). Lorsque l'on désire détecter plusieurs piétons sur des images entières, une méthode couramment utilisée consiste alors à limiter le nombre de fenêtres de détection à certaines zones d'intérêts (ROI) afin d'alléger la charge de calcul. On s'appuie alors souvent sur des systèmes prédictifs (filtres Bayésiens,...) de suivi des cibles pour mettre à jour les positions des fenêtres. Mais cette approche ne suffit pas dans le cas d'une caméra embarquée dans un environnement routier. Dans ce cas, en effet, l'évolution rapide de l'environnement impose de faire une mise à jour complète de l'ensemble des fenêtres pour intégrer de nouvelles cibles potentielles. Cette approche trouve alors certaines limites. Si ce rafraîchissement ne se fait pas suffisamment souvent, le risque existe de manquer des certaines cibles. La technique proposée est basée sur une fenêtre de détection déplacée dans toute l'image, telle que pour chaque position, la SVM fournit une information sur la présence ou non d'un piéton. L'intérêt du modèle flot de données est qu'il va rendre le cout de calcul de cette méthode indépendant du nombre de fenêtres à calculer. Ce principe, communément appelé *fenêtre glissante*, est illustré par la Fig 3.18.

6. Valeur mesurée par la librairie d'apprentissage. Exécution sur un processeur *i*7-870 et une distribution Linux Ubuntu 14.04



(a) Position initiale

(b) Deuxième fenêtre de détection (recouvrement horizontal)



(c) Dernière fenêtre de la première ligne

(d) Première fenêtre de la seconde ligne (recouvrement vertical)

Fig. 3.18: Illustration de la fenêtre de détection glissante

L'objectif de cette approche est de faire remonter des informations sur les présences potentielles de piétons dans chaque image, et ce sans aucune prédiction/hypothèse sur les positions de ces piétons. Ce système pourra également servir d'entrée à des systèmes prédictifs de plus haut niveau, qui eux pourront faire des hypothèses sur les corrélations spatiales et temporelles entre plusieurs images successives afin de faire du suivi de piétons. Considérant une image d'entrée I, l'ensemble des réponses de la SVM s'exprime alors sous la forme d'une matrice de taille ζ, ψ :

$$Y(I) = \begin{pmatrix} y_{1,1} & \cdots & y_{1,\psi} \\ \vdots & \ddots & \vdots \\ y_{\zeta,1} & \cdots & y_{\zeta,\psi} \end{pmatrix} \tag{3.29}$$

où chaque y_{ij} représente la réponse de la SVM pour une fenêtre de détection de l'image d'entrée et ζ, ψ le nombre de fenêtres dans l'image globale. Cette représentation matricielle sera utilisée par les systèmes de traitement situés en aval qui pourront exploiter ces données afin de filtrer les résultats bruts et d'améliorer la fiabilité du système. Cette matrice Y pourrait être étendue à trois dimensions dans le cas de systèmes multi-échelles et même à quatre dimensions si l'évolution temporelle des réponses du système est considérée.

Dans notre cas, la fonction de classification (Eq. 3.28) associée à un noyau linéaire pour un problème avec deux classes peut s'exprimer par un produit scalaire entre le modèle et le descripteur [CV95] :

$$y(\mathbf{x}) = signe(\mathbf{w}^T \cdot \mathbf{x} + b) \tag{3.30}$$

Dans le cas de l'utilisation de descripteurs de type HOG, le vecteur d'entrée $\mathbf{x}$ correspond à la collection des descripteurs HOG de tous les blocs composants une fenêtre de détection. La dimension du vecteur $\mathbf{x}$ est alors une fonction du nombre d'entrées de l'histogramme, du nombre de cellules composantes un bloc et du nombre de blocs composant la fenêtre de détection. Le schéma le plus utilisé pour la détection de piéton est basé sur des fenêtres de détection de dimension 64×128 pixels, avec des cellules de 8×8 pixels, des blocs composés

de 2×2 cellules et des histogrammes à 9 classes, ce qui produira un descripteur de dimension 3780 par fenêtre de détection. La fonction *signe* retourne l'étiquette associée à la classe : 1 si la valeur est positive ou égal à 0 (piéton) et -1 si la valeur est négative (non-piéton). A signaler que même si seul le signe de la fonction de classification f suffit à la détermination de la classe à laquelle appartient l'échantillon testé, sa valeur procure une information sur la distance de l'échantillon par rapport à l'hyperplan séparant les deux classes, et peut servir d'information sur la confiance du résultat de classification. Cette possibilité sera utilisée seulement lorsque le filtrage sur des images sera abordé.

Il est important de noter que la technique de fenêtre glissante proposée ci-dessus est assez peu utilisée en pratique car son coût calculatoire est considérable. Selon Mizuno [MTT+12], l'ensemble des calculs du descripteur HOG et de la classification pour une seule image 1920×1080 est de 447×10^9 opérations (53×10^9 pour une image VGA). Si le calcul doit être fait directement sur un flux vidéo en temps réel, ce traitement doit être effectué plusieurs fois par seconde. L'estimation de Mizuno ne considère que le cas d'un noyau linéaire (simple produit de deux vecteurs), il est facile d'imaginer l'impact du choix d'un noyau plus complexe sur les coûts calculatoires, comme le noyau RBF (Table.3.2) intégrant des calculs de normes et d'exponentielles. Nous montrerons au chapitre 4 que la reformulation algorithmique de cette méthode suivant le modèle flot de données, en autorisant une mise en œuvre complète du parallélisme, va permettre de rendre le temps de calcul indépendant du nombre de fenêtres à évaluer dans l'image.

Après avoir présenté une méthode d'apprentissage supervisé basée sur des machines à vecteurs supports, l'apprentissage supervisé à partir des réseaux de neurones convolutionnels est abordé.

3.7 Apprentissage supervisé à base de réseaux de neurones convolutionnels (CNN)

Les réseaux de neurones convolutionnels (Convolutional Neural Networks (CNN)) sont une variante bio-inspirée du perceptron multicouches (MultiLayer Perceptron (MLP)) dans lesquels les couches linéaires ont été remplacées par des couches convolutionnelles. Les CNN sont issus des travaux de Hubel et Wiesel [HW68] qui se sont inspirés de la structure du cortex visuel des chats. Cette structure est constituée d'un arrangement complexe de cellules où chacune est sensible à une sous région du champ visuel. Les cellules fonctionnent comme des filtres locaux sur l'image d'entrée et sont adaptées à l'exploitation des corrélations spatiales fortes naturellement présente dans les images. De ces travaux ont découlé plusieurs modèles neuro-inspirés, le NeoCognitron [Fuk80], le HMAX [SWB+07] ou bien LeNet-5 [LBBH98, LBOM12].

A la différence d'un système SVM dont la performance va dépendre de la qualité du descripteur utilisé en entrée, un réseau convolutionnel intègre à la fois la détermination du descripteur (appris par les premières couches du réseau) et un système de classification (typiquement un MLP dans le modèle LeNet-5). Cette approche est efficace car elle permet de créer implicitement un descripteur en fonction d'un problème et d'un modèle de réseau donné, en apprenant les poids des convolueurs (par rétropagation de l'erreur du gradient [LCTHS88, HN89]).

Nous nous intéressons ici au modèle LeNet-5 introduit par LeCun. Un exemple de réseau conforme à ce modèle est donné sur la Fig. 3.19.

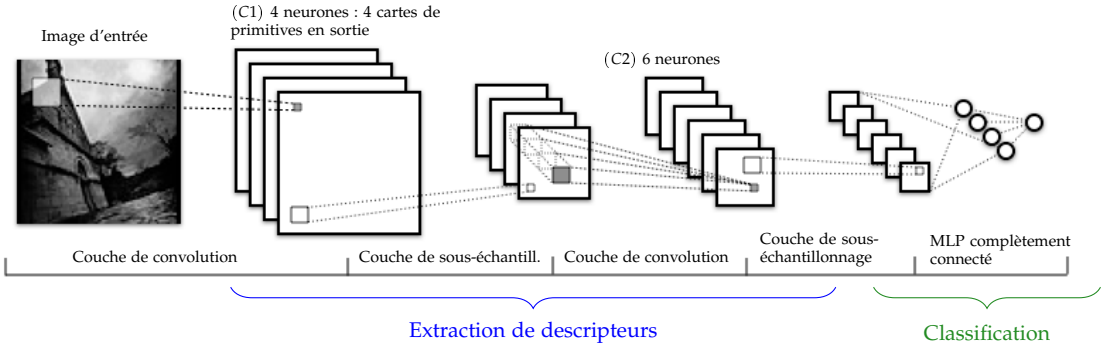


Fig. 3.19: Exemple de réseau convolutionnel Lenet, source [len15]

Comme l'illustre la Fig. 3.19, un réseau de neurones convolutionnels est composé de **couches** successives. Les données qui transitent entre les couches sont des volumes $3D$, appelées *vecteurs de cartes de primitives* (*feature map vectors*). Chaque élément de ce vecteur (sortie d'un neurone) est donc *une carte de primitives* $2D$, que l'on peut représenter sous la forme d'une image. Voici une description des couches existantes dans un réseau CNN LeNet.

3.7.1 Couche de convolution

Aussi appelé ***Filter Bank Layer***, une couche de convolution est composée d'un ensemble de *neurones*, aussi appelés *noyaux* ou *filtres*. L'entrée d'une couche de convolution est un volume $3D$, composé de D_1 cartes de primitives $2D$, chacune de dimension $W_1 \times H_1$. Chaque neurone au sein de la couche reçoit un ensemble de N cartes de primitives en entrée (inférieur ou égal à $D1$). Un exemple de neurone convolutionnel est illustré sur la Fig. 3.20 (avec $N = 3$).

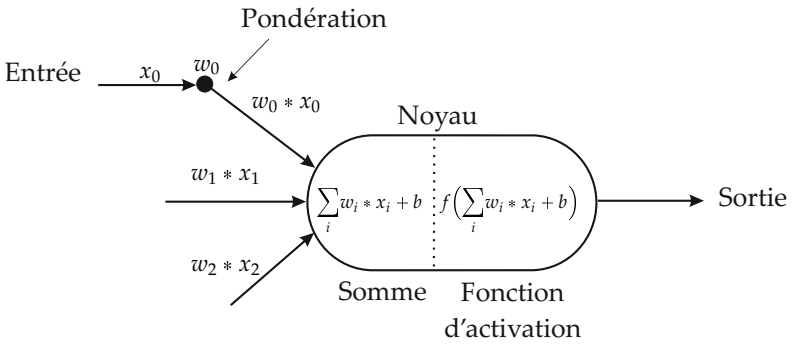


Fig. 3.20: Exemple de neurone convolutionnel

Au sein de chaque neurone, chaque carte de primitives entrante x_i (axone sur Fig. 3.20) est convoluée avec un filtre dont les valeurs sont issues de l'apprentissage (synapse). Le neurone calcule la somme des résultats de chaque convolution avant d'appliquer un biais b et une fonction d'activation f. Considérons K neurones dans une couche, chaque neurone j connecte les entrées x_i aux sorties y_j (Fig. 3.20) tel que :

$$y_j = f\left(b_j + \sum_{i \in N} w_{ij} * x_i\right), \quad j \in \{1, ..., K\} \tag{3.31}$$

où

- N est le nombre de cartes de primitives d'entrées connectées au neurone,
- w_{ij} sont des filtres de convolution de dimension $l1 \times l2$, associés à chaque carte de primitives d'entrées i pour le neurone j. Ces filtres sont appris durant la phase d'apprentissage,
- b_j est un biais issu de l'apprentissage associé au neurone j,
- $*$ est l'opérateur de convolution $2D$ discret,
- f est une fonction d'activation non linéaire, servant à ramener les sorties de chaque neurone dans la même plage de valeurs.

La sortie d'une couche de convolution est également un volume $3D$ composé de D_2 cartes de primitives de dimension $W_2 \times H_2$, dont les grandeurs sont déterminées par :

$$\begin{aligned} D_2 &= K \\ W_2 &= (W_1 - F + 2P)/S + 1 \\ H_2 &= (H_1 - F + 2P)/S + 1 \end{aligned}$$

où

- K est le nombre de neurones composant une couche,
- F est la dimension du champ réceptif du neurone (en pratique égale à la dimension des filtres de convolution $F = l_1 = l_2$),
- P est le paramètre de remplissage (*zero-padding*) des contours du volume d'entrée,
- S le pas de calcul (*stride*) qui sera utilisé pour l'évaluation du volume sur le neurone.

Si on considère, à titre d'exemple, une image d'entrée RGB de dimension 32×32, le volume d'entrée sera $32 \times 32 \times 3$. Si la couche convolutionnelle est composée de 2 neurones où chaque neurone est paramétré avec $F = 5$, $P = 1$ et $S = 1$ alors le volume de sortie du premier étage de convolution sera de dimension $30 \times 30 \times 2$ et chaque filtre sera de dimension $5 \times 5 \times 3$. Par ailleurs, afin de réduire le nombre de paramètres à déterminer au sein d'une couche de convolution, la technique de l'état de l'art repose sur le partage de paramètres (*parameter sharing*) consistant à appliquer les mêmes filtres et biais pour l'ensemble des données au sein d'une carte de primitives. Avec la technique de partages de paramètres, chaque

filtre utilise $F \times F \times D_1$ poids, soit un total de $F \times F \times D_1 \times K$ poids et K biais par couche convolutionnelle.

La dernière opération au sein d'un neurone (Fig. 3.20) applique une fonction d'activation, permettant d'introduire des non-linéarités dans le réseau. Les fonctions d'activations les plus courantes sont :

- **Sigmoïde**, définie par :

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.32}$$

  Cette fonction, représentée sur la Fig. 3.21 ramène les données d'entrée dans l'intervalle $[0, 1]$. En particulier, les valeurs largement négatives deviennent 0 et les valeurs largement positives 1. Cette fonction, utilisée pendant longtemps pour sa correspondance avec l'activation d'un neurone biologique, est un peu tombée en désuétude à cause de certains désavantages : tout d'abord, les gradients faibles ont tendance à être détruits. Ensuite, les valeurs de sortie ne sont pas centrées en 0, provoquant des variations de signe indésirable des gradients lors de la phase d'apprentissage. Enfin, le choix des poids initiaux est critique ; cssi es poids initiaux sont trop élevés alors les neurones vont saturer rapidement, ce qui limitera le nombre d'itérations lors de l'apprentissage. Ces inconvénients font que l'utilisation de la tangente hyperbolique est aujourd'hui plus courante.

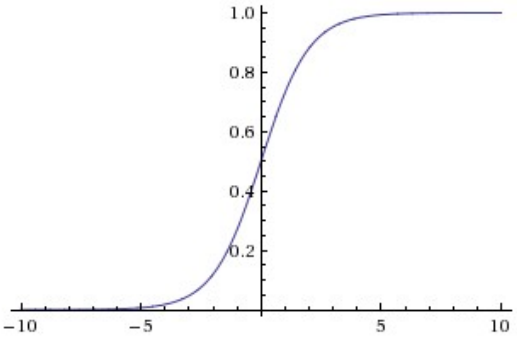


Fig. 3.21: Représentation de la fonction sigmoïde

- **Tangente hyperbolique**. La tangente hyperbolique, représenté sur la Fig. 3.22, a le même objectif que la fonction sigmoïde mais produit des données de sorties sur un ensemble centré en 0.

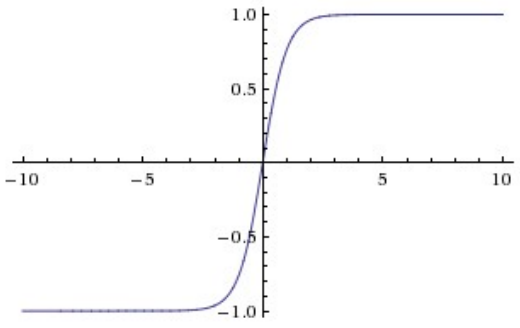


Fig. 3.22: Représentation de la fonction tangente hyperbolique.

- **ReLU** (*Rectified-Linear*). La fonction d'activation **ReLU** a été introduite par Krizhevsky dans [KSH12]. L'utilisation de cette fonction permet une convergence plus rapide (d'un facteur 6) de la descente du gradient par rapport aux fonctions classiques (sigmoïde et

tangente hyperbolique). La fonction ReLU est définie par :

$$f(x) = \begin{cases} x & \text{si } x > 0 \\ -kx & \text{si } x \leqslant 0 \text{ si, avec } k \in \mathbb{N} \end{cases} \tag{3.33}$$

Un exemple de graphe (pour $k = 0$) est donné Fig. 3.23.

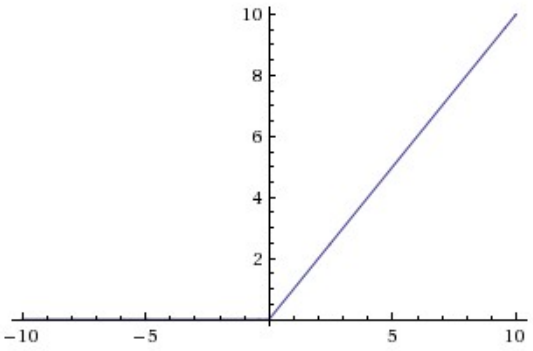


Fig. 3.23: Représentation de la Relu pour $k = 0$

3.7.2 Couche de sous-échantillonnage

Le second type couche, aussi appelée ***Feature Pooling Layer***, opère un sous-échantillonnage sur chaque carte de primitives provenant d'une couche de convolution (Fig. 3.24). La technique de sous-échantillonnage utilise généralement le maximum [AHMJP12], la moyenne [LBBH98] ou plus récemment la norme [ZF13] . Si le volume d'entrée est de dimension $D_1 \times W_1 \times H_1$ alors le volume de sortie sera de dimension $D_2 \times W_2 \times H_2$ dont les valeurs sont égales à :

$$\begin{aligned} W_2 &= ((W_1 - z)/s) + 1 \\ H_2 &= ((H_1 - z)/s) + 1 \\ D_2 &= D_1 \end{aligned}$$

où

- z correspond à la dimension du voisinage sur lequel est effectué le sous-échantillonnage,
- s est le pas entre deux applications du sous-échantillonnage ($s \leqslant z$ et $(W_1 - z)$ divisible par s).

Typiquement, ces couches sont paramétrées avec des filtres de dimension 2×2 sans recouvrement, ce qui revient à réduire la dimension de la carte de primitives d'un facteur 2 horizontalement et verticalement, soit une réduction de 75% des informations de chaque carte de primitives d'entrée (qui sera plus robuste aux variations de position des primitives dans la couche précédente).

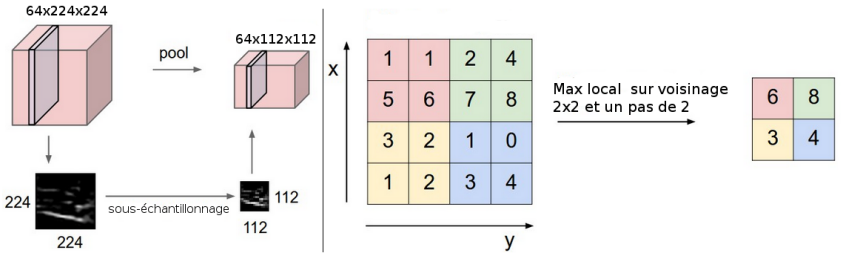


Fig. 3.24: Exemple de sous-échantillonnage ($z = s = 2$)

Il a été observé que l'utilisation d'un voisinage $z = 3$ avec un pas de recouvrement de 2 pouvait améliorer légèrement les résultats (0.4% de bonnes détections supplémentaires dans [KSH12]).

3.7.3 Couche de Classification

La dernière couche d'un CNN effectue la classification des cartes de primitives afin d'obtenir les résultats de correspondance avec un ensemble de classes. Dans le cas du modèle LeNet-5 de la Fig. 3.19, la classification est en fait effectuée par deux couches. La première, dite *complètement connectée*, composée de k unités, effectue pour chaque unité le produit scalaire de toutes les cartes de primitives d'entrées avec un vecteur appris et applique un biais et une sigmoïde. La seconde couche est composée d'unités calculant la distance euclidienne pour chaque classe de sortie y_i, à partir des j valeurs produites par la couche complètement connectée, soit :

$$y_i = \sum_{k=1}^{j} (x_k - w_{i,k})^2 \tag{3.34}$$

Depuis la proposition de Lecun, d'autres méthodes ont été proposées pour la structure la couche de classification. Krizhevsky a par exemple proposé l'utilisation de la régression logistique multinomiale [KSH12] (fonction *softmax*) pour remplacer le calcul de la distance euclidienne tout en conservant une couche complètement connectée.

Comparés aux réseaux de neurones classiques (type Artificial Neural Networks (ANN) [Hop88]), les réseaux CNN ont une connectivité partielle à l'entrée des couches de convolutions. Cette connectivité permet de réduire le nombre de paramètres à apprendre dans le réseau. La Fig. 3.25 illustre cette connectivité sur trois couches.

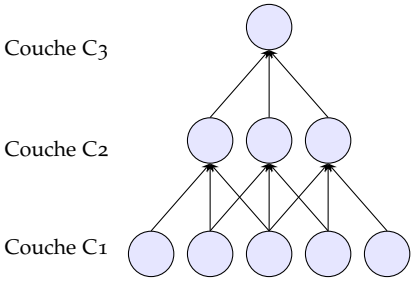


Fig. 3.25: Connectivité partielles des couches de classification d'un réseau CNN

Les réseaux de neurones convolutionnels connaissent actuellement un essor important dans la communauté de la vision car leur utilisation permet d'améliorer l'efficacité de nombreuses tâches liées à la vision (reconnaissance, classification, segmentation). Malheureusement, il existe une grande quantité de modèles possibles, différents sur le nombre de couches, le nombre de neurones par couche ou les fonctions non linéaires choisies. Chacun de ces modèles conduit à des résultats de classification différents. Plus généralement, il est très difficile de prouver qu'une configuration particulière est "optimale" pour un problème donné, ce qui laisse une grande part à l'expérimentation et montre l'intérêt des méthodes d'exploration automatique des réseaux.

Un point intéressant que l'on peut aborder sur ces systèmes concerne les récents travaux de Girshick [GDDM14], qui a remplacé la couche de classification par une SVM linéaire. Cette démarche revient à utiliser les premières couches du réseau comme un extracteur de descripteurs. Comme la qualité de classification d'une SVM est dépendante de la qualité du descripteur utilisé en entrée, les descripteurs générés par un réseau de neurones convolutionnels ont l'avantage d'être issus d'un algorithme d'apprentissage au sein duquel chaque couche converge théoriquement vers une erreur minimale, ce qui procure de meilleurs résultats qu'une machine SVM prise avec des descripteurs choisi arbitrairement.

Après avoir introduit les systèmes d'apprentissages supervisés qui seront abordés dans ces travaux, nous présentons les métriques statistiques utilisées pour l'évaluation des performances des systèmes de classification.

3.8 Évaluation des résultats de classification en ligne

Afin d'évaluer un système d'apprentissage, on applique la fonction de décision obtenue à un ensemble d'images d'entrées connues. Selon que cette image représente ou non la classe à identifier, on retrouve quatre possibilités que l'on peut regrouper dans la table de classiquement appelée *matrice de confusion*.

		Valeur attendue	
		Positif	Négatif
Valeur obtenue	Positif	Vrai positif (TP)	Faux Positif (FP)
	Négatif	Faux Négatif (FN)	Vrai Négatif (TN)

Tab. 3.3: Matrice de confusion

A partir de ces données, les métriques suivantes peut être calculées :

$$TPR \text{ (ou } Recall) = TP/P \tag{3.35}$$
$$FNR \text{ (ou } Miss\ Rate) = FN/P \tag{3.36}$$
$$FPR = FP/N \tag{3.37}$$
$$TNR = TN/N \tag{3.38}$$
$$\text{Précision} = \frac{TP + TN}{P + N} \tag{3.39}$$

où P et N correspondent respectivement au nombre total d'images positives et négatives testées. A partir de ces données, on peut utiliser différentes métriques pour l'évaluation des résultats comme les courbes Receiver Operating Characteristic (ROC) [Faw06] ou Detection Error Tradeoff (DET) [DT05]. Dans le cas où les images d'entrées ne sont pas des vignettes mais des images réelles sur lesquelles des détections sur de multiples sous-images sont possibles, la métrique d'évaluation change. On évalue alors le recouvrement entre la boîte englobante et la vérité terrain, respectivement noté BB_{dt} and BB_{gt}. Cette méthode proposée dans [DWSP12] considère une bonne détection si le recouvrement entre les deux boîtes est suffisant. En particulier dans le challenge PASCAL [EVGW+10], on attend un recouvrement de 50% minimum (Eq. 3.40). Chaque fenêtre de détection est donc évaluée par rapport à la vérité terrain. Plus le recouvrement entre les deux fenêtres sera élevé, plus la probabilité de bonne détection sera élevée. Si le recouvrement ne dépasse pas le seuil attendu, la détection sera considérée comme une fausse détection.

$$a_0 = \frac{Area(BB_{dt} \cap BB_{gt})}{Area(BB_{dt} \cup BB_{gt})} > 0.5 \tag{3.40}$$

Une autre métrique d'évaluation couramment rencontrée est la métrique Area Under Curve (AUC), déterminée par l'intégrale de la courbe ROC où le seuil de l'hyperplan normalisé varie de -1 à 1. Grâce à son invariance par rapport au seuil de l'hyperplan, cette métrique est utile pour estimer le gain du ratioTPR/FPR.

3.9 Conclusion

Nous avons introduit dans ce chapitre les notions de bases des systèmes d'apprentissages ainsi que les métriques d'évaluation associées, avec une attention particulière sur deux systèmes utilisés dans ces travaux, la machine à vecteurs supports (SVM) et les réseaux de

neurones convolutionnels (CNN). On utilisera ces notions dans les chapitres 4 et 5, qui traiteront respectivement de la détection de piétons à base de machines SVM et de la détection de caractères via un réseau de neurones convolutionnels CNN.

4

Reformulation flot de données et implantation : application aux algorithmes HOG-SVM

Ce chapitre a pour objectif démontrer l'intérêt de l'utilisation du modèle flot de données sur une application de traitement d'images, embarquée sur architecture FPGA. L'application ciblée, présentée au début de ce manuscrit, concerne la détection de piétons dans le champ de vision d'un véhicule afin de transmettre des alertes à d'autres véhicules. Comme expliqué précédemment (Sec. 2.5), le recours à un modèle de programmation adapté aux architectures matérielles, comme le modèle flot de données, permet de mettre en œuvre des outils HLS simplifiant la programmation des FPGAs tout en produisant un code HDL conduisant à de bonnes performances. Le chapitre sera articulé de la manière suivante : une formulation flot de données d'un extracteur de descripteurs HOG et d'un système de classification SVM est d'abord donnée. Puis, à partir de cette formulation, purement fonctionnelle et indépendante de tout outil HLS, une implémentation à l'aide du langage CAPH (Sec. 2.6) est proposée.

Le système de détection proposé, introduit au chapitre 3, se décompose en deux parties : un extracteur de descripteurs HOG et un système de classification SVM opérant avec une technique de fenêtrage glissant. La première partie transforme l'image d'entrée en un ensemble de descripteurs. Chaque descripteur HOG sera ici associé à une certaine zone de l'image, appelée **bloc** (voir Sec. 3.4 pour plus de détails). Afin de détecter l'ensemble des objets présents dans l'image, la scène est découpée en une collection de fenêtres de détection sur lesquelles seront évaluées les probabilités de présence de l'objet recherché [1]. Le vecteur d'entrée associé à chaque fenêtre de détection est alors composé de l'agrégation de tous les descripteurs HOG issus des blocs appartenant à la fenêtre (Sec 3.6). La SVM calcule le produit scalaire entre le vecteur d'entrée et le modèle issu de l'apprentissage supervisé afin de prendre une décision (parfois avec une probabilité associée) quant à la présence de l'objet recherché dans la fenêtre.

Au plus haut niveau, le système de détection étudié est décrit sous la forme d'un graphe flot de données sur la figure 4.1. Le flux de pixel acquis par la caméra est traité dans le module d'extraction de descripteurs qui génère un flux de descripteurs HOG. Afin de déterminer toutes les occurrences de l'objet cible dans l'image, la fenêtre de détection est déplacée dans l'image et pour chaque position, le système évalue la présence de l'objet ciblé. Les images entrent dans le graphe par la gauche en tant que flux de pixels et le système produit un flux d'images binaires où chaque bit correspond à la réponse pour la fenêtre de détection associée.

Comme expliqué dans la section 3.6, une telle approche multi-fenêtres n'est pas nouvelle dans le champ de recherche de la vision par ordinateur. Cependant, les coûts calculatoires qu'elle engendre rendent son implémentation compliquée en pratique, notamment sur des systèmes fortement contraints. La reformulation flot de données que nous allons donner de cette technique va nous permettre d'obtenir une implémentation performante de cette méthode dans laquelle le coût de calcul ne dépend plus du nombre de fenêtres à évaluer.

1. La taille de la fenêtre de détection dépend du type d'objets à détecter.

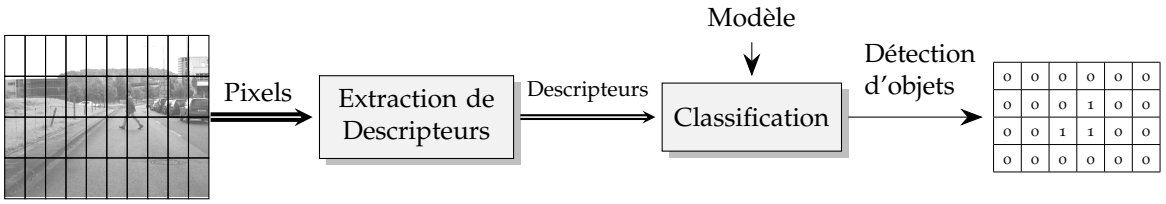


Fig. 4.1: Vue d'ensemble du système de détection d'objets flot de données

4.1 Algorithme

Le listing 4.1 présente un pseudo code de l'algorithme de détection complet suivant une formulation séquentielle. La première étape correspond à la détermination des descripteurs HOG (Sec. 3.4) et la seconde à l'évaluation de la fonction de classification pour chaque fenêtre de détection de l'image (Sec. 3.6).

```
1   // 1ere étape: Calcul des HOG sur l'image
2   // -------------------------------------
3   Pour chaque pixel (i,j) de l'image I
4       //Calcul des derivées en x et en y
5       gradx(i,j) = I(i,j+1) - I(i,j-1)
6       grady(i,j) = I(i+1,j) - I(i-1,j)
7
8       // Calcul de l'amplitude et de l'orientation du gradient au point i,j
9       magnitude = sqrt(gradx^2 + grady^2)
10      angle = arctan(grady(i,j), gradx(i,j))
11
12      // Déterminer l'entrée de l'histogramme correspondant à l'angle du gradient
13      bin = findbin(angle)
14
15      // Déterminer la position de la cellule à laquelle appartient le pixel courant
16      cell = getcell(i,j)
17
18      // Mise à jour l'histogramme correspondant
19      histogram (cell, bin) =   histogram(cell, bin) + magnitude
20  Fin
21
22  // Regroupement des cellules en blocs et Normalisation
23  Pour chaque cellule c de image I
24
25      // On récupère les histogrammes des cellules adjacentes
26      descriptor(c) = createblock(c, histogram)
27
28      // On normalise le descripteur
29      descriptor(c) = norm(descriptor(c), L2)
30  Fin
31
32  // -------------------------------------
33  // 2eme étape: classification SVM sur chaque fenêtre de l'image
34  // -------------------------------------
35  Pour toutes les fenetres w de l'image I
36      // on lit le descripteur de tous les blocs associes à la fenêtre en cours
37      finaldesc = gatherHOG(descriptor, w)
38
39      // on calcule le produit scalaire yij
40      y = svm(finaldesc, model)
41  Fin
```

Listing 4.1: Pseudo code de la formulation impérative du système de détection

La première étape calcule les descripteurs HOG à partir de l'amplitude et l'orientation des gradients calculés pour chaque pixel (première boucle) avec les opérations *gradx*, *grady*, *square root* et *arctan*. Ensuite, l'orientation calculée est discrétisée, comme illustré sur la figure 3.13 par la fonction *findbin*. L'étape suivante détermine à quelle cellule le pixel appartient (fonction *getcell*) afin d'incrémenter l'histogramme local correspondant.

La seconde boucle itère sur l'ensemble des cellules composant l'image. Pour chaque po-

sition, le descripteur associé à la cellule est créé en agrégeant les histogrammes voisins (2x2 cellules). Ensuite, le schéma de normalisation est appliqué à chaque descripteur indépendamment.

A l'issue de cette partie, l'ensemble des descripteurs HOG pour chaque bloc constituant l'image d'entrée est à disposition. L'étape de classification consiste alors à déplacer la fenêtre de détection et pour chaque position, à récupérer les descripteurs associés à la fenêtre courante (fonction *gatherHOG*). Une fois le final descripteur final constitué, il est évalué par un produit scalaire avec le modèle de l'objet souhaité afin de déterminer la présence ou non de l'objet dans la fenêtre de détection courante.

4.2 Formulation flot de données

Dans cette partie, on propose une formulation purement flot de données de l'algorithme décrit dans la partie précédente sous la forme d'un graphe d'acteurs. Les notations permettant d'expliciter à la fois le comportement des acteurs et le graphe lui-même seront d'abord présentées. Puis, l'algorithme d'extraction des descripteurs HOG (Sec. 4.2.2) et de classification par SVM (Sec. 4.2.3) seront reformulés successivement, avec le modèle ainsi défini.

4.2.1 Notations

Comme présenté au chapitre 2, l'expression d'un algorithme selon le modèle flot de données se fait sous la forme d'un réseau d'unités de calcul indépendantes (*acteurs*), échangeant des *jetons* par le biais de canaux unidirectionnels de type FIFO. Il paraît judicieux de donner d'abord, dans cette section, une formulation la plus générale possible, *c.à.d* indépendante de l'implémentation concrète qui sera présentée aux sections 4.5.1 et 4.5.2. Pour cela, on s'appuiera

- d'une part sur une notation *graphique* des réseaux,
- d'autre part sur une description purement *fonctionnelle* du comportement des acteurs de ce réseau [2].

La possibilité de donner une formulation purement fonctionnelle du comportement des acteurs est un point essentiel de la démarche. Elle découle de la transparence référentielle du modèle, au sein duquel le comportement de chaque acteur est défini indépendamment de tous les autres.

Dans notre cas, les flots de jetons traversant le réseau d'acteurs sont *structurés*. Cette structuration est réalisée en distinguant deux catégories de jetons : des jetons de *données* et des jetons de *contrôle*. Les premiers correspondent aux données proprement dites, les seconds aux "signaux" mais *vus ici comme de simples données particulières*, explicitant la structure de ces données. Considérons, par exemple l'image $n \times m$ de la Fig. 4.2.

$$\begin{array}{|cccc|}\hline p_{1,1} & p_{1,2} & \cdots & p_{1,m} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ p_{n,1} & p_{n,2} & \cdots & p_{n,m} \\ \hline \end{array}$$

Fig. 4.2: Image I correspondant au flux structuré p de l'équation défini par l'équation 4.1

Cette image peut être décrite par le flot de jetons suivant :

$$\langle\langle\, p_{1,1} \;\cdots\; p_{1,m} \,\rangle \cdots \langle p_{n,1} \;\cdots\; p_{n,m} \,\rangle\rangle \tag{4.1}$$

2. Le comportement des canaux est lui donné implicitement : c'est celui de canaux de type FIFO, de taille non bornée à ce niveau.

où "$\langle$" et "$\rangle$" sont deux jetons de contrôle qui peuvent être interprétés respectivement comme des signaux de début et de fin de liste (l'image étant vue alors comme une liste de listes). Loin d'être limitée aux seules images, cette représentation permet de décrire des structures de données arbitrairement complexes. Par exemple, un histogramme à n entrées peut se représenter comme une simple liste, délimité par les jetons "$\langle$" et "$\rangle$" :

$$\langle\, h_1 \ \cdots \ h_n \,\rangle \tag{4.2}$$

Une liste de points d'intérêts, où chaque point d'intérêt est constitué d'une valeur v et d'une paire de coordonnées x et y, pourrait être représentée par le flot suivant :

$$\langle\langle\, v_1 \langle\, x_1 \ y_1 \,\rangle\rangle \ \cdots \ \langle\, v_n \langle\, x_n \ y_n \,\rangle\rangle\rangle \tag{4.3}$$

Comme montré à la Sec 4.5.2, ce mode de structuration des flots de données se prête très bien à l'implémentation matérielle des acteurs. Mais, à ce niveau, son principal intérêt est d'autoriser une description purement fonctionnelle (applicative) du comportement des acteurs impliqués dans un algorithme. Considérons un acteur `double` dont le rôle est de multiplier par deux toutes les données d'une liste, représentée donc comme un flot structuré. Cet acteur peut être décrit par la relation suivante :

$$\texttt{double}(\langle\, x_1 \ \cdots \ x_n \,\rangle) = \langle\, 2 \times x_1 \ \cdots \ 2 \times x_n \,\rangle \tag{4.4}$$

ou de manière plus concise :

$$\texttt{double}(x_s) = \texttt{lift}(\texttt{mul2}, x_s) \tag{4.5}$$

où

- `lift` est la fonction d'ordre supérieur[3] définie par :

$$\texttt{lift}(f, \langle\, x_1 \ \cdots \ x_n \,\rangle) = \langle\, f(x_1) \ \cdots \ f(x_n) \,\rangle \tag{4.6}$$

- `mul2` la fonction qui multiplie par deux son argument :

$$\texttt{mul2}(x) = 2 \times x \tag{4.7}$$

Plusieurs variantes de la fonctionnelle `lift` peuvent être définies :

- les fonctionnelles $\texttt{lift}_2$, $\texttt{lift}_3$,...,$\texttt{lift}_m$ qui opèrent respectivement sur deux et trois, ..., m flux structurés parallèles :

$$\texttt{lift}_2(f, \langle\, x_1 \ \cdots \ x_n \,\rangle, \langle\, y_1 \ \cdots \ y_n \,\rangle) = \langle\, f(x_1, y_1) \ \cdots \ f(x_n, y_n) \,\rangle \tag{4.8}$$

$$\texttt{lift}_3(f, \langle\, x_1 \ \cdots \ x_n \,\rangle, \langle\, y_1 \ \cdots \ y_n \,\rangle, \langle\, z_1 \ \cdots \ z_n \,\rangle) = \langle\, f(x_1, y_1, z_1) \ \cdots \ f(x_n, y_n, z_n) \,\rangle \tag{4.9}$$

$$\texttt{lift}_m(f, \underbrace{\langle\, a_1 \ \cdots \ a_n \,\rangle, \ \cdots \ , \langle\, m_1 \ \cdots \ m_n \,\rangle}_{m \text{ flot parallèles}}) = \langle\, f(\underbrace{a_1, \ \cdots \ , m_1}_{m \text{ éléments}}) \ \cdots \ f(a_n, \cdots \ , m_n) \,\rangle \tag{4.10}$$

$$\tag{4.11}$$

- la fonctionnelle `mlift` qui opère sur un flot structuré à deux niveaux (une image typiquement) :

$$\texttt{mlift}(f, \langle\langle\, x_{1,1} \cdots x_{1,m} \,\rangle \cdots \langle\, x_{n,1} \cdots x_{n,m} \,\rangle\rangle) = \langle\langle\, f(x_{1,1}) \cdots f(x_{1,m}) \,\rangle \cdots \langle\, f(x_{n,1}) \cdots f(x_{n,m}) \,\rangle\rangle) \tag{4.12}$$

- les fonctionnelles $\texttt{mlift}_2$,..., $\texttt{mlift}_m$, qui sont des extensions de $\texttt{lift}_2$,$\texttt{lift}_m$ à un flot structuré à deux niveaux :

$$\begin{aligned}\texttt{mlift}_2(f, \langle\langle\, x_{1,1} \ \cdots \ x_{1,m} \,\rangle \cdots \langle\, x_{n,1} \ \cdots \ x_{n,m} \,\rangle\rangle, \langle\langle\, y_{1,1} \ \cdots \ y_{1,m} \,\rangle \cdots \langle\, y_{n,1} \ \cdots \ y_{n,m} \,\rangle\rangle) = \\ \langle\langle\, f(x_{1,1}, y_{1,1}) \ \cdots \ f(x_{1,m}, y_{1,m}) \,\rangle \cdots \langle\, f(x_{n,1}, y_{n,1}) \ \cdots \ f(x_{n,m}, y_{n,m}) \,\rangle\rangle\end{aligned} \tag{4.13}$$

3. En programmation fonctionnelle, une fonction d'ordre supérieur (*Higher Order Function*, HOF) est une fonction prenant une ou plusieurs autres fonctions en argument. Le terme de *fonctionnelle* est aussi utilisé.

4.2.2 Extraction de descripteurs HOG

La figure 4.3 présente la reformulation de la première étape de l'algorithme du listing 4.1 (le calcul du descripteur HOG) conformément au modèle flot de données. Dans cette figure, chaque boite grise correspond à un acteur et les flèches noires aux canaux de communications qui connectent les acteurs entre eux. Les images entrent en tant que flux structurés de pixels et les résultats sortent en tant que flux structurés de descripteurs, où chaque composante du descripteur est portée par un flux parallèle.

La formulation de cette partie sur le listing 4.1 est séquentielle. Le graphe de la Fig. 4.3 met par contraste en évidence le parallélisme de l'algorithme. L'approche choisie tire parti du fait que chaque entrée de l'histogramme constituant le descripteur peut être calculé indépendamment des autres (en parallèle), comme illustré sur 4.3. Pour ce faire, on réplique une chaîne d'acteurs spécifiques sur chaque entrée. Cette chaîne d'acteurs correspond à trois étapes, dédiées respectivement au calcul du gradient (acteurs `conv` et `argmax`), au calcul de l'histogramme (acteurs `hsum` et `vsum`) et à la normalisation du descripteur (acteurs `block`, `norm_factor` et `div`).

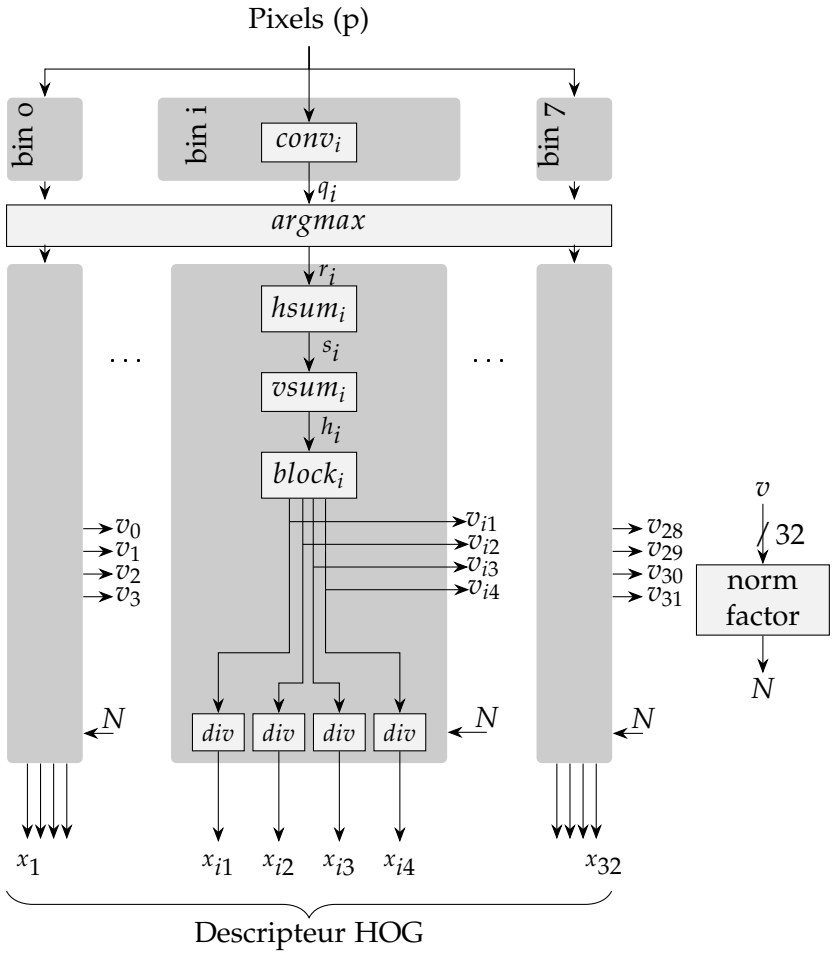


Fig. 4.3: Graphe flot de données de l'extraction de descripteur HOG

4.2.2.1 Extraction des gradients

Le calcul du gradient est une opération courante en traitement d'images mais son exécution sur des cibles matérielles n'est pas triviale car elle nécessite des opérations de racine carrée et d'arc tangente (lignes 9 et 10 du listing 4.1). Les méthodes courantes pour le calcul de la racine carrée utilisent un algorithme d'approximation comme l'algorithme de Newton-Raphson, utilisé par exemple dans [BKDB10, HSH+13]. Cependant, ces méthodes d'approximations sont itératives et le nombre d'itérations peut varier selon les opérandes. Appliqué

au modèle de calcul flot de données, cela conduit à des ratios production/consommation supérieurs à un voire variables. Cela pose alors des problèmes car chaque acteur du réseau est cadencé par l'arrivée des jetons sur ses entrées (comme introduit dans le chapitre 2). Une solution possible serait de *dérouler* la boucle d'itération de l'algorithme en un certain nombre d'étapes successives (ce qui est d'ailleurs proposé dans certains composants fournis par les fabricants de FPGA). Cette solution pose cependant un certain nombre de questions : comment choisir le nombre d'étages ? Quelle est l'erreur commise ? Pour contourner ces problèmes, certain auteurs approximent l'amplitude du gradient par la somme des valeurs absolues de chacune des composantes [4] [BM12]. Cependant, l'erreur sur le calcul de l'amplitude peut aller jusqu'à 45%. Une autre version proposée par Lee dans [LSCM12] appliquée au HOG ajoute un facteur correctif [5], ce qui réduit l'erreur commise mais cette erreur peut néanmoins atteindre 21% (comparatif disponible dans [MBB+15a]).

Considérons maintenant le calcul de l'arc tangente, deux solutions ont été proposées : utiliser des tableaux de correspondance (Look-up Tables (LUT)), comme décrit par Lee dans [LMS13] ou un algorithme d'approximation itératif comme le CORDIC [MTT+12]. La première approche requiert une quantité de mémoire importante, dépendante de la dynamique d'entrée et la précision souhaitée. La seconde approche a les mêmes inconvénients que les approches liées à l'élaboration de la racine carrée. Appliqué à l'algorithme du HOG, une troisième solution a été proposée par Bauer [BKDB10] et reprise par la suite par Hahnle [HSH+13]. Cette technique utilise le fait que les orientations des gradients vont être par la suite discrétisées par l'histogramme. L'information utile n'est donc pas réellement l'orientation exacte du gradient mais la correspondance de l'entrée de l'histogramme à laquelle cette orientation appartient. La stratégie proposée par Bauer ne calcule pas directement la valeur de l'arc tangente, mais compare itérativement les rapports des deux dérivées G_x et G_y avec des seuils pré-calculés (et enregistrés dans des LUTs). Cette méthode améliore en terme de ressources et de complexité les deux précédentes mais conserve les inconvénients sus-cités concernant le calcul de l'amplitude du gradient.

Nous proposons ici une méthode de calcul qui évalue à la fois l'amplitude et l'orientation du gradient de chaque pixel avec un coût linéaire, adaptée à un fonctionnement suivant le modèle flot de données. Cette méthode, appelée *HOG-Dot*, publiée dans [MBB+15a] pour la partie théorique et dans [MBB+15b] pour la partie implémentation, est une méthode spécialement conçue pour être optimale sur l'algorithme du HOG mais peut être également utilisée dans le cas général du calcul d'un gradient d'image.

La méthode part de la formule de la projection du gradient ∇I au point (x, y) sur un vecteur unitaire $\hat{i_k}$, avec pour relation :

$$\frac{\partial I}{\partial \hat{i_k}}(x,y) = \nabla I(x,y) \cdot \hat{i_k} \tag{4.14}$$

Dans le cas du HOG, l'espace des orientations est discrétisé en N_b échantillons équidistants sur l'intervalle $[0, \pi[$ [6]. Soit l'ensemble de vecteurs $S_{\hat{i}}$, illustré par la Fig. 4.4, et défini par :

$$S_{\hat{i}} = \left\{ \hat{i_k} \quad \middle| \, \theta_k = k\pi/N_b, \; k = 0, \ldots, N_b - 1 \right\} \tag{4.15}$$

avec

$$\hat{i_k} = \hat{x}\cos\theta_k + \hat{y}\sin\theta_k$$

Pour chaque vecteur $\hat{i_k}$ de $S_{\hat{i}}$, la projection du gradient ∇I au point (x, y) sur $\hat{i_k}$ est exprimée en remplaçant $\hat{i_k}$ dans l'Eq. 4.14 par son expression (Eq. 4.2.2.1), soit :

$$\frac{\partial I}{\partial \hat{i_k}}(x,y) = \cos\theta_k \frac{\partial I}{\partial x} + \sin\theta_k \frac{\partial I}{\partial y} \tag{4.16}$$

4. $G(x,y) = \sqrt{G_x(x,y)^2 + G_y(x,y)^2} \simeq |G_x| + |G_y|$
5. $G(x,y) \simeq |G_x| + \frac{|G_y|}{1+\sqrt{2}}$
6. Seul le demi-cercle supérieur est suffisant, voir Sec. 3.4 pour plus de détails

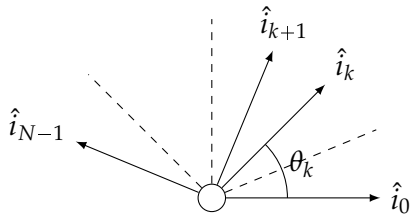


Fig. 4.4: Représentation spatiale de l'ensemble de vecteurs utilisé pour le HOG-Dot

Dans notre cas, les dérivées partielles sont exprimées par les différences des intensités des pixels voisins (Eq. 3.14 et 3.15), ce qui donne, appliqué à une image I :

$$\frac{\partial I}{\partial \hat{i}_k}(x,y) = \cos\theta_k \left[I(x+1,y) - I(x-1,y)\right] + \sin\theta_k \left[I(x,y+1) - I(x,y-1)\right] \tag{4.17}$$

Comme le montre l'Eq. 4.17, la $k^{ième}$ projection du gradient ∇I peut être simplement calculée comme la convolution de l'image d'entrée avec une noyau 3×3 qui contient les valeurs des coefficients constants $\cos\theta_k$ et $\sin\theta_k$. Il en résulte pour chaque $\hat{i}_k$, l'expression suivante :

$$\frac{\partial I}{\partial \hat{i}_k}(x,y) = \underbrace{\begin{pmatrix} 0 & -\sin\theta_k & 0 \\ -\cos\theta_k & 0 & \cos\theta_k \\ 0 & \sin\theta_k & 0 \end{pmatrix}}_{C_k} * \underbrace{\begin{pmatrix} I(x-1,y-1) & I(x,y-1) & I(x+1,y-1) \\ I(x-1,y) & I(x,y) & I(x+1,y) \\ I(x-1,y+1) & I(x,y+1) & I(x+1,y+1) \end{pmatrix}}_{I} \tag{4.18}$$

L'orientation approximée du gradient est déterminée en comparant toutes les amplitudes des projections $\frac{\partial I}{\partial \hat{i}_k}(x,y)$ sur chaque $\hat{i}_k$ et en gardant l'argument maximum, $\bar{k}$:

$$\bar{k} = argmax \left\{ \frac{\partial I}{\partial \hat{i}_k}(x,y) \mid k = 0, ..., N_b - 1 \right\} \tag{4.19}$$

L'amplitude du gradient est obtenue en prenant la valeur absolue du résultat de la convolution. Dans le cas du HOG, il est judicieux de choisir un nombre de vecteurs N_b égal au nombre d'entrées de l'histogramme, et la méthode produit respectivement l'amplitude et l'entrée de l'histogramme pour chaque pixel (x,y) de l'image. De plus, chaque convolution peut être calculée indépendamment, comme illustré par la Fig. 4.5.

L'utilisation de cette méthode présente plusieurs avantages :

- Précision de l'approximation. La méthode proposée permet de borner l'erreur commise sur le gradient en fonction du nombre de filtres parallèles choisi N_b. De plus, il a été démontré que dans le cas particulier du HOG [MBB+15a] cette méthode produit une erreur d'approximation du gradient équivalente à celle induite par la discrétisation de l'espace des orientations pour l'histogramme. Il en résulte que le recours aux calculs de la racine carrée et de l'arc tangente n'améliorerait en aucun cas la valeur produite par notre méthode.
- Parallélisme. Sur une architecture matérielle telle qu'un FPGA, chaque convolution peut se calculer en parallèle. Le calcul du maximum peut se faire avec une cascade de comparateurs. Cette technique produit un flux de données correspondant directement à chacune des composantes de l'histogramme situé en aval dans la chaîne de traitement.
- Ratio de production/consommation (comme défini à la section 2.4) des acteurs égal à un. La méthode proposée produit un jeton de sortie à chaque jeton d'entrée en une seule itération. Cette spécification permet lors de l'implémentation de faire fonctionner l'acteur exactement à la même cadence que le flot de pixels entrant.

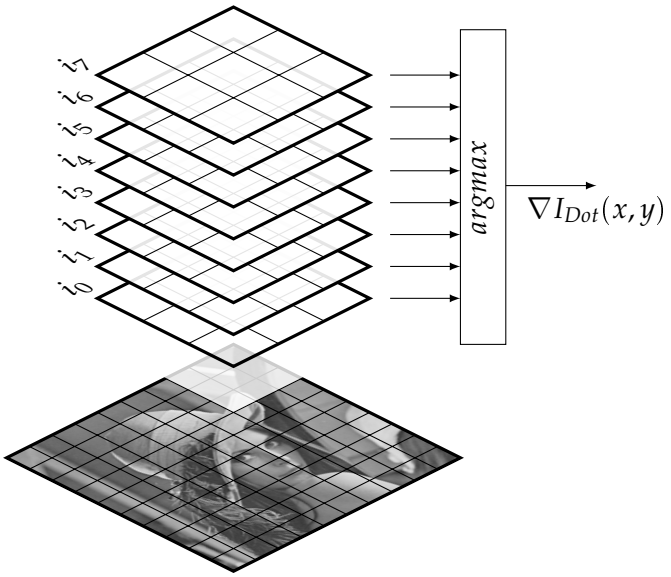


Fig. 4.5: Méthode parallèle d'extraction HOG-Dot

Les acteurs impliqués dans le graphe flot de données de la Fig. 4.3 vont maintenant être définis.

L'acteur $\texttt{conv}_k$ associé à la $k^{ième}$ orientation est défini par :

$$\texttt{conv}_k(\langle\langle\, p_{1,1}\ \ldots\ p_{1,m}\,\rangle\ \ldots\ \langle\, p_{n,1}\ \ldots\ p_{n,m}\,\rangle\rangle) = \langle\langle\, q_{k_{1,1}}\ \ldots\ q_{k_{1,m}}\,\rangle\ \ldots\ \langle\, q_{k_{n,1}}\ \ldots\ q_{k_{n,m}}\,\rangle\rangle \quad (4.20)$$

où

$$q_{k_{i,j}} = \begin{cases} 0, & \text{si } i \in \{1,n\} \vee j \in \{1,m\} \\ \cos\theta_k\, p_{i,j+1} - \cos\theta_k\, p_{i,j-1} + \sin\theta_k\, p_{i+1,j} - \sin\theta_k\, p_{i-1,j} & \text{sinon} \end{cases}$$

et

$$\theta_k = \frac{k\pi}{8},\ k \in \{0,\ldots,7\}$$

Chaque acteur $\texttt{conv}_k$ est connecté en entrée au même flux structuré p (Eq. 4.1) et applique un noyau C_k (Eq. 4.18) de dimension 3×3 différent (associé à la $k^{ième}$ orientation). Les bords de l'image où les convolutions ne peuvent pas être calculées sont mis à zéro. En sortie, l'acteur $\texttt{conv}_k$ produit un flux d'image de gradients projetés sur l'orientation associée (q_k). L'application des 8 acteurs `conv` en parallèle produit l'ensemble des composantes de l'histogramme. Il convient alors de déterminer quel canal d'entrée contient le gradient maximal. Pour cela, l'acteur `argmax` (représenté comme un seul acteur sur la Fig. 4.3 pour des raisons de lisibilité), peut se décomposer en un sous graphe d'acteurs plus élémentaires comme indiqué sur la Fig. 4.6.

Le premier acteur `abs` calcule l'amplitude de chaque vecteur projeté en prenant la valeur absolue de chaque élément.

$$\texttt{abs}(q) = \texttt{mlift}(fabs, q) \quad (4.21)$$

avec

$$fabs(x) = |x|$$

Ensuite, on retrouve un ensemble d'acteurs `comp` organisé en arbre binaire sur la Fig. 4.6. En cascadant les acteurs, on réduit la latence du système de N_b étapes à $\log_2(N_b)$ où est N_b est le nombre de valeurs à trier[7]. Chaque acteur `comp` est connecté à un n-uplet (q_1, q_2, k_1, k_2) en

7. Égale au nombre d'entrées de l'histogramme

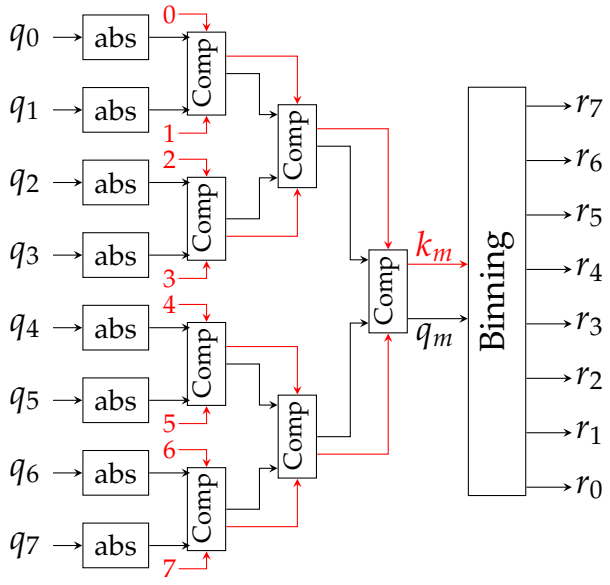


Fig. 4.6: Sous graphe d'acteurs pour la détermination de l'argmax

entrée où q_1, q_2 sont les amplitudes de deux projections du gradient et k_1, k_2 sont les indices associés aux flux q_1, q_2. En sortie, chaque acteur produit un couple $(q_m,\ k_m)$ où q_m est le maximum de q_1 et q_2, et k_m l'index de q_m. Plus formellement, l'acteur `comp` peut s'exprimer :

$$\texttt{comp}(q_1, q_2, k_1, k_2) = \texttt{mlift}_4(fcomp, q_1, q_2, k_1, k_2) = (q_m, k_m) \tag{4.22}$$

avec

$$fcomp(q_1, q_2, k_1, k_2) = (max(q_1, q_2), \quad k_1 \text{ si } q_1 \geqslant q_2 \text{ sinon } k_2)$$

Dans le cas particulier du premier étage, les flux k_1 et k_2 sont des flux de données constantes dont la valeur représente l'index du canal d'entrée q. Le dernier acteur `binning` recopie l'amplitude du gradient maximal sur la sortie correspondante et met les autres sorties à zéro.

$$\texttt{binning}(q_m, k_m) = \texttt{mlift}_2(fbinning, q_m, k_m) = (r_0, r_1, r_2, r_3, r_4, r_5, r_6, r_7) \tag{4.23}$$

avec

$$fbinning(q, k) = \{r_i = q \text{ si } i = k, 0 \text{ sinon} \mid \forall i \in \{0, ..., 7\}\}$$

On pourrait remettre en cause la présence de ce dernier acteur `binning` au motif que l'acteur en charge de l'histogramme pourrait être directement indexé par la sortie k_m. Cependant, deux raisons ont motivé le choix effectué. Premièrement, nous voulons maximiser le parallélisme de l'application et à ce titre, il est important de maintenir les N_b flux parallèles. D'autre part, ce choix va simplifier la formulation des calculs d'histogramme et éviter des problématiques d'implémentations matérielles sur la gestion du contrôle des histogrammes comme exposé dans [MSP+14].

4.2.2.2 Histogramme

Le calcul de l'histogramme est le goulot d'étranglement majeur dans l'élaboration des descripteurs HOG. C'est la raison pour laquelle les descripteurs ont rarement été implémentés sur des systèmes temps réels. Après la formulation originale de Dalal [DT05], Zhu a proposé dans [ZYCA06] d'améliorer le calcul de l'histogramme en appliquant le principe des histogrammes intégraux introduit par Porikli [Por05] pour accélérer le traitement. Cette technique permet l'évaluation rapide d'un histogramme dans une région indépendamment de la dimension et de la position de cette région par l'intersection des coordonnées de chaque extrémité de la région dans les histogrammes intégraux. La méthode de Porikli est composée

de deux étapes : la propagation des histogrammes intégraux et le calcul de l'intersection dans ces mêmes histogrammes intégraux pour déterminer l'histogramme d'une région spécifique.

Même si cette méthode a grandement amélioré les performances de calcul, elle est principalement adaptée aux architectures de processeur standard. L'étape de propagation requiert une mémorisation des histogrammes intermédiaires, équivalent à N_b images dans notre cas. La mémorisation de N_b images, ainsi que leur accès aléatoire en parallèle, même s'ils peuvent être autorisés par les modèles flot de données coûtera trop de ressources lors de la phase d'implémentation sur la cible FPGA. L'avantage de la technique de Porikli est qu'elle fournit le même temps d'élaboration quelque soit la taille de la zone sur laquelle l'histogramme est calculé. Cependant dans le HOG, il existe un paramètre supplémentaire qui va permettre de simplifier l'élaboration : la régularité de la grille sur laquelle sont déterminés les histogrammes.

L'approche que nous avons exploitée utilise la régularité du découpage de l'image en cellules (Fig. 3.13) afin de réduire le calcul de l'histogramme de chaque cellule à des accumulations en **parallèle** sur chaque classe (Fig. 4.7). Grâce aux N_b flux qui ont été maintenus en parallèle grâce à l'acteur `binning`, la détermination de chaque classe de l'histogramme sur une cellule peut se calculer par la simple accumulation des valeurs présentes sur la cellule. Comme les cellules sont adjacentes dans le HOG et que les données arrivent sur le flux, le calcul se résume à deux accumulateurs chaînés (acteur `xsum` et `vsum` sur la Fig. 4.3) par classe de l'histogramme, un pour la direction horizontale et un second pour la direction verticale.

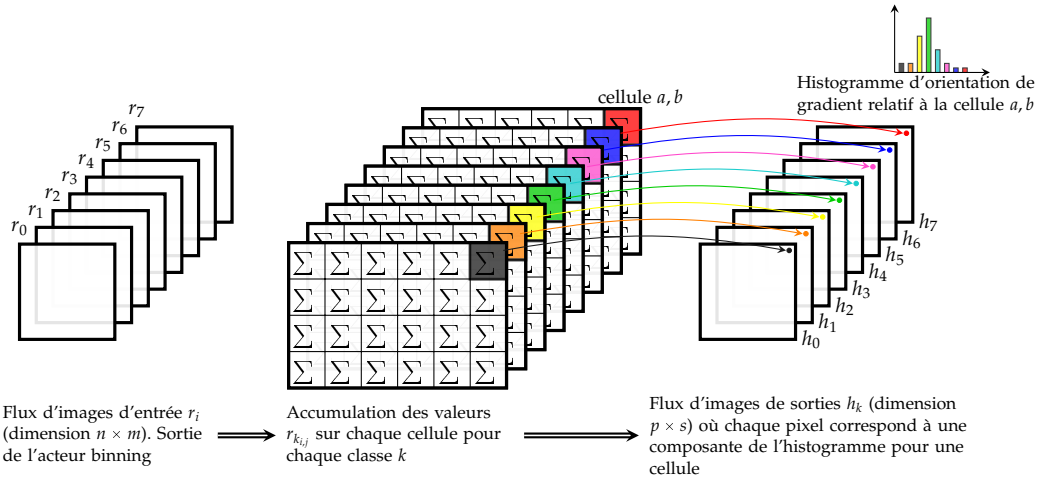


Fig. 4.7: Méthode proposée pour le calcul de l'histogramme

En terme de flux de données, l'étape d'histogramme produit en sortie N_b flux parallèles (h_k), où chaque flux porte une composante de l'histogramme. Chaque pixel d'une image h_k de la Fig. 4.7 correspond à une composante de l'histogramme pour une cellule donnée de l'image. En accédant aux N_b flux en parallèle pour une cellule donnée, l'histogramme d'orientation de gradient complet se rapportant à la cellule est obtenu.

Considérant une dimension de cellule de $c \times d$ pixels et une image d'entrée r_i de $n \times m$ pixels, où les dimensions de l'image sont divisibles par la taille de la cellule [8], les cellules de coordonnées (ii, jj) dans les images h_k sont données par $ii = \{1, 2, ..., \frac{n}{c}\}$ et $jj = \{1, 2, ..., \frac{m}{d}\}$ et les images h_k peuvent être exprimées par :

$$\forall (ii, jj) \in \mathbb{N}_+^2, \quad h_k(ii, jj) = \sum_{j=d\cdot(jj-1)+1}^{d\cdot jj} \sum_{i=c\cdot(ii-1)+1}^{c\cdot ii} r_{k_{i,j}} \tag{4.24}$$

Le calcul des histogrammes est assuré par deux acteurs `hsum` et `vsum`, chacun en charge de l'accumulation dans une direction. Le premier acteur `hsum` accumule les valeurs dans la

8. En supposant que $n = ck$ et $m = dk$ pour $k \in \mathbb{N}_+$

direction horizontale pour chaque classe. Sur chaque ligne de l'image, l'acteur accumule d valeurs consécutives et produit en sortie la somme ; soit, formellement :

$$\texttt{hsum}_d(\langle\langle \underbrace{r_{1,1} \ \cdots \ r_{1,m}}_{\text{1 ligne : } m \text{ éléments}} \rangle \cdots \langle r_{n,1} \ \cdots \ r_{n,m} \rangle\rangle) = \langle\langle \underbrace{x_{1,1} \ \cdots \ x_{1,s}}_{\text{1 ligne : } s \text{ éléments}} \rangle \cdots \langle x_{n,1} \ \cdots \ x_{n,s} \rangle\rangle \quad (4.25)$$

avec

$$s = m/d \quad \text{et} \quad x_{i,jj} = \sum_{j=d\cdot(jj-1)+1}^{d\cdot jj} r_{i,j}$$

L'acteur `hsum` réduit le nombre d'éléments du flux d'un facteur d par rapport au flux d'entrée comme il n'opère que sur les lignes des images r_i. L'acteur `vsum` effectue le même calcul mais sur la direction verticale. Sur chaque flux parallèle, l'acteur `vsum` est connecté à la sortie de l'acteur `hsum` (Fig. 4.3) et peut se formuler comme suit :

$$\texttt{vsum}_c(\langle\langle \underbrace{x_{1,1} \ \cdots \ x_{1,s} \rangle \cdots \langle x_{n,1} \ \cdots \ x_{n,s}}_{n \text{ lignes (de } s \text{ éléments)}} \rangle\rangle) = \langle\langle \underbrace{h_{1,1} \ \cdots \ h_{1,s} \rangle \cdots \langle h_{p,1} \ \cdots \ h_{p,s}}_{p \text{ lignes (de } s \text{ éléments)}} \rangle\rangle \quad (4.26)$$

avec

$$p = n/c \quad \text{et} \quad h_{ii,j} = \sum_{i=c\cdot(ii-1)+1}^{c\cdot ii} x_{i,j}$$

La composition des acteurs `hsum` et `vsum` donne la fonction d'histogramme :

$$\texttt{hist}(x) = \texttt{vsum}_c \circ \texttt{hsum}_d(x) \quad (4.27)$$

L'application de la fonction `hist` conserve une structure d'images (deux niveaux de délimiteurs) mais réduit le nombre de données dans les deux directions d'un facteur $c \times d$, dans notre cas 64 ($c = d = 8$). Si la cadence d'activation est commune à tous les acteurs du réseau (égale à la fréquence d'arrivée des pixels dans le graphe), cette réduction signifie que le nombre de données circulant sur les canaux de communication situés en aval de la fonction `hist` sera 64 fois moins important que la cadence possible des acteurs. Cette réduction de la dimension des flux de données pourra être exploitée lorsque l'on abordera certaines optimisations dans le chapitre 6, notamment concernant la sérialisation des données dans l'application.

4.2.2.3 Normalisation

Une fois les histogrammes h_k déterminés sur chaque cellule, la création du descripteur se fait par une étape de regroupement des cellules en blocs. Le schéma de regroupement standard [DT05, HSH+13, BKDB10, LSCM12] définit un bloc comme le regroupement de 2×2 cellules adjacentes (cf. Fig. 3.14). Avec ce schéma, et pour un histogramme à 8 classes, un descripteur HOG aura $4 \times 8 = 32$ composantes. Chaque descripteur, un par bloc, est ensuite normalisé afin de réduire la sensibilité aux variations d'illumination. Reformulé suivant le modèle flot de données, cette étape est certainement la plus complexe à appréhender car elle nécessite une façon de penser l'algorithme très différente de l'approche impérative. Considérons l'exemple d'une image d'histogramme h_k (Fig. 4.7) de dimension 4×4 cellules avec le schéma de regroupement standard, on obtiendra les 9 blocs illustrés sur la Fig. 4.8. Chaque cellule a été numérotée dans l'image h_k de gauche et chaque bloc obtenu est décrit en fonction du numéro des cellules qui le composent.

Chaque composante du voisinage 2×2 prise sur l'ensemble des blocs de l'image est en fait un fragment de l'image originale. Par exemple, la première composante des 9 blocs générés correspond aux cellules $1, 2, 3, 5, 6, 7, 9, 10, 11$ (surligné sur la partie droite de la Fig. 4.8),

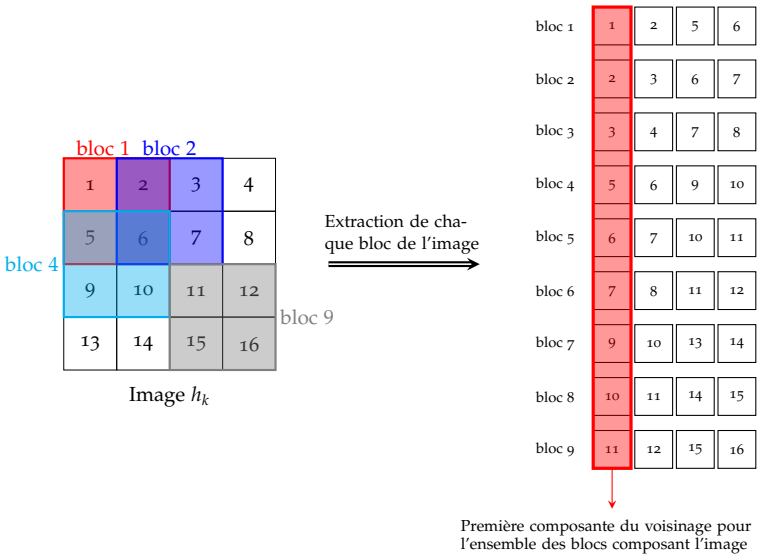


Fig. 4.8: Exemple d'extraction de voisinage

c'est-à-dire à l'image d'origine h_k sans la dernière colonne ni la dernière ligne de cellules. La seconde composante (cellules $2, 3, 4, 6, 7, 8, 10, 11, 12$) correspond à l'image h_k sans la première colonne ni dans la dernière ligne. La troisième composante (cellules $5, 6, 7, 9, 10, 11, 13, 14, 15$) correspond à l'image h_k sans la première ligne ni la dernière colonne et la dernière composante (cellules $6, 7, 8, 10, 11, 12, 14, 15, 16$) correspond à l'image h_k sans la première ligne ni la première colonne. D'une manière générale, à chaque flux d'image h_k (Fig. 4.7) peut être associé quatre nouveaux flux d'images, notés $(v_{4k}, v_{4k+1}, v_{4k+2}, v_{4k+3})$, chacun représentant une composante du voisinage. L'obtention des images v_i se fait par recopie partielle des images h_k en ne conservant pour chaque composante que la partie correspondante, comme illustré sur la Fig. 4.9.

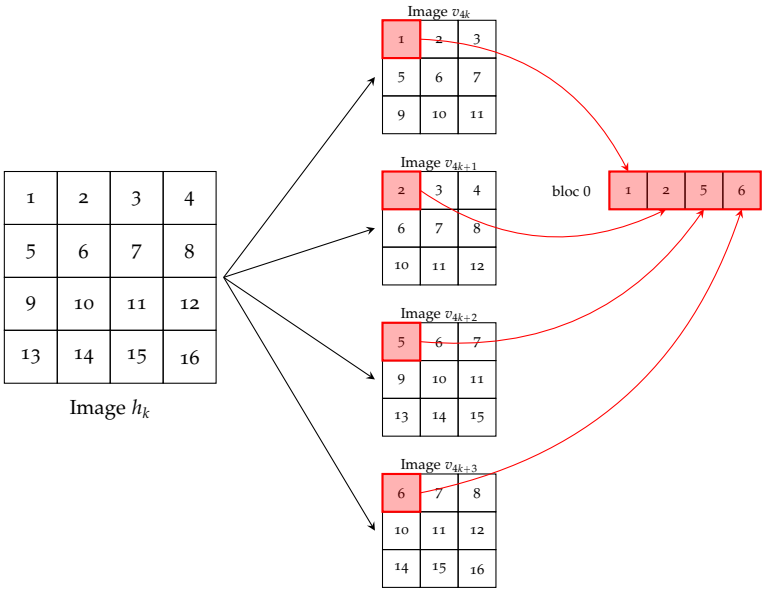


Fig. 4.9: Extraction de voisinage flot de données

Cette approche requiert la mémorisation de la première ligne de cellules ainsi que la première cellule de la seconde ligne sur chaque h_k. Sur l'exemple de la Fig. 4.9, le premier bloc à produire est composé des cellules $1, 2, 5, 6$, et l'acteur doit donc attendre que la cellule 6 de l'image h_k soit disponible en entrée afin de pouvoir fournir le voisinage complet en sortie sur les quatre flux (v_i).

En résumé, l'acteur `block` réalisant l'extraction du voisinage 2×2 pour une image h_k est

défini par :

$$\texttt{block}(h_k) = (v_{4k}, v_{4k+1}, v_{4k+2}, v_{4k+3}) \tag{4.28}$$

avec

$$\begin{aligned}
h_k &= \langle\langle h_{k_{1,1}} \cdots h_{k_{1,s}} \rangle \cdots \langle h_{k_{p,1}} \cdots h_{k_{p,s}} \rangle\rangle \\
v_{4k} &= \langle\langle h_{k_{1,1}} \cdots h_{k_{1,s-1}} \rangle \cdots \langle h_{k_{p-1,1}} \cdots h_{k_{p-1,s-1}} \rangle\rangle \\
v_{4k+1} &= \langle\langle h_{k_{1,2}} \cdots h_{k_{1,s}} \rangle \cdots \langle h_{k_{p-1,2}} \cdots h_{k_{p-1,s}} \rangle\rangle \\
v_{4k+2} &= \langle\langle h_{k_{2,1}} \cdots h_{k_{2,s-1}} \rangle \cdots \langle h_{k_{p,1}} \cdots h_{k_{p,s-1}} \rangle\rangle \\
v_{4k+3} &= \langle\langle h_{k_{2,2}} \cdots h_{k_{2,s}} \rangle \cdots \langle h_{k_{p,2}} \cdots h_{k_{p,s}} \rangle\rangle
\end{aligned}$$

En appliquant à chaque image d'histogramme h_k (*i.e.* chaque composante de l'histogramme), on obtient finalement la totalité du descripteur HOG en parallèle $(v_0, ..., v_{31})$. Les 32 images v_i (*i.e.* flux de blocs) générées sont de dimension inférieure aux images h_k d'une ligne et d'une colonne. Afin de simplifier les notations pour la suite des acteurs, on exprimera les dimensions des images v_i en fonction de nouveaux indices α, β obtenus à partir des anciennes dimensions p, s des images histogrammes h_k :

$$v_i = \langle\langle v_{i_{1,1}} \cdots v_{i_{1,\beta}} \rangle \cdots \langle v_{i_{\alpha,1}} \cdots v_{i_{\alpha,\beta}} \rangle\rangle, \text{ avec } \alpha = p-1 \text{ et } \beta = s-1 \tag{4.29}$$

Comme indiqué au paragraphe 3.5, cette étape est indispensable au bon fonctionnement du système d'apprentissage. Il a été montré par ailleurs dans [DT05] que le choix du schéma de norme avait un impact sur la qualité de l'espace de description. La table 4.1 rapporte les différentes normes possibles ainsi que leur impact sur la qualité de détection et le type d'opérations requises. Le schéma le plus efficace est la normalisation de type L2Hys. Ce schéma, typique du HOG, requiert premièrement une étape de normalisation, puis de seuillage des valeurs normalisées (typiquement fixé à 0.2) puis une seconde étape de normalisation. Le calcul de chaque facteur de normalisation requiert une opération racine carré, une somme et une multiplication, faisant resurgir les problématiques évoquées lors de la formulation de l'extraction des gradients concernant les méthodes d'approximation de l'opération de racine carré. Bien qu'il soit envisageable de procéder à une formulation d'un schéma aussi complexe, les autres schémas étudiés par Dalal [DT05] offrent des performances similaires (bien que légèrement moins bonne) mais impliquent des calculs nettement moins lourds.

Norme	L2Hys	L2	L1Sqrt	L1	Aucune
Formule	$x_i = \frac{x_i}{\sqrt{\|\|x_i\|\|_2^2+\epsilon^2}}$ avec $x_i <= 0.2$	$x_i = \frac{x_i}{\sqrt{\|\|x_i\|\|_2^2+\epsilon^2}}$	$x_i = \sqrt{\frac{x_i}{\|\|x_i\|\|_1+\epsilon}}$	$x_i = \frac{x_i}{\|\|x_i\|\|_1+\epsilon}$	$x_i = x_i$
Qualité de Détection (TPR à 10^{-4} FFPW) [DT05]	0.90	0.90	0.90	0.85	0.63
Opérations arithmétiques	64 additions 2 multiplications 2 racines carrée 2 divisions 1 seuillage	32 additions 1 multiplication 1 racine carrée 1 division	32 additions 1 division 1 racine carrée	32 additions 1 division	

Tab. 4.1: Comparatif des schémas de normalisation

Parmi les schémas existants, la normalisation L1 a été choisie. Ce schéma offre en effet un bon compromis entre simplicité des calculs et qualité de la détection (perte de cinq pour cent de bonnes détections par rapport au meilleur schéma possible, pour un taux de faux positifs

de 10^{-4} FFPW). Sur la Fig. 4.3, cette étape de normalisation est assurée par deux acteurs : `norm_factor` et `div`. Le premier calcule le facteur de normalisation :

$$\texttt{norm\_factor}(v_0,...,v_{31}) = \texttt{mlift}_{32}(fnorm, v_0,...,v_{31}) = \langle\langle N_{1,1} \ \cdots \ N_{1,\beta}\rangle \cdots \langle N_{\alpha,1} \ \cdots \ N_{\alpha,\beta}\rangle\rangle \tag{4.30}$$

avec

$$fnorm(v_0,...,v_{31}) = \sum_{k=0}^{31} v_{k_{i,j}} = ||v_{i,j}||$$

Le second `div` normalise chaque composante du descripteur en la divisant par le facteur N ainsi calculé :

$$\texttt{div}(v_k, N) = \texttt{mlift}_2(fl1, v_k, N) = \langle\langle x_{k_{1,1}} \ \cdots \ x_{k_{1,\beta}}\rangle \cdots \langle x_{k_{\alpha,1}} \ \cdots \ x_{k_{\alpha,\beta}}\rangle\rangle \tag{4.31}$$

avec

$$fl1(v_k, N) = \frac{v_k}{N}$$

Si on note $\mathbf{x_{b_{k,l}}}$ le descripteur HOG, associé à un bloc quelconque k, l de l'image, soit :

$$\mathbf{x_{b_{k,l}}} = \begin{pmatrix} x_{0_{k,l}} \\ x_{1_{k,l}} \\ \cdots \\ x_{31_{k,l}} \end{pmatrix} \tag{4.32}$$

Alors on notera $\mathbf{x_h}$ l'**ensemble** des descripteurs x_i disponible à la fin du processus d'extraction des descripteurs HOG pour une image complète, soit :

$$\mathbf{x_h} = \begin{pmatrix} x_0 \\ x_1 \\ \cdots \\ x_{31} \end{pmatrix} = \begin{pmatrix} \langle\langle x_{0_{1,1}} \ \cdots \ x_{0_{1,\beta}}\rangle \cdots \langle x_{0_{\alpha,1}} \ \cdots \ x_{0_{\alpha,\beta}}\rangle\rangle \\ \langle\langle x_{1_{1,1}} \ \cdots \ x_{1_{1,\beta}}\rangle \cdots \langle x_{1_{\alpha,1}} \ \cdots \ x_{1_{\alpha,\beta}}\rangle\rangle \\ \cdots \\ \langle\langle x_{31_{1,1}} \ \cdots \ x_{31_{1,\beta}}\rangle \cdots \langle x_{31_{\alpha,1}} \ \cdots \ x_{31_{\alpha,\beta}}\rangle\rangle \end{pmatrix} \tag{4.33}$$

Quelques remarques sur la formulation proposée pour cette partie.

1. Tout d'abord, avec cette formulation, l'algorithme consomme en entrée un flux de pixels et produit en sortie un flux de descripteurs HOG et ceci indépendamment de la taille de l'image d'entrée. La seule contrainte est que l'image soit un multiple de la dimension des cellules sur lesquelles sont calculés les histogrammes. Il a été volontairement fait en sorte de garder l'extraction des descripteurs HOG indépendante du système de classification en aval afin de permettre l'utilisation de différents systèmes où chacun pourrait être connecté au même flux de données. On peut imaginer non pas un seul mais un ensemble de systèmes de classification où chacun exploite les mêmes flux de descripteurs mais paramétrés différemment (taille de la fenêtre de détection et modèle), offrant la possibilité de détecter plusieurs objets en parallèle sur chaque image (piétons, voitures, motos, mandolines,...).
2. Chaque composante du descripteur HOG étant portée par un canal de communication séparé, le parallélisme de l'application est maximal. Par la suite, il sera facile d'accéder à l'ensemble des composantes de chaque descripteur de manière parallèle et synchrone (toutes les composantes sont disponibles au même instant).
3. La structure de chaque flux de données de sortie x_i correspond à une image dans laquelle chaque pixel correspond à une composante d'un descripteur. La position du pixel dans l'image est liée à la position du descripteur dans l'image originale. Cette représentation est primordiale car l'information de position du descripteur va être nécessaire pour le regroupement des descripteurs au sein d'une même fenêtre de détection.

4. La dimension de chaque image de sortie est réduite par rapport à l'image d'entrée. Avec les notations utilisées dans cette section, l'image sera de dimension $\alpha = n/c - 1$ et $\beta = m/d - 1$. Dans notre cas, avec une image d'entrée de résolution 1280×1024 pixels et une cellule 8×8, chaque x_i sera de dimension 159×127. Même en considérant l'ensemble des 32 flux parallèles, la quantité de donnée est tout de même réduite d'un facteur $2,03$ (facteur $64,9$ par flux de données).

4.2.3 Classification par SVM Linéaire

La section précédente a proposé une formulation flot de données de l'extraction des descripteurs HOG pour une image. Toutefois, le système présenté produit un flot de descripteurs au sein duquel chaque descripteur est associé à un bloc de l'image, alors que le système de classification visé opère sur une **fenêtre**. La détermination du descripteur complet associé à une fenêtre se fait par l'agrégation de l'ensemble des descripteurs des blocs compris dans cette fenêtre.

Afin d'évaluer le résultat de la détection sur une fenêtre, il faut calculer le produit scalaire du descripteur complet associé à cette fenêtre avec le modèle issu de la phase d'apprentissage. Pour cela, la linéarité de l'équation 3.30 est utilisée pour séparer le calcul en produit scalaires partiels comme suit :

$$y(\mathbf{x}) = signe(\mathbf{w}^T \cdot \mathbf{x} + b) = signe\Big(\sum_{i=1}^{n_y}\sum_{j=1}^{n_x}({\mathbf{w}_{\mathbf{b}_{i,j}}}^T \cdot \mathbf{x}_{\mathbf{b}_{i,j}}) + b\Big) \tag{4.34}$$

où n_x et n_y correspondent aux dimensions de la fenêtre de détection, exprimés respectivement en nombre de blocs dans le sens horizontal et vertical, $\mathbf{x}_{\mathbf{b}_{i,j}}$ est le descripteur à la position i, j au sein de la **fenêtre considérée**[9] et $\mathbf{w}_{\mathbf{b}_{i,j}}$ est le vecteur partiel du modèle associé au descripteur $\mathbf{x}_{\mathbf{b}_{i,j}}$.

Dans le cas de la formulation séquentielle (Listing. 4.1), chaque fenêtre potentielle de l'image est évaluée, avec un pas de recouvrement entre deux fenêtres égal à un bloc. Le passage d'un ensemble de blocs à un ensemble de fenêtres recouvrantes se fait par de simples lectures des descripteurs $\mathbf{x}_{\mathbf{b}_{i,j}}$ contenus en mémoire. Au sein du modèle flot de données, cette opération d'agrégation des descripteurs est un exemple typique d'opération délicate à formuler. Le problème se pose en ces termes : pour chaque descripteur d'entrée $\mathbf{x}_{\mathbf{b}_{i,j}}$ qui arrive en entrée, à quelle(s) fenêtre()s ce descripteur appartient-il et avec quelle(s) partie(s) du modèle doit-il être multiplié ? La réponse est que chaque descripteur $\mathbf{x}_{\mathbf{b}_{i,j}}$ (de dimension 32) appartient à $n_x \times n_y$ fenêtres potentielles dans lesquelles le descripteur sera multiplié avec des parties différentes du modèle (car le bloc occupe une position différente dans chaque fenêtre).

Afin de simplifier la mise en œuvre de ce schéma, un schéma de fenêtrage sans recouvrement est proposé. Avec cette approche (illustrée sur la Fig. 4.10), chaque descripteur $\mathbf{x}_{\mathbf{b}_{i,j}}$ ne peut appartenir qu'à une seule fenêtre de détection. Cette spécification permet de calculer les produits scalaires partiels sur le flux, sans se soucier de l'appartenance du descripteur $\mathbf{x}_{\mathbf{b}_{i,j}}$ entrant à un ensemble de fenêtres de détection.

9. A ne pas confondre avec la position du descripteur dans l'image complète de l'Eq. 4.32. Les deux sont égaux à la seule condition qu'il n'y ait qu'une fenêtre de détection dans toute l'image d'entrée

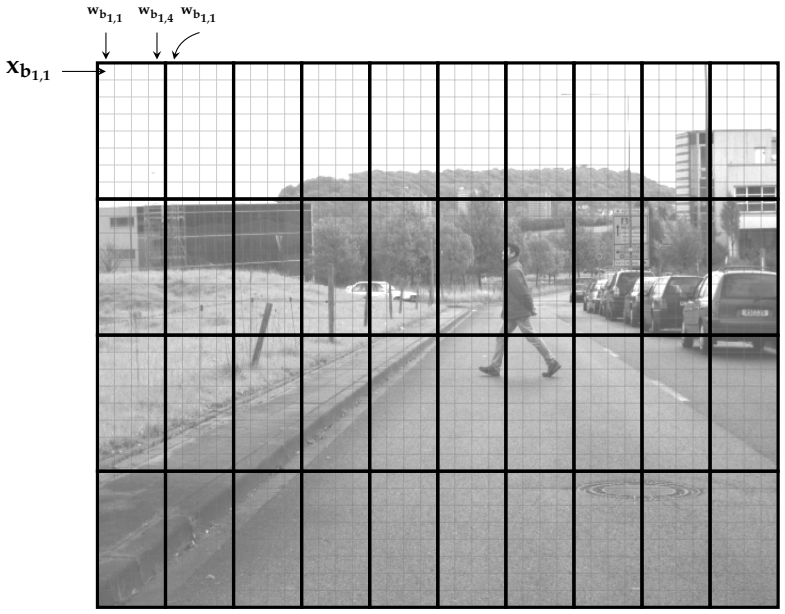


Fig. 4.10: Illustration de la fenêtre de détection glissante sans recouvrement. Les fenêtres de détection sont surlignées en gras. Chaque descripteur HOG $\mathbf{x}_{\mathbf{b}_{i,j}}$ (représenté par un carré gris) n'appartient qu'à une seule fenêtre de détection, ce qui permet de simplifier le graphe flot de données du système de classification. Chaque poids partiel $\mathbf{w}_{\mathbf{b}_{i,j}}$ qui pondère le vecteur $\mathbf{x}_{\mathbf{b}_{i,j}}$ dépend de la position du bloc $\mathbf{x}_{\mathbf{b}_{i,j}}$ dans la fenêtre globale. Sur cet exemple est dessiné une fenêtre de détection de dimension de 4×8 blocs pour des raisons de lisibilité. Dans le cas pratique de la détection de piéton, cette fenêtre sera de dimension 7×15 blocs.

Cette approche, même si elle présente des inconvénients notamment en termes de qualité de détection en conditions réelles, autorise une première formulation flot de données du système qui constituera la brique de base d'un système complet flot de données gérant les recouvrements, la gestion des recouvrements se faisant par simple mise en parallèle de 105 briques de base ($n_x \times n_y$ dans notre cas).

Le graphe flot de données décrivant le fonctionnement de cette brique de base est donné, Fig. 4.11.

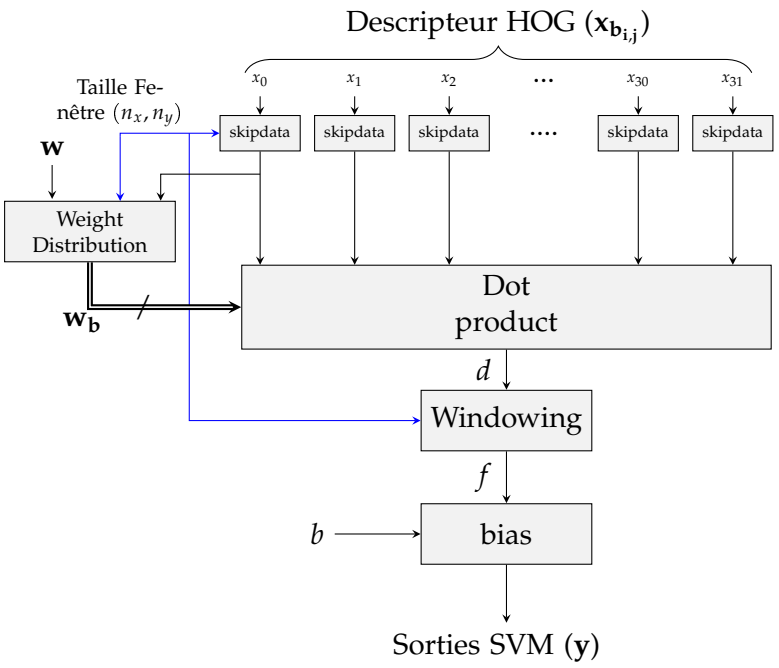


Fig. 4.11: Graphe flot de données du système de classification

Le premier acteur, appliqué à chaque composante du descripteur est l'acteur `Skipdata`. Cet acteur a pour seul rôle de retirer certaines données du flux de descripteurs, données qui peuvent être vues comme des artefacts liés au mécanisme de fenêtrage.

Le rôle de cet acteur est illustré sur la Fig. 4.12. Sur cet exemple, chaque rectangle représente une **cellule** (entrée de l'acteur `block`) et chaque fenêtre de détection (surlignée en gras) est composée de 8×16 cellules (soit à 7×15 blocs). Pour chaque cellule de l'image, l'acteur `block` génère un bloc correspondant. Le flot de blocs issu de l'acteur `block` comprend alors des blocs qui se trouvent à l'intersection de deux fenêtres de détection (blocs striés sur la Fig. 4.12). Ce sont ces blocs qui sont supprimés par l'acteur `skipdata`.

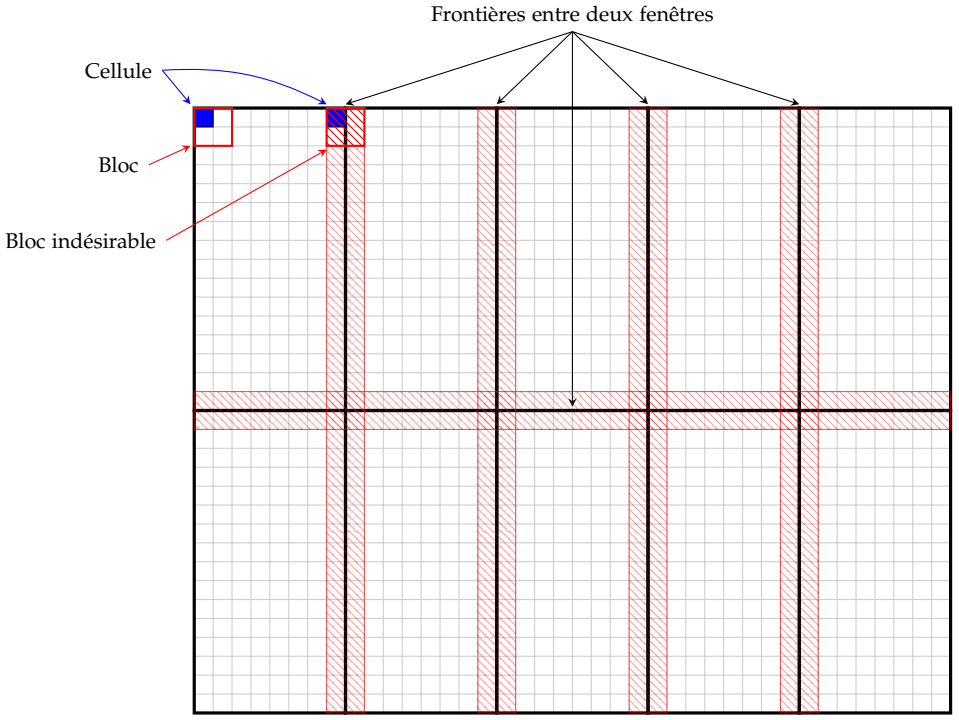


Fig. 4.12: Fonctionnement de l'acteur `skipdata`

Fonctionnellement, l''acteur `skipdata` se défini par :

$$\texttt{skipdata}_{n,m}(\langle\langle\, x_{k_{1,1}} \;\cdots\; x_{k_{1,\beta}}\,\rangle \ldots \langle\, x_{k_{\alpha,1}} \;\cdots\; {}_{k_{\alpha,\beta}}\,\rangle\rangle) = \langle\langle\, xs_{k_{1,1}} \;\cdots\; xs_{k_{1,\mu}}\,\rangle \cdots \langle\, xs_{k_{\lambda,1}}, \cdots, xs_{k_{\lambda,\mu}}\,\rangle\rangle \tag{4.35}$$

où

$$xs_{k_{i,j}} = x_{k_{i,j}} \quad \text{si } i \bmod (n_y+1) \neq 0 \vee j \bmod (n_x+1) \neq 0$$

Le flux sortant de l'acteur `SkipData` est de dimension (λ, μ), avec

$$\lambda = \alpha - \left(\frac{\alpha+1}{n_y+1} - 1\right) \text{ et } \mu = \beta - \left(\frac{\beta+1}{n_x+1} - 1\right) \tag{4.36}$$

Afin de ne pas introduire une notation supplémentaire, on continue d'utiliser la notation $(\mathbf{x_b})$ pour le flux de descripteur mais celle-ci référera au nouveau flux de descripteur qui vient d'être créé.

L'acteur central du graphe (Fig. 4.11) est l'acteur `dot_product` qui calcule les produits scalaires partiels de chaque descripteur $(\mathbf{x_b})$ avec le poids associé $(\mathbf{w_b})$. L'acteur `dot_product` est connecté à 64 flux d'entrées, 32 pour le descripteur $\mathbf{x_b}$ et 32 pour les poids associés $\mathbf{w_b}$ et produit en sortie un seul flux portant les produits scalaires partiels.

$$\texttt{dot\_product}(\mathbf{x_b}, \mathbf{w_b}) = \texttt{mlift}_{64}(fdot, \mathbf{x_b}, \mathbf{w_b}) = \langle\langle\, d_{1,1} \;\cdots\; d_{1,\mu}\,\rangle \cdots \langle\, d_{\lambda,1} \cdots d_{\lambda,\mu}\,\rangle\rangle \tag{4.37}$$

avec

$$fdot(\mathbf{x_b}, \mathbf{w_b}) = \mathbf{x_b} \cdot \mathbf{w_b} = \sum_{k=0}^{31} x_k \cdot w_k$$

La difficulté réside ici dans la distribution du bon poids au bon instant en fonction du fenêtrage. Cette tâche est effectuée par un second acteur, `weight_distribution`. Le vecteur de poids $\mathbf{w}$ déterminé par l'apprentissage est liée à une unique fenêtre de détection et chaque poids partiel $\mathbf{w_b}$ dépend de sa position dans la fenêtre. Les poids au sein d'une même fenêtre ne sont donc pas distribués de manière continue. Par exemple, le vecteur $\mathbf{x_{b_{1,1}}}$ est multiplié par le poids $\mathbf{w_{b_{1,1}}}$, le vecteur $\mathbf{x_{b_{1,4}}}$ par le poids $\mathbf{w_{b_{1,4}}}$ tandis que le vecteur $\mathbf{x_{b_{1,5}}}$ est de nouveau multiplié par le $\mathbf{w_{b_{1,1}}}$. Ce comportement oblige l'acteur de distribution à prendre en compte à la fois la dimension de la fenêtre de détection n_x, n_y et la dimension du flux de descripteur pour gérer le ré-indexage des poids. Comme la dimension du flux de descripteurs est à priori inconnue, l'acteur `weight_distribution` doit être piloté par une composante du vecteur $\mathbf{x_b}$ qui donne la structure du flux correspondant (et indirectement les dimensions).

Le vecteur poids global $\mathbf{W}$ est défini par un tableau à trois dimensions dont les deux premières correspondent à la position du descripteur dans la fenêtre de détection. La dernière dimension correspond à la profondeur du descripteur HOG. En considérant une fenêtre de détection composée de $n_x \times n_y$ blocs, chaque descripteur étant de dimension d, le tableau $\mathbf{W}$ sera alors de la forme : $\mathbf{W}[n_x][n_y][d]$. Cette structuration en trois dimensions permet d'exprimer plus simplement la relation entre les index du flux d'activation x_0 et des flux de sorties $\mathbf{w_b}$. En effet, chaque index peut s'exprimer en fonction de l'indice courant du jeton d'entrée $x_{0_{i,j}}$ et des dimensions de la fenêtre n_x, n_y. De plus, la même notation vectorielle que celle précédemment est utilisée pour les descripteurs $\mathbf{x_b}$, $\mathbf{w_{b_{i,j}}} = \mathbf{W}[i][j]$ correspond à l'ensemble des 32 composantes du vecteur poids pour un descripteur donné.

Partant de là, l'acteur `weight_distribution` produit les poids de SVM $\mathbf{w_b}$ en fonction des coordonnées du descripteur d'entrée, de manière synchrone à l'entrée x_0 et en fonction des dimensions de la fenêtre de détection n_x et n_y.

Fonctionnellement, l'acteur `weight_distribution` s'exprime comme suit :

$$\texttt{weight\_distribution}_{\mathbf{W},n_x,n_y}(x_0) = \mathbf{w_b} \tag{4.38}$$

avec

$$x_0 = \langle\langle x_{0_{1,1}} \ \cdots \ x_{0_{1,\mu}} \rangle \cdots \langle x_{0_{\lambda,1}} \ \cdots \ x_{0_{\lambda,\mu}} \rangle\rangle$$

$$\mathbf{w_b} = \begin{pmatrix} \langle\langle w_{0_{1,1}} \ \cdots \ w_{0_{1,\mu}} \rangle \cdots \langle w_{0_{\lambda,1}} \cdots \ w_{0_{\lambda,\mu}} \rangle\rangle \\ \langle\langle w_{1_{1,1}} \ \cdots \ w_{1_{1,\mu}} \rangle \cdots \langle w_{1_{\lambda,1}} \cdots \ w_{1_{\lambda,\mu}} \rangle\rangle \\ \cdots \\ \langle\langle w_{31_{1,1}} \ \cdots \ w_{31_{1,\mu}} \rangle \cdots \langle w_{31_{\lambda,1}} \ \cdots \ w_{31_{\lambda,\mu}} \rangle\rangle \end{pmatrix}$$

$$w_{k_{i,j}} = \mathbf{W}[ii][jj][k], \quad ii = i \bmod n_x, \quad jj = j \bmod n_y$$

Une fois les produits scalaires partiels calculés, il suffit de les regrouper en fonction des fenêtres pour obtenir le résultat final. Avec le schéma de fenêtrage sans recouvrement (Fig. 4.10), cette opération est équivalente à l'opération effectuée par l'acteur d'histogramme (Eq. 4.24), à savoir une accumulation des valeurs sur une grille rectangulaire.

$$\forall (ii, jj) \in \mathbb{N}_+^2, \quad f(ii, jj) = \sum_{j=n_y \cdot (jj-1)+1}^{n_y \cdot jj} \ \sum_{i=n_x \cdot (ii-1)+1}^{n_x \cdot ii} d_{i,j} \tag{4.39}$$

L'acteur `windowing` est défini comme la composition des acteurs `vsum` et `hsum` :

$$\texttt{windowing}_{n_x,n_y}(d) = \texttt{vsum}_{n_x} \circ \texttt{hsum}_{n_y}(d) = f \tag{4.40}$$

avec

$$d = \langle\langle d_{1,1} \ \cdots \ d_{1,\mu} \rangle \ldots \langle d_{\lambda,1} \ \cdots \ d_{\lambda,\mu} \rangle\rangle$$
$$f = \langle\langle f_{1,1} \ \cdots \ f_{1,\psi} \rangle \ldots \langle f_{\zeta,1} \ \cdots \ f_{\zeta,\psi} \rangle\rangle, \zeta = \lambda / n_x \text{ et } \psi = \mu / n_y$$

Finalement, le dernier acteur `bias` ajoute la valeur du paramètre b comme défini par le système d'apprentissage et produit le résultat de détection en fonction du signe de l'expression finale :

$$\texttt{bias}_b(f) = \texttt{mlift}(fbias, f) = y \tag{4.41}$$

avec

$$fbias(f) = \begin{cases} 1 & \text{si } f + b > 0 \\ 0 & \text{sinon} \end{cases}$$

Même si seul le signe est utile pour la détermination de la classe à laquelle appartient l'échantillon d'entrée, la valeur de l'expression $f + b$ donne également une information sur la distance de l'échantillon par rapport à la frontière de décision. Cette valeur sera notamment utile pour le filtrage des données en aval.

4.3 Méthode d'implémentation

La démarche générale suivie pour l'implémentation de la formulation précédemment effectuée est illustrée sur la Fig. 4.13.

La première étape consiste en une validation algorithmique ceci afin de quantifier l'impact des paramètres de l'algorithme sur la qualité du système de détection. Ces paramètres concernent la méthode HOG-Dot mais également le typage effectif des données. En effet, une implémentation sur cible matérielle opère en pratique sur des données représentées par des nombres entiers, avec un représentation en virgule fixe des nombres décimaux. Il est dès lors essentiel de quantifier les erreurs d'approximations dues à l'utilisation d'une arithmétique en virgule fixe sur les performances de détection du système.

La seconde étape est la transcription des acteurs et du réseau associé en CAPH, le langage qui sera utilisé pour l'implémentation. A partir du code CAPH, la génération des codes SystemC et VHDL est automatique [10].

La dernière étape concerne la validation de la transcription VHDL RTL générée par CAPH sur une architecture réelle. Cette partie est primordiale pour valider la méthodologie proposée sur un algorithme complexe et mettre en avant certaines problématiques issus de la formulation flot de données sur l'implémentation finale.

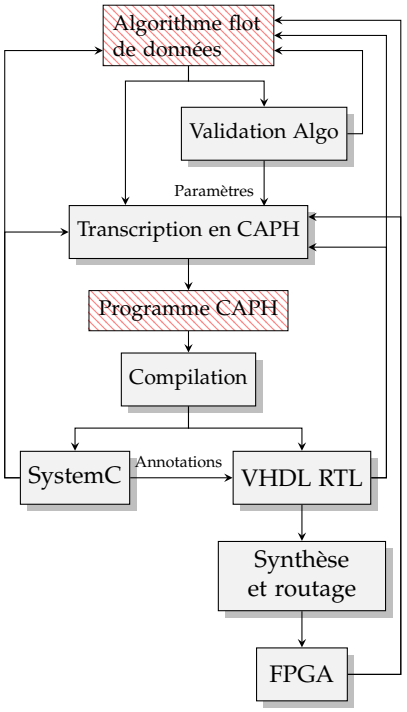


Fig. 4.13: Méthodologie d'implémentation utilisée.

4.4 Validation algorithmique

Comme indiqué plus haut, les algorithmes utilisés nécessitent une étape de plus haut niveau afin d'effectuer l'apprentissage mais aussi de quantifier l'impact des approximations effectuées, qu'elles soient liées au niveau algorithmique ou à la dynamique des données. Pour cela, un environnement de développement basé sur les librairies OpenCV [Brao0] et SVMLight [Joa99] a été utilisé. Cette implémentation logicielle de référence (ILR), est une transcription logicielle de la formulation flot de données afin d'évaluer l'impact de celle-ci sur les performances de détection. La librairie OpenCV fournit une implémentation de référence (issue de [DT05]) pour l'extraction des descripteurs HOG. La librairie SVMLight gère le

10. La synthèse et le placement routage sont assurés par l'outil du fabricant Altera (Quartus II 13.1)

processus d'apprentissage, dont la mise au point peut être empirique à cause des algorithmes d'optimisation qui procurent des résultats différents en fonction des paramètres utilisés, des approximations effectuées ou encore des bases de données d'apprentissage choisies.

Concernant la représentation en virgule fixe, trois catégories de données sont concernées : **les noyaux de convolution**, le **descripteur normalisé** et le **modèle** issu de l'apprentissage. Afin de choisir la meilleure représentation de chacune de ces valeurs, des expérimentations ont été menées sur l'implémentation haut-niveau du système (première étape de la Fig. 4.13). Le protocole de qualification est le suivant :

1. Chaque catégorie est évaluée indépendamment des autres afin de limiter l'influence des propagations d'erreur de précision.
2. Le paramètre utilisé est la dynamique de chaque donnée.
3. Les mesures effectuées sont obtenues par :

   - **apprentissage sur la base de données de l'INRIA** [DT05] (version étendue). La base de données est composée de 12183 images négatives et de 2861 images positives. L'erreur d'apprentissage est rapportée par les estimateurs Xi-Alpha [Joa02] et LOO. Ces estimateurs produisent une information sur la séparabilité de l'espace de description approximée en fonction de la dynamique choisie pour chaque paramètre.
   - **test sur la base de test de l'INRIA**. Cette base est composée de 1126 images positives et 453 négatives. Ce test permet de quantifier la perte de performances avec des images différentes de celles utilisées pour l'apprentissage. De ce test sont extraites les évolutions des métriques TPR, FPR en fonction de la dynamique sur les deux bases.

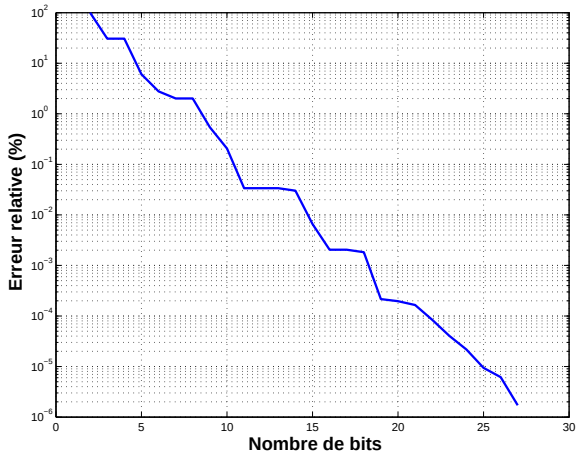
Une fois toutes les expérimentations menées, on effectue une estimation du système complet. Une validation croisée à k-parts (Sec. 3.5.4.1) est rajoutée sur le test final afin d'obtenir une estimation de la sensibilité du système approximé par rapport à la base d'images utilisée. Ce dernier test permet notamment de détecter les sur-apprentissage.

4.4.1 Dynamique des noyaux de convolution

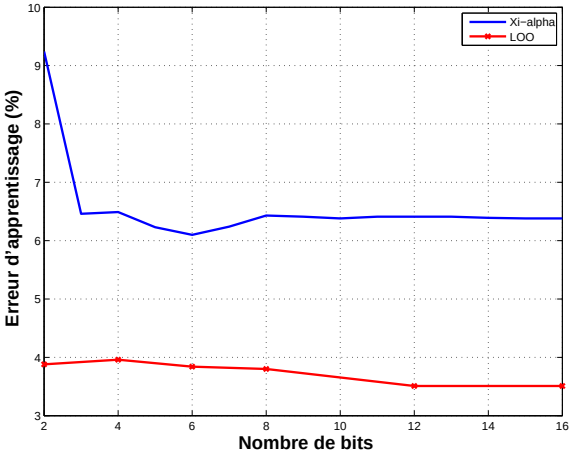
La Fig. 4.14a montre l'évolution de l'erreur relative maximale, commise sur l'ensemble des noyaux de convolutions utilisés dans la méthode *HOG-Dot*. Par erreur relative, on entend ici l'écart entre la valeur réelle d'un coefficient et sa valeur approximée, résultant de son encadrage en virgule fixe. Cette erreur est calculée pour chaque dynamique n par :

$$err = \max\left(\frac{|2^{-n} \times (ceil(2^n \times \cos(\frac{k\pi}{8})) - \cos(\frac{k\pi}{8})|}{|\cos(\frac{k\pi}{8})|}\right), \quad \text{pour } k \in \{0,...,7\} \tag{4.42}$$

où fonction *ceil* est la fonction de troncature d'un nombre décimal. La Fig. 4.14a rapporte l'évolution de l'erreur d'apprentissage en fonction du nombre de bits utilisé. L'erreur d'apprentissage donnée par l'estimateur Xi-alpha se stabilise autour des 6% pour une dynamique de 3 bits et n'évolue plus lorsque la dynamique croît. L'estimateur LOO indique également une erreur quasi constante, autour de 3.5%, indépendamment de la dynamique. La lecture de ces courbes indique que l'utilisation de plus de 3 bits pour la représentation des noyaux sera inutile, et n'améliorera pas la qualité de l'espace de description (donc les performances de classification) bien que l'erreur relative atteigne dans ce cas de 30%.



(a) Erreur relative

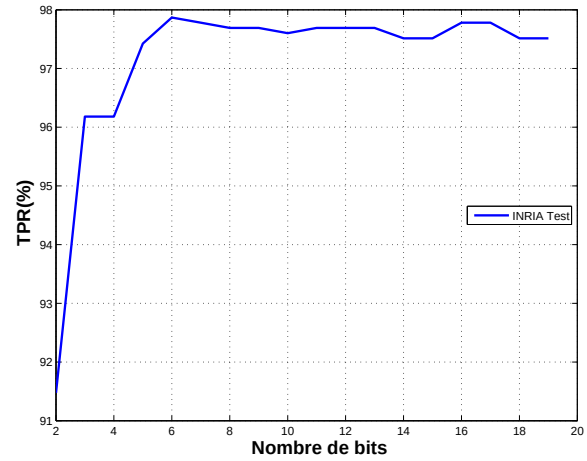


(b) Estimateur Xi-Alpha

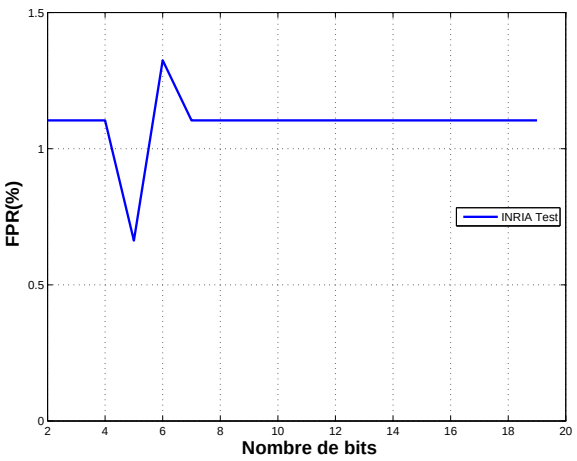
Fig. 4.14: Evolution de l'erreur relative et de l'estimateur Xi-Alpha en fonction de la représentation des noyaux de convolution.

Afin de confirmer ces résultats, l'évolution des TPR et FPR en fonction de la dynamique est représentée sur la Fig.4.15 pour le test effectué sur la base INRIA. Les taux de détection se stabilisent respectivement autour de leur maximum et minimum pour une dynamique de 6 bits, qui est une valeur supérieure aux résultats précédemment obtenus avec l'erreur d'apprentissage.

Le cas le plus défavorable issu des deux observations précédentes est choisi pour la représentation des noyaux de convolutions, soit 6 bits.



(a) TPR



(b) FPR

Fig. 4.15: Evolution du TPR et FPR en fonction de la dynamique utilisée pour la représentation des noyaux de convolution. Les valeurs des TPR et FPR sont données pour l'hyperplan définie par l'apprentissage.

4.4.2 Dynamique du descripteur HOG

Le second facteur dont il convient d'évaluer l'impact est la dynamique du descripteur HOG normalisé. Cette dynamique du descripteur impacte à la fois la qualité de détection mais aussi les performances d'implémentation. En effet, les ressources nécessaires à l'implantation des 32 diviseurs parallèles dépendent directement de la dynamique de leurs opérandes. La Fig.4.16 montre l'évolution des estimateur Xi-Alpha et LOO en fonction du nombre de bits

utilisé pour la dynamique du descripteur. La valeur de l'estimateur Xi-Alpha démarre à 35% d'erreur pour une représentation sur 2 bits et décroît de manière exponentielle jusqu'à se stabiliser autour des 6% pour une représentation sur 8 bits. L'estimateur LOO apporte une information différente (théoriquement plus précise que l'estimateur Xi-alpha) et montre que l'erreur d'apprentissage se stabilise à partir de seulement 4 bits.

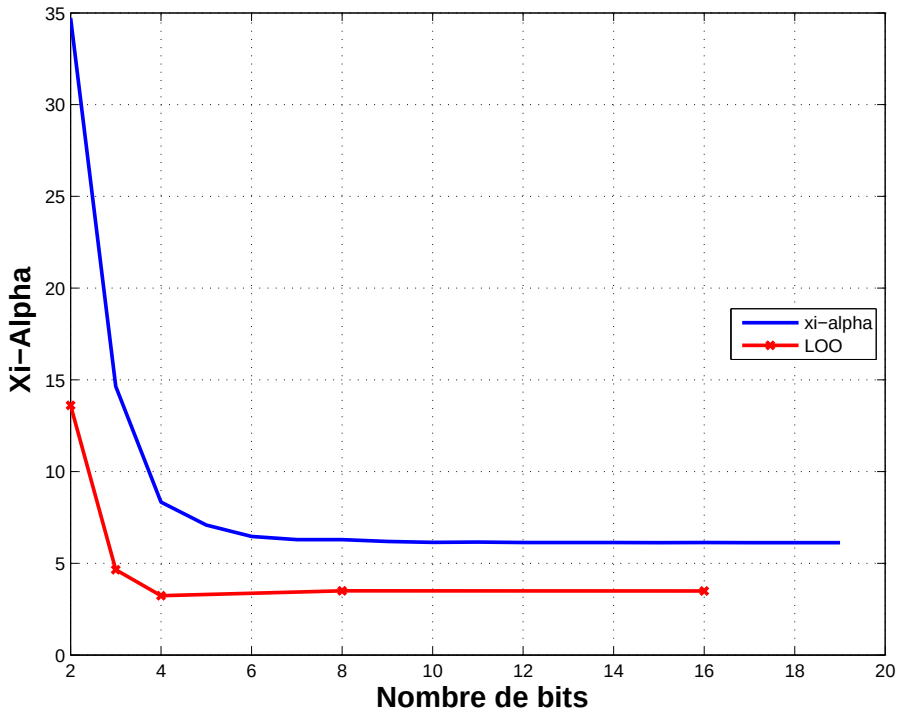
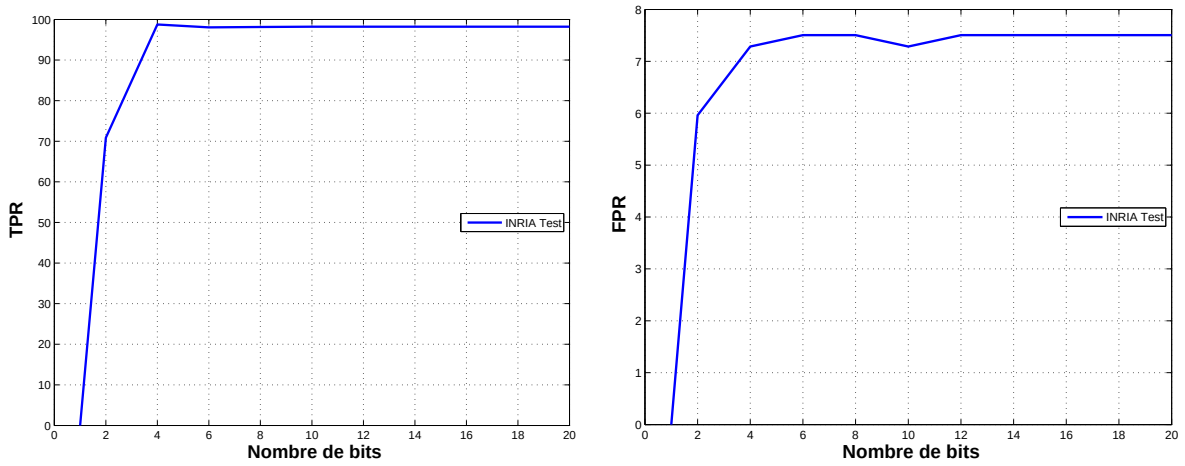


Fig. 4.16: Evolutions des estimateurs Xi-Alpha et LOO en fonction de la dynamique du descripteur HOG

Afin de confirmer ces résultats, qui peuvent paraître surprenants, la Fig. 4.17 montre l'évolution des taux de bonnes et fausses détections sur la classification de la base test INRIA, toujours en fonction de la dynamique du descripteur. Les tests sur les deux bases confirment l'information fournie par l'estimateur LOO, à savoir que les taux de détection se stabilisent passé 4 bits de dynamiques. Comme les valeurs des TPR et FPR sont obtenues par rapport à un hyperplan différent sur chaque apprentissage, une variation (*offset*) peut s'introduire dans les résultats. Afin d'évaluer plus précisément le couple TPR/FPR en fonction du nombre de bits, il convient de tracer les courbes ROC et de mesurer la métrique AUC associée. Les résultats précédemment obtenus sur la Fig. 4.17 permettent de limiter le nombre de courbes ROC à tracer afin de garder de la lisibilité (il faudrait sinon une courbe par dimension possible).

La Fig. 4.18 compare les courbes ROC pour les dynamiques $2, 3, 4, 8, 16$ bits, obtenues sur la base de test de l'INRIA. A cela, nous avons rajouté une version en virgule flottante (simple précision) et la version de référence OpenCV, intégrant un schéma de norme plus complexe (L2Hys). La métrique AUC est calculée pour chaque courbe par une méthode d'intégration par trapèzes. Les représentations sur 2 et 3 bits dégradent les performances comme constaté sur la Fig. 4.17. Les dynamiques $4, 8$ et 16 bits procurent des résultats similaires à la représentation flottante effectuant une norme $L1$ (AUC de 0.99). Il est important de souligner que la représentation en virgule fixe du descripteur sur 4 bits, soit 16 valeurs possibles par composante, ne dégrade aucunement les performances de classification par rapport à une représentation en virgule flottante. Cependant, l'utilisation du schéma de norme d'OpenCV, plus complexe, permet une amélioration des performances, comme discuté lors de la formulation de l'étape de normalisation du descripteur (Sec. 4.2.2.3).

En conclusion, même si les résultats montrent qu'une représentation sur 4 bits suffit dans ces conditions de mesures, l'implémentation réelle utilise une représentation sur 8 bits car celle-ci n'engendre pas de surcout important de ressources dans les opérateurs arithmétiques



(a) TPR *(b)* FPR

Fig. 4.17: Evolution du TPR et FPR en fonction de la dynamique utilisée pour la représentation des composantes du vecteur HOG. Les valeurs des TPR et FPR sont données pour l'hyperplan défini par l'apprentissage.

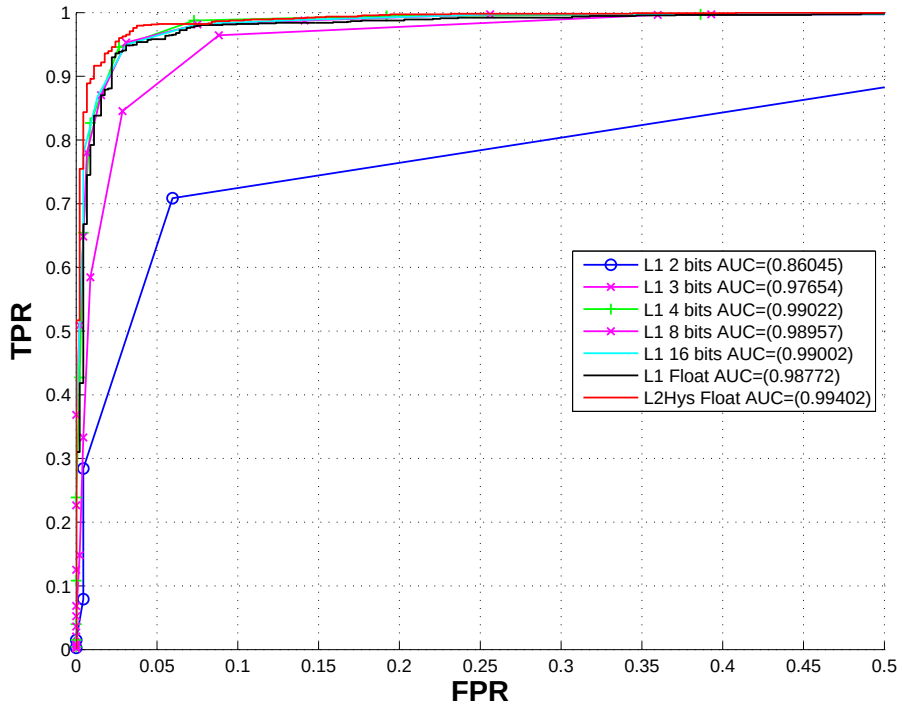


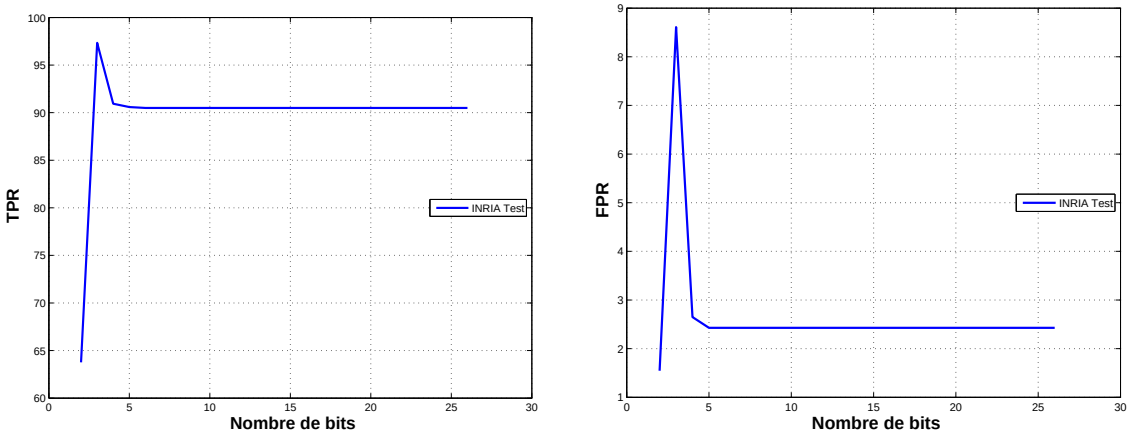
Fig. 4.18: Courbe ROC sur la précision du descripteur pour le jeu de test de l'INRIA dont la métrique AUC est calculée pour chaque courbe. Une précision de 2 bits dégrade les performances du système. A partir d'une dynamique sur 4 bits, les performances sont similaires à celles obtenues par une normalisation de type L1 flottante. Afin d'améliorer le système il convient de modifier le schéma de norme.

sur le FPGA.

4.4.3 Dynamique du modèle SVM

Le dernier paramètre dont la valeur peut impacter les performances du système est la dynamique du vecteur utilisé pour coder les poids de la SVM. Cette grandeur ne peut pas être mesurée par l'erreur d'apprentissage (estimateur Xi-Alpha et LOO) car l'apprentissage est obligatoirement effectué en virgule flottante. La donnée est seulement évaluée via les

résultats de classification sur la base INRIA, en fonction de la dynamique souhaitée pour le modèle issu de l'apprentissage, comme illustré sur la Fig. 4.19.



(a) TPR *(b)* FPR

Fig. 4.19: Evolution du TPR et FPR en fonction de la dynamique utilisée pour la représentation du modèle d'apprentissage. Les valeurs des TPR et FPR sont données pour l'hyperplan défini par l'apprentissage.

De même que pour la dynamique du descripteur, les performances de détection ne sont plus impactées dès lors que la dynamique du modèle est supérieure à 4 bits. A noter que les valeurs importantes des TPR et FPR relevées pour une dynamique de 3 bits sont dues à une variation importante du biais obtenu entre deux apprentissages successifs.

4.4.4 Synthèse

Dans cette section, on évalue maintenant l'impact combiné de chacune des approximations étudiées séparément dans la section précédente. Pour cela, les résultats précédents sont réutilisés, à savoir : des dynamiques de 6 bits pour les noyaux de convolution, 4 bits pour le descripteur et le modèle SVM.

La Fig. 4.20 montre les courbes ROC obtenues avec notre implémentation logicielle (ILR) simulant les approximations en fixe virgule d'une part et les résultats fournis par l'implémentation OpenCV de référence d'autre part.

La formulation proposée fournit de moins bons résultats que l'implémentation OpenCV, principalement à cause du schéma de normalisation utilisé qui est moins élaboré. L'écart de performances se situe sur le taux de faux positifs qui est plus important sur le système flot de données. Toutefois, ce taux important de bruit sera diminué par un système de filtrage (Sec. 4.9).

Sur cette évaluation finale, une validation croisée à 10 parts a été effectuée. Les résultats montrent une précision (Eq 3.35) moyenne de 98.47 avec une variance de 0.11. La faible variance obtenue indique que le système n'est pas sur-entraîné.

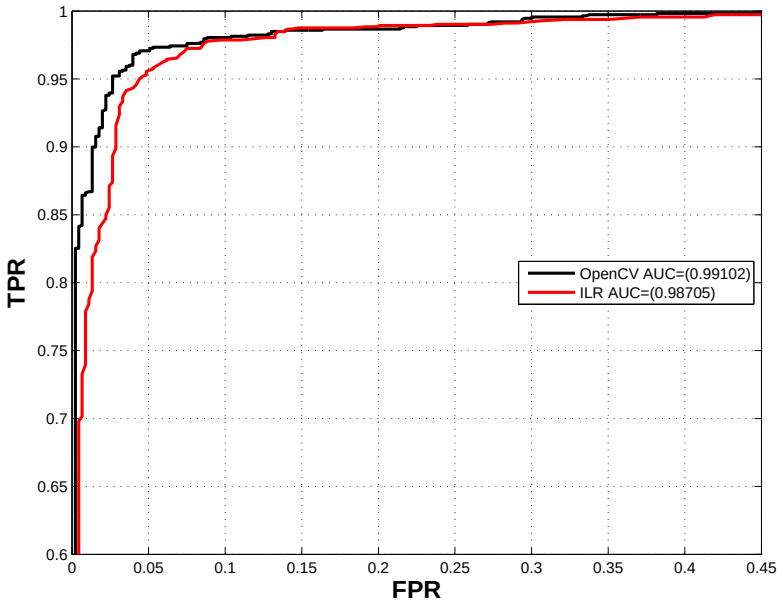


Fig. 4.20: Courbes ROC et métrique AUC obtenues à partir du descripteur OpenCV et du descripteur approximé en virgule fixe

4.5 Transcription en CAPH

4.5.1 Description du réseau d'acteurs en CAPH

Un programme CAPH contient à la fois la description du réseau d'acteurs constituant l'application et du comportement de chacun de ces acteurs. Cette section se focalise sur le premier aspect, c'est-à-dire la description du réseau d'acteurs associé à l'application de détection. Le réseau est décrit précédemment sur les Figs. 4.3 et 4.11.

Le langage de description de réseaux d'acteurs au sein de CAPH est fonctionnel, polymorphique et supporte les fonctions de cablage (*wiring functions*). Ces fonctions permettent la définition de schémas de connexion (*graph pattern*), qui peuvent être réutilisés, facilitant ainsi la description de réseaux complexes avec un ou plusieurs niveaux de hiérarchie. Le langage fournit par ailleurs un ensemble de fonctions d'ordre supérieur ou "fonctionnelles" ([Ser15] pour plus de détails). Notamment, la fonctionnelle `map` qui prend deux arguments : une fonction f et un ensemble de canaux de communications $(x_1, ..., x_n)$ et applique la fonction f à chaque canal :

$$\texttt{map}\ f\ (x_1, ..., x_n) = (f\ x_1, ..., f\ x_n) \tag{4.43}$$

Le premier argument peut être un simple acteur mais également une autre fonction. La fonctionnelle `map` permet la réplication d'acteurs, comme illustré sur la Fig. 4.21.

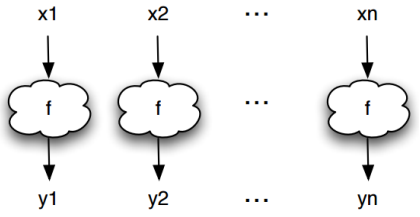


Fig. 4.21: Illustration de la fonction `map` : $(y_1, \cdots, y_n) = \texttt{map}\ f\ (x_1, \cdots, x_n)$

Une variante de la fonctionnelle `map` est la fonction `mapi`, définie par :

$$\texttt{map}\ i\ f\ (x_1, \cdots, x_n) = (f\ 0\ x_1, \cdots, f\ (n-1)\ x_n) \tag{4.44}$$

Le listing 4.2 donne la description en langage CAPH du réseau d'acteurs formulé dans la première partie de ce chapitre. Les descriptions des acteurs sont disponibles dans l'annexe A.

Ce listing contient la déclaration de plusieurs fonctions `convs`, `argmax`, `hists`, `norm`, `hog` et `svm`. Chacune de ces fonctions correspond à une étape de l'algorithme précédemment décrit. Les fonctions `hog` et `svm` correspondent en particulier à l'extraction de descripteurs et au système de classification. Les mots clés `stream` et `port` désignent les entrées/sorties du système. Un `stream` est assimilable à un canal de type FIFO connecté à une entrée (resp. sortie) du système. un `port` est vu comme un **registre** pouvant est lu/écrit depuis l'extérieur. Le graphe flot de données, correspondant, généré par le compilateur CAPH est donné sur la Fig. 4.22.

```
1  -- Fonction d'instanciation d'un acteur de convolution
2  net conv_hv kernel norm pad i =
3      let rec (o,z,zz) = my_conv2s33_hv (kernel,norm,pad) (i,z,zz) in o;
4
5  -- Fonction d'instanciation d'un ensemble de convolueurs parallèles
6  net convs rep x =
7      let ff i =  conv_hv (coeff[i]) 9 o in
8      mapi ff (rep x);
9
10 -- Fonction d'instanciation de l'argmax pour la méthode HOG-Dot
11 net argmax ii =
12     let (io,i1,i2,i3,i4,i5,i6,i7) = map abs ii in
13     let (mag01,bin01) = comp_init (0,1) (io, i1) in
14     let (mag23,bin23) = comp_init (2,3) (i2, i3)  in
15     let (mag45,bin45) = comp_init (4,5) (i4, i5)  in
16     let (mag67,bin67) = comp_init (6,7) (i6, i7)  in
17     let (mag0123,bin0123) = comp(mag01,mag23,bin01,bin23) in
18     let (mag4567,bin4567) = comp(mag45,mag67,bin45,bin67) in
19     let (mag,bin) = comp(mag0123, mag4567, bin0123, bin4567) in
20         binning (mag,bin);
21
22 -- Application des acteurs d'histogrammes sur chaque classe
23 -- xs: cellule en x
24 -- ys: cellule en y
25 net hists xs ys hx =
26     let hist xs ys i = vsum ys (hsum xs (i)) in
27     map (hist xs ys) (hx);
28
29 -- Extraction du voisinage et normalisation
30 net norm hx =
31     let (vo,...,v31) = map block_extraction  (hx) in
32     let N = norm_factor(vo,...,v31) in
33     let (no,...,n31) = rep32 N in
34     map2 div ((vo,...,v31), (no,...,n31));
35
36 -- Function d'instanciation du reseau d'extraction de descripteurs HOG
37 net hog xs ys i =  norm (hists xs ys ( argmax (convs rep8 i)));
38
39 -- Fonction d'instanciation de la svm parallèle
40 net svm k xx x0 x1 x2 ... x31 =
41     let (ko,...k31)= kern (k) xx in
42     let wi_xi =  dot_u (xo,...,x31,ko,...,k31) in
43     let xwin = xwindowing wi_xi in
44     let ywin = ywindowing xwin in
45     decision oS ywin;
46
47 -- Description du réseau complet
48 net y = svm w n m b (hog 8 8 i);
49
50 stream i: unsigned<8> dc from "/dev/camo";
51 stream y: unsigned<1> dc to "/dev/disp1";
52
53 port w : unsigned<8> from  "/dev/porto" init o;
54 port n: unsigned<8> from "/dev/port1" init o;
55 port m: unsigned<8> from "/dev/port2" init o;
56 port b: unsigned<8> from "/dev/port3" init o;
```

Listing 4.2: Description du réseau d'acteurs en CAPH pour l'application HOGSVM

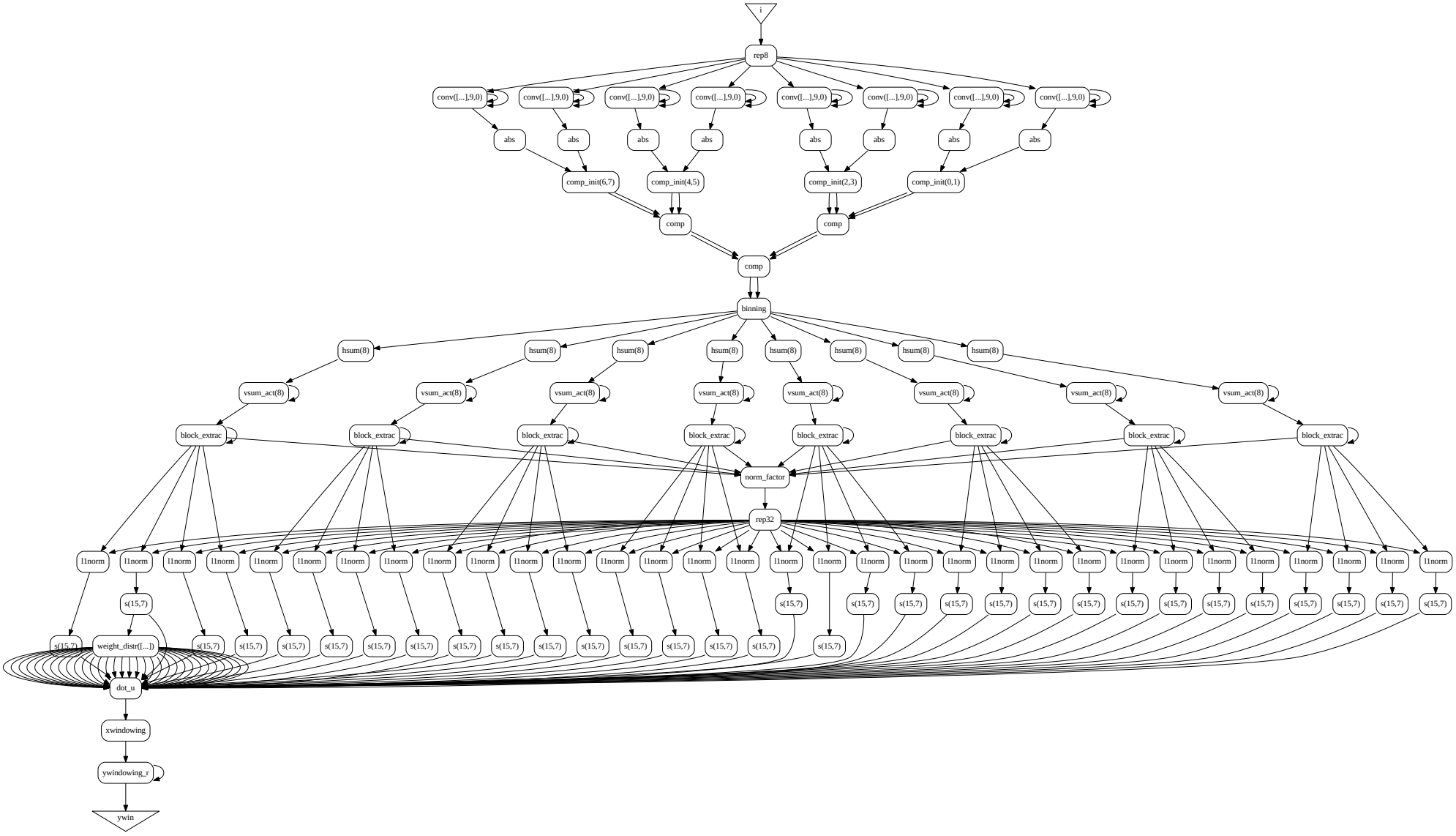


Fig. 4.22: Graphe flot de données de l'application correspondant au listing 4.2

4.5.2 Expression des acteurs en CAPH

Le code de l'ensemble des acteurs utilisés dans le programme du listing 4.2 est disponible à l'annexe A Cette section se limitera à la description de deux acteurs. Les acteurs choisis à titre d'exemple sont `hsum` et `vsum`. Ces acteurs servent au calcul des histogrammes d'orientations de gradients.

La description d'un acteur comprend une interface et un corps. L'interface regroupe la déclaration des entrées, des sorties et des paramètres (avec les types associés). Lors de la construction du réseau, entrées et sorties sont connectées par des canaux de communication et des valeurs statiques sont associées aux paramètres à ce moment là. Le corps de l'acteur contient la déclaration de variables locales et un ensemble de règles d'activation. Les variables locales permettent de mémoriser des valeurs entre plusieurs activations successives. Les règles d'activations sont spécifiées après le mot clé **`rules`**. Chaque règle comporte un ensemble de *patterns* (pat_i) impliquant les entrées et/ou les variables locales et un ensemble d'expressions (exp_i) décrivant les modifications sur les sorties et/ou les variables locales lors de l'activation de cette règle. Chaque règle est de la forme :

$$\mid \quad (qual_1 : pat_1, \cdots, qual_n : pat_n) \rightarrow (qual'_1 : exp_1, \cdots, qual'_n : exp_n)$$

où $qual$ (respectivement $qual'$) désigne une entrée ou une variable locale (respectivement une sortie ou une variable locale).

Comme on l'a vu à la section. 4.2.1, le modèle de calcul utilisé repose sur la distinction au niveau des données circulant entre les acteurs, entre jetons de données et jetons de contrôle. En CAPH, la distinction entre les jetons de contrôle et de données est assurée en associant à ces données un type de données algébrique. Ce type `dc`, est défini comme suit :

```
type t dc = SoS | EoS | Data of t
```

où `SoS` (Start of Structure), `EoS` (End of Structure) et `Data` sont des *constructeurs* encodant respectivement les jetons de contrôle "<", ">" et les jetons de données. Le paramètre t dénote une variable de type [11].

Une description en CAPH de l'acteur `hsum`, introduit à la Sec. 4.2.2.2 est donnée sur le listing 4.3. Sur cet exemple, on a utilisé une notation abrégée pour les constructeurs du type `dc` : `'<` désigne le constructeur `SoS`, `'>` celui de `EoS` et `'t` le constructeur `Data of t`.

Listing 4.3: Listing CAPH de l'acteur `hsum`

```
1  actor hsum (k:unsigned<8>)
2    in  (a:unsigned<s> dc )
3    out (c:unsigned<s> dc )
4
5  var s : {WSF, WSL, Sum} = WSF
6  var xsum  : unsigned<s>
7  var j     : unsigned <8>
8
9  rules
10 |(s:WSF, a:'<)                -> (s:WSL, c:'<)
11 |(s:WSL, a:'<)                -> (s:Sum, c:'<, xsum:0, j:0)
12 |(s:WSL, a:'>)                -> (s:WSF, c:'>)
13 |(s:Sum, a:'x) when j<(k-1)   -> (s:Sum, xsum:xsum+x, j:j+1)
14 |(s:Sum, a:'x) when j=(k-1)   -> (s:Sum, c:'(xsum+x), xsum:0, j:0)
15 |(s:Sum, a:'>)                -> (s:WSL, c:'>);
```

Le comportement de l'acteur `hsum` peut se décrire avec six règles d'activations (lignes 10-15) et trois variables internes s, $xsum$ et j. La variable s est une variable d'état pouvant prendre trois valeurs : `WSF`, `WSL` et `Sum`. Dans l'état initial (s = `WSF`), l'acteur attend le jeton `SoS` (début

11. Le système de typage de CAPH est similaire à ceux équipant les langages de programmation fonctionnels modernes comme Haskell or ML. Il supporte en particulier le polymorphisme paramétrique, c'est-à-dire la définition de types paramétrés par d'autres types.

d'image) sur le canal a pour passer dans l'état `WSL` et générer le même jeton sur le canal c de sortie. Dans l'état `WSL`, l'acteur attend un autre jeton `SoS` (début de ligne) sur le canal a. L'activation de la règle génère le jeton de début de ligne sur le canal de sortie c, initialise l'accumulateur $xsum$ et la variable de compteur j à 0. Dans l'état `Sum`, lorsqu'une donnée $Data(x)$ est présente sur le canal d'entrée, deux règles sont activables suivant la valeur du compteur interne j. Si j est inférieur à $k-1$, l'acteur additionne la valeur lue en entrée (x) avec le contenu de l'accumulateur $xsum$, sinon il produit en sortie le résultat de l'accumulation et réinitialise les variables internes à 0. Le nombre de valeurs à accumuler k est donné en paramètre statique de l'acteur. Les paramètres statiques sont fixés lors de l'instanciation de l'acteur dans un réseau et restent constants lors de l'exécution. La variable d'état s reste égale à `Sum` jusqu'à l'arrivée du jeton de fin de ligne, qui déclenche l'écriture du même jeton en sortie et met la variable d'état à `WSL`. Une fois de nouveau dans l'état `WSL`, si le jeton suivant correspond à un début de ligne, l'acteur retourne dans l'état `Sum` pour débuter l'accumulation sur une nouvelle ligne, et si le jeton correspondant à la fin de structure (fin d'image), l'acteur retourne dans l'état `WSF`.

Le second acteur décrit est l'acteur `vsum` qui effectue l'accumulation des données dans la direction verticale. Par exemple, pour une image de quatre lignes et de quatre pixels :

$$\texttt{vsum}_4(\langle\langle 1\ 2\ 3\ 4\ \rangle\langle 0\ 20\ 3\ 4\ \rangle\langle 0\ 0\ 3\ 4\ \rangle\langle 0\ 0\ 0\ 40\ \rangle\rangle) = \langle\langle 1\ 22\ 9\ 52\ \rangle\rangle$$

La description de cet acteur est légèrement plus complexe car elle nécessite la mémorisation d'une ligne de données avant de débuter l'accumulation. De plus, pour des raisons d'optimalité des ressources matérielles allouées (raisons détaillées dans le chapitre 6), on utilise un canal de communication rebouclé pour la mémorisation des données, comme illustré dans la Fig. 4.23.

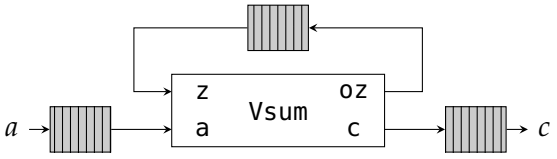


Fig. 4.23: Schémas de connexions de l'acteur `vsum`

Le comportement de l'acteur `vsum` est défini sur le listing 4.4.

Listing 4.4: Listing CAPH de l'acteur `vsum`

```
1  actor vsum_act (k:unsigned<8>)
2      in  (a:unsigned<16> dc,  z:unsigned<16>)
3      out (c:unsigned<16> dc, oz:unsigned<16>)
4
5  var s : {WaitSoF,Wait1L, FirstLine , WaitSoL,ACC,WaitkLine,WriteRes} = WaitSoF
6  var j : unsigned <8>
7
8  rules
9  | (s:WaitSoF,    a:'<)              -> (s:Wait1L, c:'<)
10 | (s:Wait1L,     a:'>)              -> (s:WaitSoF,c:'>)
11 | (s:Wait1L,     a:'<)              -> (s:FirstLine)
12 | (s:FirstLine , a:'x)              -> (s:FirstLine , oz:x )
13 | (s:FirstLine , a:'>)              -> (s:WaitSoL, j:1)
14 | (s:WaitSoL,    a:'<)              -> (s:ACC)
15 | (s:ACC,        a:'x, z:acc)       -> (s:ACC,  oz:x+acc)
16 | (s:ACC,        a:'>) when j<(k-2) -> (s:WaitSoL, j:j+1)
17 | (s:ACC,        a:'>)              -> (s:WaitkLine, j: 0)
18 | (s:WaitkLine , a:'<)              -> (s:WriteRes, c:'< )
19 | (s:WriteRes,   a:'x, z:acc)       -> (s:WriteRes, c:'(x+acc) )
20 | (s:WriteRes,   a:'>)              -> (s:Wait1L, c:'> );
21
22 -- Mapping function
23 net vsum k ii = let rec (oo,z1) = vsum_act k (ii,z1) in oo;
```

Le fonctionnement de l'acteur est défini via douze règles d'activations (lignes 9-20), une variable d'états s et une variable de comptage j. Le paramètre k, donné en argument de l'acteur, définit le nombre de lignes sur lequel s'effectue l'accumulation.

Comme pour l'acteur précédent, il n'y a qu'une seule règle possible dans l'état initial (`WaitSoF`), déclenchée par l'arrivée du jeton de début de structure. Son activation recopie le jeton de début de structure sur le canal c et fait passer l'acteur dans l'état `Wait1L`. Les trois règles suivantes correspondent à la mémorisation de la première ligne de données. Pour ce faire, l'acteur recopie les données d'entrée (canal a) sur le canal oz (entrée de la FIFO rebouclée). La variable j est réinitialisée à 1 lors de l'arrivée du jeton de fin de structure, marquant la fin de la première ligne de données. Dans cet état, la seule règle activable est celle déclenchée par l'arrivée d'un jeton de début de ligne (non recopié en sortie), qui envoie dans l'état `ACC`. Dans l'état `ACC`, chaque donnée lue en entrée (canal a) est ajoutée à la donnée de la ligne précédente (lue sur le canal oz) et ce résultat intermédiaire est renvoyé sur le canal oz. A réception du jeton de fin de ligne, deux règles sont activables en fonction de la valeur de la variable interne j. Si la valeur de j est inférieure à $k-2$, l'acteur retourne dans l'état `WaitSoL` et la variable j est incrémentée, ce qui signifie que l'on continue à lire des données en entrée.

Dans le cas contraire, cela signifie que l'acteur va écrire les résultats en sortie lors de la prochaine ligne. En effet dans l'état `WriteRes`, pour chaque donnée de la même ligne, l'activation de la règle (ligne 19) produit en sortie le résultat de la somme entre la valeur d'entrée et la valeur lue sur le canal z (correspondant à l'accumulation verticale précédemment effectuée). Cette règle est active jusqu'à ce que le jeton de fin de structure soit présent en entrée, renvoyant l'acteur dans l'état `Wait1L`. Dans l'état `Wait1L`, le type de jeton qui arrivera en entrée indiquera le comportement de l'acteur : si ce jeton est un début de structure, l'acteur recommence l'accumulation des k lignes, sinon l'acteur retourne à l'état initial.

De même que pour l'acteur `hsum` précédemment décrit, il est important que la dimension du flux d'entrée soit multiple de la valeur donnée en paramètre k car aucun chemin autre que celui que nous venons de décrire ne permet de retour à l'initial. Cet acteur peut cependant être étendu avec quelques règles d'activations pour gérer le cas d'une image non multiple de la dimension de la cellule.

4.6 Simulation du code CAPH

Cette étape vise à valider, par simulation (indépendamment de la cible FPGA finale), le code décrit à la section précédente. On utilise pour cela les deux "backends" du compilateur CAPH : SystemC et VHDL (la simulation via l'interpréteur du langage n'est pas exploitée ici car elle conduit ici à des temps d'exécution trop importants).

En l'absence de simulation CAPH, la simulation SystemC a pour objectif de valider l'algorithme et de fournir des informations sur l'occupation maximale de chaque canal de communication. Les annotations générées sont ensuite réutilisées par le second "backend" lors de la génération du code VHDL.

La Fig. 4.24 montre les courbes ROC issues des simulations faites avec OpenCV, notre implémentation logicielle de référence ILR (utilisée dans la Sec. 4.4) d'une part et le code SystemC et VHDL généré par le compilateur CAPH d'autre part, sur le jeu de test de l'INRIA. Les courbes sont quasiment confondues, et les AUC égales à 10^{-3} près. Ce résultat valide la méthodologie d'implémentation proposée à la Sec. 4.3. Il montre par ailleurs qu'il sera, si besoin, possible de palier les problématiques de ressources matérielles par une simulation haut-niveau, sachant que la transcription du code VHDL procurera des résultats équivalents.

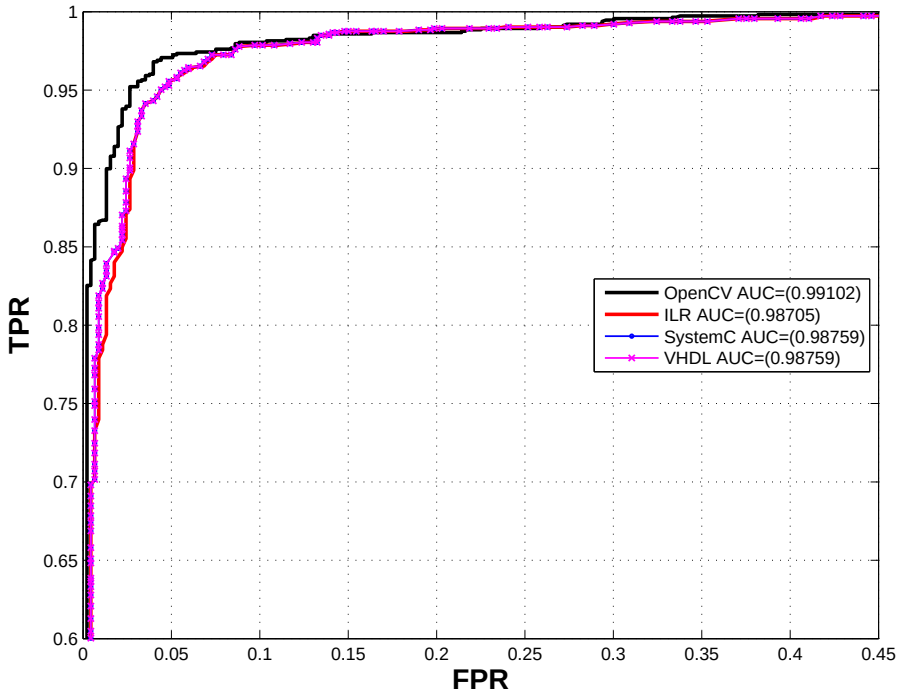


Fig. 4.24: Courbes ROC issus de la simulation du code généré en SystemC et en VHDL RTL

4.7 Implantation

On décrit ici la dernière étape du flot de conception, à savoir l'intégration du code VHDL RTL généré sur une plateforme réelle. Comme annoncé au chapitre 1, l'implantation est faite sur une plateforme de type smart-camera.

4.7.1 Plateforme de test

La plateforme cible, nommée DreamCam [BB14], est représentée sur la Fig. 4.26. Elle est équipée d'un imageur 1.3 Megapixels (E2V ou Aptina) qui offre de nombreuses possibilités de configuration, notamment le sous-échantillonnage et les régions d'intérêts (ROI) multiples. Le coeur du système est un FPGA Cyclone III de la société Altera. Le FPGA est connecté à six mémoires SRAM contenant chacune un Méga-mots de 16 bits. Chaque mémoire a un bus d'adresse et de données privé, permettant un fonctionnement parallèle. La caméra originale a été étendue pour supporter des communications sans fil et ainsi répondre aux contraintes de mobilité à la base de ces travaux (Fig. 4.25). Le protocole de communication, issu des réseaux de capteurs, IEEE 802.15.4 [iee15] (bande passante maximale théorique 250 Ko/s) est utilisé. Pour cela, un composant de la société Microchip (MRF214J) est en charge de la couche physique du protocole. Les autres couches réseau sont gérées par une bibliothèque de communication implantée dans un microcontroleur [12].

D'un point de vue logique, la plateforme DreamCam peut être vue comme une architecture en quatre modules (Fig. 4.26). Le premier module fournit une interface avec l'imageur externe et récupère le flux de pixels qui sera communiqué au module de traitement. Le module de traitement est ici généré par le compilateur CAPH. Les deux modules restants sont des éléments de communication, un cœur de processeur (*softcore*) et un composant USB. Le *softcore* est en charge des communications réseaux mais aussi de la récupération les données issues de la partie traitement et de la configuration du système (bus de paramètres). Afin de simplifier sa mise en oeuvre, un système d'exploitation multi-tâches (ERIKA [eri15]), intégrant par défaut le support des communications 802.15.4, est utilisé. La communication USB

12. La nature du contrôleur a évolué, il s'agissait initialement d'un microcontroleur PIC32, qui a été remplacé par un softcore NIOSII et sera implémenté dans un hardcore Cortex-A9 dans la prochaine génération de caméra

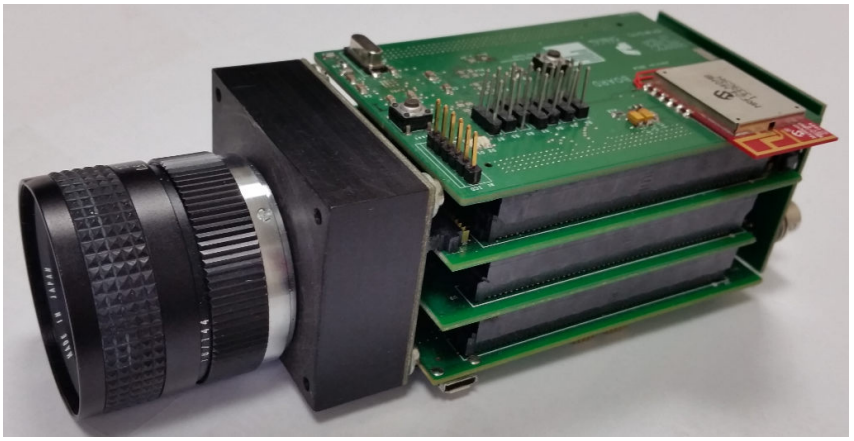


Fig. 4.25: Plateforme de test DreamCam - Caméra intelligente pour le réseau de capteurs

permet à la fois la transmission de flux de données avec des débits plus élevés mais surtout l'envoi d'images (ou de données) depuis l'extérieur, permettant d'utiliser des vecteurs de test connus et ainsi de déboguer les applications efficacement.

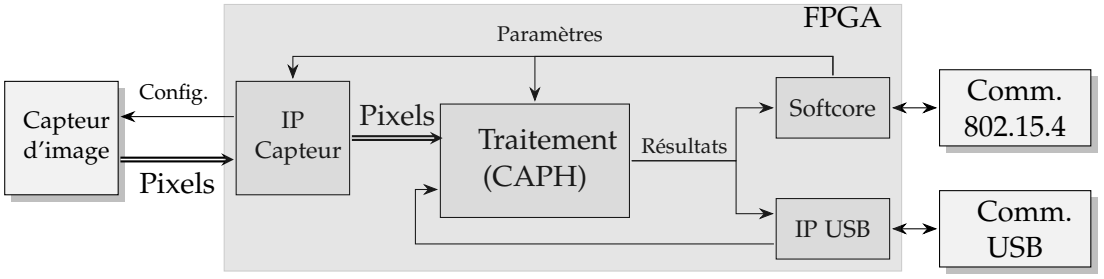


Fig. 4.26: Architecture de caméra intelligente pour le réseau de capteurs.

4.7.2 Intégration sur la plateforme DreamCam

L'intégration du code VHDL RTL généré par le compulateur CAPH au sein de l'architecture décrite par la Fig. 4.26 suppose un mécanisme d'interfaçage adéquat. Ce mécanisme, schématisé sur la Fig. 4.27, doit notamment assurer la mise en forme des signaux à destination et en provenance du réseau d'acteurs et la gestion de paramètres (vus comme des *ports* au niveau du programme CAPH).

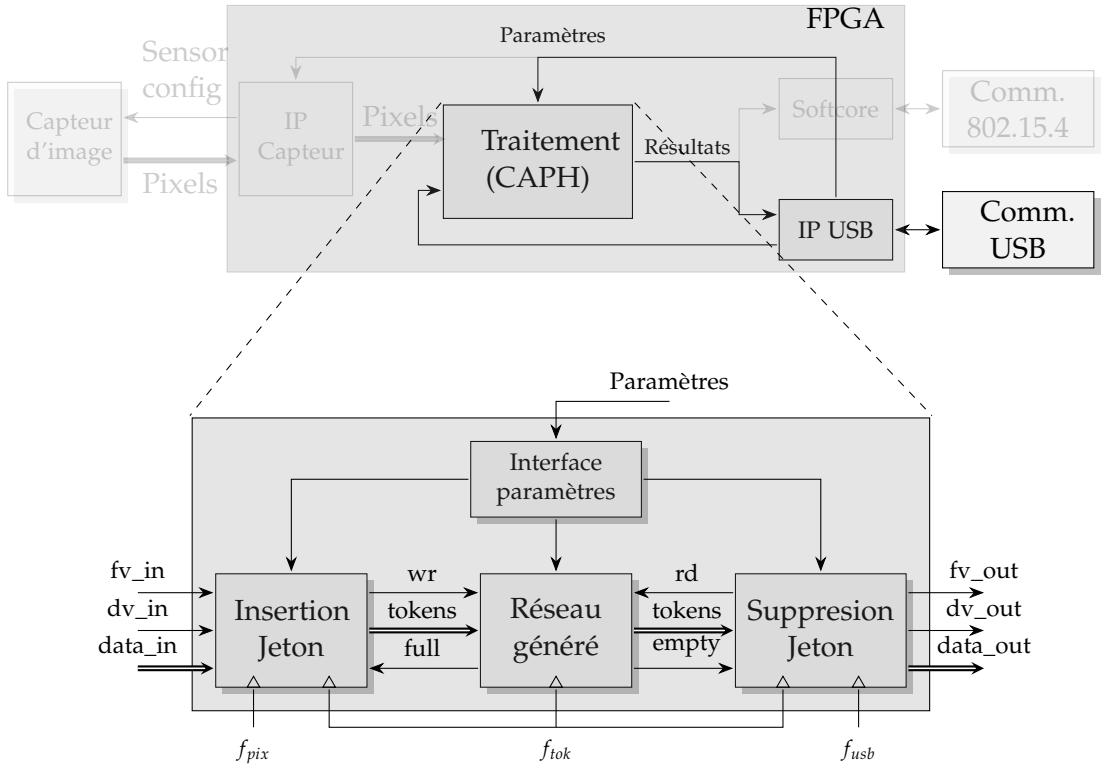


Fig. 4.27: Configuration de test pour la validation de l'application sur la DreamCam.

Voici la description des flots de données, illustrées sur la Fig. 4.27, entre l'USB et le bloc de traitement :

- *fv_in* : signal de validation de l'image d'entrée. Ce signal est à 1 tant que l'image courante est en cours de traitement. Ce signal est remis à 0 à chaque inter-trame.
- *dv_in* : signal de validation de la donnée présente sur le bus de données *data_in*. Ce signal peut être discontinu.
- *data_in* : bus de données.
- *tokens* : flux de jetons entrant dans le (resp sortant du) réseau CAPH (de type `dc` ici). Les deux bits de poids forts de chaque jeton correspondent à l'entête, permettant de distinguer les délimiteurs ($\langle$ ou $\rangle$) des données.
- *wr* et *full* : signaux de contrôle de la première FIFO du réseau d'acteurs.
- *rd* et *empty* : signaux de contrôle de la dernière FIFO du réseau d'acteurs.
- *fv_out*, *dv_out* et *data_out* : même comportement que les signaux *fv_in*, *dv_in* et *data_in*.
- *Paramètres* : paramètres envoyés depuis le contrôleur USB. Les paramètres peuvent êtres des registres pour les blocs autour de CAPH (dimensions des images d'entrée et de sorties, encodage des jetons,...) ou des valeurs associées aux *ports* du programme CAPH.

La logique de contrôle nécessaire au bon fonctionnement du système est composée de quatre blocs :

- Un bloc d'insertion de jetons. Ce bloc génère le flot de jetons entrant dans le réseau CAPH à partir des signaux de contrôle (*fv_in* et *dv_in*). L'insertion des jetons a un coût en surface (non négligeable suivant la complexité du traitement) et dépendant de la structuration des données utilisée dans l'implémentation CAPH. L'utilisation du type `dc` avec des données de type image requiert l'insertion de deux jetons par ligne de données plus deux jetons par image. Il est nécessaire d'augmenter la fréquence de la partie traitement par rapport à la fréquence du flux de pixels. Si on considère une image

de dimension $W \times H$, la fréquence de traitement f_{tok} par rapport à la fréquence pixels f_{pix} doit être à minima de :

$$f_{tok} = f_{pix}\left(1 + \frac{2H+2}{WH}\right) \tag{4.45}$$

Dans notre cas, les spécifications de l'imageur (ou de l'envoi d'image par USB) font qu'il existe un certain nombre de cycles de repos pendant lesquels aucune donnée d'entrée n'est valide. Cette particularité permet l'insertion des jetons sans aucune augmentation de la fréquence. Cependant le modèle d'acteur défini dans la version du compilateur utilisée dans les travaux décrits ici impose une contrainte sur la fréquence des acteurs qui doit être double de celle des données d'entrée. On a donc pour cette phase de test :

$$f_{tok} = 2 \cdot f_{pix} \tag{4.46}$$

- Un bloc de suppression de jeton. Ce bloc effectue la transcription entre le flot de jetons généré par le réseau d'acteurs et le composant USB. Cette transcription implique le décodage des jetons de contrôle pour la génération des signaux (*fv_out* et *dv_out*) ainsi qu'une adaptation en fréquence.
- Un bloc pour l'interfaçage de paramètres. Ce bloc autorise des configurations dynamiques du système via la communication extérieure. Ces configurations peuvent être de paramètres tels que la dimension d'image pour les blocs de création/suppression de jetons ou bien la valeur d'un **port** du réseau d'acteurs.
- Un bloc de communication USB bidirectionnelle. La possibilité d'envoyer des vecteurs de tests unitaires dans la caméra afin de valider l'implémentation est une spécification incontournable de l'intégration sur la plateforme. Aussi comme nous n'utilisons pas le *softcore* ici, les paramètres peuvent envoyer également via le contrôleur USB, permettant un paramétrage dynamique du système.

4.7.3 Résultats

Cette section présente les résultats après placement routage sur le FPGA. Les outils utilisés pour ces résultats sont :

- L'environnement Altera Quartus II version 13.0.1.232 pour le Cyclone III $EPC3120F780C7$. La configuration de l'outil est celle par défaut, sans aucune optimisation spécifique en fréquence ou en surface.
- Le compilateur CAPH version 2.6.3

L'implémentation matérielle est testée sur la base de données Daimler Pedestrian Benchmark Dataset [EG09]. Il est important de préciser que la base de test utilisée n'est pas corrélée avec la base d'images utilisée pour l'apprentissage (bien que Daimler propose sa propre base d'apprentissage). Ce type d'évaluation *cross-dataset* est assez peu rapporté dans la littérature. A notre connaissance, seul Enzweiler [EG09] et plus récemment Benenson [BOH+14] ont quantifié la perte de fiabilité due à un changement de bases de données. Utiliser la même base en apprentissage et test limite l'évaluation des capacités de généralisation du système (même lorsque les jeux de données sont indépendants). S'ils sont issus d'une même base, le contexte sera globalement le même, et certains paramètres comme la variation de l'illumination, la taille des personnes, leurs vêtements, les couleurs dans les rues, la structure de l'environnement (conditionnant les négatifs) peuvent induire un phénomène de sur-apprentissage (si le descripteur n'est pas assez robuste) qui ne sera pas détecté lors des tests, ce qui conduit à surestimer les capacités de détection et de généralisation. C'est pourquoi nous avons fait le choix de décorréler les bases d'apprentissage et de test. Les résultats de détection seront statistiquement moins bons, mais le portage du système dans un environnement inconnu devrait procurer des résultats similaires.

De la base Daimler, nous avons extrait une séquence de 35 images où un piéton traverse la chaussée. (séquence disponible dans l'annexe B). Cette sélection s'explique par le besoin de tester l'effet des paramètres et des étapes de l'algorithme dans des conditions relativement maîtrisées. Cette séquence peut être découpée en deux parties. La première partie (images $1-18$) propose des conditions favorables au bon fonctionnement de l'algorithme. Un piéton dont la taille est assez proche de la dimension de la fenêtre de détection traverse la chaussée. La seconde partie (images $19-30$) est beaucoup moins favorable, car le véhicule se rapproche du piéton et celui-ci devient plus grand que la taille de la fenêtre de détection.

La Fig. 4.28 illustre un exemple de détection sur une image de la séquence choisie.

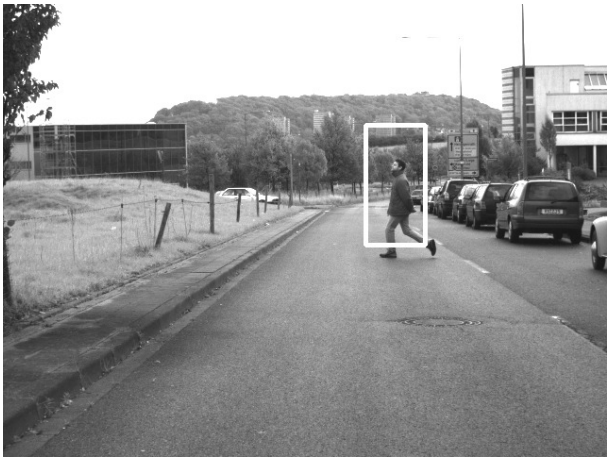


Fig. 4.28: Exemple de détection de l'implémentation matérielle générée par CAPH sur la base Daimler [EG09]

Ce résultat permet de valider la dernière étape de la méthodologie proposée, à savoir le passage d'une simulation VHDL RTL à une implémentation sur une plateforme réelle. Avant d'aborder la gestion des recouvrements, il est d'abord proposé une analyse des résultats d'implémentation du système actuel.

La table 4.2 montre les résultats après placement routage. Nous avons reporté dans cette table les informations suivantes :

- nombre d'éléments logiques spécifiques au FPGA choisi (LE),
- correspondance en terme de LUT (4 entrées) et de registres, pour une image d'entrée de résolution 1280×960 (résolution maximale avec notre plateforme).
- nombre de bits mémoires requis par chaque acteur,
- nombre de blocs SRAM M9K instanciés ,
- nombre de blocs arithmétiques DSP,
- fréquence maximale pour chaque partie de l'algorithme (Gradient, Histogramme, Normalisation, SVM).

Aussi les ressources utilisées par les FIFOs rebouclées pour la mémorisation de lignes localement à un acteur (comme l'acteur `vsum` de la Fig. 4.23) sont comptabilisées au sein de l'acteur (et non aux ressources liées aux FIFOs).

Tab. 4.2: Ressources matérielles requises par le réseau CAPH pour une résolution d'image de 1280×960

Opérations	Acteurs	Eléments Logiques	LUT	Reg	Mémoire (*bits*)	M9K	DSP	F_{max} (MHz)
Gradient	`rep8`	18	7	11	0	0	0	71.16
	8× `conv_33`	4422	3693	2430	195472	46	0	
	8× `abs`	246	240	113	0	0	0	
	4× `comp_init`	132	128	77	0	0	0	
	3× `comp`	112	108	58	0	0	0	
	3× `binning`	101	101	84	0	0	0	
Histogramme	8× `hsum`	735	728	400	0	0	0	79.24
	8× `vsum`	1927	1681	1302	20864	8	0	
Normalisation	8× `block_extrac`	2890	2435	2162	23760	8	0	21.2
	`norm_factor`	161	160	17	0	0	0	
	`rep32`	28	27	18	0	0	0	
	32× `l1norm`	9181	9165	560	0	0	0	
SVM Linéaire	`weight_distr`	3744	3455	326	0	0	0	59.61
	`dot_prod`	666	665	30	0	0	32	
	32× `skipdata`	1165	1148	640	0	0	0	
	`xwindowing`	113	112	66	0	0	0	
	`ywindowing`	158	158	89	0	0	0	
	`decision`	10	0	0	0	0	0	
FIFO		11663	8928	9935	0	0	0	
Total		**37163** (**31**%)	**32940**	**18315**	**240096**	**62/432**	**32/576**	**21.2**

Le réseau complet requiert 37163 éléments logiques soit environ 31% des ressources logiques disponibles. Sur ces 37163 éléments, 32940 LUT 4 entrées sont utilisées ainsi que 18315 registres. Du côté de la mémoire, le code généré alloue 240096 bits répartis dans 64 blocs *M9K*. Enfin l'implémentation utilise 32 blocs arithmétiques DSP. Voici l'analyse de ces résultats en détail :

- L'utilisation des convolutions standards de la bibliothèque CAPH permet l'instanciation automatique de blocs mémoires SRAM pour la mémorisation des lignes. Cependant, la mise en parallèle des 8 acteurs `conv33` engendre une duplication de la mémorisation du voisinage. En effet, chaque acteur va mémoriser en interne les deux mêmes lignes de pixels, créant des allocations de mémoires inutiles. Une solution efficace serait d'utiliser un acteur en charge de l'extraction de voisinage et un ensemble d'acteurs opérant chacun leur calcul indépendamment mais ceci se ferait au prix d'un routage accru. On étudiera au chapitre 6 le compromis associé aux choix d'implémentations des convolueurs.
- La formulation proposée pour le calcul de l'histogramme procure de bons résultats. La logique requise est de 332 LE par classe et la fréquence de fonctionnement avoisine les $80MHz$. La séparation en deux acteurs distincts a permis une optimisation des ressources. Si nous avions suivi la même approche que pour les convolutions lors de la formulation, c'est-à-dire mémoriser l'ensemble du voisinage nécessaire avant de calculer la valeur de l'histogramme comme cela avait été d'abord proposé dans [SBB14], nous aurions été obligé de mémoriser 8 fois plus de données par ligne et 7 lignes au lieu d'une par acteur (soit 448 fois plus de données au total).
- La partie critique de l'application est la normalisation. A elle seule, cette étape mobilise 44% des ressources totales. Ceci est principalement dû à l'utilisation des 32 acteurs `l1norm` en parallèle. Chaque acteur `l1norm` effectue un calcul de division, nécessitant environ 600 LE, ce qui explique les chiffres reportés pour cette étape dans la colonne correspondante de la table 4.2. Ceci dit, le point le plus critique reste la fréquence maxi-

male de fonctionnement résultante : 21MHz. On rappelle que la fréquence du réseau d'acteurs doit être deux fois supérieure à la fréquence du flux de pixels (Eq. 4.46). Une fréquence de fonctionnement de 21MHz implique une cadence de traitement de l'ordre de 10.5 images par seconde avec la résolution choisie. L'examen des chemins critiques de l'application, rapportés par l'outil Quartus (Fig. 4.29), montre que cette fréquence faible est due à la logique combinatoire nécessaire aux diviseurs.

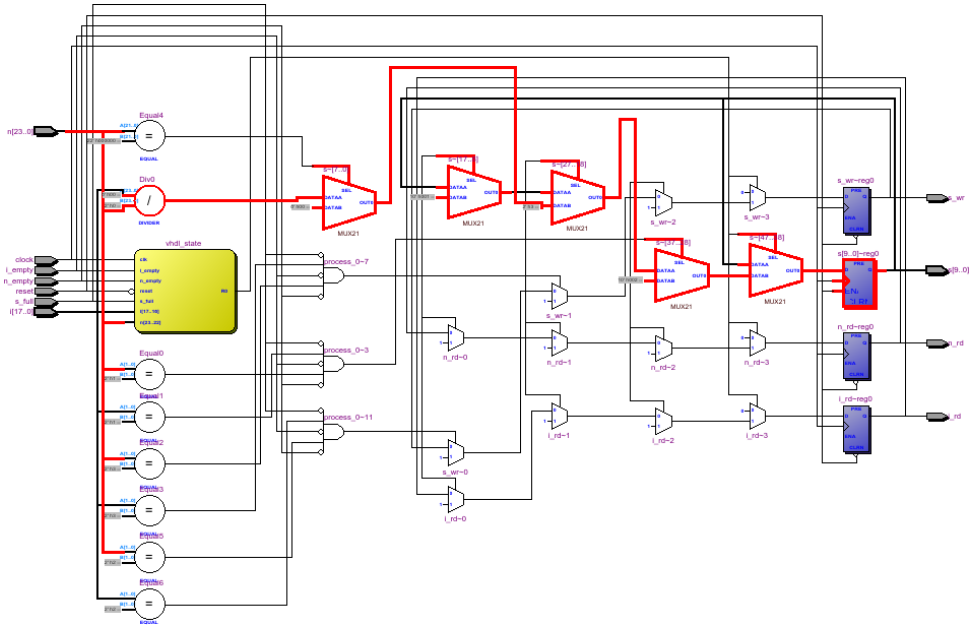


Fig. 4.29: Schéma RTL d'un acteur `l1_norm` avec le chemin critique (en rouge) rapporté par l'outil Quartus

Avec une implémentation HDL développée "à la main", il est possible de pallier ce problème en utilisant pour l'opération de division un composant optimisé, en général fourni par le fabricant de FPGA (LPM_DIVIDE chez Altera, LogiCORE chez Xilinx par exemple). Le développeur peut alors optimiser la fréquence de fonctionnement en ajustant le nombre d'étages de pipeline, quitte à augmenter la latence de l'opération dans ce cas.

Lorsque le code HDL est généré par un outil de HLS comme CAPH, cette approche est plus difficile à mettre en œuvre. CAPH offre la possibilité d'insérer des acteurs écrits "à la main" dans le code généré (il suffit que leur interface soit conforme au modèle d'exécution). Dans ce cas, cela revient à faire le même travail que ferait un développeur HDL concernant les diviseurs. Cette approche biaise toutefois les résultats, l'intérêt premier d'un outil de HLS étant justement d'éviter au développeur de faire du développement HDL. Une seconde solution sera proposée en perspective de recherche dans les conclusions de cette thèse.

- Les résultats concernant l'implémentation de la partie SVM sont mitigés. Le résultat intéressant concerne l'acteur de produit scalaire (`dot_product`) qui instancie automatiquement des blocs arithmétiques DSP, preuve que l'outil de synthèse interprète correctement le code généré. La présence des acteurs `skipdata` est discutable. On pourrait se passer de cet acteur en fusionnant les acteurs `block_extrac` et `skipdata`. L'acteur de distribution de poids `weight_distr` utilise également beaucoup de logique. La majeure partie de celle-ci sert à mémoriser les poids du modèle qui sont dans cette version encodés statiquement. Il est possible d'utiliser les mémoires externes de la plateforme DreamCam avec l'utilisation d'un port afin de déplacer le stockage du modèle, et ainsi économiser des ressources logiques.
- Les FIFOs d'inter-connexion entre acteurs représentent environ 31% du total des ressources utilisées. Le développeur a donc intérêt à limiter le nombre d'acteurs s'il désire

optimiser l'usage de ces ressources. Cependant, limiter le nombre d'acteurs implique des acteurs plus complexes, ce qui en retour conduit à augmenter la logique combinatoire au sein de chaque acteur, et donc à diminuer la fréquence. Nous traiterons en profondeur ce compromis sur un exemple de convolution dans le chapitre 6.

- La latence globale du réseau est de 164443 cycles d'activation, pour une image de résolution 1280×960 pixels, soit 8.02 ms pour une fréquence de 20.1 MHz. Cette valeur, obtenue par une mesure en simulation VHDL RTL, est définie comme le nombre de cycles d'horloge pixels entre l'arrivée du premier pixel à l'entrée du réseau et le premier résultat de classification produit en sortie. Exprimé par rapport à la résolution de l'image, le système produit le premier résultat de sortie alors que seulement 13% des pixels de l'image ont été fourni par le capteur d'image.

Le dernier tableau 4.3 présente les résultats pour l'ensemble du dispositif déployé sur la plateforme. Le coût de l'encapsulation des signaux de contrôle dans les jetons (blocs insertion/suppression de jetons) permettant le passage de l'interface standard (`fv`, `dv`, `data`) à la représentation utilisée dans CAPH est assez faible (250 LE et un bloc mémoire SRAM). Le composant USB requiert 1200 éléments logiques avec 11 mémoires qui permettent la synchronisation de la caméra avec une application externe. Au total, le déploiement requiert 38624 LE (regroupant x LUT-4 et x registres) soit 40% des ressources logiques totales, 74/432 blocs mémoires et 32/576 blocs arithmétiques

Tab. 4.3: Ressources matérielles totales

IP	Eléments Logiques	LUT	Reg	Mémoire (*bits*)	M9K	DSP
CAPH	37163	32940	18315	240096	62	32
Insertion jeton	162	151	116	8192	1	0
Suppression jeton	99	68	69	0	0	0
Comm. USB	1200	799	946	55488	11	0
Total	48073 (41%)	43743	18647	240096	62/432	32/576

4.8 Gestion du recouvrement

La version du système de détection décrit à la section précédente est essentiellement une preuve de concept. En effet, le système décrit ne gère pas les recouvrements entre deux fenêtres successives. Par conséquent, les piétons se trouvant à l'intersection de deux ou plusieurs fenêtres ne seront pas détectés.

En pratique, cet inconvénient est rédhibitoire. Le problème majeur n'est pas lié à la formulation flot de données car la gestion du recouvrement peut se faire simplement par une mise en parallèle de 105 unités SVM du type de celle décrite ci-dessus (soit autant d'unités parallèles que de recouvrements possibles dans une fenêtre). Malheureusement, les ressources requises par ces 105 unités risquent alors de dépasser celles disponibles sur notre plateforme.

La table 4.4 montre les ressources requises sur la cible DreamCam pour plusieurs configurations différentes : 1, 7, 10 et 12 unités SVM. Chaque configuration intègre l'extraction de descripteurs HOG avec un nombre d'unités SVM différent, et toutes les unités sont connectées en parallèle. Avec seulement douze unités, soit environ 11% du nombre d'unités SVM nécessaires à gestion complète du recouvrement, les capacités maximales de la plateforme sont atteintes.

Tab. 4.4: Ressources matérielles totales requises suivant le nombre de recouvrements pour le Cyclone III

Nombre d'unités SVM	Elements Logiques	Mémoire (bits)	M9K	DSP
1	37163 (31%)	240096	62	32
7 (recouvr. hor)	82673 (69%)	240096	62	224
10	101149 (85%)	240096	62	320
12	116008 (97%)	240096	62	384

Des mesures ont donc été effectuées sur un FPGA offrant plus de ressources (Stratix V). L'architecture d'un Stratix V est légèrement différente du celle du Cyclone III, les éléments logiques sont basés sur les nouvelles architectures de blocs logiques ALM. Toutefois, selon le constructeur Altera [alta], un ALM est équivalent à 2.5 LE. La table 4.5 montre les résultats obtenus sur le Stratix V.

Tab. 4.5: Ressources matérielles totales requises suivant le nombre de recouvrements pour le Stratix V

Nombre d'unités SVM	Logic (ALM)	Mémoire (bits)	*M20K*	*DSP*
1	18525 (7%)	240096	32	42
2	21620 (8%)	240096	32	72
4	27773 (11%)	240096	32	132
7 (recouvr. hor)	40884 (16%)	240096	32	224
14(recouvr. hor + 2 vert)	63212 (24%)	240096	32	432
28(recouvr. hor + 4 vert)	109272 (42%)	240096	32	852
56	223106 (85%)	240096	32	1692

Avec la moité des unités SVM requises, le FPGA Stratix V atteint également ses limites. Toutefois, les différentes expérimentations montrent que les ressources logiques ainsi que le nombre de blocs DSPs évoluent linéairement en fonction du nombre d'unités SVM instanciés (Fig. 4.30). Une interpolation linéaire permet donc d'estimer les ressources nécessaires pour la totalité de l'application.

Pour 105 unités SVM, cette estimation donne 400220 ALM et 3161 blocs DSP sur le Stratix. Transposée à la plateforme DreamCam, un million de blocs logiques LE seraient nécessaires, soit respectivement 8 fois les ressources logiques et 5.4 fois les blocs DSP disponibles. On est donc assez loin de ce qui est physiquement implantable sur cette plateforme

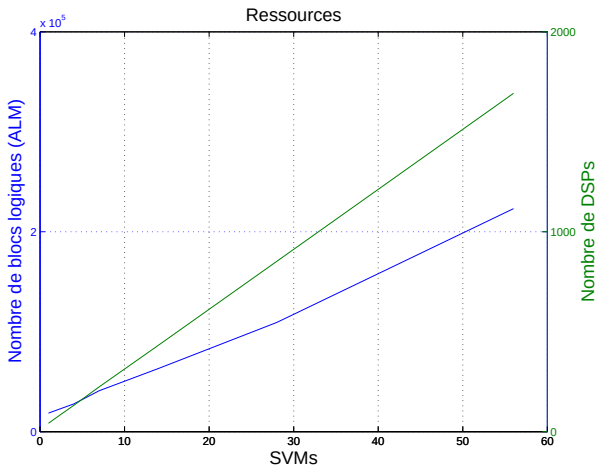


Fig. 4.30: Evolution des ressources matérielles en fonction du nombre d'unités SVMs

La solution à ce problème passe donc probablement par une reformulation conséquente de la partie SVM afin de l'adapter aux contraintes matérielles imposées par la plateforme. Une telle reformulation pourrait par exemple s'inspirer de la technique décrite dans [MBQ+16], proposée dans le cadre d'une implémentation purement HDL, c'est-à-dire sans le recours à un outil de HLS, et qui exploite les redondances entre SVM afin de minimiser les ressources matérielles requises.

Afin que les contingences matérielles ne nous empêchent pas d'évaluer les performances de notre implémentation en termes de détection, nous avons choisi d'effectuer cette évaluation avec l'implémentation logicielle de référence (ILR) telle que décrite à la Sec. 4.4 et non avec le code produit par le compilateur CAPH. Cette démarche se justifie dans la mesure où, comme on a vu à la Sec. 4.6, les performances des deux implémentations sont très proches. Les résultats obtenus sur la séquence Daimler, sont illustrés sur la Fig. 4.31

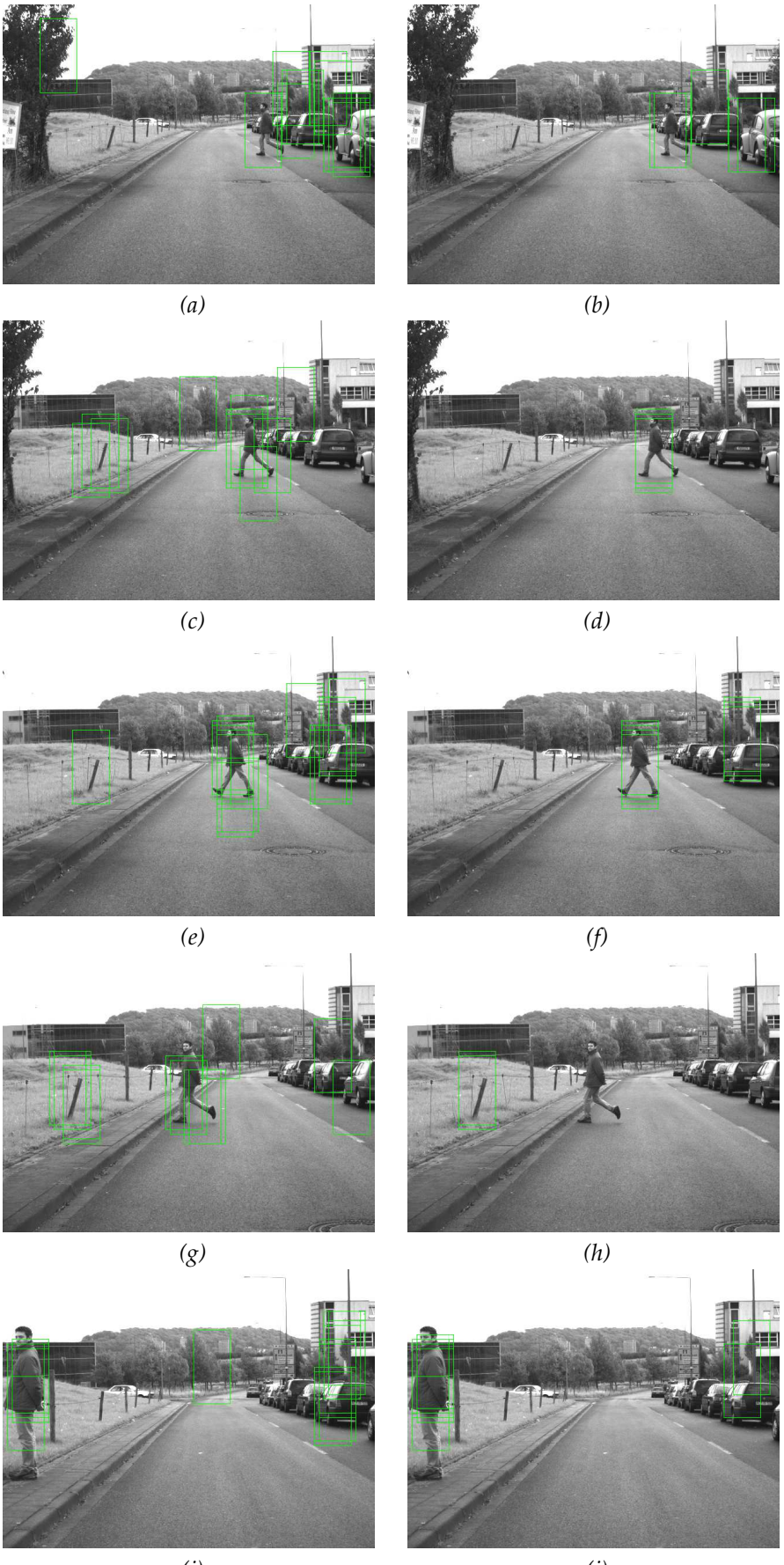


Fig. 4.31: Images de détection de piétons sur la base de données Daimler. Le côté gauche représente les sorties obtenues par l'implémentation logicielle de référence. Le côté droit a été obtenu en utilisant OpenCV

La Fig. 4.32 décrit l'évolution de l'AUC, où une détection est considérée bonne si le recouvrement entre la vérité terrain et la fenêtre de détection est supérieure à 50% (Eq. 3.40).

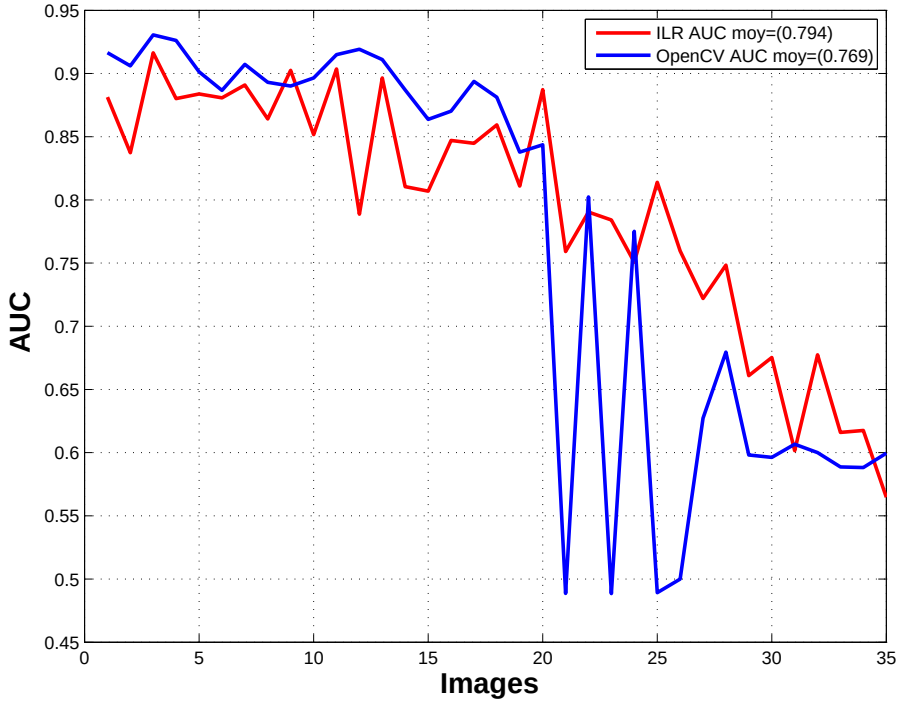


Fig. 4.32: Evaluation des AUC obtenues avec notre ILR d'une part et OpenCV d'autre part sur la séquence d'images extraite de la base Daimler

Sur la première partie de la séquence (images $1-18$), l'implémentation OpenCV se comporte mieux que l'implémentation logicielle de référence, avec une AUC moyenne de 0.90. A noter que cette valeur est plus faible que celle obtenue lors des essais sur la base INRIA (0.99 sur la Fig. 4.20), démontrant la nécessité du changement de base pour tester l'implémentation dans des conditions difficiles, et ainsi évaluer les capacités de généralisation du système au monde réel. Notre implémentation conduit en moyenne à une AUC de 0.86. Sur la seconde partie (images $19-30$), les résultats s'inversent. L'implémentation OpenCV ne détecte pas le piéton une image sur deux car celui-ci sort des conditions optimales de fonctionnement de l'algorithme alors que notre implémentation est moins sensible à ces nouvelles conditions (voir Figs. 4.31g et 4.31h). Ces résultats peuvent s'expliquer par la qualité des espaces de description. Le descripteur OpenCV est plus discriminant que le descripteur reformulé, procurant de meilleurs résultats dans des conditions optimales de fonctionnement mais cette forte discrimination devient un inconvénient dans le cas où les images d'entrées ne correspondent plus exactement aux conditions utilisées lors de l'apprentissage. L'AUC moyenne obtenue avec l'implémentation OpenCV sur la seconde partie est de 0.63 alors que celle obtenue avec notre implémentation est de 0.72. Théoriquement, il est possible d'ajuster les seuils de détection/rejet (avec la valeur du biais) afin de limiter les effets observés sur l'implémentation OpenCV. Toutefois, cette technique revient à paramétrer le système en fonction de son implémentation, des approximations et même des bases utilisées. Sur l'ensemble de la séquence, L'AUC moyenne de la séquence complète est de 0.769 pour OpenCV et de 0.794 pour notre implémentation. Avec les conditions de tests utilisées, les résultats de détection sont donc équivalents, ce qui peut paraître surprenant au vu des approximations effectuées. Cependant les résultats sont beaucoup moins fiables que prévu lorsque les conditions de fonctionnement ne sont plus idéales. Afin d'améliorer la fiabilité du système, on propose donc d'étudier un système de filtrage neuro-inspiré permettant de prendre en compte la cohérence/continuité spatiale et temporelle sur une séquence d'images. Cette amélioration est décrite dans la section suivante.

4.9 Amélioration de la fiabilité de détection par filtrage neuro-inspiré

On a pu constater que la variabilité des images d'une part et des approximations sur lesquelles repose notre implémentation d'autre part peuvent conduire à une détérioration des performances finales de détection. On décrit dans cette section un mécanisme de post-filtrage visant à améliorer les performances. Le système, tel que décrit jusqu'à présent, produit en sortie des cartes de détection complètes (Eq. 3.29), où un degré de confiance peut être associé à chaque fenêtre de détection dans l'image [13], permettant d'appliquer directement sur les sorties un champ neuronal dynamique (Dynamic Neural Field (DNF)). Les DNFs peuvent être décrits comme une approche bio-inspirée et massivement parallèle permettant d'exécuter des estimations probabilistes au niveau image (modélisant le fonctionnement du cortex visuel à une échelle mésoscopique), avec une formulation différente mais fondée sur des principes similaires au filtrage Bayésien, ou du suivi de zones d'intérêt (Kalman ou filtrage à particules). Nous pouvons espérer que les sorties de la SVM garantiront une cohérence temporelle entre images successives et spatiale dans l'image (continuité de mouvement). L'addition d'un modèle de mouvement des cibles devrait également permettre d'améliorer la qualité de détection tout en réduisant la quantité de faux positifs du filtrage. La suite du paragraphe va introduire les équations classiques (stationnaires), le lecteur peut se référer à [QG11] pour les détails sur la version prédictive des DNF intégrant des modèles de mouvement linéaire. L'équation classique d'un DNF modélise les variations du potentiel moyen d'une population neuronale dans des colonnes corticales, formant un champ 2D du cortex visuel. Le potentiel à la position $\vec{x}$ et à l'instant t dans ce type de champs est définit par $u(\vec{x},t)$, où les stimuli d'entrées sont notés $y(\vec{x},t)$, correspondant aux sorties de la SVM sur l'image entière pour l'image au temps t. Une version simplifiée de l'équation du DNF à une couche est donnée par :

$$\tau \frac{\partial u(\vec{x},t)}{\partial t} = -u(\vec{x},t) + c(\vec{x},t) + y(\vec{x},t) \tag{4.47}$$

où c est le terme de compétition latérale décidant de la sélection des cibles, et défini par :

$$c(\vec{x},t) = \int_{x' \in M} w(\vec{x},\vec{x'}) \sigma(u(\vec{x'},t)) d\vec{x'} \tag{4.48}$$

où σ est une fonction d'activation non linéaire (classiquement une sigmoïde), et $w(\vec{x},\vec{x'})$ est une fonction de poids, représentant les connexions latérales et satisfaisant l'Eq. 4.49. L'écart type de la composante excitatrice a contrôle la taille attendue de la cible, permettant un suivi pour une échelle donnée, mais avec une certaine robustesse aux variations d'échelles, alors que l'écart type b de la composante inhibitrice détermine la distance minimale entre deux cibles acquises. A et B sont respectivement les amplitudes des composantes excitatrices et inhibitrices du noyau. Ils contrôlent donc l'influence de la compétition latérale relativement aux données d'entrées bruitées (ici les sorties SVM). En terme probabiliste, ils pondèrent la distribution de probabilité à priori de la position de la cible avec les observations de l'image en cours.

$$w(\vec{x},\vec{x'}) = Ae^{-\frac{|\vec{x}-\vec{x'}|^2}{a^2}} - Be^{-\frac{|\vec{x}-\vec{x'}|^2}{b^2}} \tag{4.49}$$

Afin de rendre possible la simulation du modèle DNF, des discrétisations spatiales (basées sur une grille) et temporelles (schéma d'intégration d'Euler) sont faites. Utilisant une formulation matricielle, l'Eq. 4.47 devient :

$$U(t+dt) = \left(1 - \frac{dt}{\tau}\right) U(t) + \frac{dt}{\tau}\left(C + \overline{Y}(t)\right) \tag{4.50}$$

où $\overline{Y}$ est obtenu par seuillage de Y, puis normalisation dans l'intervalle $[0,1]$. Cette opération a pour effet de mettre toutes les valeurs entre $-\mu$ et 0 où μ correspond à la tolérance du côté

13. Obtenue par la suppression de la fonction *signe* dans l'équation de la SVM 3.30

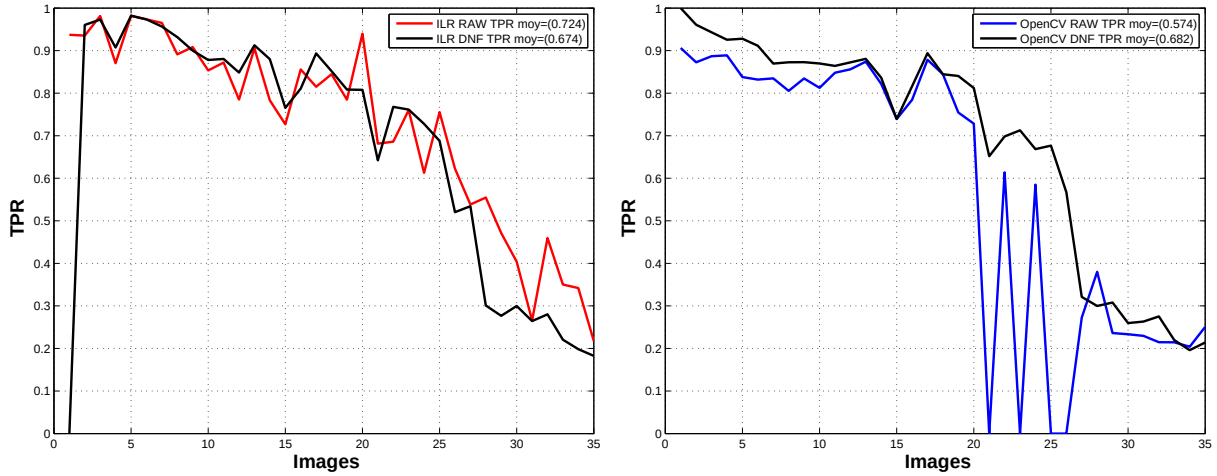
négatif du séparateur de la classification. L'augmentation de μ permet de retenir un nombre de faux positifs plus important. Comme les DNF sont spécifiquement conçus pour gérer des rapports signal sur bruit faibles et filtrer efficacement les faux positifs, choisir $\mu > 0$ revient à augmenter le TPR. Néanmoins, nous avons choisi de garder ce paramètre μ à 0 pour que la comparaison des résultats soient justes avec les résultats bruts de sortie du système. Le dernier terme C, représentant la compétition spatiale, s'écrit alors :

$$C = W^{+} * \sigma(U(t)) - B \times \Sigma(U(t)) \tag{4.51}$$

où W^{+} est une version matricielle de la part excitatrice du noyau de la fonction w (Eq. 4.49) et $\Sigma(U(t))$ est la somme des éléments de la matrice $\sigma(U(t))$. Cette approximation est possible en définissant $b = +\infty$, ciblant un seul piéton à chaque image, et limitant aussi la convolution à un noyau d'excitation réduit (5×9).

L'implémentation du système de filtrage est actuellement logicielle mais de récentes recherches ont montré que son déploiement était possible sur une architecture matérielle [VThG15]. On continue de s'intéresser à la même séquence d'images que précédemment, répondant aux conditions de fonctionnement du système de filtrage (un seul piéton). Nous avons rapporté l'impact du filtrage DNF sur plusieurs métriques : TPR, FPR et AUC et pour chaque algorithme (OpenCV et CAPH). Pour chaque métrique et chaque algorithme, les valeurs moyennes sont rapportées sur les graphiques.

La première métrique mesurée concerne le taux de bonnes détections (Fig. 4.33). Le DNF a un excellent impact sur le TPR OpenCV (Fig. 4.33b), qui est supérieur (0.68) à celui des résultats bruts (0.57). Le système atténue notamment les pertes de performances liées à la disparition du piéton d'une image à l'autre (images $19 - 26$). L'impact sur les bonnes détections provenant du système matériel est moins important. La Fig. 4.33a montre que le DNF a un besoin d'une image d'initialisation. Sur la première partie de la séquence, le filtrage maintient le TPR à son niveau alors que sur la deuxième partie de la séquence, les résultats filtrés sur les bonnes détections sont statistiquement un peu moins bons. Ces résultats sont satisfaisants car le système de filtrage a pour objectif principal de filtrer les faux positifs sans détériorer le taux de vrais positifs. Les données générées par l'implémentation sont plus bruitées, ce qui explique en partie que le système soit plus efficace sur les données OpenCV.

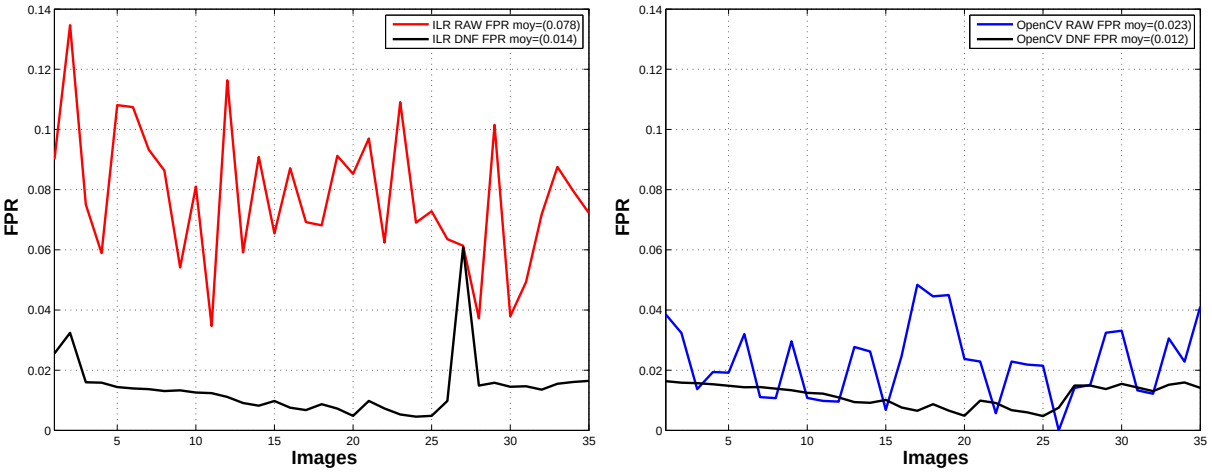


(a) CAPH TPR *(b)* OpenCV TPR

Fig. 4.33: Impact du filtrage DNF sur le taux de bonnes détections

La Fig. 4.34 illustre l'impact du filtrage sur les faux positifs. Dans le cas de l'implémentation matérielle (Fig. 4.34a), le taux moyen de faux positifs sur la séquence complète passe de 7.8% à 1.4%. Dans le cas de l'implémentation OpenCV (Fig. 4.34b), le taux moyen de faux positifs passe de 2.3% à 1.2%. En sortie de filtrage, les deux implémentations produisent un taux de fausses alarmes équivalents. L'impact du filtrage DNF est donc très concluant sur les faux

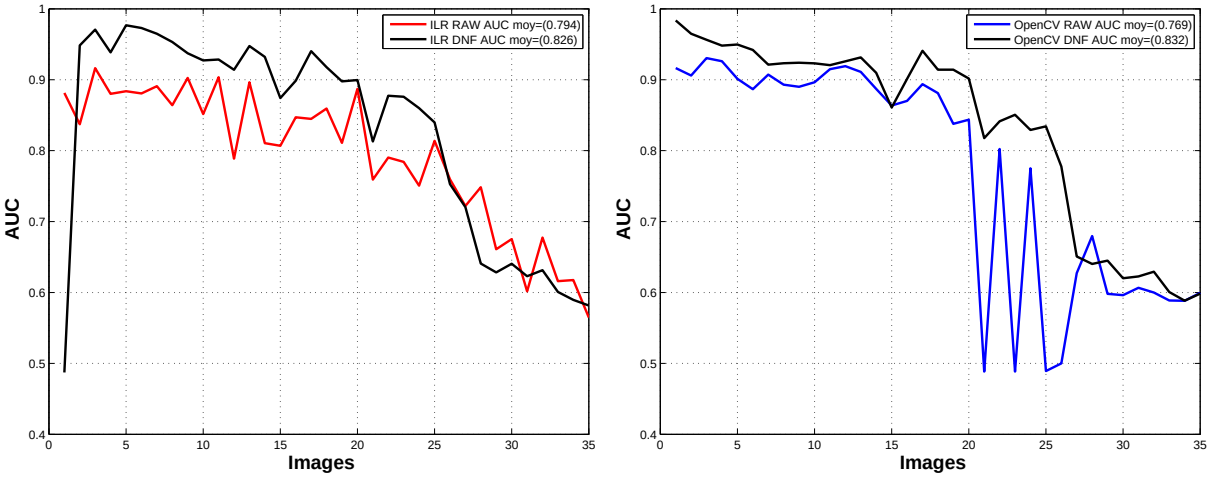
positifs car il permet de compenser les pertes de précision introduites par les approximations faites dans la formulation du descripteur HOG matériel.



(a) CAPH FPR *(b)* OpenCV FPR

Fig. 4.34: Impact du filtrage DNF sur le taux de faux positifs

D'un point de vue global (Fig. 4.35), le système de filtrage proposé améliore la fiabilité du système dans les deux cas mais sa contribution est différente. Pour le cas de l'implémentation matérielle, le DNF réduit fortement le taux de faux positifs, augmentant ainsi la valeur de l'AUC mais réduit aussi légèrement le TPR. Dans le cas de l'implémentation OpenCV, le système réduit assez peu les faux positifs car il y en a peu dans les données brutes mais améliore le taux de bonnes détections, notamment en lissant les pertes temporaires du piéton dans la séquence. Au final, la métrique AUC moyenne est équivalente entre les deux implémentations (0.832 et 0.826), ce qui signifie que le système de filtrage permet de compenser les approximations faites lors de la formulation du descripteur HOG flot de données.



(a) CAPH AUC *(b)* OpenCV AUC

Fig. 4.35: Impact du filtrage DNF sur la qualité globale de détection

Avant de passer aux conclusions de ce chapitre, il est proposé sur la Fig. 4.36 les résultats de détection avant et après filtrage sur quelques images de la séquence (La séquence complète est disponible dans l'annexe. B).

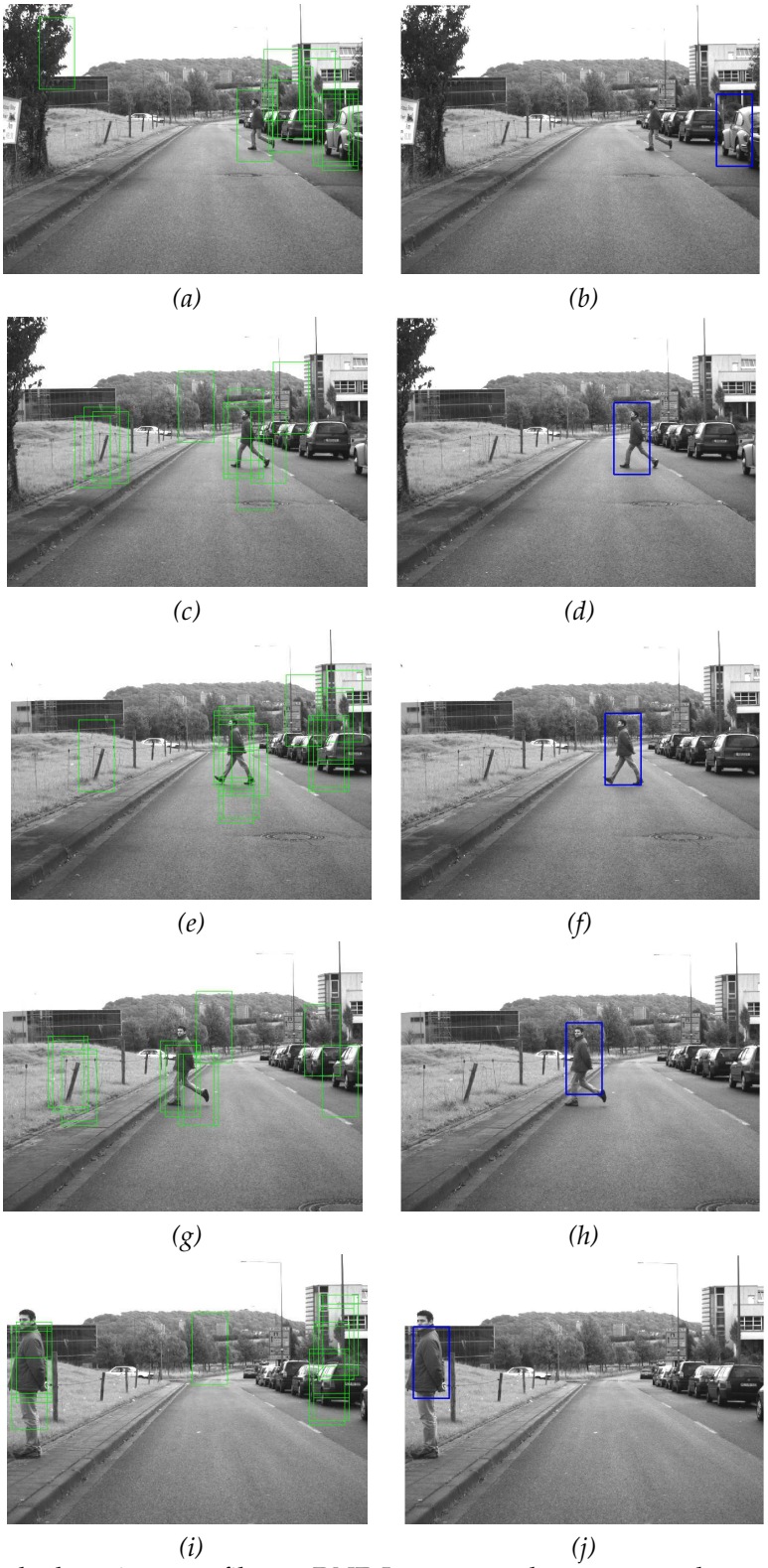


Fig. 4.36: Résultats de détection avec filtrage DNF. Le côté gauche représente les résultat bruts fournis par la formulation CAPH. Le côté droit représente les résultats après le système de filtrage. Le système a eu besoin dans ce cas d'une image d'initialisation mais produit ensuite de bons résultats de détection.

4.10 Conclusions

On a proposé dans ce chapitre une formulation flot de données d'un algorithme d'extraction de descripteurs HOG et d'un système de classification par SVM. Cette formulation, purement fonctionnelle, exploite des techniques originales comme la méthode *HOG-Dot* pour l'extraction des gradients ou bien le calcul des histogrammes en parallèle. L'impact des approximations en virgule fixe utilisées dans les opérations arithmétiques nécessaires à l'implémentation d'un algorithme sur une cible FPGA a été quantifié. Les mesures ont montré qu'une dynamique faible (inférieur à 8 bits) permettait d'obtenir de bons résultats de classification. Enfin, une transcription en CAPH a été effectuée à partir de cette formulation. Cette transcription a permis de générer une implémentation matérielle automatiquement, sans utiliser de langage de description matérielle.

Le déploiement sur une plateforme réelle a montré qu'un outil de HLS basé sur le modèle flot de données, tel que CAPH, permet une implémentation efficace d'algorithmes complexes sur une cible FPGA. Le code généré, de manière totalement automatique, permet un fonctionnement à 10 images par secondes avec une résolution 1280×960.

Le travail mené a toutefois mis en évidence certaines limitations de la méthode. On a montré par exemple que la présence des diviseurs introduit une limite significative de la fréquence maximale de fonctionnement, sauf à optimiser manuellement le code généré. On a aussi montré que la gestion des recouvrements entre les fenêtres de détection, nécessaire pour la fiabilité de détection sur des images réelles ne peut se faire sans une reformulation de l'algorithme tenant compte du caractère limité des ressources offertes part la plateforme cible.

L'évaluation du système a été faite sur des images différentes de celles issues de la base d'apprentissage, ce qui a permis de tester le système dans des conditions plus difficiles. Les résultats bruts, même s'ils sont conformes aux mesures statistiques effectuées sur les jeux de données peuvent paraître insuffisants comparés à des observations que ferait un humain. L'implémentation OpenCV "rate" des piétons alors que notre implémentation introduit du bruit (*i.e.* des faux positifs). Un des avantages du système est qu'il fournit pour chaque image, l'ensemble des détections. A partir de ces informations, nous avons proposé une méthode de filtrage exploitant simultanément les corrélations spatiales et temporelles entre les réponses du système. Cette approche permet d'améliorer de manière significative les performances.

Une limitation de l'application concerne la gestion de l'espace d'échelles. Nous avons observé sur la séquence de test que les résultats de détection chutent lorsque la taille du piéton ne correspond plus à celle de la fenêtre de détection. L'approche classique de la littérature pour la gestion de l'espace d'échelles consiste en une pyramide gaussienne [BA83]. La pyramide gaussienne produit un ensemble d'images sous-échantillonnées sur lesquelles on applique l'ensemble des traitements HOG-SVM. Cette méthode a l'avantage d'être aisément parallélisable. Le système de filtrage DNF devra être étendu pour prendre en compte l'ensemble des échelles, mais les résultats de détection devraient en être encore améliorés.

Les tests effectués afin de quantifier la qualité du système de détection ont été appliqués à un échantillon de la base complète Daimler. Ce choix a été fait essentiellement dans le but de *maîtriser* un certain nombre de paramètres lors des tests. Cependant, une évaluation statistique des performances sur l'intégralité de la base est à envisager dans le futur afin d'obtenir des résultats plus précis sur la fiabilité de notre système.

Le système de filtrage DNF peut également évoluer avec le support de cibles multiples. Actuellement la compétition entre les fenêtres de détection est évaluée de manière globale. Les expérimentations ont fournies de bons résultats car les conditions de tests étaient favorables mais une séquence d'images plus élaborées, avec de nombreuses cibles, limiterait le bon fonctionnement du filtrage.

Seul le descripteur HOG a été utilisé ici pour la détection. La présentation des descripteurs dans le chapitre 3 a montré que les méthodes récentes exploitent un ensemble de descripteurs,

souvent agrégé avant la classification. L'extension du système avec l'ajout d'autres descripteurs, tel qu'un LBP, permettrait d'améliorer la qualité de détection.

5

Reformulation flot de données et implantation : application aux réseaux de neurones convolutionnels (CNN)

L'objectif de ce chapitre est de démontrer que la méthode proposée dans le cadre d'un système d'apprentissage à base de SVM peut être transposée à un autre type de systèmes, en particulier les CNN. La communauté de la vision, actuellement active dans ce domaine de recherche, explore l'impact de différentes configurations des réseaux CNN pour un problème donné. Ces configurations diffèrent en particulier sur la profondeur du réseau (le nombre de couches) ou bien le choix des fonctions d'activation effectuées sur chaque couche.

Dans ce contexte, la transcription automatique d'un réseau obtenu avec un outil logiciel (Caffe, Torch, Matlab) vers une implémentation sur cible FPGA permettrait un déploiement rapide et systématique. Pour cela, on propose le modèle flot de données comme intermédiaire entre la description du réseau mathématique et son implémentation matérielle. Associée au compilateur CAPH pour la génération du code VHDL, cette approche permet d'assurer que la transformation vers l'implémentation matérielle d'un réseau CNN sera efficace. Ce chapitre donnera une première formulation fonctionnelle d'un réseau de neurones convolutionnel. Ensuite, les éléments constituant la partie de classification d'un réseau CNN seront formulés selon le modèle flot de données [1]. Enfin, des exemples de réseaux seront transcrits en CAPH.

5.1 CNN et FPGA

Actuellement, l'étude des réseaux de neurones convolutionnels est un domaine de recherche incontournable dans la communauté de la vision. Dans ce contexte, l'intérêt des FPGA est immédiat dans la mesure où le calcul des convolutions $2D$ discrètes, base des premières couches d'un réseau, se prête naturellement à une implémentation matérielle. Les premières publications sur le sujet datent de 2009 avec Farabet qui propose la première architecture matérielle spécialisée pour les réseaux CNN avec l'architecture CNP [FPHL09], présentée dans la Fig. 5.1.

1. La partie apprentissage est effectuée hors-ligne

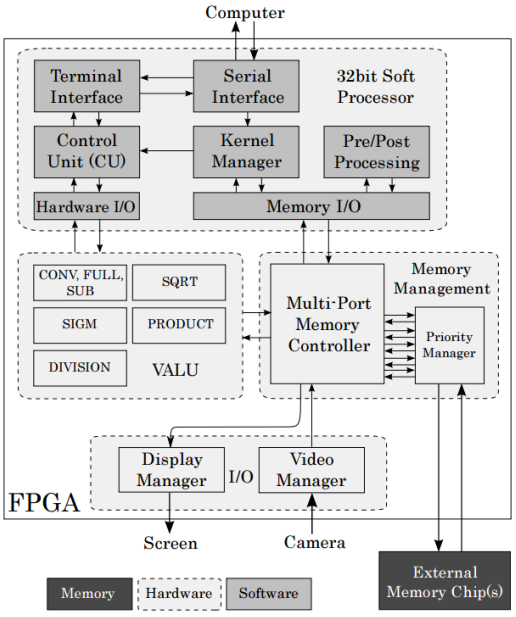


Fig. 5.1: Architecture du CNP de Farabet, source [FPHL09]

Cette architecture contient une unité de contrôle (CU), un pipeline parallèle d'unité de calcul arithmétique et logique (VALU), une unité de contrôle des I/O et une interface mémoire. L'unité de contrôle CU est un *softcore* 32 bits basé sur une architecture PowerPC, utilisé pour ordonnancer les calculs effectués sur l'unité VALU. Cet unité VALU implémente des opérateurs spécifiques aux réseaux de neurones convolutionnels, comme présenté dans le chapitre 3 (Convolutions 2D, sous-échantillonnage, sigmoïde, racine carrée, division). Les communications avec le NCP sont assurées par deux contrôleurs matériels DVI, gérant respectivement l'acquisition de données vidéo et l'affichage de données. Le dernier élément est le contrôleur mémoire en charge de multiplexer les flux de données entre les différents éléments. Cette architecture a été implantée dans un FPGA Xilinx Spartan 3 ou une application de détection de visage (résolution QVGA) est effectuée par un réseau LeNet-5 [LBBH98].

Par la suite, Farabet a également proposé sur une seconde architecture, cette fois basée sur le modèle flot de données [FMC+11, Far13]. La figure 5.2 montre cette architecture, appelée NeuFlow, constituée d'une grille bidimensionnelle d'éléments de traitement configurables (Processing tiles PT) reliés à travers un réseau de multiplexeurs. L'interface avec la mémoire externe et la gestion les transferts internes à la grille sont gérées par un DMA. Le système est supervisé par une unité de contrôle implanté dans un processeur et le système est reconfigurable dynamiquement.

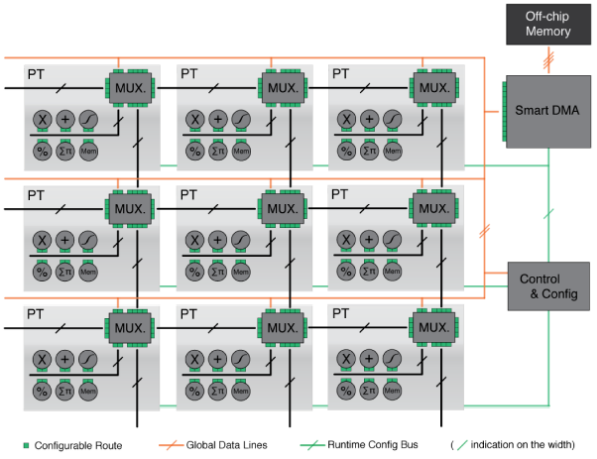


Fig. 5.2: Architecture Flot de données de Farabet pour les CNN

En dehors des ces travaux, les approches existantes dans la littérature utilisent des FPGAs comme simple coprocesseur de calculs, de la même manière qu'un GPU. Chakraddhar a proposé une architecture de coprocesseur dynamiquement configurable [CSJC10]. La configuration dynamique dans cette approche est synonyme de multiplexage dynamique entre des unités statiques. [2] On retrouve également cette approche coprocesseur dans [SJC+09] ou plus récemment Ovtcharov [ORK+15] qui exploite un serveur de calculs (dual-socket Xeon server) équipé d'un ensemble de cartes de la société Catapult où chacune intègre un FPGA Stratix V et 8GB de mémoire DDR3.

2. A ne pas confondre avec la reconfiguration dynamique des FPGAs récents, qui recharge une partie du bitstream localement afin de modifier le traitement.

5.2 Formulation flot de données d'un réseau CNN

Comme dans le chapitre 4, on commence par donner une description flot de données, sous la forme d'un réseau d'acteurs dont le comportement est spécifié de manière purement fonctionnelle, d'une application. Cette application est une application de reconnaissance de caractères dans des images 32×32, réalisée à l'aide d'un réseau de type LeNet-5 décrit à la Sec. 3.7 et sur la Fig. 5.3.

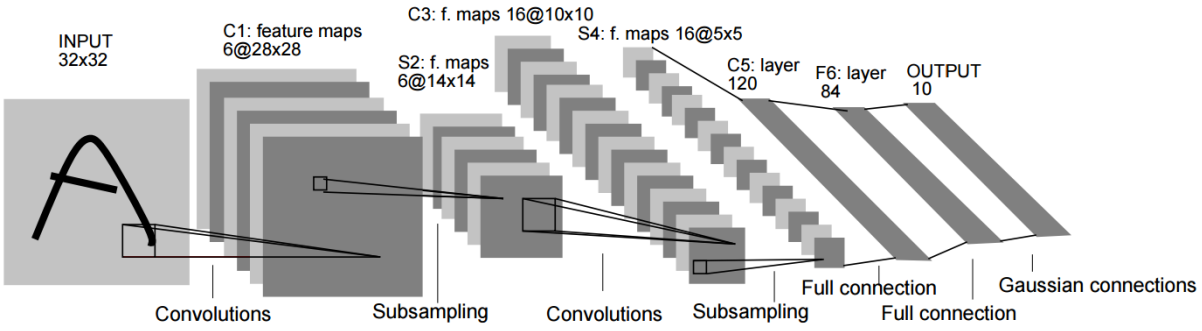


Fig. 5.3: Architecture d'un réseau CNN LeNet-5 pour la reconnaissance de caractères, source [LBBH98]

L'image d'entrée est de résolution 32×32 pixels en niveau de gris. Conformément au modèle décrit à la section 3.7, l'application peut se décomposer en sept couches :

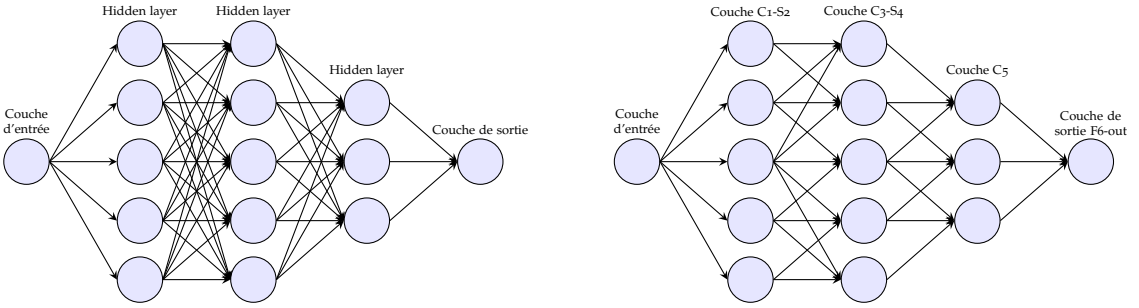
- couche $C1$: couche de convolution, composée de 6 cartes de primitives et de filtres 5×5.
- couche $S1$: couche de sous-échantillonnage, composée de 6 cartes de primitives et de filtres 2×2.
- couche $C3$: couche de convolution, composée de 16 cartes de primitives et de filtres 5×5.
- couche $S4$: couche de sous-échantillonnage, composée de 16 cartes de primitives et de filtres 2×2.
- couche $C5$: couche de convolution, composée de 120 cartes de primitives et de filtres 5×5.
- couche $F6$: couche complètement connectée.
- couche OUTPUT : couche de sortie. Chaque élément calcule la distance euclidienne entre le vecteur d'entrée et les paramètres appris.

Dans la suite, on va décrire chaque couche sous la forme d'une collection d'acteurs flot de données opérant sur des données structurées. On représente pour cela une carte de primitives f sous la forme d'une *image* de n lignes de m pixels :

$$f = \langle \langle f_{1,1} \ \cdots \ f_{1,m} \rangle \cdots \langle f_{n,1} \ \cdots \ f_{n,m} \rangle \rangle \tag{5.1}$$

5.2.1 Couches de convolution

Contrairement au réseau de neurones artificiels (ANN [MRG+86]), les réseaux CNN ont une connectivité partielle d'une couche de convolution à l'autre. Cette connectivité permet de limiter le nombre de paramètres à apprendre.



(a) Réseau de neurones classiques (ANN) *(b)* Réseau de neurones convolutionnels (CNN)

Fig. 5.4: Comparaison de la connectivité des réseaux de neurones ANN et CNN

Comme le montre la Fig. 5.4b, chaque neurone au sein d'une couche de convolution est connecté à un ensemble de cartes de primitives d'entrée. Considérons une couche composée de n neurones connectée à m cartes de primitives. Cette couche peut s'écrire :

$$\texttt{ConvLayer}_{\mathbf{b},\mathbf{W},f_{act}} = \texttt{mapi}\ \phi \tag{5.2}$$

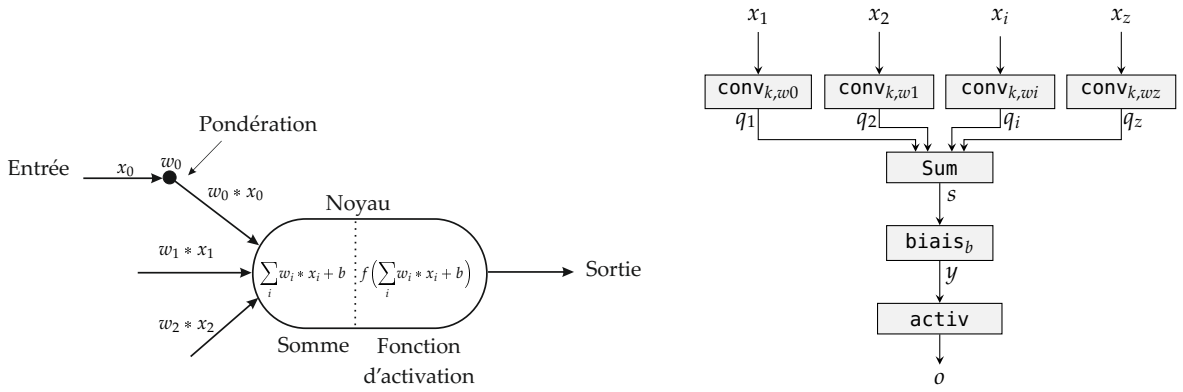
avec

$$\phi(i,x) = \texttt{neurone}_{\mathbf{b}_i,\mathbf{W}_i,f_{act}}\ (\texttt{sparse\_extract}(x))$$

où

- $x = (x_1, \cdots, x_m)$ est l'ensemble des cartes de primitives disponibles à l'entrée de la couche,
- `sparse_extract` est une fonction qui extrait les z connexions de l'ensemble des m cartes d'entrées pour chaque neurone (chaque neurone reçoit z cartes en entrée),
- $\mathbf{b}$ est le vecteur de biais avec $\mathbf{b} = (b_1, \cdots, b_n)^T$,
- $\mathbf{W} = (\mathbf{w_1}, \cdots, \mathbf{w_n})^T$ correspond à l'ensemble des poids associés à une couche,
- f_{act} est la fonction d'activation, identique pour chaque neurone.
- `mapi` est la fonctionnelle de réplication introduite à la Sec. 4.5.1

Pour chacune des z cartes de primitives connectées en entrée, un neurone calcule la convolution de cette carte avec un filtre de poids appris, comme illustré sur la Fig. 5.5.



(a) Vue schématique *(b)* Graphe flot de données correspondant

Fig. 5.5: Modèle de neurone.

Les axones entrants (x_i du modèle) correspondent aux cartes de primitives d'entrées (f_i) du graphe. Une synapse correspond à un acteur de convolution. Les dendrites sont représentés par les canaux q_i. Le noyau est la composition de trois acteurs. Le premier `sum` calcule la

combinaison linéaire des données sur les dendrites. Un second acteur ajoute un biais et enfin un dernier acteur correspond à la fonction d'activation. Fonctionnellement, le graphe flot de données de la Fig. 5.5b peut s'exprimer par :

$$\texttt{neurone}_{b,\mathbf{w},f_{act}} = \texttt{activ}_{f_{act}} \circ \texttt{biais}_b \circ \texttt{sum} \circ \texttt{mapi}\ \phi \tag{5.3}$$

avec

$$\phi(i,x) = \texttt{conv}_{\mathbf{w}_i}(x)$$

où b est le biais d'apprentissage, $\mathbf{w} = (w_1, \cdots, w_z)^T$ regroupe les poids obtenus lors de l'apprentissage et f_{act} est la fonction d'activation du neurone. Chaque élément w_i du vecteur $\mathbf{w}$ est composé de $k \times k$ éléments (équivalent à la dimension du filtre de convolution). Dans la suite, chaque acteur impliqué dans l'Eq. 5.3 est décrit séparément.

5.2.1.1 Acteur conv

Comme introduit dans le chapitre 3.7, les opérateurs de convolution ont plusieurs paramètres liés à la constance des filtres dans les cartes de primitives, le déplacement entre deux opérations successives (*stride*) et la gestion des contours (*padding*)).

On utilise ici la technique de *parameter sharing*, souvent préconisée afin de limiter le nombre de paramètres à apprendre dans un réseau. Cette technique consiste à appliquer le même filtre de convolution à l'ensemble des entrées de la carte de primitives. Par ailleurs, aucun *padding* n'est utilisé : les contours de l'image où les résultats des convolutions n'ont pas de sens ne sont pas produits en sortie. Une convolution avec un noyau de dimension $k \times k$ diminue alors les dimensions de l'image d'entrée d'une valeur $(k-1)$ dans chaque direction. Dans notre, un acteur de convolution avec un noyau w de dimension $k \times k$ (k impair), connecté à une carte de primitive f peut donc se formuler par :

$$\texttt{conv}_{k,w}(\langle\langle f_{1,1} \cdots f_{1,m}\rangle \cdots \langle f_{n,1} \cdots f_{n,m}\rangle\rangle) = \langle\langle q_{1,1} \cdots q_{1,s}\rangle \cdots \langle q_{r,1} \cdots q_{r,s}\rangle\rangle \tag{5.4}$$

où

$$q_{i,j} = \sum_{b=-\frac{k-1}{2}}^{\frac{k-1}{2}} \sum_{a=-\frac{k-1}{2}}^{\frac{k-1}{2}} w_{a,b} \cdot f_{i+a,j+b} \qquad \text{si } i \in \left]\frac{k-1}{2}, n - \frac{k-1}{2}\right] \ \wedge \ j \in \left]\frac{k-1}{2}, m - \frac{k-1}{2}\right]$$

Les flux de sorties q seront de dimension $r \times s$ avec :

$$r = n - (k-1) \text{ et } s = m - (k-1) \tag{5.5}$$

5.2.1.2 Acteur Sum

Cet acteur fait la somme de ses entrées. Il s'exprime simplement avec la fonctionnelle $\texttt{mlift}_m$ introduite à la Sec. 4.2.1, où m correspond aux nombres de flux d'entrées (ici z) :

$$\texttt{sum}(q_1, \cdots, q_z) = \texttt{mlift}_z(fsum, q_1, \cdots, q_z) = \langle\langle s_{1,1} \cdots s_{1,m}\rangle \cdots \langle s_{n,1} \cdots s_{n,m}\rangle\rangle \tag{5.6}$$

où

$$fsum(q_1, \cdots, q_z) = \sum_{k=1}^{z} q_k$$

5.2.1.3 Acteur bias

L'acteur `bias` correspond à l'ajout de biais, il est défini par :

$$\begin{aligned}\texttt{bias}_b(\langle\langle s_{1,1} \cdots s_{1,m}\rangle \cdots \langle s_{n,1} \cdots s_{n,m}\rangle\rangle) &= \texttt{mlift}\ (fbias_b, s)\\ &= \langle\langle y_{1,1} \cdots y_{1,m}\rangle \cdots \langle y_{n,1} \cdots y_{n,m}\rangle\rangle \end{aligned} \quad (5.7)$$

avec

$$fbias_b(s) = s + b$$

5.2.1.4 Acteur activ

L'acteur `activ` applique une fonction d'activation sur chaque élément en entrée. Cet acteur s'exprime par :

$$\texttt{activ}_{f_{act}}(\langle\langle y_{1,1} \cdots y_{1,m}\rangle \cdots \langle y_{n,1} \cdots y_{n,m}\rangle\rangle) = \texttt{mlift}(f_{act}, y) \quad (5.8)$$

Comme expliqué à la Sec. 3.7, il existe plusieurs possibilités pour la fonction f_{act}. Pour l'essentiel, les travaux menés dans le cadre des ANN sur l'impact des approximations sur une architecture matérielle sont transposables aux CNN. Par exemple, la fonction sigmoïde peut être implantée dans des tables de correspondance (LUT) [NLM+09], ou approximée par des développements de Taylor au second ordre [dCFEB13]. La fonction tangente hyperbolique peut également exploiter des LUTs [KLK02], des algorithmes d'approximations par CORDIC [TK15] ou des polynômes de Chebyshev [BMD13]. Dans ces travaux, la fonction d'activation dite ReLU [KSH12] est choisi pour la simplicité de son implémentation.

5.2.2 Couches de sous-échantillonnage

Les couches de sous-échantillonnage appliquent le même traitement sur chaque carte de primitives en sortie d'une couche de convolution (Fig. 5.6).

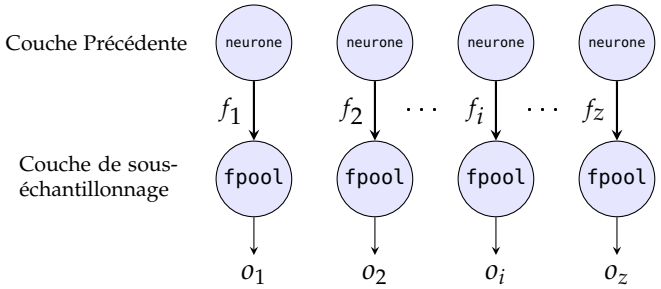


Fig. 5.6: Connectivité des couches de sous-échantillonnage

La couche de sous-échantillonnage, illustrée sur la Fig. 5.6, avec z cartes de primitives d'entrées et une opération de sous-échantillonnage `fpool`, se formule donc simplement via la fonctionnelle `map` :

$$\texttt{PoolLayer}_{fpool} = \texttt{map fpool} \quad (5.9)$$

La fonction `fpool` se base généralement sur le maximum (Fig. 5.7) [AHMJP12], la moyenne [LBBH98] ou plus récemment la norme [ZF13]. Les fonctions au sein de ces couches appliquent des traitements locaux sur des régions rectangulaires de dimension $(k \times k)$ avec un pas de déplacement (*stride*), potentiellement différent de k [KSH12].

L'acteur choisi $\texttt{maxpool}_k$ réalise un sous-échantillonnage sur la base du maximum local. Le principe est illustré sur la Fig. 5.7. L'entrée est ici une image 4×4 représentée par le flux

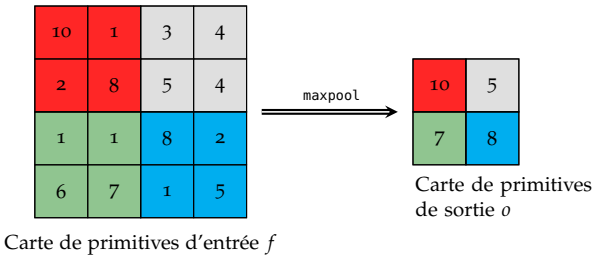


Fig. 5.7: Exemple d'opération de sous-échantillonnage avec maximum local

structuré : $\langle\langle 10\ 1\ 3\ 4\rangle\ \langle 2\ 8\ 5\ 4\rangle\langle 1\ 1\ 8\ 2\rangle\langle 6\ 7\ 1\ 5\rangle\rangle$. La sortie est une image 2×2 représentée par le flux structuré : $\langle\langle 10\ 5\rangle\ \langle 7\ 8\rangle\rangle$.

L'acteur `maxpool`$_z$ se décrit naturellement par la composition de deux acteurs, `maxpoolh` et `maxpoolv`, opérant respectivement sur la dimension horizontale et verticale d'une carte de primitives :

$$\texttt{maxpool}_k = \texttt{maxpoolv}_k \circ \texttt{maxpoolh}_k \tag{5.10}$$

5.2.2.1 Acteur `maxpoolh`

Considérons un sous-échantillonnage de $k \times k$ éléments et une carte de primitives d'entrée f de $n \times m$ éléments, où les dimensions de la carte de primitives sont divisibles par la taille du voisinage, le premier acteur horizontal `maxpoolh` est défini formellement, pour chaque ligne $i \in \{1, n\}$:

$$\texttt{maxpoolh}_k(\langle\langle \underbrace{f_{1,1} \cdots f_{1,m}}_{1 \text{ ligne : } m \text{ éléments}} \rangle \cdots \langle f_{n,1} \cdots f_{n,m} \rangle\rangle) = \langle\langle \underbrace{x_{1,1} \cdots x_{1,s}}_{1 \text{ ligne : } s \text{ éléments}} \rangle \cdots \langle x_{n,1} \cdots x_{n,s} \rangle\rangle \tag{5.11}$$

avec

$$s = m/k$$
$$x_{i,jj} = \max\left\{ f_{i,j} \;\middle|\; j = \{(k \cdot (jj-1)) + 1, \cdots, (k \cdot jj)\}, \quad i \in \{1, \cdots, n\},\ jj \in \{1, \cdots, s\} \right\}$$

A partir de la séquence donnée en exemple (Fig. 5.7), l'application de l'acteur `maxpoolh`, illustrée sur la Fig. 5.8, produit par exemple le flux suivant : $\langle\langle 10\ 4\rangle\langle 8\ 5\rangle\langle 1\ 8\rangle\langle 7\ 5\rangle\rangle$.

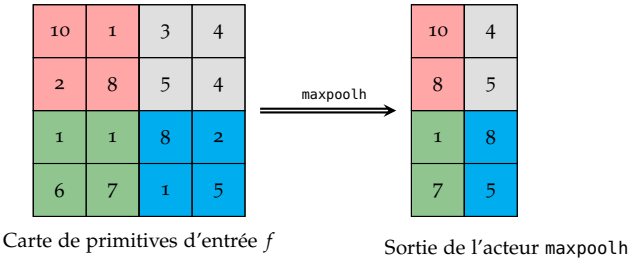


Fig. 5.8: Exemple d'opération de l'acteur `maxpoolh`.

5.2.2.2 Acteur `maxpoolv`

Le second acteur `maxpoolv`, connecté à la sortie de l'acteur `maxpoolh` applique la même opération mais dans la direction verticale :

$$\texttt{maxpoolv}_z(\langle\langle \underbrace{x_{1,1} \cdots x_{1,s} \rangle \cdots \langle x_{n,1} \cdots x_{n,s}}_{n \text{ lignes (de } s \text{ éléments)}} \rangle\rangle) = \langle\langle \underbrace{o_{1,1} \cdots o_{1,s} \rangle \cdots \langle o_{p,1} \cdots o_{p,s}}_{p \text{ lignes (de } s \text{ éléments)}} \rangle\rangle \tag{5.12}$$

avec

$$p = n/k$$
$$o_{ii,j} = \max\left\{ x_{i,j} \;\middle|\; i = \{(k\cdot(ii-1)+1,\cdots,(k\cdot ii)\}, \quad ii \in \{1,\cdots,p\}, j \in \{1,\cdots,s\} \right\}$$

A partir de la séquence produite par l'acteur `maxpoolh` : $\langle\langle 10\,4\rangle\langle 8\,5\rangle\langle 1\,8\rangle\langle 7\,5\rangle\rangle$, l'application de l'acteur `maxpoolv`, illustrée sur la Fig. 5.9, produit en sortie le flux : $\langle\langle 10\,5\rangle\ \langle 7\,8\rangle\rangle$.

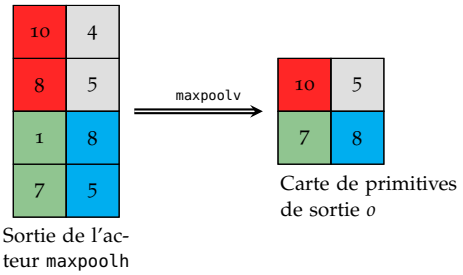


Fig. 5.9: Exemple d'opération de l'acteur `maxpoolv`.

5.2.3 Couche complètement connectée

Alors que les neurones dans les couches de convolution ont une connectivité partielle en entrée, les neurones présents dans les dernières couches dites *Fully-Connected* (FC) sont complètement connectés à l'ensemble des sorties de la couche précédente, comme illustré sur la Fig. 5.10. Au sein de ces couches, chaque acteur opère un produit scalaire entre l'ensemble des éléments présents dans chaque carte de primitives en entrée et un vecteur issu de l'apprentissage.

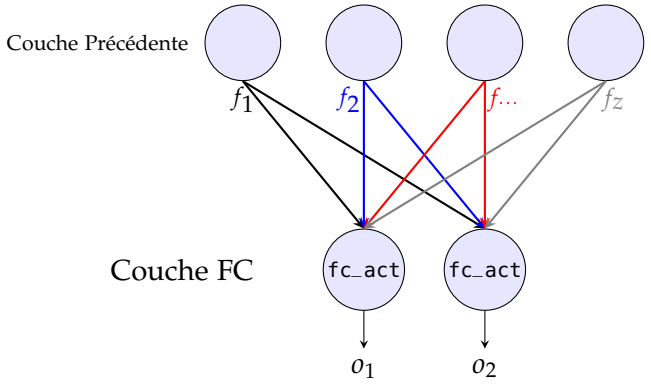


Fig. 5.10: Illustration de la couche complètement connectée

Une couche FC, telle que celle illustrée sur la Fig. 5.10, s'exprime donc fonctionnellement par :

$$\texttt{FCLayer}_{\mathbf{b},\mathbf{W},n,m} = \texttt{mapi}\ \phi \tag{5.13}$$

avec

$$\phi(i,f) = \texttt{fc\_act}_{\mathbf{b}_i,\mathbf{W}_i,n,m}\ (f) \tag{5.14}$$

où

- $f = (f_1, \cdots, f_z)$ est l'ensemble des cartes de primitives disponibles à l'entrée de la couche,
- $\mathbf{b}$ est le vecteur de biais avec $\mathbf{b} = (b_1, \cdots, b_z)^T$
- $\mathbf{W} = (\mathbf{w_1}, \cdots, \mathbf{w_z})^T$ correspond à l'ensemble des poids contenus dans une couche
- n, m sont des constantes indiquant les dimensions des cartes de primitives f.

La fonction `fc_act` représente l'opération arithmétique qui doit être effectuée au sein de chaque acteur d'une couche FC. Cette fonction prend en entrée un ensemble de cartes de primitives et produit en sortie une "carte" réduite à un **scalaire**.

$$\texttt{fc\_act}_{\mathbf{w},b,n,m}(f_1,\ \cdots,\ f_z) = \texttt{mlift}_z(fct_{\mathbf{w},b,n,m}, f_1,\ \cdots,\ f_z) = o \tag{5.15}$$

avec

$$f_k = \langle \langle f_{k_{1,1}} \ \cdots \ f_{k_{1,m}} \rangle \cdots \langle f_{k_{n,1}} \ \cdots \ f_{k_{n,m}} \rangle \rangle \quad \forall k \in \{1, \cdots, z\}$$

$$fct_{\mathbf{w},b,n,m}(f_1,\ \cdots,\ f_z) = b + \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{z} w_{k_{i,j}} \times f_{k_{i,j}}$$

La fonction $\texttt{fc\_act}_{\mathbf{w},b,n,m}$ peut se décomposer en un graphe de cinq acteurs élémentaires, illustrés sur la Fig. 5.11 : $\texttt{distp}_{\mathbf{w},n,m}$, `dot_product`, $\texttt{hsum}_m$, $\texttt{vsum}_n$ et $\texttt{biais}_b$.

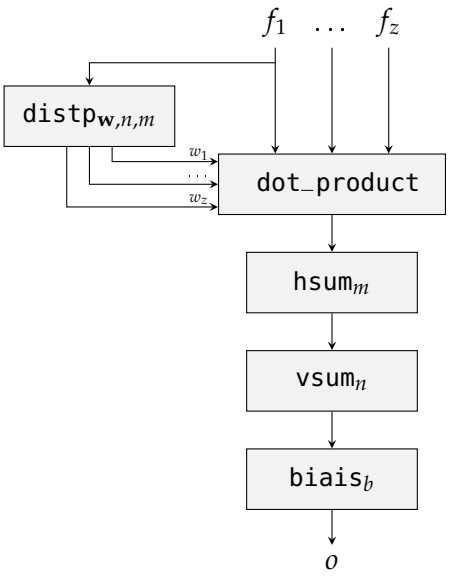


Fig. 5.11: Graphe flot de données correspondant à la fonction `fc_act`

Le graphe flot de données de la Fig. 5.11 peut se décrire par l'équation :

$$\texttt{fc\_act}_{\mathbf{w},b,n,m}(f_1, \cdots, f_z) = \texttt{biais} \circ \texttt{vsum}_n \circ \texttt{hsum}_m \circ \texttt{dot\_product}(w_1, \cdots, w_z, f_1, \cdots, f_z) \tag{5.16}$$

avec

$$(w_1, \cdots, w_z) = \texttt{distp}_{\mathbf{w},n,m}(f_1)$$

Les acteurs $\texttt{hsum}_m$, $\texttt{vsum}_n$ et $\texttt{biais}_b$ ont déjà été définis précédemment dans les Sec. 4.2.2.2 et 4.2.3.

5.2.3.1 Acteur `dot_product`

L'acteur `dot_product` calcule le produit scalaire partiel des primitives d'entrée avec les poids correspondants. Cet acteur est défini par :

$$\texttt{dot\_product}(f_1, \cdots, f_z, w_1, \cdots, w_z) = \texttt{mlift}_{2z}(fdot, f_1, \cdots, f_z, w_1, \cdots, w_z) \tag{5.17}$$

avec

$$fdot(f_1, \cdots, f_z, w_1, \cdots, w_z) = \sum_{k=1}^{z} w_k \cdot f_k$$

5.2.3.2 Acteur `distp`

La distribution des poids correspondants est assurée par l'acteur `distp`. Le vecteur de poids $\mathbf{w} = (w_1, \cdots, w_z)^T$, déterminé par l'apprentissage, correspond à l'ensemble des poids associés aux cartes de primitives. Chaque élément de ce vecteur $w_{k_{i,j}}$ dépend à la fois de la carte de primitives (k) et de la position de la primitive au sein de la carte (i, j). L'acteur `distp` est activé par une carte de primitive (f_1 sur Fig. 5.11). Le vecteur poids global $\mathbf{w}$ est défini par un tableau à trois dimensions dont la première correspond à l'index de la carte de primitive et les deux autres dimensions correspondent aux éléments dans les deux directions possibles. Le tableau $\mathbf{w}$, associé à z entrées chacune de dimension $n \times m$, est donc de la forme : $\mathbf{w}[z][n][m]$, et l'acteur de distribution de poids se formule :

$$\texttt{distp}_{\mathbf{w},n,m}(f_1) = (w_1, \cdots, w_z) \tag{5.18}$$

avec

$$f_1 = \langle \langle f_{1_{1,1}} \ \cdots \ f_{1_{1,m}} \rangle \cdots \langle f_{1_{n,1}} \ \cdots \ f_{1_{n,m}} \rangle \rangle$$

$$w_k = \langle \langle w_{k_{1,1}} \ \cdots \ w_{k_{1,m}} \rangle \cdots \langle w_{k_{n,1}} \cdots \ w_{k_{n,m}} \rangle \rangle, \quad \forall k \in \{1, \cdots, z\}$$

$$w_{k_{i,j}} = \mathbf{w}[k][i][j], \ k \in \{1, \cdots, z\}, \ i \in \{1, \cdots, n\}, \ j \in \{1, \cdots, m\}$$

Ensuite, la composition des acteurs `hsum` et `vsum` paramétrés avec les dimensions des cartes de primitives (n, m dans notre cas) permet de calculer le produit scalaire final. Le dernier acteur ajoute seulement un biais (similaire à l'acteur Eq. 5.7).

5.2.4 Couche de classification

La dernière couche d'un réseau CNN est une couche de classification. L'objectif de cette couche est de déterminer à quelle classe y appartient le vecteur d'entrée $\mathbf{x}$ (composé de tous les éléments de la couche précédente) parmi K classes possibles. Cette couche est complètement connectée aux entrées de la couche précédente (Fig. 5.12).

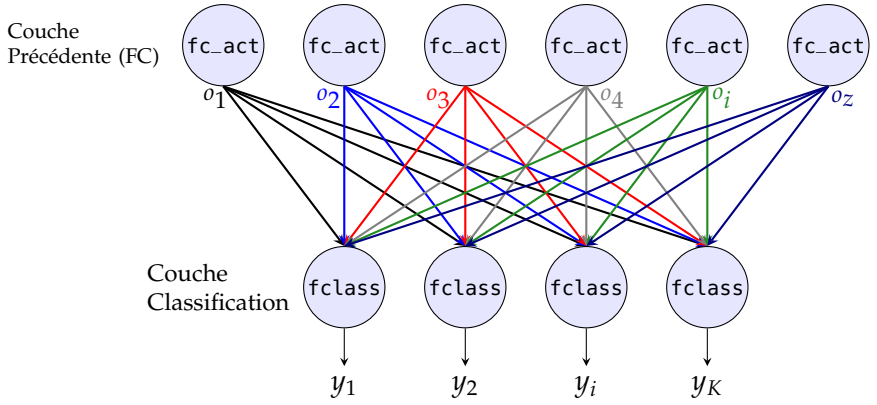


Fig. 5.12: Illustration de la couche de classification

La couche de classification de la Fig. 5.12 a pour expression :

$$\texttt{ClassLayer}_{\mathbf{W}} = \texttt{mapi}\ \phi \tag{5.19}$$

avec

$$\phi(i, \mathbf{x}) = \texttt{fclass}_{\mathbf{w}_i}(\mathbf{x})$$

où $\mathbf{x} = (x_1, \cdots, x_z)^T$ est l'ensemble des données d'entrées de la couche. De même que pour les opérations d'activations, il existe plusieurs solutions possibles pour la fonction de classification. Le modèle original de Lecun propose l'utilisation d'un ensemble de calculs de distance euclidienne entre le vecteur d'entrée et des paramètres appris [LBBH98]. La fonction de classification `fclass` pour un acteur k de la couche peut se formuler dans ce cas :

$$\texttt{fclass}_{\mathbf{w}_i}(\mathbf{x}) = \sum_{j=1}^{z} (x_j - w_{i,j})^2 = y_k \tag{5.20}$$

L'opération se décompose en un graphe d'acteurs élémentaires, illustré sur la Fig. 5.13.

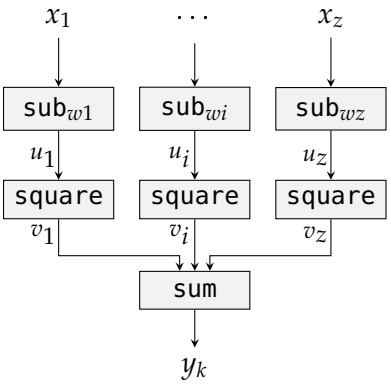


Fig. 5.13: Graphe flot de données correspondant à la fonction `fclass`

La fonction `fclass`(Eq. 5.20) peut également s'exprimer par la composition des acteurs du graphe de la Fig. 5.13.

$$\texttt{fclass}_{\mathbf{w}} = \texttt{sum} \circ \texttt{map square} \circ \texttt{mapi}\ \phi \tag{5.21}$$

avec

$$\phi(i, x) = \texttt{sub}_{\mathbf{w}_i}(x)$$

A chaque entrée x_k avec $k \in \{1, \cdots, z\}$ est soustraite un poids issu de l'apprentissage par l'acteur `sub`, tel que :

$$\texttt{sub}_w(x_k) = x_k - w_k = u_k \tag{5.22}$$

L'acteur suivant `square` élève le résultat de la soustraction au carré.

$$\texttt{square}(u_k) = u_k^2 = v_k \tag{5.23}$$

Le dernier acteur est l'acteur `sum`, précédemment défini à l'Eq. 5.6, qui accumule les résultats de chaque entrée pour fournir la réponse finale de la classe y_k.

Le réseau LeNet complet, présenté sur la Fig. 5.3, se formule par la composition des couches précédemment définies :

$$\begin{aligned}\texttt{LeNet} = \texttt{ClassLayer}_{\mathbf{W_6}} \circ \texttt{FCLayer}_{\mathbf{b_5}, \mathbf{W_5}, n, m} \circ \texttt{PoolLayer}_{fpool} \circ \\ \texttt{ConvLayer}_{\mathbf{b_3}, \mathbf{W_3}, f_{act}} \circ \texttt{PoolLayer}_{fpool} \circ \texttt{ConvLayer}_{\mathbf{b_1}, \mathbf{W_1}, f_{act}}\end{aligned} \tag{5.24}$$

5.3 Transcription en CAPH

Cette section s'intéresse à la transcription en CAPH du réseau précédemment formulé. On montre comment la construction d'un réseau CNN se décrit, puis on propose une exploration de différentes configurations possibles. Tous les acteurs ainsi que les fonctions d'ordre supérieur utilisées sont décrits dans l'annexe C.

Le langage de description de réseau de CAPH étant fonctionnel, la transcription de la formulation flot de données d'un CNN est immédiate. Afin de rendre la description plus lisible, on décrit d'abord un réseau simplifié par rapport à celui étudié à la section précédente (moins de neurones par couche). Les résultats décrits à la Sec. 5.4 ont été obtenus avec un complet. La table 5.1 résume la structure de ce réseau simplifié. La première couche contient 6 neurones comportant chacun des filtres de dimensions 3×3 et une seule connexion d'entrée (image 30×30 de la base de données MNIST [LBBH98]). La seconde couche applique une fonction de sous-échantillonnage avec un filtre 2×2 sur chaque carte de primitives de la couche précédente. La troisième couche est composée de quatre neurones, où chaque neurone est connecté à 3 cartes de primitives et applique également un filtre 3×3. La couche complètement connectée contient 4 unités et la couche de classification seulement deux classes. Bien que ce premier réseau ne puisse pas répondre directement au problème de la classification de caractères, il permet d'expliquer sur un exemple simple la transcription en CAPH de la formulation flot de données.

Tab. 5.1: Composition d'une premier réseau LeNet

Couche	Type	Neurones	Dimension du vecteur de cartes de primitives entrant	Opération
C1	Convolution	6	$3 \times 3 \times 1$	ReLU
S2	Sous-Échantillonnage		$2 \times 2 \times 6$	Max
C3	Convolution	4	$3 \times 3 \times 3$	ReLU
S4	Sous-Échantillonnage		$2 \times 2 \times 4$	Max
F5	Complètement connectée	4	$6 \times 6 \times 4$	Produit Scalaire
Classif	Classification	2	$1 \times 1 \times 4$	Distance Euclidienne

La transcription en CAPH de ce réseau est disponible dans le listing 5.1 et le graphe flot de données sur la Fig. 5.14.

Listing 5.1: Listing CAPH du réseau LeNet simplifié

```
1  -- Couche C1: 6 neurones
2  net (ts1,ts2,ts3,ts4,ts5,ts6) = convs conv233c_wb_opt rep6 weights_C1_3x3 o biais_C1 relu i;
3
4  -- Couche S2: Subsampling maxpool filtre 2x2
5  net (t1,t2,t3,t4,t5,t6) = map (maxpool 2 2) (ts1,ts2,ts3,ts4,ts5,ts6);
6
7  -- Couche C3: 4 neurones:
8  net (os1,os2,os3,os4) = convlayer conv233c_wb_opt weights_N1 o biais_C1 sum3 relu
9                            ((t1,t2,t3),(t1,t2,t3),(t4,t5,t6),(t4,t5,t6));
10
11 -- Couche S4: Subsampling maxpool filtre 2x2
12 net (o1,o2,o3,o4) = map (maxpool 2 2) (os1,os2,os3,os4);
13
14 -- Couche F5: FC
15 net (x11,x12,x13,x14) = fclayer distr weights_FC 6 6 biais_FC) o1 (o1,o2,o3,o4);
16
17 -- Couche OUTPUT: classification distance enclidienne
18 net yo= classif weights_CLASSo sum4 (x0,x1,x2,x3);
19 net y1= classif weights_CLASS1 sum4 (x0,x1,x2,x3);
20
21 -- I/Os
22 stream i    : signed<16> dc from "imgs/minst30.txt";
23 stream yo   : signed<16> dc to "res/yo.txt";
24 stream y1   : signed<16> dc to "res/y1.txt";
```

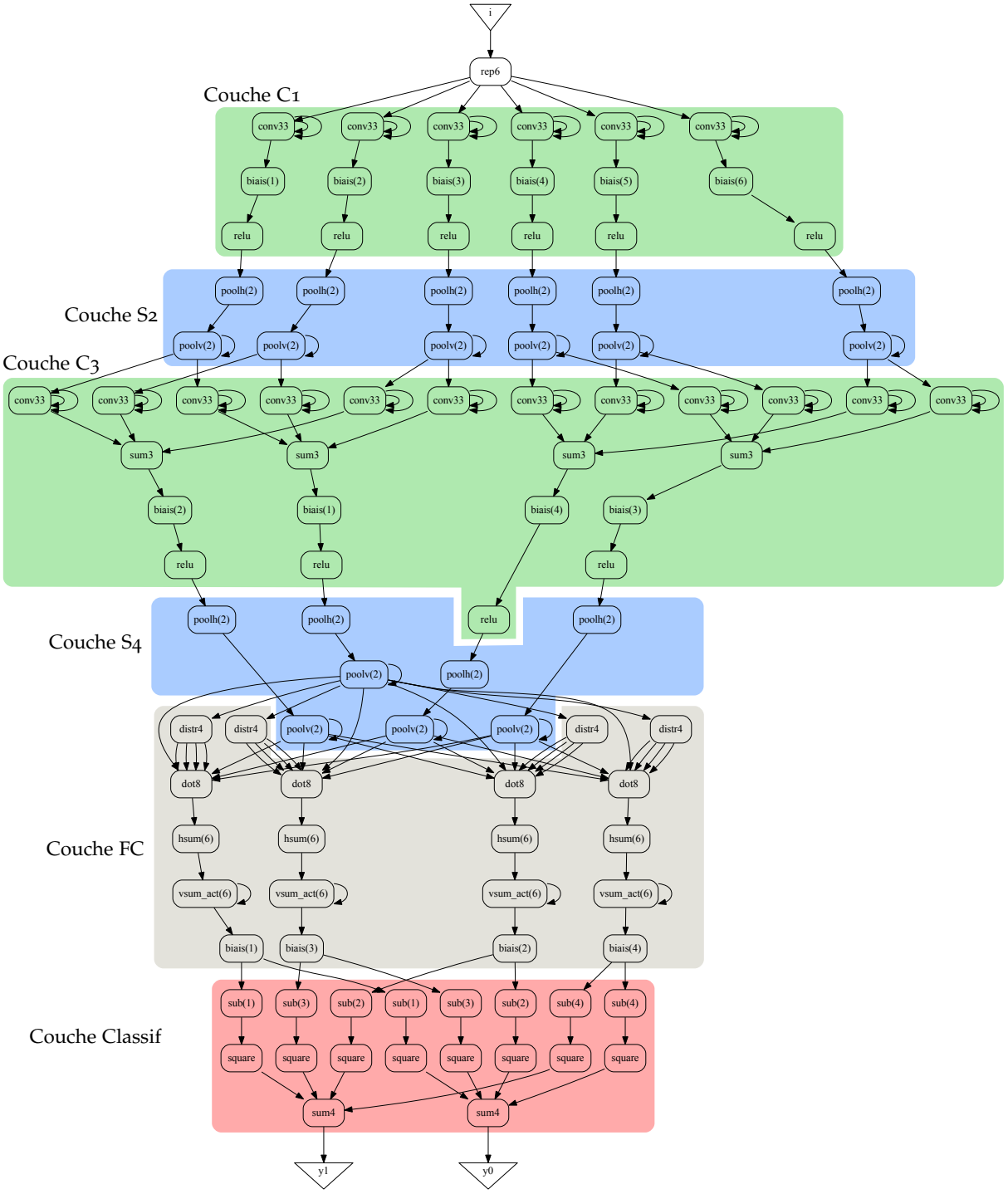


Fig. 5.14: Graphe d'acteurs généré par CAPH à partir de la transcription du Listing. 5.1

La première fonction décrite est la fonction `convlayer` (Eq. 5.2) correspondant aux couches de convolution *C*1 et *C*3, qui s'appuie sur une fonction intermédiaire `neurone` (Eq. 5.3). Ces deux fonctions sont disponibles sur le listing 5.2.

Listing 5.2: Listing CAPH de la fonction `ConvLayer`

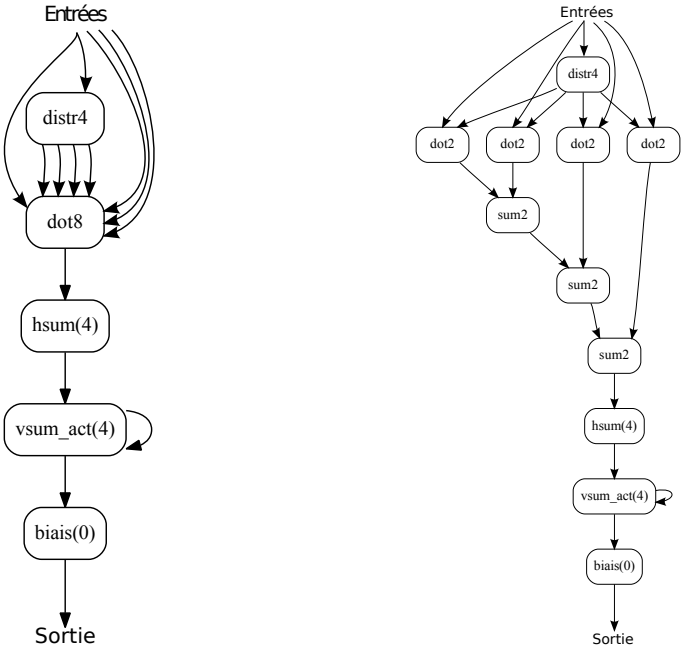
```
1  -- Fonction Neurone
2  ----------------------------------------------------------------------
3  -- tx: t-uplet représentant le flux d'entrée
4  -- actor_conv: nom de l'acteur à utiliser pour la convolution
5  -- kernels_weights: tableau de poids
6  -- shift: facteur de normalisation des convolutions
7  -- biais_weight: 1 biais par neurone.
8  -- sumx: acteur de somme des conv
9  -- actor_activation: acteur d'activation
10
11 net neurone actor_conv kernels_weights shift biais_weight sumx actor_activation tx =
12     let ff i tx =  actor_conv (kernels_weights[i]) shift tx in
13         actor_activation(biais biais_weight (sumx (mapi ff (tx)))) ;
14
15 -- Fonction convlayer: Couche de neurones
16 ----------------------------------------------------------------------
17 -- conv_act: nom de l'acteur à utiliser pour la convolution (ex:conv33)
18 -- weights: tableau de poids 3D
19 -- shift: facteur de normalisation des convolutions
20 -- biais: tableau de biais
21 -- sum_act: acteur de somme (depend du nombre de conv dans un neurone)
22 -- fact : acteur d'activation
23 -- ttx: t-uplet of t-uplet où chaque t-uplet correspond aux cartes de primitives entrantes d'un neurone
24
25 net convlayer conv_act weights shift biais sum_act fact ttx =
26     let ff i ttx = neurone_act conv_act (weights[i]) o (biais[i]) sum_act fact ttx in
27         mapi ff (ttx);
```

La fonction `convlayer` est utilisée à la ligne 8 du listing. 5.1. Le cas de la première couche de convolution est légèrement différent car chaque neurone n'est connecté qu'à une seule entrée. La fonction `convs` utilisée (ligne 2) est une version simplifiée de la fonction `convlayer` dans lequel l'acteur de sommation est supprimé.

La transcription des couches de sous-échantillonnage est équivalente à leur définition (Eq. 5.9). Les dimensions du filtre de sous-échantillonnage (ici 2×2) sont données en paramètres de la fonction `maxpool`. La fonction `maxpool` (Eq.5.10) est la composition de deux acteurs `poolh` et `poolv`. Par ailleurs, il pourrait être intéressant de pouvoir définir en CAPH des acteurs d'ordre supérieur (acteurs paramétrés avec des fonctions) pour `poolh` et `poolv`. Dans ce cas, il serait possible de modifier l'opération effectuée pendant le sous-échantillonnage par le simple paramétrage de l'acteur lors de son instanciation.

La couche complètement connectée est construite à la ligne 15. Il n'est pas possible à l'heure actuelle de transcrire la fonction`FClayer` (Eq. 5.13) en CAPH exactement comme elle a été définie précédemment. En effet, il existe une limitation provenant de l'interface de l'acteur `dot_product`. Cet acteur, défini à l'Eq. 5.17 et illustré sur le graphe Fig. 5.14, calcule le produit scalaire de l'ensemble des entrées de la couche avec des poids associés (provenant de l'acteur `distp`). Afin de transcrire la fonction `FClayer`, il faudrait être en mesure de définir un acteur `dot_product` *générique*, pouvant accepter en entrée un t-uplet de taille quelconque, ce qui n'est pas supporté avec la version actuelle des outils.

Deux solutions sont alors possibles. La première consiste à utiliser une fonction spécifique à une configuration du réseau. Cette approche permet d'instancier l'acteur `dot8` (graphe de la Fig. 5.14) en explicitant les connexions. La seconde solution consiste à substituer l'acteur `dot_product` par une composition d'acteurs de produits scalaires partiels (`dot2` et `sum2`) connectés deux à deux via les fonctions `fold` et map2 de CAPH. Cette technique permet de définir la fonction pour n'importe quelle configuration du réseau mais engendrera un surcout lors de l'implémentation (par rapport à un seul acteur `dot_product`). Cette substitution est illustrée sur la Fig. 5.15.



(a) Fonction `fc_act` original

(b) Fonction `fc_act` avec substitution de l'acteur de produit scalaire

Fig. 5.15: Graphes correspondant à la fonction `fc_act`

Le listing 5.3 illustre la seconde proposition. La définition de la fonction `FCLayer` se fait par l'intermédiaire de la fonction `fc_act`, définie à l'Eq. 5.16 et de la fonctionnelle *nappi*, introduite dans la version 2.7.1. La fonctionnelle *nappi* est similaire à la fonctionnelle *mapi*, à la seule différence que le nombre de réplication est fixé de manière nombre arbitraire.

Listing 5.3: Listing CAPH de la fonction `FCLayer`

```
1  -- Fonction FC_act
2  ----------------------------------------------------------------------
3  -- tx: t-uplet de flux d'entrée
4  -- x: flux d'entrée pour activation de la distribution de poids (un element de nx)
5  -- distr_act: acteur de distribution de poids
6  -- kernels_weights: tableau 2D de poids
7  -- n: nombre de lignes des cartes de primitives
8  -- m: nombre de colonnes des cartes de primitives
9  -- b: biais (scalaire)
10
11 net fc_act distr_act kernels_weights n m b x tx=
12     let w = distr_act (kernels_weights,n,m) (x) in
13         let yy = foldl sum2 ( map2 (dot2) ( (tx), (w) ))
14             in biais b (vsum n (hsum m (yy)));
15
16
17 -- Fonction FCLayer: instancie un ensemble de FC_act en fonction de la dimension du t-uplet d'entrée
18 ----------------------------------------------------------------------
19 -- tx: t-uplet de flux d'entrée
20 -- x: flux d'entrée pour activation de la distribution de poids (un élément de nx)
21 -- distr_act: acteur de distribution de poids
22 -- kernels_weights: tableau 3D de poids
23 -- n: nombre de lignes des cartes de primitives
24 -- m: nombre de colonnes des cartes de primitives
25 -- biais_weights: tableau 1D de biais
26
27 net fclayer distr_act kernels_weights n m biais_weights x tx =
28     let ff i tx  = fc_act distr_act (kernels_weights[i]) n m (b[i]) x tx in
29         nappi 4 ff (tx);
```

La dernière fonction à transcrire en CAPH est la fonction `ClassLayer`. Comme pour la

fonction précédente, il est impossible de la transcrire directement à l'aide des primitives actuelles du langage. Toutefois, il est possible d'exprimer la fonction `fclass` (Eq. 5.20).

Listing 5.4: Listing CAPH de la fonction `ClassLayer`

```
1 -- Fonction fclass: Classification: Distance euclidienne
2 -- tx: flux d'entrée pour activation de la distribution de poids (un élément de nx)
3 -- sum_act: acteur de sommation
4 -- weights: tableau 1D de poids
5 ------------------------------------------------------------------------
6 net fclass weights sum_act tx =
7     let ff i tx =  sub (weights[i]) x in
8         sum_act (map square (mapi ff (tx))) ;
```

Maintenant que les éléments de base de la construction d'un réseau CNN ont été donnés, on s'intéresse à l'exploration de différentes solutions concernant la problématique de reconnaissance de caractères.

5.4 Exploration de réseaux et implantations

On propose dans cette section d'explorer différentes configurations de réseaux dans le cadre d'une application de reconnaissance de caractères. L'exploration doit permettre de déterminer un réseau qui maximise les performances de classification tout en minimisant les ressources matérielles requises par l'implémentation. Pour cela, l'environnement de développement Caffee [JSD+14] est utilisé afin de gérer le processus d'apprentissage, la mesure de fiabilité des résultats ainsi que la génération des paramètres pour le réseau CAPH (poids issus de l'apprentissage). Le premier réseau correspond à celui proposé par LeCun [LBBH98] où les fonctions d'activation ont été remplacées par des fonctions *ReLU* et les filtres de convolution sont de dimension 3×3 pour des raisons d'implémentation. Le second réseau divise par deux le nombre d'unités pour les deux dernières couches $C5$ et FC. Le troisième réseau divise encore le nombre d'unités des dernières couches $C5$ et FC par trois. Le dernier réseau est le plus dégradé où la troisième couche de convolutions $C5$ a été totalement supprimée. La table 5.2 récapitule les différentes configurations explorées, où les champs N et Op correspondent respectivement au nombre d'unités et aux opérations effectuées au sein de chaque couche.

Tab. 5.2: Réseaux explorés

Réseau 1			Réseau 2		
Couches	N	Op	Couches	N	Op
C1	6	Conv 3 × 3 + ReLU	C1	6	Conv 3 × 3 + ReLU
S2	6	Max	S2	6	Max
C3	16	Conv 3 × 3 + ReLU	C3	8	Conv 3 × 3 + ReLU
S4	16	Max	S4	8	Max
C5	120	Conv 3 × 3 + ReLU	C5	60	Conv 3 × 3 + ReLU
F5	84	Prod. Scalaire	F5	42	Prod. Scalaire
Classif	10	Dist. Euclid	Classif	10	Dist. Euclid
Réseau 3			Réseau 4		
Couches	N	Op	Couches	N	Op
C1	6	Conv 3 × 3 + ReLU	C1	6	Conv 3 × 3 + ReLU
S2	6	Max	S2	6	Max
C3	8	Conv 3 × 3 + ReLU	C3	8	Conv 3 × 3 + ReLU
S4	8	Max	S4	8	Max
C5	20	Conv 3 × 3 + ReLU			
F5	14	Prod. Scalaire	F5	14	Prod. Scalaire
Classif	10	Dist. Euclid	Classif	10	Dist. Euclid

5.4.1 Estimation de l'apprentissage

Le premier aspect à évaluer dans ce comparatif est l'impact de la configuration du réseau sur les performances de détection. L'environnement de développement utilisé intègre à la fois le processus d'apprentissage et l'évaluation des résultats sur la base de données MNIST (100 images de test). La Fig. 5.16 illustre le taux de bonnes détections en fonction du nombre d'itérations de l'algorithme d'apprentissage pour l'ensemble des réseaux. Les mesures ont été effectuées sur 10000 itérations de l'algorithme d'apprentissage mais seules les 2000 premières sont rapportées sur la Fig. 5.16 car les valeurs n'évoluent plus par la suite et n'apportent donc pas d'informations supplémentaires.

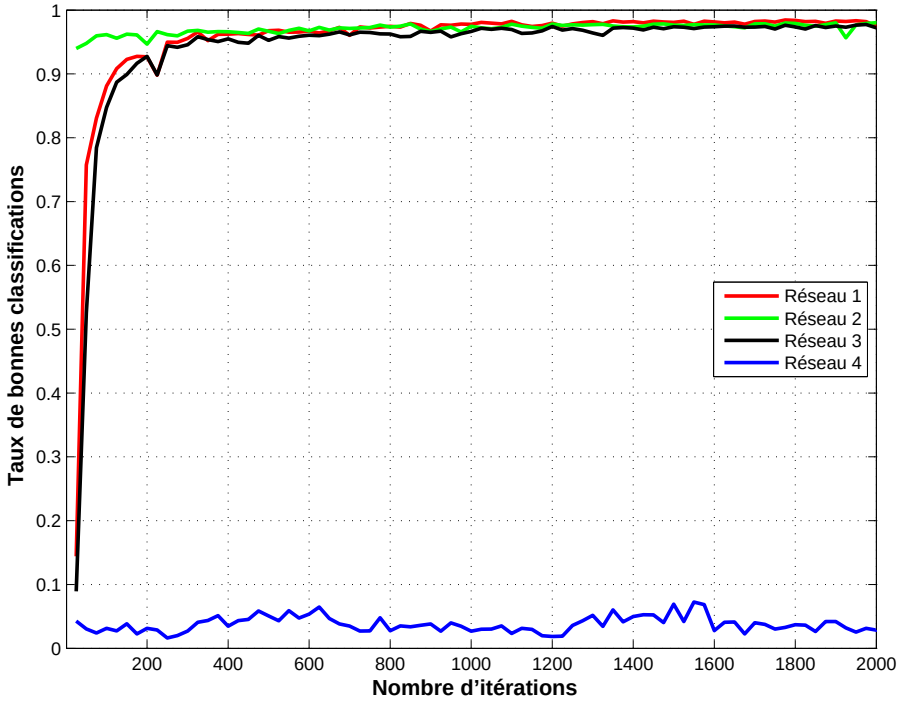


Fig. 5.16: Comparatif des taux de bonnes détections en fonction du nombre d'itérations de l'algorithme de classification sur les quatre réseaux explorés

Les trois premiers réseaux convergent assez rapidement (environ 1000 itérations) vers des taux de bonnes détections de l'ordre de 0.99. Le réseau original de LeCun est le plus efficace car il contient le plus d'éléments. Sur les 10000 itérations, le TPR maximal relevé est de 0.9912 et la valeur moyenne est de 0.9892 (on ne tient pas compte des 1000 première itérations de l'apprentissage). Le second réseau fournit un TPR maximum de 0.9867 avec une valeur moyenne de 0.9842. Le troisième réseau fournit un 0.9857 avec une valeur moyenne de 0.9831. Le dernier réseau lui ne fonctionne pas car il a été trop dégradé. La présence de la dernière couche de convolution est donc impérative au bon fonctionnement de la couche de classification. De cette expérimentation, on observe que la dégradation de nombre d'éléments dans chaque couche a un impact sur les performances de classification. Toutefois, la perte de fiabilité de détection des réseaux 2 et 3 est assez faible, respectivement 0.5% et 0.61%.

5.4.2 Implantations

Dans cette section sont proposés les résultats d'implantation des différents réseaux explorés. Le processus de synthèse s'est fait pour une cible Stratix V $5SGSED8N3F45I4$ (comme pour les systèmes SVMs à la Sec. 4.8) pour des raisons de ressources maximales. La table 5.3 rapporte les ressources obtenues.

Réseau	ALM (%)	Mémoire requise (bits)	Blocs M20K	Mémoire allouée (bits)	DSP	Freq (MHz)
$R2$	327786 (124.0)	120332			2983	
$R3$	84218 (32.0)	104262	434	8888320	456	61
$R4$	33870 (13.0)	41130	114	2334720	319	64

Tab. 5.3: Résultats d'implémentation sur Stratix V des différents réseaux

La simulation et la synthèse du premier réseau se sont avérés impossibles avec les outils habituels (Quartus) compte-tenu de sa taille. Pour le second réseau, nous n'avons pu obtenir que des résultats de synthèse partiels. En effet, les ressources requises étant supérieures à

la capacité maximale de la cible choisie, le processus de placement routage sur le FPGA n'a pas pu aboutir. En conséquence, certaines informations sont manquantes dans la table. 5.3, comme la fréquence maximale ou bien le nombre de blocs mémoire alloués sur la cible. Les deux réseaux $R3$ et $R4$ ont ou, eux, être synthétisés complètement. On ne discutera que de $R3$ car, comme on l'a vu $R4$ ne donne pas de résultat satisfaisant en termes de classification.

Les ressources d'implémentations requises par ce réseau sont résumées sur la table 5.4. Les résultats sont fournis par couche et par ensemble d'acteurs, et sont rapportés comme suit : nombre de blocs logiques ALM, mémoire requise, blocs mémoire $M20K$ alloués, mémoire implémentée sur la cible, nombre de blocs DSP et enfin fréquence maximale de la couche. Chaque couche a été synthétisée séparément, la somme des ressources ne sera donc pas égale à la valeur donnée dans la table. 5.3. L'objectif de la table 5.4 est de mettre en avant la distribution des ressources logiques entre les différents éléments reformulés afin d'extraire des informations sur les efforts d'optimisation à effectuer dans le futur.

Tab. 5.4: Ressources matérielles requises par le réseau $R3$. Chaque couche a été synthétisée séparément.

Opérations	Acteurs	ALM	Mémoire (*bits*)	M20K	Mémoire allouée (bits)	DSP	F_{max} (MHz)
Couche C1	`rep6`	9	0	0	0	0	63.9
	6× `conv_33`	1282	7560	12	245760	34	
	6× `biais`	37	0	0	0	0	
	6× `ReLU`	53	0	0	0	0	
	18× `fifo`	714	0	0	0	0	
Couche S2	6× `poolh`	255	0	0	0	0	64.2
	6× `poolv`	192	0	0	0	0	
	16× `fifo`	117	0	0	0	0	
Couche C3	24× `conv_33`	5335	16416	48	983040	113	61.1
	8× `sum`	64	0	0	0	0	
	8× `biais`	110	0	0	0	0	
	8× `ReLU`	64	0	0	0	0	
	64× `fifo`	2438	0	0	0	0	
Couche S4	8× `poolh`	301	0	0	0	0	62.2
	8× `poolv`	871	0	0	0	0	
	16× `fifo`	700	0	0	0	0	
Couche C5	80× `conv_33`	18443	54701	160	3276800	310	61.2
	20× `sum`	500	0	0	0	0	
	20× `biais`	300	0	0	0	0	
	10× `ReLU`	160	0	0	0	0	
	220× `fifo`	8686	0	0	0	0	
Couche FC	14× `dot`	1799	0	0	0	280	63.7
	14× `distp`	1476	0	0	0	0	
	14× `hsum`	390	0	0	0	0	
	14× `vsum`	656	0	0	0	0	
	14× `biais`	180	0	0	0	0	
	644× `fifo`	23407	0	0	0	0	
Couche Classif	140× `sub`	1715	0	0	0	0	62.2
	140× `square`	920	0	0	0	140	
	10× `sum`	730	0	0	0	0	
	430× `fifo`	16243	0	0	0	0	

Les ressources requises par le réseau $R3$ sont de 84218 éléments logiques ALM, soit 32% de la surface disponible. En ce qui concerne la mémorisation, 120332 bits sont nécessaires mais les blocs mémoires sont alloués par des acteurs différents, ce qui conduit à l'utilisation de 434

blocs distincts. Chaque bloc mémoire correspond à un espace de 20 Kbits dans ce FPGA, ce qui conduit au final à l'allocation de 8.8 Mbits, soit un taux d'utilisation réel de 1.2% des blocs mémoires. Ce résultat montre que l'allocation de bloc mémoires SRAM pour la mémorisation de lignes n'est pas toujours la solution optimale. La fréquence de fonctionnement du réseau se situe à 61 MHz. En considérant que la fréquence des acteurs est le double de celle des données, ce résultat correspondant à une capacité de traitement de 29785 images par seconde pour des images de résolution 32×32 (dimension des images du jeu de données MNIST). Exprimée sur des résolutions plus courantes, ceci permettrait de traiter des images VGA à 99 images par seconde et des images $720p$ (dimension maximale de l'imageur de la plateforme DreamCam) à 24 images par seconde.

Les résultats montrent que le nombre de canaux de communication FIFO est important, plus d'un millier dont plus la plupart sont repartis entre la dernière couche de convolution, la couche complètement connectée et la couche de classification. Ces FIFOs représentent au total 56% des ressources logiques utilisées. Il est clair dans cas que le surcout engendré par l'utilisation du modèle flot de données est significatif, un effort de reformulation dans le but de factoriser les acteurs doit être fait afin de réduire le nombre de canaux. La première modification possible serait de fusionner certains acteurs élémentaires. par exemple, au sein d'un neurone, les acteurs de `sum`, `biais` et `ReLU` pourraient être regroupés en un seul acteur. De même, les deux acteurs de sous-échantillonnage pourrait être fusionnés, tout comme les acteurs de soustraction `sub` et de carré `square` dans la couche de classification.

Le second problème provient des blocs mémoires instanciés dans chaque convoluer pour la mémorisation du voisinage. Finalement, le synthétiseur alloue sur cette cible deux blocs mémoires SRAM $M20K$ par instance. Comme les lignes dans cet exemple ne comportent que peu de données, il en résulte un gaspillage des ressources mémoire.

5.5 Conclusions

Dans ce chapitre, une première formulation flot de données d'un réseau de neurones convolutionnels a été donnée. Les acteurs et fonctions élémentaires proposés couvrent les opérations les plus couramment utilisées [KSH12]. Les performances obtenues en termes de fréquence de fonctionnement du code généré sont satisfaisantes. Les ressources logiques nécessaires sont toutefois importantes dans la mesure où la formulation utilisée conduit à une exploitation complète du parallélisme de données. La solution obtenue constitue donc une solution particulière du compromis surface-fréquence classiquement associé à ce type d'implantation. Ceci montre au passage de l'intérêt de disposer d'outils permettant d'explorer de la manière la plus aisée possible ce compromis.

Les perspectives sur ce domaine de recherche sont multiples. Tout d'abord sur les aspects HLS, l'existence d'un outil de transcription automatique (Caffee vers VHDL) est une demande forte de la communauté de la vision. En effet, automatiser cette transcription permettrait à la communauté de la vision et de l'apprentissage d'accéder à des technologies performantes et ceci de manière totalement transparente. Associée à une plateforme de développement (telle que la DreamCam), un tel outil permettrait également de valider le fonctionnement d'un réseau sur une plateforme réelle simplement (et non sur des bases de données). Une première version de cet outil est déjà fonctionnelle. Cet outil, appelé *Caphee*, est basé sur la transcription en CAPH de la formulation proposée dans ce chapitre, ainsi que sur les acteurs déjà formulés et implémentés. Le modèle flot de données sert de représentation intermédiaire entre la librairie d'apprentissage (Caffee) et l'implémentation matérielle.

Notons au passage que, du point de vue du modèle d'exécution, les réseaux de neurones convolutionnels ont un profil d'exécution totalement déterministe, au sens où les ratios production/consommation ne dépendent pas des données. Cette propriété devrait permettre à terme de spécialiser les techniques de compilation utilisée par CAPH en vue d'optimiser le code RTL produit.

D'une manière générale, le travail décrit dans ce chapitre a mis en évidence plusieurs optimisations qui pourrait être intégrées au compilateur CAPH. La première consisterait par exemple à factoriser automatiquement la mémorisation des voisinages nécessaires au calcul des convolutions, actuellement effectuée au sein de chaque acteur de convolution. La seconde consisterait en un mécanisme de sérialisation automatique de certaines opérations. Chaque couche de sous-échantillonnage réduit le nombre de données dans les cartes de primitives (d'un facteur 4 sur chaque couche dans notre cas). Alors que le nombre de neurones augmente à chaque couche, la mise en place d'une sérialisation des données après chaque couche de sous-échantillonnage permettrait de factoriser les opérations et le nombre de canaux, maîtrisant ainsi la largeur du graphe. Il est même envisageable, en fonction de la dimension d'images utilisée, d'exploiter la fréquence élevée obtenue afin de également multiplexer temporellement les calculs, permettant de réduire encore la dimension du graphe (et ainsi le nombre de ressources) tout en maintenant une rapidité de traitement. Les gains en terme d'implémentation pourrait permettre la mise en œuvre de réseaux encore jamais implanté à ce jour sur une architecture FPGA.

Enfin, une autre piste de réflexion pour le futur serait d'intégrer la taille des représentations en virgule fixe des données comme paramètre de l'apprentissage, ce qui n'a pas été exploré dans ces travaux. A notre connaissance, peu de travaux sont disponibles dans la littérature sur le sujet [GAGN15, TDP15]. Il serait notamment possible de transposer les techniques dites d'*approximate computing* [HO13], en élargissant les métriques existantes (qui évaluent le compromis précision/efficacité énergétique) avec les performances de classification ou le résultat de la fonction de coût calculée à chaque itération de l'algorithme.

6

Problématiques de transcription du modèle flot de données vers les systèmes matériels

L'objectif de ce chapitre est d'aborder les problématiques de transcription du modèle flot de données vers les systèmes matériels à travers l'utilisation de l'outil CAPH. Pour cela, l'impact de la notion clé de *granularité* sera étudiée sur un cas simple de convolution. La question de la sérialisation des données au sein d'un graphe flot de données sera également abordée. Le chapitre proposera quelques recommandations et perspectives d'évolution des modèles utilisés pour la génération de code VHDL à partir d'une transcription CAPH.

6.1 Le choix de la granularité d'un acteur/réseau : cas de la convolution 3x3

Une problématique couramment rencontrée avec l'utilisation du modèle flot de données est le choix de la granularité. Cette notion est liée à la complexité d'un acteur ou d'un réseau. Dans notre cas, une application avec un *grain fin* est définie comme une application dans laquelle les acteurs sont *élémentaires*, c'est-à-dire comportant peu de règles d'activations (typiquement moins d'une dizaine). Une application avec un *grain gros* utilisera des acteurs complexes (nombreuses règles d'activations).

L'approche typique dans le développement d'une application flot de données consiste à utiliser des acteurs avec un grain fin et de les composer afin d'obtenir le programme souhaité. Cependant, les performances finales de l'application (fréquence, LUT, registres, mémoire) sont dépendantes du nombre de canaux de communications utilisés et de la complexité de chaque acteur.

Afin d'évaluer l'impact de la granularité sur les performances finales, nous avons exploré cinq manières différentes de formuler le graphe d'une convolution 3×3, toutes fonctionnellement équivalentes. Les convolutions produisent une donnée en sortie pour chaque donnée consommée en entrée du graphe. Si le voisinage n'est pas complètement disponible, des valeurs spécifiques (0 par exemple) sont générées. Ce comportement est légèrement différent des convolutions centrées du chapitre 4 ou des convolutions supprimant les bords du chapitre 5, car il conduit à décaler l'image d'une ligne vers le bas et d'un pixel vers la droite par rapport à l'image d'entrée. Ce point est sans effet significatif sur les aspects qui nous intéressent ici. Les cinq formulations étudiées sont :

1. Une première formulation au sein de laquelle l'extraction du voisinage se fait à l'aide d'un graphe d'acteurs élémentaires de retard pixel et retard ligne. Cette forme utilise les acteurs de la bibliothèque standard CAPH (`D1P`, `D1LI` et `MADDN`). Les retards ligne sont réalisés par une mémorisation des pixels au sein de l'acteur dans un tableau.

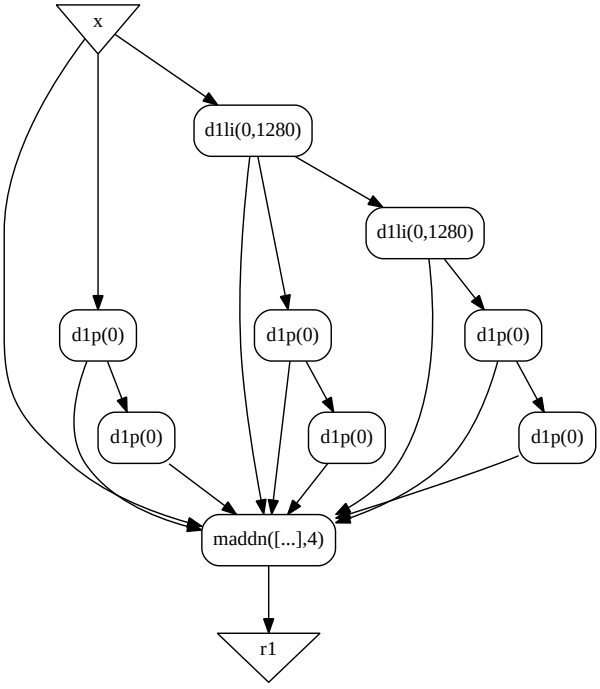
2. Une variante de cette première formulation dans laquelle la mémorisation de la ligne au sein de l'acteur de retard ligne est réalisée via une FIFO externe rebouclée, (acteur `d1lr`).
3. Une variante au sein de laquelle l'extraction de voisinage se fait avec un acteur de *mise sous forme locale* (MSFL) . Dans cette variante, un acteur dédié effectue une extraction explicite du voisinage 3×3 de chaque pixel, remplaçant ainsi les acteurs `D1P` et `D1L`. L'acteur `MADDN` pour le calcul de la convolution reste inchangé.
4. Une formulation mono-acteur au sein de laquelle un seul acteur est en charge de l'extraction explicite du voisinage et du calcul de la convolution. Comme pour la seconde et la troisième formulation, la mémorisation des lignes se fait également par des canaux de communication rebouclés.
5. Une variante de la version précédente dans laquelle les coefficients du masque de convolution sont fournis via des ports asynchrones, permettrait un changement dynamique du noyau de convolutions.

Les conditions d'expérimentations sont les suivantes :

- Un FPGA Cyclone V $5CSXFC6D6F31C6$ est utilisé. Le composant intégre 41910 blocs ALM, 112 blocs arithmétiques DSP ainsi que 553 blocs mémoires $M10K$.
- Le paramétrage de l'outil Quartus (version 15.0.2) est celui par défaut, avec une optimisation des ressources équilibrée entre fréquence et surface.
- La fréquence de fonctionnement souhaitée est fixée à 50 MHz. Nous cherchons en effet à éviter une optimisation en fréquence, qui conduirait à des résultats en termes de consommation de ressources peu significative.

6.1.1 Première formulation : D1P/D1LI/ADD

Le code CAPH et le graphe correspondant sont donnés sur la Fig. 6.1. Le graphe s'exprime par la composition d'une fonction de *mise sous forme locale* `neigh33` et d'un acteur d'addition normée *maddn*.



(a) Réseau d'acteurs

```
-- Voisinage 1x3
net neigh13 v x =
  let y2 = d1p v x in
  let y3 = d1p v y2 in
  (x, y2, y3);

-- Voisinage 3x3
net neigh33 z x =
  let xz = d1li z x in
  let xzz = d1li z xz in
  let (y11,y12,y13) = neigh13 z x in
  let (y21,y22,y23) = neigh13 z xz in
  let (y31,y32,y33) = neigh13 z xzz in
  y11,y12,y13,y21,y22,y23,y31,y32,y33;

-- Fonction
net conv (kernel, norm, pad) i =
    maddn (kernel, norm) (neigh33 pad i);
```

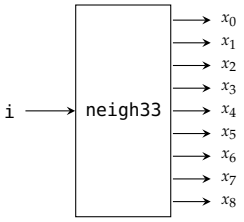
(b) Listing CAPH

Fig. 6.1: Réseau et code CAPH de la première formulation

Dans cette première formulation, la fonction `neigh33` transforme un flux structuré de données en un ensemble de flux correspondant au voisinage 3×3 de l'entrée. Cette opération est illustrée sur la Fig. 6.2a où x_4 est considéré comme le pixel dont le voisinage est généré.

x_8	x_7	x_6
x_5	x_4	x_3
x_2	x_1	x_0

(a) Voisinage 3×3



(b) Schéma de connexions

Considérons le flux d'entrée structuré suivant, représentant une image de n lignes de m pixels :

$$i = \langle\langle i_{1,1} \dots i_{1,m} \rangle\langle i_{2,1} \dots i_{2,m} \rangle\langle i_{3,1} \dots i_{3,m} \rangle \dots \langle i_{n,1} \dots i_{n,m} \rangle\rangle \tag{6.1}$$

L'application de la fonction de `neigh33` sur l'entrée i produira les flux suivants :

$$\begin{aligned}
x_0 &= \langle\langle v \dots v \rangle\langle v \dots v \rangle\langle v, v, i_{3,3} \dots i_{3,m} \rangle \dots \langle v, v, i_{n,3} \dots i_{n,m} \rangle\rangle \\
x_1 &= \langle\langle v \dots v \rangle\langle v \dots v \rangle\langle v, v, i_{3,2} \dots i_{3,m-1} \rangle \dots \langle v, v, i_{n,2} \dots i_{n,m-1} \rangle\rangle \\
x_2 &= \langle\langle v \dots v \rangle\langle v \dots v \rangle\langle v, v, i_{3,1} \dots i_{3,m-2} \rangle \dots \langle v, v, i_{n,1} \dots i_{n,m-2} \rangle\rangle \\
x_3 &= \langle\langle v \dots v \rangle\langle v \dots v \rangle\langle v, v, i_{2,3} \dots i_{2,m} \rangle \dots \langle v, v, i_{n-1,3} \dots i_{n-1,m} \rangle\rangle \\
x_4 &= \langle\langle v \dots v \rangle\langle v \dots v \rangle\langle v, v, i_{2,2} \dots i_{2,m-1} \rangle \dots \langle v, v, i_{n-1,2} \dots i_{n-1,m-1} \rangle\rangle \\
x_5 &= \langle\langle v \dots v \rangle\langle v \dots v \rangle\langle v, v, i_{2,1} \dots i_{2,m-2} \rangle \dots \langle v, v, i_{n-1,1} \dots i_{n-1,m-2} \rangle\rangle \\
x_6 &= \langle\langle v \dots v \rangle\langle v \dots v \rangle\langle v, v, i_{1,3} \dots i_{1,m} \rangle \dots \langle v, v, i_{n-2,3} \dots i_{n-2,m} \rangle\rangle \\
x_7 &= \langle\langle v \dots v \rangle\langle v \dots v \rangle\langle v, v, i_{1,2} \dots i_{1,m-1} \rangle \dots \langle v, v, i_{n-2,2} \dots i_{n-2,m-1} \rangle\rangle \\
x_8 &= \langle\langle v \dots v \rangle\langle v \dots v \rangle\langle v, v, i_{1,1} \dots i_{1,m-2} \rangle \dots \langle v, v, i_{n-2,1} \dots i_{n-2,m-2} \rangle\rangle
\end{aligned}$$

Le premier acteur utilisé pour l'extraction du voisinage est un retard pixel `d1p`, défini par :

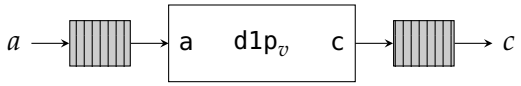
$$\texttt{d1p}_v(\langle x_1, x_2 \dots, x_n \rangle) = \langle v, x_1, \dots, x_{n-1} \rangle \tag{6.2}$$

L'implémentation de cet acteur est décrite sur la Fig. 6.2. Cet acteur opère sur des flux structurés d'images en retardant le flux d'un jeton pour chaque ligne de l'image. L'acteur utilise une variable d'état s et un buffer z pour créer le décalage. Les deux premières règles permettent de gérer le niveau de structure associées aux images. La règle à la ligne 9 est activée en début de chaque ligne de l'image, et réinitialise le buffer z à la valeur v (v est un paramètre statique de l'acteur). Pour chaque jeton de données (ligne 10), la valeur du buffer z est écrite sur la sortie c et la valeur du buffer z est mise à jour avec la valeur d'entrée lue sur le canal a.

En utilisant deux acteurs `d1p` chaînés, on obtient l'extraction d'un voisinage 1×3, définie par la fonction `neigh13` (listing. 6.1b). Le second acteur utilisé dans le graphe est l'acteur de retard ligne `d1li`.

$$\texttt{d1li}_{v,maxw}(\langle\langle x_{1,1}, \dots, x_{1,m} \rangle \dots \langle x_{n,1}, \dots, x_{n,m} \rangle\rangle) = \langle\langle v, \dots, v \rangle \dots \langle x_{n-1,1}, \dots, x_{n-1,m-1} \rangle\rangle \tag{6.3}$$

L'implémentation de cet acteur est illustrée sur la Fig. 6.3. L'acteur prend deux paramètres statiques (v et $maxw$), utilise cinq états et deux variables internes z (tableau) et i (indice du tableau). Le paramètre $maxw$ permet de fixer la dimension du tableau interne. Les deux premières règles permettent également de gérer les jetons délimitant la structure d'une image. Au début de la première ligne de l'image, la variable i est initialisé à 0. L'état $S2$ (ligne 12)



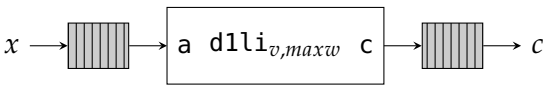
(a) Schéma de connexions

```
1  actor d1p (v:$t)
2    in (a:$t dc)
3    out (c:$t dc)
4  var s : {S0,S1,S2} = S0
5  var z : $t
6  rules
7  | (s:S0, a:'<) -> (s:S1, c:'<)
8  | (s:S1, a:'>) -> (s:S0, c:'>)
9  | (s:S1, a:'<) -> (s:S2, c:'<, z:v)
10 | (s:S2, a:'p) -> (s:S2, c:'z, z:p)
11 | (s:S2, a:'>) -> (s:S1, c:'>);
```

(b) Listing CAPH

Fig. 6.2: Acteur `d1p`

correspond à la mémorisation de la première ligne où les jetons d'entrée sont stockés dans le tableau z et l'acteur produit en sortie des valeurs v (donné en paramètre). Les états $S3$ et $S4$ représentent la boucle de calcul principale. Pour chaque donnée, l'acteur produit en sortie le contenu du tableau $z[i]$, et en même temps, continue de stocker les données d'entrée dans le tableau z. A chaque fin de ligne, l'acteur retourne dans l'état $S3$ où le jeton suivant déterminera le comportement. Si celui ci est un jeton de début de ligne l'acteur retourne dans l'état $S4$ en réinitialisant l'adresse du tableau i à 0. Si le jeton suivant est celui de fin d'image, l'acteur retourne dans l'état initial $S0$.



(a) Schéma de connexions

```
1  actor d1li (v:$t, maxw:int)
2    in (a:$t dc)
3    out (c:$t dc)
4  var s : {S0,S1,S2,S3,S4}=S0
5  var z : $t array[maxw] = [v | i=0 to maxw-1]
6  var i : int
7  rules
8  | (s:S0, a:'<) -> (s:S1, c:'<)
9  | (s:S1, a:'>) -> (s:S0, c:'>)
10 | (s:S1, a:'<) -> (s:S2, c:'<, i:0)
11 | (s:S2, a:'>) -> (s:S3, c:'>)
12 | (s:S2, a:'p) -> (s:S2, c:'v, z[i]:p, i:i+1)
13 | (s:S3, a:'>) -> (s:S0, c:'>)
14 | (s:S3, a:'<) -> (s:S4, c:'<, i:0)
15 | (s:S4, a:'p) -> (s:S4, c:'z[i], z[i]:p, i:i+1)
16 | (s:S4, a:'>) -> (s:S3, c:'>);
```

(b) Listing CAPH

Fig. 6.3: Acteur `d1li`

Le dernier acteur `maddn` fait le produit des neuf entrées (issue de la fonction `neigh33`) avec un noyau de convolution $\mathbf{k}$.

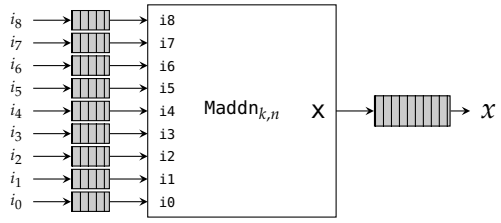
$$\texttt{maddn}_{\mathbf{k},n}(x) = \texttt{mlift}_9(prod_{k,n}) \tag{6.4}$$

avec

$$prod_{k,n}(i_0,\cdots,i_8) = \Big(\sum_{l=0}^{8} i_{l_{i,j}} \times k_l\Big)/2^n$$
$$i_l = \langle\langle\, i_{l_{1,1}} \ldots i_{l_{1,m}} \,\rangle \ldots \langle i_{l_{n,1}} \ldots i_{l_{n,m}} \,\rangle\rangle, \quad l \in \{1,\cdots,8\}$$

Cet acteur, illustré sur la Fig. 6.4 n'est composé que de trois règles d'activations. Les deux premières concernent la gestion des jetons de début et fin de structure, qui sont recopiés en sortie. La dernière règle traite les jetons de données. Chaque valeur est multipliée par le poids correspondant et la somme des neufs produits est décalée du facteur de normalisation n.

Les résultats d'implémentation sur FPGA Cyclone V sont disponibles dans la table 6.1 en fonction de la dimension d'image d'entrée, ceci afin d'observer l'impact de celle-ci. On donne de gauche à droite, le nombre de blocs ALM requis, la mémoire requise (exprimée en bits



(a) Schéma de connexions

```
actor maddn(k:signed<m> array[9], n:unsigned<4>)
 in(io:signed<m> dc, i1:signed<m> dc, i2:signed<m> dc,
    i3:signed<m> dc, i4:signed<m> dc, i5:signed<m> dc,
    i6:signed<m> dc, i7:signed<m> dc, i8:signed<m> dc)
 out(o:signed<m> dc)
rules
|(io:'<,i1:'<,i2:'<,i3:'<,i4:'<,i5:'<,i6:'<,i7:'<,i8:'<) -> o:'<
|(io:'>,i1:'>,i2:'>,i3:'>,i4:'>,i5:'>,i6:'>,i7:'>,i8:'>) -> o:'>

|(io:'x0,i1:'x1,i2:'x2,i3:'x3,i4:'x4,i5:'x5,i6:'x6,i7:'x7,i8:'x8)->
    o:'((k[0]*x0+k[1]*x1+k[2]*x2+k[3]*x3+k[4]*x4+k[5]*x5+k[6]*x6+k
        [7]*x7+k[8]*x8)>>n);
```

(b) Listing CAPH

Fig. 6.4: Acteur `maddn`

(*Mem req.*) et en nombre de blocs mémoires SRAM $M10K$), le nombre de blocs arithmétiques DSP et la fréquence maximale de fonctionnement du réseau rapporté par l'outil Quartus (avec les spécifications données en début de cette section).

Taille d'image	ALM (%)	Mémoire requise (bits)	Blocs M10K	Mémoire allouée (bits)	DSP	Freq (MHz)
64×64	2687 (6.4)	0	0	0	0	82.67
128×128	4468 (10.6)	0	0	0	0	82.04
256×256	8031 (19.1)	0	0	0	0	69.91
512×512	15096 (36.0)	0	0	0	0	69.91
840×640	24176 (57.7)	0	0	0	0	73.12
1280×960	36382 (86.8)	0	0	0	0	68.19

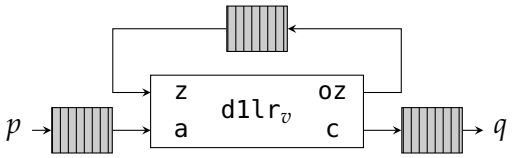
Tab. 6.1: Résultats d'implémentation sur Cyclone V de la formulation `d1li/d1p/maddn`

Les résultats de la table 6.1 montrent que cette formulation n'utilise que des éléments logiques ALM. Ceci s'explique par l'usage du tableau `z` pour la mémorisation des lignes au sein de l'acteur `d1li`. Malheureusement, le synthétiseur n'est pas en mesure d'interpréter que les accès à ce tableau sont équivalents à des accès à une mémoire et infère dans ce cas un ensemble de registres. Le nombre d'éléments utilisés est directement proportionnel à la largeur de l'image d'entrée, ce qui est handicapant dans le cas de grandes images. Dans notre cas, une convolution sur une image de dimension 1280×960 requiert plus de 85% des ressources logiques du FPGA pour la mémorisation des deux lignes précédentes. La fréquence moyenne est de $74MHz$ sur cet exemple (avec un écart type $\sigma = 6.43$), avec un maximum de $82MHz$ et chutant à $70MHz$ lorsque la surface allouée augmente.

6.1.2 Deuxième formulation : D1P/D1LR/ADD

La seconde version proposée est assez proche de la première. Elle n'en diffère qu'au niveau de l'acteur de retard ligne utilisé dans la fonction `neigh33`. L'acteur `d1lr` utilisé ici réalise la mémorisation de la ligne via une FIFO externe rebouclée. La représentation de l'acteur `d1lr` est donnée sur la Fig. 6.5. La description de l'acteur est similaire à celle de l'acteur `d1li` à ceci près que :

- les variables `z` et `i` ont disparues,
- chaque écriture dans le tableau $z[i]$ est remplacée par une écriture sur le canal `oz`,
- chaque lecture du tableau est remplacée par une lecture sur le canal `z`,
- une règle est ajoutée pour le vidage de la FIFO entre deux images successives [1]
- Pour fonctionner, la sortie `oz` de l'acteur doit être rebouclée sur l'entrée `z`. C'est le rôle de la fonction `d1l` qui encapsule le sous graphe correspondant à la Fig. 6.5a.



(a) Schéma de connexions

```
1  actor d1lr (v:$t)
2    in (a:$t dc, z:$t)
3    out (c:$t dc, oz:$t)
4  var s : {So,S1,S2,S3,S4}=So
5  rules
6  |(s:So, a:'<)        -> (s:S1, a:'<)
7  |(s:S1, a:'>)        -> (s:So, a:'>)
8  |(s:S1, a:'<)        -> (s:S2, a:'<)
9  |(s:S2, a:'>)        -> (s:S3, a:'>)
10 |(s:S2, a:'p)        -> (s:S2, a:'v, oz:p)
11 |(s:S3, a:'>)        -> (s:So, a:'>)
12 |(s:S3, a:'<)        -> (s:S4, a:'<)
13 |(s:S4, a:'p, z:'z) -> (s:S4, a:'z, oz:p)
14 |(s:S4, a:'>)        -> (s:S3, a:'>)
15 |(s:So, z:'z)        -> (s:So); -- vidage fifo
16
17 -- fonction d'encapsulation
18 net d1l z i =
19   let rec (o,l) = d1lr z (i,l) in o;
```

(b) Listing CAPH

Fig. 6.5: Acteur `d1lr`

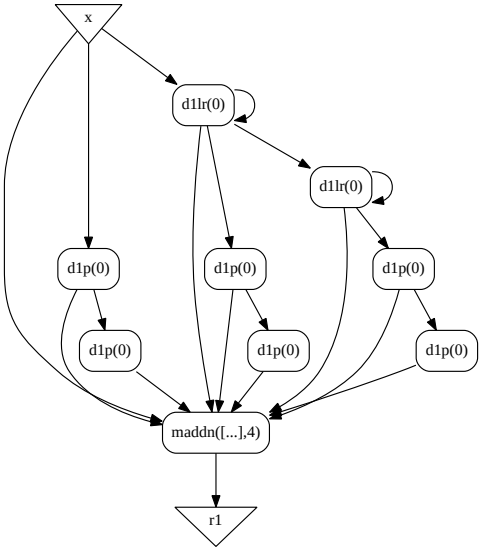


Fig. 6.6: Réseau CAPH pour la formulation `d1lr/d1p/maddn`. Les FIFOs rebouclées sont représentées par un arc rebouclé sur les acteur `d1lr`

La Fig. 6.6 illustre le réseau d'acteurs obtenu avec le listing. 6.5b en remplaçant l'acteur de retard ligne `d1li` par la fonction `d1lr`. Les résultats d'implémentation de ce réseau sont donnés dans la table 6.2.

On constate que le nombre d'éléments logiques ne dépend plus de la dimension de l'image et ce nombre reste faible (moins de 3% des ressources logiques totales). Comparé à la formulation précédente, les empreintes logique sont réduites d'un facteur 2 pour une image de dimension 64×64 et d'un facteur 30 pour une dimension 1280×960.

En contrepartie, bien entendu, de la mémoire est allouée. On note qu'ici l'outil de synthèse instancie des blocs mémoires synchrones SRAM double-port dédiés ($M10K$) automatiquement. Ceci est possible grâce à l'utilisation des FIFOs rebouclées, utilisées pour la mémorisation du voisinage.

Pour analyser ces résultats, il faut savoir que la bibliothèque de support VHDL fournie par CAPH comprend en fait deux modèles de FIFO. Ces deux modèles ont la même interface et le même comportement fonctionnel. La différence se situe sur l'interprétation qu'en font

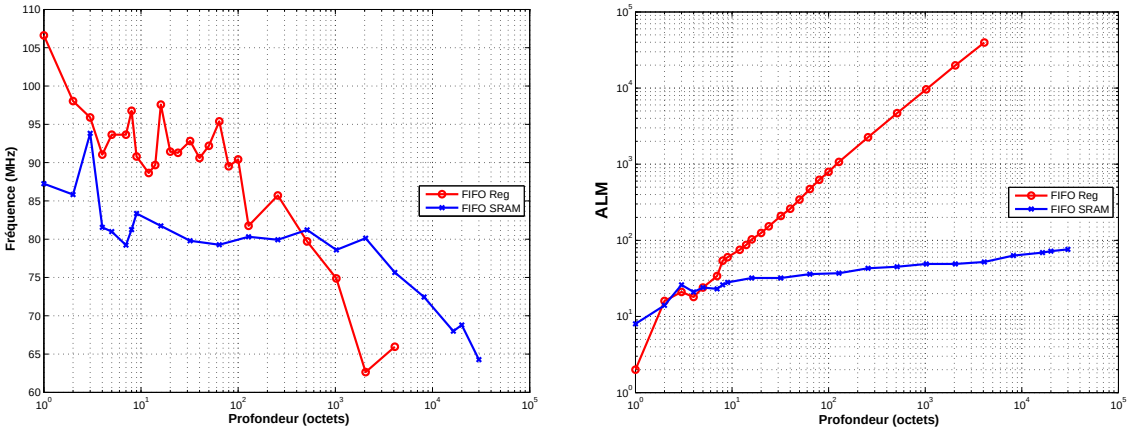
1. Le vidage de la FIFO peut également être fait pendant la mémorisation de la première ligne

Taille d'image	ALM (%)	Mémoire requise (bits)	Blocs M10K	Mémoire allouée (bits)	DSP	Freq (MHz)
64 × 64	1200 (2.8)	2144	2	20480	0	87.6
128 × 128	1207 (2.9)	4192	2	20480	0	82.0
256 × 256	1216 (2.9)	8288	2	20480	0	79.8
512 × 512	1222 (2.9)	16480	4	40960	0	83.8
840 × 640	1231 (2.9)	26976	4	40960	0	80.3
1280 × 960	1239 (2.9)	41056	8	81920	0	81.1
1920 × 1080	1229 (2.9)	61536	8	81920	0	79.1
4096 × 2160	1225 (2.9)	131168	32	327680	0	83.5

Tab. 6.2: Résultats d'implémentation de la formulation `d1lr/d1p/maddn`.

les outils de synthèse (voir annexe D). Le premier modèle, basé un ensemble de registres à décalage, va générer une FIFO qui sera instanciée sous forme de blocs logiques (LUT + registres). Le second modèle est décrit sous la forme d'une mémoire circulaire avec deux pointeurs, un pour la lecture et un pour l'écriture. Son fonctionnement peut se ramener à celui d'une mémoire, que les outils de synthèse sont capables d'instancier sous la forme de blocs mémoire SRAM [2]. L'utilisation de ces deux modèles permet une meilleure gestion des ressources matérielles. Le *seuil de basculement* entre les deux est un paramètre clé de la génération du code VHDL, car sa valeur définira si une FIFO sera implémentée en éléments logiques ou utilisera un bloc mémoire SRAM.

Ce paramètre est empirique car il dépend à la fois de l'architecture des éléments logiques, du nombre de blocs mémoire disponibles sur le FPGA choisi, mais également de l'application finale. Lors de la génération du code VHDL du réseau d'acteurs CAPH, il est possible de modifier la valeur de ce seuil (par défaut égale à 32) afin d'optimiser au mieux les ressources requises par les canaux de communications. La Fig. 6.7 présente une évaluation pratique des deux modèles pour un FPGA Cyclone V.



(a) Evolution de la fréquence maximale de la FIFO en fonction de la profondeur

(b) Evolution du taux d'occupation des ressources logiques ALM en fonction de la profondeur

Fig. 6.7: Comparatif de performances en ressources et fréquence des deux modèles de FIFOs utilisés dans CAPH.

Dans l'exemple qui nous intéresse, le nombre d'éléments dans la FIFO rebouclée est de 64 (égal au nombre de pixels dans une ligne). Ce nombre étant supérieur au seuil de basculement entre les deux modèles, le modèle utilisé pour la FIFO rebouclée conduit bien à l'implantation

2. L'interprétation a été validée sur les deux fabricants principaux du marché des FPGAs, Altera et Xilinx.

d'une mémoire SRAM double ports

Notons toutefois que l'instanciation de blocs SRAM peut impliquer une surconsommation des ressources mémoires. Pour une dimension de 64×64 par exemple, l'implémentation requiert 2144 bits mémoire. L'outil de synthèse va utiliser deux blocs $M10K$ car il est impossible de regrouper les deux FIFOs en un seul bloc. La mémoire totale réellement allouée alors est de 20480 bits soit une utilisation réelle de 10.4% de chaque bloc mémoire. Les résultats pour les deux dimensions suivantes (respectivement 128×128 et 256×256) montrent que les ressources nécessaires sont égales, avec une utilisation plus efficace des mémoires SRAM (resp. 20.4% et 40.7%). D'un point de vue de la fréquence, cette formulation fournit une fréquence de fonctionnement moyenne de $82MHz$ et aussi plus stable ($\sigma = 2.87$) lorsque la dimension d'image augmente.

6.1.3 Troisième formulation : Version MSFL/MADDN

Dans cette troisième version, on propose de remplacer la fonction de mise sous forme locale `neigh33` par un seul acteur. L'objectif est de limiter le nombre d'acteurs, réduisant ainsi le nombre de canaux de communication dans le but d'économiser des ressources matérielles. Le code de cet acteur est donné sur la Fig. 6.8c. Comme l'acteur `d1lr` présenté à la section précédente, l'acteur `msfl_33` utilise deux FIFOs externes rebouclées pour mémoriser les lignes précédentes. Les variables internes `x1,x2,x4,x5,x7,x8` stockent le voisinage, selon le schéma indiqué sur la Fig. 6.2a.

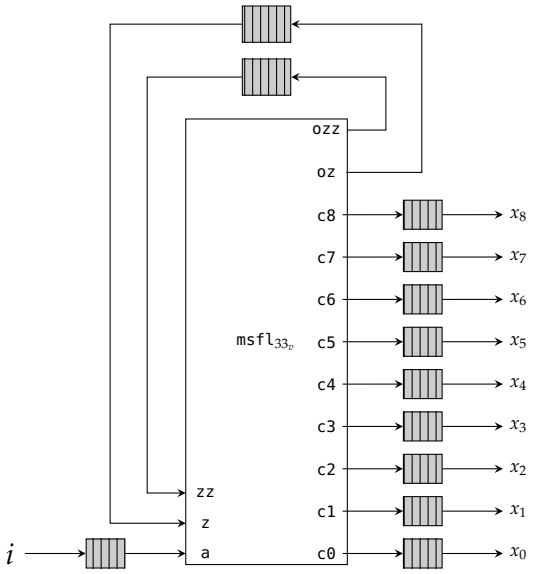
Les premières règles (lignes 18 à 24) correspondent à la mémorisation des deux premières lignes de l'image. Pendant cette phase, l'acteur produit en sortie la valeur du paramètre v sur l'ensemble des sorties. Les données entrantes lues sur le canal `a` sont systématiquement renvoyées sur le canal `oz`. La seconde mémorisation a lieu à partir de l'état `BufL2` où les données lues sur le canal `z` sont renvoyées sur le canal `ozz`. A partir de l'état `WaitNL`, l'acteur a dans ses deux FIFOs rebouclées les données des deux lignes précédentes. Les états `Bufpix1` et `Bufpix2` permettent de construire le voisinage 3×3 où les données entrantes (canaux `a`, `z` et `zz`) sont mémorisées dans les variables internes de l'acteur. Dans l'état `Prod` (ligne 32), pour chaque jeton entrant, le voisinage complet est produit sur les neuf sorties.

Comme pour l'acteur `d1lr`, une fonction d'encapsulation (`msfl_m`) permet d'abstraire les connexions des FIFOs rebouclées. Les résultats d'implémentation de cette formulation sont disponibles dans la table 6.3.

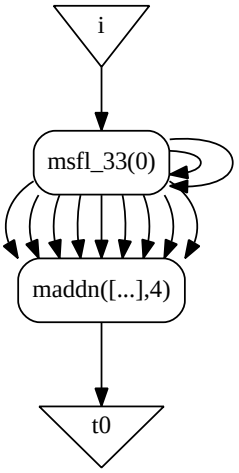
Taille d'image	ALM (%)	Mémoire requise (bits)	Blocs M10K	Mémoire allouée (bits)	DSP	Freq (MHz)
64×64	306 (0.7)	2144	2	20480	0	93.7
128×128	314 (0.7)	4192	2	20480	0	91.0
256×256	318 (0.7)	8288	2	20480	0	93.4
512×512	327 (0.8)	16480	4	40960	0	88.0
840×640	324 (0.8)	26976	4	40960	0	91.8
1280×960	328 (0.8)	41056	8	81920	0	89.6
1920×1080	338 (0.8)	61536	8	81920	0	85.7
4096×2160	338 (0.8)	131168	32	327680	0	84.8

Tab. 6.3: Résultats d'implémentation de la formulation msfl/maddn

Le nombre d'ALM a diminué d'un facteur 3.77 en moyenne par rapport à la formulation précédente. Cette diminution s'explique par la réduction du nombre de canaux (13 contre 20 dans la précédente). L'usage des FIFOs rebouclées pour la mémorisation des lignes, même si cette approche rend la description de l'acteur moins lisible, permet de forcer l'utilisation



(a) Schéma de connexions



(b) Réseau CAPH pour la formulation MSFL/MADDN

```
1  -- Acteur
2  actor msfl_33(v:signed<m>)
3    in (a:signed<m> dc, z:signed<m> dc, zz:signed<m> dc)
4   out (co:signed<m> dc, c1:signed<m> dc, c2:signed<m> dc, c3:signed<m> dc,
5        c4:signed<m> dc, c5:signed<m> dc, c6:signed<m> dc, c7:signed<m> dc,
6        c8:signed<m> dc, oz:signed<m> dc, ozz:signed<m> dc)
7
8  var s : {WaitSoF,WaitSoL1,BufL1,WaitSoL2,BufL2,WaitNL,Bufpix1,Bufpix2,Prod} = WaitSoF
9
10 var x1 :signed<m>
11 var x2 :signed<m>
12 var x4 :signed<m>
13 var x5 :signed<m>
14 var x7 :signed<m>
15 var x8 :signed<m>
16
17 rules
18 | (s:WaitSoF , z:z, zz:zz)       -> (s:WaitSoF)  -- vidage fifos
19 | (s:WaitSoF ,a:'<)              -> (s:WaitSoL1, co:'<, c1:'<, c2:'<, c3:'<, c4:'<, c5:'<, c6:'<, c7:'<, c8:'<)
20 | (s:WaitSoL1,a:'<)              -> (s:BufL1,    co:'<, c1:'<, c2:'<, c3:'<, c4:'<, c5:'<, c6:'<, c7:'<, c8:'<, oz:'<)
21 | (s:BufL1,    a:'p)            -> (s:BufL1,    co:'v, c1:'v, c2:'v, c3:'v, c4:'v, c5:'v, c6:'v, c7:'v, c8:'v, oz:'p)
22 | (s:BufL1,    a:'>)            -> (s:WaitSoL2, co:'>, c1:'>, c2:'>, c3:'>, c4:'>, c5:'>, c6:'>, c7:'>, c8:'>, oz:'>)
23 | (s:WaitSoL2,a:'<, z:'<)       -> (s:BufL2,    co:'<, c1:'<, c2:'<, c3:'<, c4:'<, c5:'<, c6:'<, c7:'<, c8:'<, ozz:'<, oz:'<)
24 | (s:BufL2,    a:'x0,z:'x3)     -> (s:BufL2,    co:'v, c1:'v, c2:'v, c3:'v, c4:'v, c5:'v, c6:'v, c7:'v, c8:'v, ozz:'x3,oz:'x0)
25 | (s:BufL2,    a:'>, z:'>)      -> (s:WaitNL,   co:'>, c1:'>, c2:'>, c3:'>, c4:'>, c5:'>, c6:'>, c7:'>, c8:'>, ozz:'>, oz:'>)
26 | (s:WaitNL,  a:'<, z:'<,zz:'<) -> (s:Bufpix1, co:'<, c1:'<, c2:'<, c3:'<, c4:'<, c5:'<, c6:'<, c7:'< ,c8:'<, ozz:'<, oz:'<)
27 | (s:WaitNL,  a:'>, z:'<,zz:'<) -> (s:WaitSoF, co:'>, c1:'>, c2:'>, c3:'>, c4:'>, c5:'>, c6:'>, c7:'>, c8:'>)
28 | (s:Bufpix1, a:'x0,z:'x3,zz:'x6)->(s:Bufpix2, co:'v, c1:'v, c2:'v, c3:'v, c4:'v, c5:'v, c6:'v, c7:'v, c8:'v,
29                                                 x7:x6, x4:x3, x1:x0, ozz:'x3, oz:'x0)
30
31 | (s:Bufpix2, a:'x0, z:'x3, zz:'x6) -> (s:Prod, co:'v, c1:'v, c2:'v, c3:'v, c4:'v, c5:'v, c6:'v, c7:'v,c8:'v,
32                                                 x8:x7, x5:x4, x2:x1, x7:x6, x4:x3, x1:x0, ozz:'x3, oz:'x0)
33
34 | (s:Prod, a:'x0, z:'x3, zz:'x6) -> (s:Prod, co:'x0, c1:'x1, c2:'x2, c3:'x3, c4:'x4, c5:'x5, c6:'x6, c7:'x7, c8:'x8,
35                                                 x8:x7, x5:x4, x2:x1, x7:x6, x4:x3, x1:x0, ozz:'x3, oz:'x0)
36
37 | (s:Prod, a:'>, z:'>, zz:'>) -> (s:WaitNL, co:'>, c1:'>, c2:'>, c3:'>, c4:'>, c5:'>, c6:'>, c7:'> , c8:'>,ozz:'>, oz:'> );
38
39 -- Fonction d'encapsulation
40 net msfl33_m  pad i =
41     let rec (o0,o1,o2,o3,o4,o5,o6,o7,o8,z,zz) = msfl_33 (pad) (i,z,zz) in (o0,o1,o2,o3,o4,o5,o6,o7,o8);
```

(c) Listing CAPH

Fig. 6.8: Acteur `msfl`

de blocs mémoires. La fréquence maximale de l'acteur reste assez élevée (moyenne 89.7MHz, $\sigma = 3.5$) bien que la complexité de l'acteur ait également augmenté.

Toutefois, les performances proposées par cette solution se font au détriment de la lisibilité et de la flexibilité du code. La présence de neuf canaux de sortie ainsi que la gestion des canaux rebouclés rend la description de l'acteur peu lisible. Elle rend aussi sa généralisation à un voisinage $n \times n$ plus difficile (par contraste, la fonction `neigh33` introduite sur le listing 6.1b) est facilement généralisable à des voisinages plus importants).

6.1.4 Quatrième formulation : mono-acteur

La quatrième version propose de fusionner les deux acteurs `msfl` et `maddn` en un seul acteur `conv33`. Cette version a l'avantage de supprimer encore des FIFOs du graphe, en intégrant le calcul de la convolution et la mémorisation du voisinage dans le même acteur. Fonctionnellement, cet acteur `conv33` peut être exprimé sous la forme :

$$\texttt{conv}_{\mathbf{k},v,n}(p) = o \tag{6.5}$$

avec

$$p = \langle \langle p_{1,1} \dots p_{1,m} \rangle \langle p_{2,1} \dots p_{2,m} \rangle \langle p_{3,1} \dots p_{3,m} \rangle \dots \langle p_{n,1} \dots p_{n,m} \rangle \rangle$$
$$o = \langle \langle v \dots v \rangle \langle v \dots v \rangle \langle v, v, o_{3,3} \dots o_{3,m} \rangle \dots \langle v, v, o_{n,3} \dots o_{n,m} \rangle \rangle$$

où

$$o_{i,j} = \begin{cases} v, & \text{si } i \in \{1,2\} \vee j \in \{1,2\} \\ (k_0 \cdot p_{i,j} + k_1 \cdot p_{i,j-1} + k_2 \cdot p_{i,j-2} + k_3 \cdot p_{i-1,j} + k_4 \cdot p_{i-1,j-1} + \\ k_5 \cdot p_{i-1,j-2} + k_6 \cdot p_{i-2,j} + k_7 \cdot p_{i-2,j-1} + k_8 \cdot p_{i-2,j-2}) >> n & \text{sinon} \end{cases}$$

Le code de l'acteur donné sur la Fig. 6.9c. Son fonctionnement est très proche de celui de l'acteur `msfl_33`, précédemment expliqué. La différence majeure se trouve sur l'état `S8` où l'on effectue le calcul de la convolution avec le noyau **k**, ainsi que la normalisation du résultat (décalage de n bits) alors que l'état `Prod` de l'acteur `msfl_33` se contentait de produire les données du voisinage en parallèle sur les neufs canaux de sortie. Le réseau d'acteurs devient trivial et n'est plus composé que d'un seul acteur `conv33`, comme illustré sur la Fig. 6.9b.

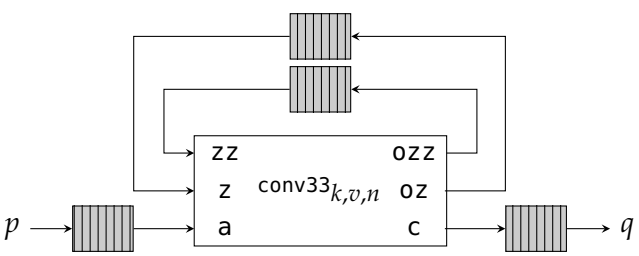
La table 6.4 montre les résultats d'implémentation pour cette formulation. Le nombre d'éléments logiques a encore été réduit, toujours dû à la diminution du nombre de canaux, mais de manière moins sensible (13% de réduction par rapport à la formulation précédente). La fréquence moyenne est de $87MHz$ avec un écart type de $\sigma = 3.11MHz$. La lisibilité du code a également été amélioré par rapport à la version précédente, par la diminution du nombre de sorties à spécifier sur chacune des règles.

Taille d'image	ALM (%)	Mémoire requise (bits)	Blocs M10K	Mémoire allouée (bits)	DSP	Freq (MHz)
64 × 64	261 (0.4)	2144	2	20480	0	84
128 × 128	273 (0.4)	4192	2	20480	0	89.4
256 × 256	287 (0.5)	8288	2	20480	0	86.7
512 × 512	284 (0.5)	16480	4	40960	0	89.7
840 × 640	292 (0.5)	26976	4	40960	0	82
1280 × 960	302 (0.5)	41056	8	81920	0	87.7
1920 × 1080	302 (0.5)	61536	8	81920	0	91.2
4096 × 2160	299 (0.5)	131168	32	327680	0	85.4

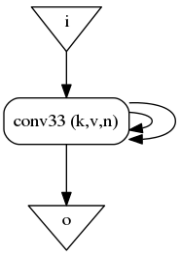
Tab. 6.4: Résultats d'implémentation de la formulation mono-acteur

6.1.5 Cinquième formulation : mono-acteur avec ports

Dans les quatre versions présentées ci-dessus, le nombre de blocs DSP alloués est resté à 0, signe que les multiplications étaient implantées de manière *cablée* par l'outil de synthèse. Ceci s'explique par le fait que le noyau de convolution était toujours spécifié sous la forme d'un paramètre statique.



(a) Schéma de connexions



(b) Réseau CAPH pour la formulation mono acteur

```
actor conv33 (k:int<s,m> array[9], n:int, v:int<s,m>)
  in (a:int<s,m> dc, z:int<s,m>, zz:int<s,m>)
  out (c:int<s,m> dc, oz:int<s,m>, ozz:int<s,m>)

var s : {S0,S1,S2,S3,S4,S5,S6,S7,S8} = S0
var x1 : int<s,m>
var x2 : int<s,m>
var x4 : int<s,m>
var x5 : int<s,m>
var x7 : int<s,m>
var x8 : int<s,m>
rules
| (s:S0, z:z, z:zz)           -> (s:S0) --vidage fifos
| (s:S0, a:'<)               -> (s:S1, c:'<)
| (s:S1, a:'>)               -> (s:S0, c:'>)
| (s:S1, a:'<)               -> (s:S2, c:'<)
| (s:S2, a:'p)               -> (s:S2, c:'v, oz:p)
| (s:S2, a:'>)               -> (s:S3, c:'>)
| (s:S3, a:'<)               -> (s:S4, c:'<)
| (s:S4, a:'x0, z:x3 )       -> (s:S4, c:'v, ozz:x3, oz:x0)
| (s:S4, a:'>)               -> (s:S5, c:'>)
| (s:S5, a:'>)               -> (s:S0, c:'>)
| (s:S5, a:'<)               -> (s:S6, c:'<)
| (s:S6, a:'x0, z:x3, zz:x6) -> (s:S7, c:'v, x7:x6, x4:x3, x1:x0, ozz:x3, oz:x0)
| (s:S7, a:'x0, z:x3, zz:x6) -> (s:S8, c:'v, x8:x7, x5:x4, x2:x1, x7:x6, x4:x3, x1:x0, ozz:x3, oz:x0)
| (s:S8, a:'x0, z:x3, zz:x6) -> (s:S8, c:'((k[0]*x0+k[1]*x1+k[2]*x2+k[3]*x3+k[4]*x4+k[5]*x5+k[6]*x6+k[7]*x7+k[8]*x8)>>n),
                                x8:x7, x5:x4, x2:x1, x7:x6, x4:x3, x1:x0, ozz:x3, oz:x0)
| (s:S8, a:'>)               -> (s:S5, c:'>);

net conv233 kernel norm pad i = let rec (o,z,zz) = conv233a (kernel,norm,pad) (i,z,zz) in o;
```

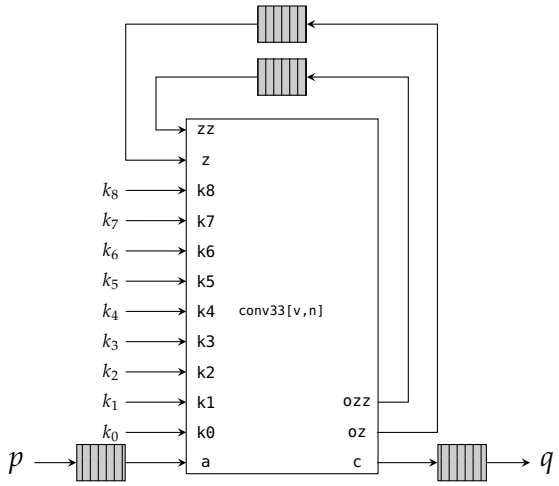
(c) Listing CAPH

Fig. 6.9: Acteur `conv33`

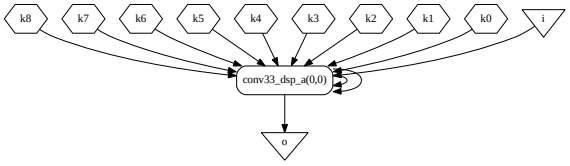
Au sein d'un FPGA, les blocs arithmétiques DSP sont des blocs dédiés aux calculs des multiplications. On peut alors se demander ce que l'utilisation de ces blocs pourrait avoir comme impact sur les performances du calcul d'une convolution. L'optimisation faite par le synthétiseur qui consiste à transformer la multiplication en une opération câblée peut se contourner de deux manières. La première consiste à modifier certains paramètres de l'outil manuellement. La seconde force l'utilisation de DSP en spécifiant les coefficients de la convolution non plus par des valeurs statiques mais via des ports d'entrées/sorties.

Le mécanisme de ports est apparu dans la version 2.5 du langage, avec pour objectif la communication asynchrone des acteurs avec l'environnement extérieur. Un port peut fonctionner en entrée ou en sortie dans un réseau. S'il est utilisé en entrée, cela permet le paramétrage dynamique de certaines valeurs de l'application. Le comportement du port est alors équivalent à une FIFO d'une seule place jamais vide. A chaque lecture, l'acteur obtient la valeur présente dans la FIFO mais sans consommer la donnée. La valeur de la FIFO peut être mise à jour depuis l'extérieur de manière asynchrone. En VHDL les ports sont implantés par des registres, plus la logique de contrôle pour être conforme aux interfaces des FIFOs de CAPH. Si le port est utilisée en entrée, ce registre peut être écrit depuis l'extérieur et être lu depuis n'importe quel acteur du réseau.

Dans notre cas, spécifier les coefficients du masque de convolution sous la forme de valeurs fournies via des ports d'entrées/sorties conduit au réseau décrit sur la Fig. 6.10.



(a) Schéma de connexions



(b) Réseau de l'acteur avec ports

```
actor conv233_dsp_a (n:int, v:int<s,m>)
  in (a:int<s,m> dc,
      k0:int<s,m>, k1:int<s,m>, k2:int<s,m>, -- noyaux via ports
      k3:int<s,m>, k4:int<s,m>, k5:int<s,m>,
      k6:int<s,m>, k7:int<s,m>, k8:int<s,m>,
      z:int<s,m>, zz:int<s,m>)
  out (c:int<s,m> dc, oz:int<s,m>, ozz:int<s,m>)

var s : {S0,S1,S2,S3,S4,S5,S6,S7,S8} = S0
var x1 : int<s,m>
var x2 : int<s,m>
var x4 : int<s,m>
var x5 : int<s,m>
var x7 : int<s,m>
var x8 : int<s,m>
rules
| (s:S0, a:'<)                -> (s:S1, c:'<)
| (s:S1, a:'>)                -> (s:S0, c:'>)
| (s:S1, a:'<)                -> (s:S2, c:'<)
| (s:S2, a:'p)                -> (s:S2, c:'v, oz:p)
| (s:S2, a:'>)                -> (s:S3, c:'>)
| (s:S3, a:'<)                -> (s:S4, c:'<)
| (s:S4, a:'x0, z:x3 )        -> (s:S4, c:'v, ozz:x3, oz:x0)
| (s:S4, a:'>)                -> (s:S5, c:'>)
| (s:S5, a:'>)                -> (s:S0, c:'>)
| (s:S5, a:'<)                -> (s:S6, c:'<)
| (s:S6, a:'x0, z:x3, zz:x6) -> (s:S7, c:'v, x7:x6, x4:x3, x1:x0, ozz:x3, oz:x0)
| (s:S7, a:'x0, z:x3, zz:x6) -> (s:S8, c:'v, x8:x7, x5:x4, x2:x1, x7:x6, x4:x3, x1:x0, ozz:x3, oz:x0)

| (s:S8, a:'x0, z:x3, zz:x6, k0:k0, k1:k1, k2:k2, k3:k3, k4:k4, k5:k5, k6:k6, k7:k7, k8:k8) ->
                                (s:S8, c:'((k0*x0+k1*x1+k2*x2+k3*x3+k4*x4+k5*x5+k6*x6+k7*x7+k8*x8)>>n),
                                 x8:x7, x5:x4, x2:x1, x7:x6, x4:x3, x1:x0, ozz:x3, oz:x0)

| (s:S8, a:'>)                -> (s:S5, c:'>);

-- Fonction d'ordre supérieure
net conv233_dsp norm pad i k0 k1 k2 k3 k4 k5 k6 k7 k8 = let rec (o,z,zz) = conv233_dsp_a (norm,pad) (i,k0,k1,
    k2,k3,k4,k5,k6,k7,k8,z,zz) in o;
```

(c) Listing CAPH

Fig. 6.10: Réseau CAPH pour la formulation mono acteur avec ports

Les résultats d'implémentation de cette dernière formulation sont donnés dans la table 6.5. Comme annoncé, cette formulation permet bien l'exploitation des blocs arithmétiques DSP. L'empreinte logique est constante mais le gain obtenu par le déport du calcul arithmétique de la convolution sur les blocs DSP est compensé par la logique nécessaire à la gestion des neufs ports (utilisant des registres). Par ailleurs, la fréquence maximale de fonctionnement a chuté d'environ 15% (moyenne 74.5MHz), ce qui n'est pas négligeable. Cette chute de fréquence s'explique par le temps de transmission des données pour aller jusqu'aux blocs DSP.

Taille d'image	ALM (%)	Mémoire requise (bits)	Blocs M10K	Mémoire allouée (bits)	DSP	Freq (MHz)
64 × 64	272 (0.6)	2144	2	20480	9	71.8
128 × 128	274 (0.6)	4192	2	20480	9	71.3
256 × 256	287 (0.7)	8288	2	20480	9	72.8
512 × 512	292 (0.7)	16480	4	40960	9	78.6
840 × 640	298 (0.7)	26976	4	40960	9	75.2
1280 × 960	294 (0.7)	41056	8	81920	9	78.5
1920 × 1080	294 (0.7)	61536	8	81920	9	71.6
4096 × 2160	307 (0.7)	131168	32	327680	9	77.9

Tab. 6.5: Résultats d'implémentation de la formulation mono-acteur avec ports

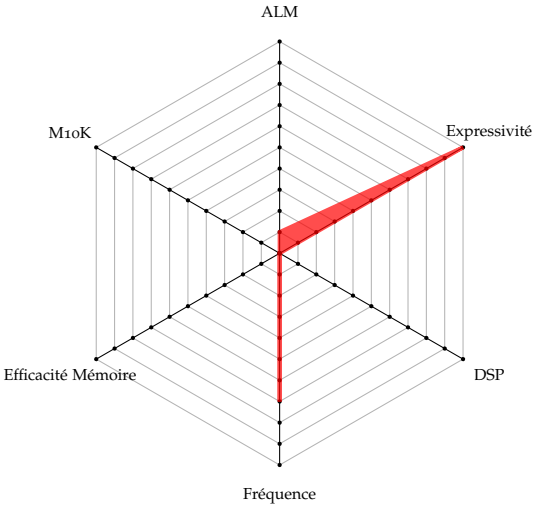
Pour finir, notons que, bien que les résultats d'implémentation obtenus n'apportent pas de gain significatif en termes de ressources logiques ou de fréquence maximale de fonctionnement, cette dernière formulation présente par ailleurs un autre intérêt : celui d'autoriser la modification *dynamique* des noyaux de convolutions (c'est-à-dire le changement du masque de pendant l'exécution du programme sur le FPGA.

6.1.6 Bilan comparatif

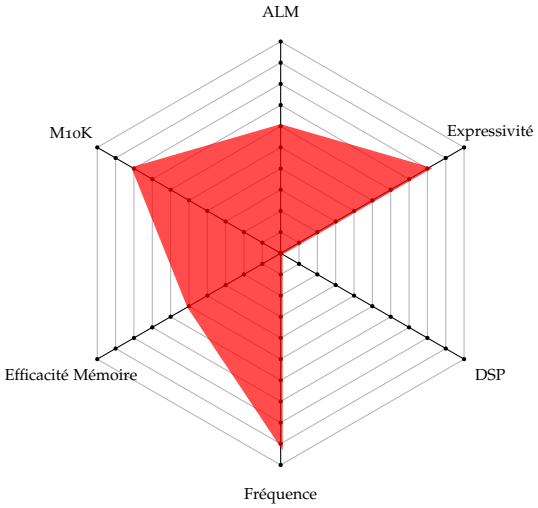
Deux remarques générales sur les différentes formulation peuvent être faites :

- La manière de stocker une ligne de pixels au sein d'un acteur a un impact important les ressources finales. Dans le cas d'une mémorisation de ligne en interne de l'acteur (première formulation), chaque pixel va utiliser des registres du FPGA. Les ressources logiques nécessaires deviennent alors dépendantes de la dimension d'image. Une méthode, basée sur l'utilisation de canaux rebouclés à l'extérieur de l'acteur permet l'exploitation des blocs mémoires du FPGA. Néanmoins, l'allocation de mémoire SRAM a un inconvénient. Chaque bloc mémoire n'ayant que deux ports, lorsque ces ports sont affectés à un processus, les ressources mémoires ne peuvent plus être partagées, même si l'espace mémoire allouée par le processus n'exploite que partiellement cette mémoire.
- La formulation qui conduit aux meilleurs résultats est la version **mono-acteur sans ports**. Le nombre de d'éléments logiques est faible, les mémoires SRAM sont bien exploitées et la fréquence finale élevée (autour de 90 *MHz*). Cette forme n'est cependant la plus efficace que si l'on ne considère que le calcul d'une seule convolution. Dans le cadre d'un ensemble de convolutions effectuées en parallèle sur le même flux d'entrée, chaque acteur mémorisera les mêmes deux lignes de voisinage. Il serait dans ce cas préférable de "factoriser" la mémorisation des lignes en revenant à un seul acteur `msfl` suivi d'une collection d'acteurs `maddn`.

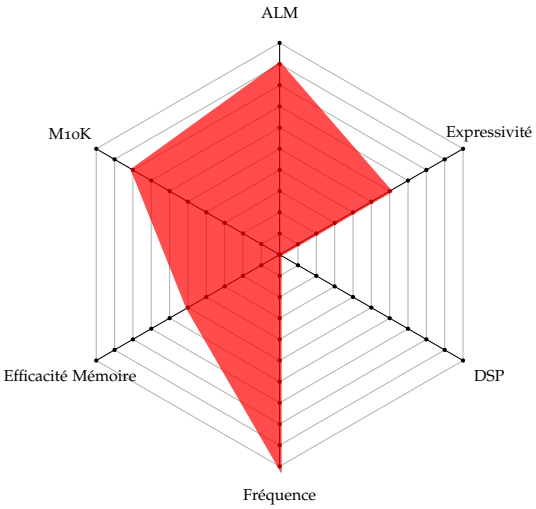
Dans un second temps, afin de juger de l'impact des paramètres mis en exergue par les différentes formulations, nous avons choisi de quantifier ces paramètres sous la forme de diagrammes de type "radar chart". Les résultats sont résumés sur la Fig. 6.11. Six paramètres sont quantifiés : *ALM*, *M10K*, *Efficacité Mémoire*, *Fréquence maximale*, *nombre de blocs DSP* et *Expressivité*. Les cinq premiers sont technologiques. Le dernier, plus subjectif, correspond au nombre de lignes de codes nécessaires, mais aussi à l'effort de formulation à effectuer (il est plus naturel pour un programmeur d'indexer un tableau que de gérer des séquences de lectures/écritures dans une FIFO rebouclée par exemple).



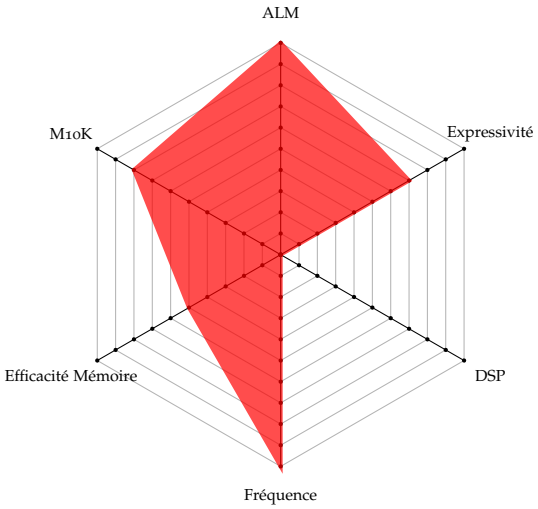
(a) D1LI/D1P



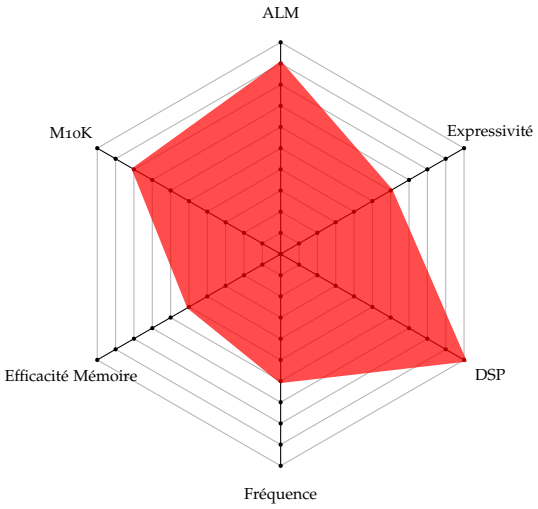
(b) D1LR/D1P



(c) MSFL/ADD



(d) Mono Act



(e) Mono-acteur avec ports

Fig. 6.11: Comparaison des résultats d'implémentation des différentes formulations de convolution 3×3 pour une image de dimensions 1280×960. Ce graphique montre les résultats de manières qualitatives. Les résultats précis sont disponibles dans les tables 6.1, 6.2, 6.3, 6.4, 6.5.

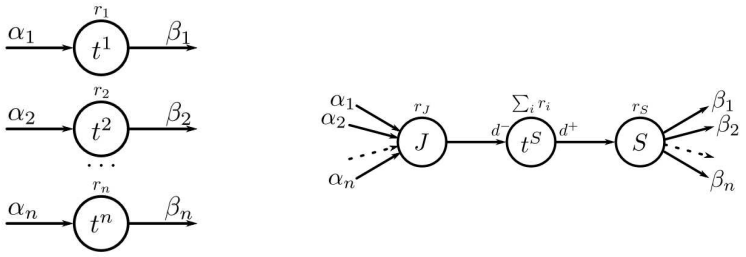
Voici les conclusions que l'on peut tirer de ces diagrammes :

- Le surcoût dû à l'utilisation des FIFOs est loin d'être négligeable. Il est donc préférable d'écrire des acteurs avec une grosse granularité plutôt qu'une collection de petits acteurs élémentaires si on se place du point de vue des ressources. Cependant cette approche diminue d'une part la lisibilité et la compréhension du comportement des acteurs, et d'autre part la réutilisation des acteurs (il est plus facile de réutiliser des acteurs "simples" généraux que des acteurs complexes spécialisés). Par exemple, l'acteur `msfl33` et les acteurs `d1lr/d1p/maddn` offrent la même fonctionnalité mais l'acteur `msfl33` n'est défini que pour un voisinage 3×3 alors que les acteurs `d1lr` et `d1p` sont exploitables par toute fonction formant un voisinage $N \times N$.
- Les variables locales de type tableau au sein des acteurs sont à utiliser avec précaution. En effet, chaque bit est alors implanté dans un registre. Si les accès à un tableau peuvent être décrits sous la forme de lecture/écriture séquentielles dans une FIFO, alors il est préférable d'utiliser une formulation exploitant des FIFOs rebouclées car elle conduira à une meilleur allocation des ressources finales.
- Le mécanisme de ports asynchrones permet de forcer l'instanciation des modules arithmétiques DSPs et des noyaux de convolutions dynamiques. On a toutefois montré qu'utiliser ce mécanisme afin de limiter la consommation de ressources matérielles n'était pas forcément probant. En effet, l'usage des blocs DSP, s'il permet de limiter d'un côté le nombre de blocs logiques affectés aux multiplications se paie d'un autre côté par l'utilisation de logique et de registres supplémentaires pour contrôler les DSP. Par ailleurs, il a été observé une baisse sensible de la fréquence maximale de fonctionnement des applications.

6.2 Sérialisation/désérialisation des jetons

Cette section s'intéresse au problème général du compromis surface/fréquence en synthèse numérique, à travers la question de la *sérialisation* des données. On entend par sérialisation ici la technique visant à réduire le nombre de ressources logiques requises par un traitement et fondée sur le multiplexage de ces ressources. Dans le cadre plus spécifique des modèles flot de données, Carpov dans [CCS13] a notamment abordé cette question par le biais de celle de la fusion d'acteurs au sein du modèle cyclo-static [BELP96]. L'approche est illustrée sur la Fig. 6.12.

Soit $S = \{t^1, ..., t^n\}$ un ensemble d'acteurs réalisant la même opération. Les acteurs sont supposés ici avoir un seul port d'entrée (α_i) et un seul de sortie (β_i). L'ensemble des acteurs S peut être réduit à un seul acteur t^S, exécutant la même opération et présentant la même interface que chaque acteur t^i. Les données d'entrées (α_i) sont multiplexées par un acteur J et les sorties (β_i) sont démultiplexées par l'acteur S.



(a) Ensemble d'acteurs équivalent

(b) Acteurs fusionnés

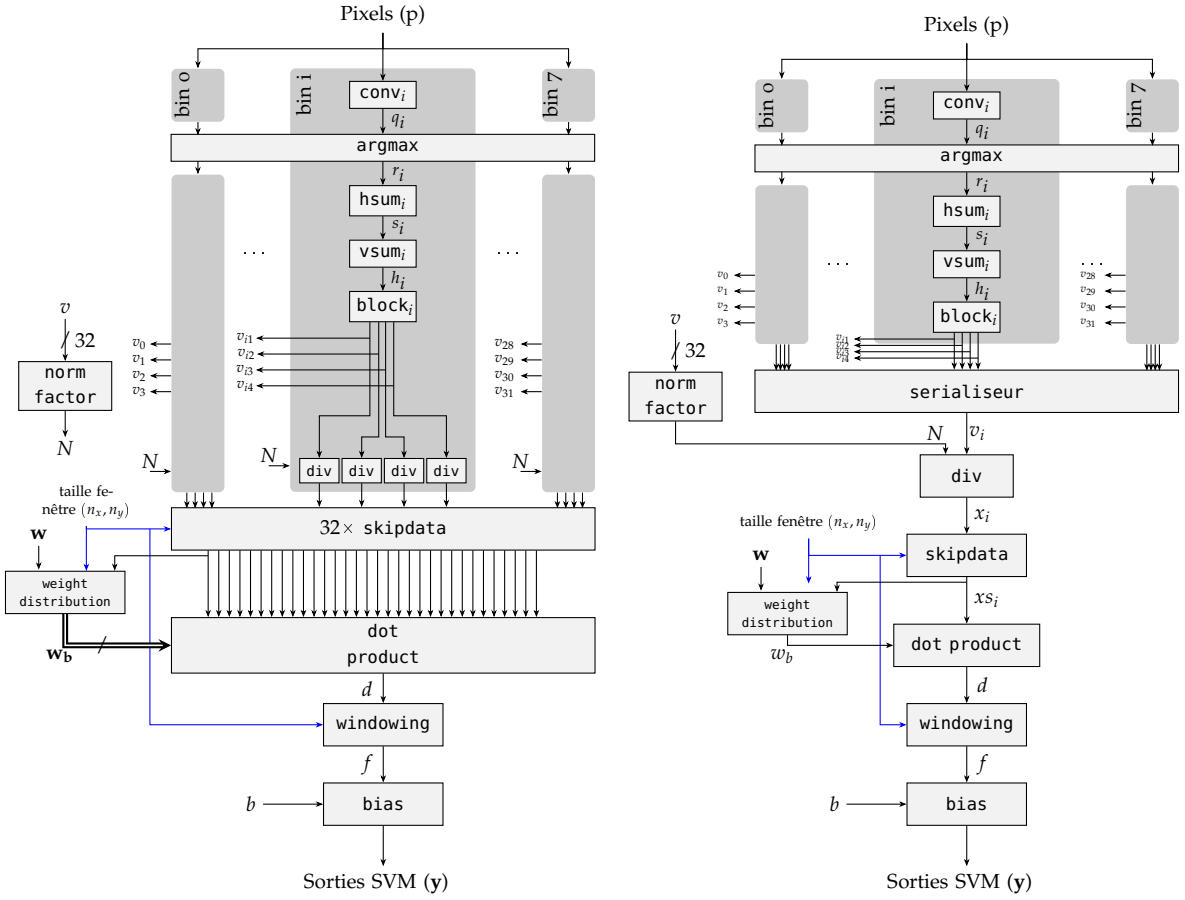
Fig. 6.12: Fusion d'acteurs, source [CCS13]

La technique de sérialisation illustrée sur la Fig. 6.12 n'est applicable que si les données peuvent être temporellement multiplexées. Plus précisément, si on suppose t_{α_i} le temps moyen entre deux arrivées de jetons sur la canal α_i et t_j le temps entre deux activations de l'acteur de sérialisation J, on doit avoir :

$$\sum_i^n t_{\alpha_i} \leqslant n \times t_J \tag{6.6}$$

où n correspond au nombre de canaux à sérialiser. Dans le cas des programmes CAPH, une contrainte supplémentaire liée au cadencement commun de tous les acteurs du réseau fait que $t_{\alpha_i} = t_J = t_{pix}$ où t_{pix} est le temps entre l'arrivée de deux pixels successifs. Il est alors possible de sérialiser les données à la seule condition que le nombre de données ait été diminué d'un facteur n en amont par des acteurs dont le ratio production/consommation est inférieur à 1.

Le comportement dynamique des acteurs de CAPH - qui rend la détermination statique du rapport production/consommation impossible dans le cas général - fait que la technique de sérialisation exposée ci-dessus est difficile à appliquer de manière automatique. Toutefois, il existe de nombreuses applications où elle peut être appliquée de manière *ad hoc* via une reformulation adéquate. C'est le cas de l'application de détection de piétons décrite au chapitre 4 pour laquelle le descripteur HOG peut être sérialisé avant sa normalisation. Dans cette application, la sérialisation est rendue possible par la présence des acteurs de calcul d'histogramme qui réduisent la quantité de données d'un facteur 64, alors que le descripteur contient 32 composantes. Les deux graphes flot de données (avant et après sérialisation) sont illustrés sur la Fig. 6.13.



(a) Graphe flot de données initial de l'application HOG-SVM

(b) Graphe flot de données après sérialisation

Fig. 6.13: Graphes flot de données de l'application HOG-SVM et après sérialisation du descripteur

L'intégration d'un acteur de sérialisation au sein de la formulation du graphe de la Fig. 6.13b a un impact sur la formulation d'autres acteurs tel que `weight_distribution` ou bien `dot_product`. L'acteur `weight_distribution` produit maintenant un seul canal de sortie sur lequel les 32 composantes sont envoyées en série. L'acteur `dot_product` effectue le produit scalaire du descripteur sérialisé. Nous avons transcrit en CAPH ces modifications et implanté le programme ainsi modifié sur la plateforme DreamCam. Les résultats sont décrits dans la table. 6.6

Tab. 6.6: Impact de la sérialisation sur les ressources matérielles de l'application HOG-SVM

	Logic Elements	LUT	Reg	Mémoire (*bits*)	M9K	DSP	Fmax
Graphe initial	37163	32940	18315	240096	62	32	21.2
Après sérialisation	21117	19171	11303	240096	62	1	21.9

L'utilisation de la sérialisation sur l'application HOG-SVM a permis de réduire l'empreinte logique de 56%, ce qui n'est pas négligeable. Le nombre de blocs arithmétiques nécessaire est lui, fort logiquement, divisé par 32. L'impact sur la fréquence maximale n'est toutefois pas significatif sur cet exemple à cause de la présence des diviseurs, qui imposent d'emblée une fréquence relativement basse.

Un paramètre qui peut avoir un impact significatif sur les performances de l'acteur de sérialisation est le nombre d'entrées à sérialiser. La Fig. 6.14 montre l'évolution de la fréquence de l'acteur en fonction du nombre d'entrées du sérialiseur pour les deux architectures de FPGA utilisées dans cette thèse (Cyclone III et V d'Altera). Les conditions d'expérimentations

choisies sont plutôt favorables à une fréquence élevée car seul l'acteur de sérialisation est implanté, ce qui signifie qu'une faible partie des ressources du FPGA est allouée. Dans le cas d'une application plus complexe, le taux d'occupation du FPGA par les autres composants rajoutera des contraintes supplémentaires qui auront pour effet d'impacter plus négativement la fréquence maximale.

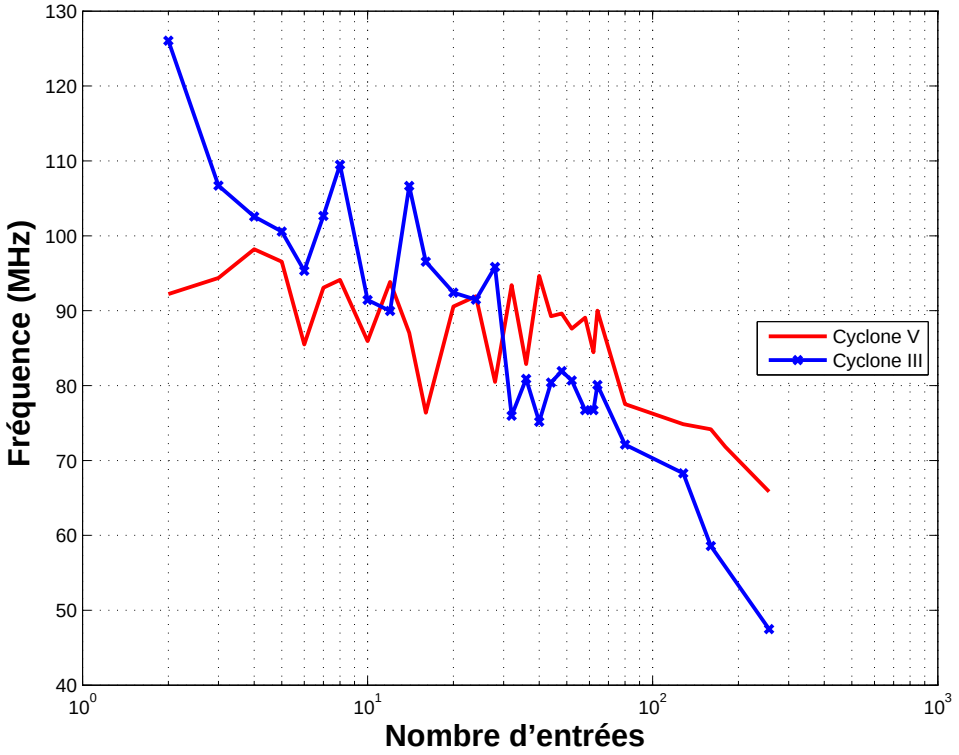


Fig. 6.14: Evolution de la fréquence maximale de l'acteur de sérialisation en fonction du nombre d'entrées

Les résultats de la Fig 6.14 montrent que la fréquence maximale de fonctionnement de l'acteur de sérialisation chute avec le nombre d'entrées. Sur la courbe du cyclone III, la fréquence maximale pour deux entrées est de 126 *Mhz*, puis de 75 *MHz* pour 32 entrées (dimension du descripteur HOG sérialisé) et continue de diminuer au fur et à mesure que le nombre d'entrées augmente. Ces résultats s'expliquent par le modèle de machine d'états utilisé lors de la génération des acteurs (voir manuel de référence de CAPH [Ser15] et l'annexe E pour un exemple). En effet, plus le nombre d'entrées/sorties augmente, plus l'état central généré comporte de tests à effectuer sur les règles d'activation. La Fig 6.14 montre également que les dernières architectures Cyclone V sont moins sensibles à ce problème. Les ALM ont un mode de fonctionnement LUT étendue (présenté dans la section 2.1) permettant d'optimiser les ressources justement pour ce cas de figure.

Les expérimentations menées dans cette section montrent que la sérialisation des jetons dans un graphe flot de données développé en CAPH est une technique qui peut permettre de diminuer les ressources matérielles requises sur le FPGA ciblé. Il faut toutefois vérifier la capacité de l'acteur de sérialisation à multiplexer temporellement les jetons, et que la perte en fréquence ne soit pas inacceptable vis à vis du gain en ressources logiques.

6.3 Conclusions et perspectives

La possibilité d'explorer rapidement l'espace de conception est un intérêt majeur d'un outil de synthèse haut-niveau comme CAPH. Il devient alors aisé de tester un ensemble de solutions pour le déploiement d'une application sur une cible matérielle afin de choisir la plus performante. L'exploration proposée dans ce chapitre sur un exemple de convolution a montré qu'il existe souvent plusieurs solutions. Chaque solution offre un compromis différent entre expressivité du code et performances matérielles.

Toutefois, comme avec tout outil, certaines règles lors de la description sont souhaitables afin d'optimiser au mieux les performances d'une application. Notamment, la mémorisation en interne de données de type tableau en CAPH peut conduire à un usage significatif des ressources de type registre. Il a été montré qu'une formulation utilisant des FIFOs rebouclées permettait au contraire l'exploitation de blocs mémoire SRAM dédiés. Toutefois cette approche complexifie la compréhension de la description du comportement de l'acteur et n'est possible que si les accès au tableau sont séquentiels. Une piste d'amélioration serait de modifier la transcription VHDL des tableaux en fonction de la dimension de ceux-ci. De manière similaire à l'approche utilisée pour les FIFOs, un composant décrivant le fonctionnement d'une mémoire pourrait être directement implanté au sein de l'acteur lors de l'utilisation d'un tableau. Dans ce cas, pour chaque règle d'activation utilisant le tableau, le pointeur d'adresse de lecture de la mémoire serait associée à l'expression utilisée pour l'accès d'une donnée dans la partie gauche des règles d'activation. Le pointeur d'écriture serait lui associé à l'expression utilisée dans la partie droite des règles. Les mémoires SRAM étant double ports, cette amélioration n'est toutefois possible que si le nombre d'accès est inférieur ou égal à deux écritures et deux lectures par règle. Une version simplifiée de cette proposition d'amélioration a d'ailleurs été implantée dans la dernière version de l'outil CAPH (2.7.0, dec 2015), pour les acteurs de retard ligne (`d1li`).

Avec les premières versions du compilateur CAPH, le modèle de machines d'états utilisé lors de la génération de la représentation VHDL intercalait des états intermédiaires, pendant lesquels les commandes des FIFOs étaient redescendues à 0. Cette contrainte a un impact fort sur les performances du code généré. En effet, elle impose d'avoir une fréquence d'activation des acteurs deux fois supérieure à celle des pixels en entrée du réseau. Les FIFOs, quelque soit le modèle utilisé, peuvent supporter des lectures et écritures continues. De cette constatation a découlé une étude permettant d'améliorer les performances du code généré. La dernière version du compilateur (2.7.0), disponible en décembre 2015, s'appuie désormais sur un modèle d'acteur sans états de repos et un contrôle optimisé des FIFOs afin que les deux fréquences susdites puissent être identiques. La suppression des états de repos fait par ailleurs que la machine d'états ne comporte plus qu'un seul état, ce qui simplifie le code RTL généré. En termes de performances, le nombre d'images par seconde maximal que peuvent traiter les applications précédemment développées est donc multiplié par deux soit, 20 images/sec pour l'application de détection de piétons et 45 images/sec pour le réseau CNN, avec la simple utilisation de la nouvelle version du compilateur.

Les expérimentations menées sur la possibilité de sérialiser les jetons dans un réseau CAPH ont montré un impact positif sur les ressources matérielles. Toutefois, le modèle flot de données sur lequel repose CAPH rend difficile l'automatisation de cette approche. La première limitation est la cadence unique de tous les acteurs. Avoir la possibilité de modifier les fréquences de fonctionnement de certaines parties du réseau permettrait d'ouvrir plus d'opportunités à l'utilisation de cette approche. La seconde limitation vient du caractère *dynamique* du modèle flot de données sur lequel repose CAPH, qui interdit à priori et dans le cas général une prédiction statique (à la compilation) du scénario d'activation des règles au sein des acteurs et des taux de remplissage des FIFO entre les acteurs. Pour certaines applications, toutefois, ce caractère dynamique n'est pas exploité et on peut se ramener à un modèle flot de données statique, pour lequel les ratios production/consommation et les taux de remplissage des FIFOs sont effectivement calculables à la compilation. Dans ce cas, les techniques de

transformation/optimisation comme la sérialisation prennent un intérêt certain.

Par ailleurs, le mécanisme de ports d'entrées/sorties asynchrones a plusieurs avantages. Il permet notamment la modification dynamique de certaines valeurs du programme lors de son exécution sur le FPGA. Ce mécanisme a également un intérêt dans la définition du modèle flot de données de CAPH, qui en plus d'être dynamique devient **paramétré** (tel que le PIMM [DPN+13]). En effet, le taux de consommation et de production des acteurs peuvent varier en fonction de la valeur d'un port.

Enfin, la dernière perspective d'évolution abordée ici concerne la question de la fusion automatique d'acteurs flot de données. Ce sujet a été déjà abordé dans de précédents travaux [Jan11, TLRW14, BEL+15]. Le modèle de machine d'états proposé par Janneck [Jan11] est un bon point de départ pour l'étude de la fusion d'acteur CAPH. L'exploration effectuée sur les différentes formes de réseau pour la convolution est en fait une démarche manuelle de ce que pourrait fournir automatiquement l'analyse du regroupement des acteurs. A partir du modèle de CAPH et d'heuristiques, il pourrait être possible de proposer un ensemble de méthodes fournissant des résultats de ressources différents, simplifiant le travail d'exploration mené par le développeur.

7

Conclusion et Perspectives

7.1 Conclusions

Cette thèse a montré que l'utilisation du modèle flot de données permet de répondre aux problématiques de synthèse haut-niveau pour l'implémentation d'algorithmes de traitement d'images sur FPGA. Il est en particulier montré que ce modèle se prête à la description et à l'implémentation d'applications allant au delà du simple traitement d'images bas niveau. Les deux applications traitées ici, bien que se rattachant toutes deux au même domaine de l'apprentissage supervisé, sont en effet suffisamment différentes à la fois dans leur structure et dans la reformulation pour valider de manière générale la méthode.

La première application, constituée des algorithmes HOG et SVM appliqués à la détection de piétons a requis un effort de reformulation important. Une part significative de cet effort a consisté à expliciter le parallélisme de données et de flux que les formulations séquentielles laissent implicites. De cet effort a découlé la proposition de nouvelles formulations respectant les contraintes du modèle flot de données tout en maximisant le parallélisme exploitable. Toutefois, certaines limites concernant les performances du code généré ont été atteintes. En ce qui concerne la fiabilité de classification du système, il a été observé que la dégradation majeure de la qualité de détection provenait du choix du type de normalisation lors de la détermination du descripteur HOG et non de la représentation en virgule fixe des données. Afin de compenser cette dégradation, qui conduit pratiquement à une augmentation du nombre de faux positifs, un système de filtrage a été proposé. Ce système permet de réduire le taux de faux positifs tout en conservant le taux de bonnes détections, permettant ainsi d'atteindre le niveau de fiabilité fourni par les implémentations logicielles de référence.

La seconde application a abordé les réseaux de neurones convolutionnels (CNNs). Le parallélisme de tâches et la faible dépendance de données au sein de ces réseaux facilitent la reformulation suivant le modèle flot de données. Ceci dit, les premiers résultats d'implémentation montrent que les ressources nécessaires pour une implémentation sur FPGA peuvent être importantes. C'est là qu'une méthodologie et des outils autorisant une exploration rapide du compromis performances de classification/ressources requises prennent tout leur sens.

Les travaux décrits ici ont par ailleurs permis d'extraire des informations plus générales sur les problématiques d'implémentation du modèle flot de données sur FPGA, notamment lors de l'utilisation de l'outil de HLS CAPH. Parmi les règles de codage mises en évidence, on peut citer notamment l'utilisation de canaux rebouclés pour la mémorisation des lignes dans le cas des acteurs opérant sur un voisinage et la sérialisation des données.

7.2 Perspectives

Les perspectives de recherche et d'évolution des travaux se situent à deux niveaux : au niveau applicatif d'une part et au niveau des outils de HLS d'autre part.

Au niveau applicatif, la première perspective portant sur l'application de détection de piétons consisterait à revoir la formulation flot de données des unités SVM afin de supporter le recouvrement des fenêtres de détection tout en rendant possible l'exécution du code généré sur notre plateforme de vision (DreamCam). Ensuite, afin d'améliorer la fiabilité de détection du système, la gestion de plusieurs échelles est souhaitable. Le passage à un système multi-échelles se fait assez simplement en pratique par la mise en parallèle du système déjà formulé, avec l'ajout d'un pyramide gaussienne. Ces évolutions nécessiteront également une modification du système de filtrage, qui devra exploiter l'ensemble des résultats sur chacune des échelles afin d'améliorer les performances de classification. Enfin, l'ajout d'un descripteur de texture de type LBP dans la détermination de l'espace de description est souhaitable. Les récentes évaluations [YPW+15, BOH+14] ont en effet montré que ce descripteur, couplé avec un descripteur HOG, améliore les performances de détection.

Concernant les réseaux de neurones convolutionnels, deux axes de recherche peuvent être approfondis. Le premier axe concerne les aspects algorithmiques avec l'intégration des problématiques liées à la représentation des données comme des paramètres à optimiser lors de la phase d'apprentissage, notamment en exploitant les techniques dites de l'*approximate computing*. Le second axe concerne les aspects méthodologiques, avec la création d'un outil spécifique pour l'implémentation automatique de réseaux CNN vers des implémentations FPGA. Un tel outil pourrait soit s'appuyer explicitement sur le langage CAPH comme langage intermédiaire, soit utiliser une représentation intermédiaire dérivée de celle sur laquelle CAPH repose mais exploitant certaines spécificités des CNNs afin d'optimiser l'implantation. Il serait intéressant d'intégrer à cet outil une exploration automatique des configurations possibles de réseaux en utilisant des heuristiques afin de maximiser à la fois la classification et les ressources, afin d'obtenir le meilleur compromis pour un problème donné.

Au niveau des outils de HLS, et plus particulièrement du langage CAPH, le code généré avec les récentes optimisations permet d'atteindre des performances tout à fait satisfaisantes tout en conservant une bonne portabilité. Le futur de l'outil se situe donc sur les transformations automatiques qu'il serait possible de faire dans certains cas et qui permettraient d'améliorer encore ces performances, tant au niveau de la consommation des ressources que de la fréquence maximale de fonctionnement. En effet, le comportement dynamique de l'outil, assez peu exploité en pratique, se paie cher en terme de ressources sur l'ensemble des applications. Il empêche notamment la mise en œuvre automatique de transformations à la compilation telles que la sérialisation des données ou la détermination exacte de la profondeur des canaux de communication.

Au niveau du langage lui même, deux axes de développement pourraient être la possibilité de paramétrer le comportement des acteurs avec des fonctions et le support d'un type *vecteur* représentant un ensemble de canaux. La première fonctionnalité simplifierait la description des acteurs opérant de la même manière sur la même structure de flots, en factorisant une partie des règles d'activations (par exemple, l'acteur de sous-échantillonnage dans les réseaux CNN). La seconde fonctionnalité permettrait de définir des acteurs opérant sur un ensemble de canaux, sans avoir à figer précisément le nombre de canaux dans l'interface de l'acteur.

En terme de performances du code généré par l'outil CAPH, la problématique posée par l'utilisation d'un composant de division a mis en avant la question de la latence d'une instruction d'un acteur. Le modèle actuel de CAPH considère que chaque règle d'activation s'exécute en un seul cycle d'horloge sur le FPGA. Dans certains cas, comme celui de l'opération de division, cette contrainte implique une logique combinatoire trop importante et donc une chute de la fréquence maximale de fonctionnement. Il se pose alors la question du support d'instructions pouvant prendre plusieurs cycles d'horloge lors de leur exécution. L'enjeu

est d'insérer la gestion d'un *pipeline* automatiquement au sein des acteurs tout en continuant de masquer la notion de cycle d'horloge au niveau du langage de description.

La méthode de programmation proposée dans cette thèse permet d'abstraire certaines problématiques liées à la programmation des circuits reconfigurables pour le traitement vidéo. Dans le contexte applicatif des réseaux de capteurs, cette méthodologie pourrait être une composante d'une méthodologie plus générale, offrant l'abstraction, voire la virtualisation, de plateformes hétérogènes dans l'Internet des objets.

Bibliographie

[AC86] Arvind and David E. Culler. Annual review of computer science vol. 1, 1986. chapter Dataflow Architectures, pages 225–253. Annual Reviews Inc., Palo Alto, CA, USA, 1986.

[AC10] Sylvain Arlot and Alain Celisse. A survey of cross-validation procedures for model selection. *Statist. Surv.*, 4 :40–79, 2010.

[AHMJP12] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, and Gerald Penn. Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 4277–4280. IEEE, 2012.

[AHP06] Timo Ahonen, Abdenour Hadid, and Matti Pietikainen. Face description with local binary patterns : Application to face recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 28(12) :2037–2041, December 2006.

[alta] Altera White Paper : Guidance for Accurately Benchmarking FPGAs. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01040.pdf. Consultation : 2015-11-18.

[Altb] Altera. Cyclone III Device Handbook - Chapter : Logic Elements and Logic Array Blocks in the Cyclone III Device family. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyc3/cyclone3_handbook.pdf. Consultation : 2015-10-30.

[Altc] Altera. Cyclone V Device Handbook. https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyclone-v/cyclone5_handbook.pdf. Consultation : 2015-11-09.

[BA83] Peter J Burt and Edward H Adelson. The laplacian pyramid as a compact image code. *Communications, IEEE Transactions on*, 31(4) :532–540, 1983.

[Bac78] John Backus. Can programming be liberated from the von neumann style ? : a functional style and its algebra of programs. *Communications of the ACM*, 21(8) :613–641, 1978.

[BB01] Bishnupriya Bhattacharya and Shuvra S Bhattacharyya. Parameterized dataflow modeling for dsp systems. *Signal Processing, IEEE Transactions on*, 49(10) :2408–2421, 2001.

[BB14] Merwan Birem and François Berry. Dreamcam : A modular fpga-based smart camera architecture. *Journal of Systems Architecture*, 60(6) :519–527, 2014.

[BDT13] Shuvra S Bhattacharyya, Ed F Deprettere, and Bart D Theelen. Dynamic dataflow graphs. In *Handbook of Signal Processing Systems*, pages 905–944. Springer, 2013.

[BEJ+11] Shuvra S Bhattacharyya, Johan Eker, Jörn W Janneck, Christophe Lucarz, Marco Mattavelli, and Mickaël Raulet. Overview of the mpeg reconfigurable video coding framework. *Journal of Signal Processing Systems*, 63(2) :251–263, 2011.

[BEL+15] J. Boutellier, J. Ersfolk, J. Lilius, M. Mattavelli, G. Roquier, and O. Silven. Actor merging for dataflow process networks. *Signal Processing, IEEE Transactions on*, 63(10) :2496–2508, May 2015.

[BELP96] Greet Bilsen, Marc Engels, Rud Lauwereins, and Jean Peperstraete. Cycle-static dataflow. *Signal Processing, IEEE Transactions on*, 44(2) :397–408, 1996.

[BETVG08] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Comput. Vis. Image Underst.*, 110(3) :346–359, June 2008.

[Bir15] Merwan Birem. *Localisation et détection de fermeture de boucle basées saillance visuelle : algorithmes et architectures matérielles*. PhD thesis, 2015.

[BKDB10] S. Bauer, S. Kohler, K. Doll, and U. Brunsmann. Fpga-gpu architecture for kernel svm pedestrian detection. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*, pages 61–68, June 2010.

[BM12] Saket Bhardwaj and Ajay Mittal. A survey on various edge detector techniques. *Procedia Technology*, 4(0) :220 – 226, 2012.

[BMD13] Darío Baptista and Fernando Morgado-Dias. Low-resource hardware implementation of the hyperbolic tangent for artificial neural networks. *Neural Computing and Applications*, 23(3-4) :601–607, 2013.

[BOH+14] R. Benenson, M. Omran, J. Hosang, , and B. Schiele. Ten years of pedestrian detection, what have we learned ? In *ECCV, CVRSUAD workshop*, 2014.

[BP66] Leonard E Baum and Ted Petrie. Statistical inference for probabilistic functions of finite state markov chains. *The annals of mathematical statistics*, pages 1554–1563, 1966.

[Bra00] G. Bradski. Opencv library. *Dr. Dobb's Journal of Software Tools*, 2000.

[Bre01] Leo Breiman. Random forests. *Machine learning*, 45(1) :5–32, 2001.

[CCS13] Sergiu Carpov, Loïc Cudennec, and Renaud Sirdey. Throughput constrained parallelism reduction in cyclo-static dataflow applications. *Procedia Computer Science*, 18 :30 – 39, 2013. 2013 International Conferevnce on Computational Science.

[Chi96] David Maxwell Chickering. Learning bayesian networks is np-complete. In *Learning from data*, pages 121–130. Springer, 1996.

[CSH+10] Jie Chen, Shiguang Shan, Chu He, Guoying Zhao, Matti Pietikäinen, Xilin Chen, and Wen Gao. Wld : A robust local image descriptor. *IEEE Trans. Pattern Anal. Mach. Intell.*, 32(9) :1705–1720, 2010.

[CSJC10] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 247–257. ACM, 2010.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3) :273–297, 1995.

[CVBM02] Olivier Chapelle, Vladimir Vapnik, Olivier Bousquet, and Sayan Mukherjee. Choosing multiple parameters for support vector machines. *Mach. Learn.*, 46(1-3) :131–159, March 2002.

[dCFEB13] Ines del Campo, Raul Finker, Javier Echanobe, and Koldo Basterretxea. Controlled accuracy approximation of sigmoid function for efficient fpga-based implementation of artificial neurons. *Electronics Letters*, 49(25) :1598–1600, 2013.

[Den74] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 362–376, London, UK, UK, 1974. Springer-Verlag.

[Des14] Desnos Desnos, Karol. *Memory Study and Dataflow Representations for Rapid Prototyping of Signal Processing Applications on MPSoCs*. Theses, INSA de Rennes, September 2014.

[DEY+09] Andreas Dahlin, Johan Ersfolk, Guifu Yang, Haitham Habli, and Johan Lilius. The canals language and its compiler. In *SCOPES '09 : Proceedings of th 12th International Workshop on Software and Compilers for Embedded Systems*, 2009.

[DKP03] Kaibo Duan, S Sathiya Keerthi, and Aun Neow Poo. Evaluation of simple performance measures for tuning svm hyperparameters. *Neurocomputing*, 51 :41–59, 2003.

[Dos14] Niraj P Doshi. *Multi-dimensional local binary pattern texture descriptors and their application for medical image analysis*. PhD thesis, Loughborough University, 2014.

[DPN+13] Karol Desnos, Maxime Pelcat, Jean-François Nezan, S Bhattacharyya, Shuvra, and Slaheddine Aridhi. PiMM : Parameterized and Interfaced dataflow Meta-Model for MPSoCs runtime reconfiguration. In *13th International Conference on Embedded Computer Systems : Architecture, Modeling and Simulation (SAMOS XIII)*, pages 41 – 48, Samos, Greece, July 2013.

[DPR+15] O Rama Devi, EV Prasad, LSS Reddy, V Sree Lasya, and V Sai Siddartha. Robust rule based local binary pattern for face recognition. *International Journal of Advanced Research in Computer Science*, 6(3), 2015.

[DT05] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In Cordelia Schmid, Stefano Soatto, and Carlo Tomasi, editors, *International Conference on Computer Vision & Pattern Recognition*, volume 2, pages 886–893, June 2005.

[DWSP09] Piotr Dollár, Christian Wojek, Bernt Schiele, and Pietro Perona. Pedestrian detection : A benchmark. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 304–311. IEEE, 2009.

[DWSP12] Piotr Dollar, Christian Wojek, Bernt Schiele, and Pietro Perona. Pedestrian detection : An evaluation of the state of the art. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34(4) :743–761, 2012.

[EG09] Markus Enzweiler and Dariu M Gavrila. Monocular pedestrian detection : Survey and experiments. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(12) :2179–2195, 2009.

[EJ03] Johan Eker and Jorn Janneck. Cal language report. Technical report, Tech. Rep. ERL Technical Memo UCB/ERL, 2003.

[eri15] Erika enterprise website. http://erika.tuxfamily.org/drupal/, 2015.

[EVGW+10] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 88(2) :303–338, 2010.

[Far13] Clément Farabet. *Towards Real-Time Image Understanding with Convolutional Networks*. Theses, Université Paris-Est, December 2013.

[Faw06] T. Fawcett. An introduction to ROC analysis. *Pattern recognition letters*, 2006.

[FB97] Mark A Friedl and Carla E Brodley. Decision tree classification of land cover from remotely sensed data. *Remote sensing of environment*, 61(3) :399–409, 1997.

[FGMR10] Pedro F Felzenszwalb, Ross B Girshick, David McAllester, and Deva Ramanan. Object detection with discriminatively trained part-based models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(9) :1627–1645, 2010.

[FMC+11] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. Neuflow : A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 109–116. IEEE, 2011.

[FPHL09] Clément Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun. Cnp : An fpga-based processor for convolutional networks. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 32–37. IEEE, 2009.

[Fri96] Jerome H. Friedman. Another approach to polychotomous classification. Technical report, Department of Statistics, Stanford University, 1996.

[FS+96] Yoav Freund, Robert E Schapire, et al. Experiments with a new boosting algorithm. In *ICML*, volume 96, pages 148–156, 1996.

[Fuk80] Kunihiko Fukushima. Neocognitron : A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36 :193–202, 1980.

[Gab46] Dennis Gabor. Theory of communication. part 1 : The analysis of information. *Journal of the Institution of Electrical Engineers-Part III : Radio and Communication Engineering*, 93(26) :429–441, 1946.

[GAGN15] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *arXiv preprint arXiv :1502.02551*, 2015.

[GAL09] Wei Gao, Haizhou Ai, and Shihong Lao. Adaptive contour features in oriented granular space for human detection and segmentation. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 1786–1793. IEEE, 2009.

[GDDM14] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jagannath Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on*, pages 580–587. IEEE, 2014.

[GDGN03] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Spark : A high-level synthesis framework for applying parallelizing compiler transformations. In *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 461–466. IEEE, 2003.

[GR12] Varghese George and Jan M Rabaey. *Low-energy FPGAs—Architecture and Design*, volume 625. Springer Science & Business Media, 2012.

[GSAK00] Maya Gokhale, Jan Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented fpga computing in the streams-c high level language. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pages 49–56. IEEE, 2000.

[GT10] Mentor Graphics and H Tool. Catapult c, 2010.

[Gue07] Yann Guermeur. *Multi-Class SVMs, Theory and Applications*. Habilitation à diriger des recherches, Université Henri Poincaré - Nancy I, November 2007.

[HALL06] Chang Huang, Haizhou Ai, Yuan Li, and Shihong Lao. Learning sparse features in granular space for multi-view face detection. In *Automatic Face and Gesture Recognition, 2006. FGR 2006. 7th International Conference on*, pages 401–406, April 2006.

[han] Handle-C. https://www.mentor.com/products/fpga/handel-c/. Consultation : 2015-11-4.

[HKM+08] Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon, and Rodric Rabbah. Optimus : Efficient realization of streaming applications on fpgas. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pages 41–50, New York, NY, USA, 2008. ACM.

[HN89] R. Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural Networks, 1989. IJCNN., International Joint Conference on*, pages 593–605 vol.1, 1989.

[HO13] Jie Han and Michael Orshansky. Approximate computing : An emerging paradigm for energy-efficient design. In *Test Symposium (ETS), 2013 18th IEEE European*, pages 1–6. IEEE, 2013.

[Hoa78] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8) :666–677, 1978.

[Hop88] John J Hopfield. Artificial neural networks. *Circuits and Devices Magazine, IEEE*, 4(5) :3–10, 1988.

[HOT06] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural computation*, 18(7) :1527–1554, 2006.

[HSH+13] M. Hahnle, F. Saxen, M. Hisung, U. Brunsmann, and K. Doll. Fpga-based real-time pedestrian detection on high-resolution images. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2013 IEEE Conference on*, pages 629–635, June 2013.

[HT98] Trevor Hastie and Robert Tibshirani. Classification by pairwise coupling. In *Proceedings of the 1997 Conference on Advances in Neural Information Processing Systems 10*, NIPS '97, pages 507–513, Cambridge, MA, USA, 1998. MIT Press.

[HW68] David H Hubel and Torsten N Wiesel. Receptive fields and functional architecture of monkey striate cortex. *The Journal of physiology*, 195(1) :215–243, 1968.

[HWY+09] E. Hung, S.J.E. Wilton, Haile Yu, T.C.P. Chau, and P.H.W. Leong. A detailed delay path model for fpgas. In *Field-Programmable Technology, 2009. FPT 2009. International Conference on*, pages 96–103, Dec 2009.

[iee15] Eee 802.15 wpan task group 4. http://www.ieee802.org/15/pub/TG4.html, 2015.

[imp] Impulse c. http://www.impulsec.com/. Consultation : 2015-11-4.

[Jan11] Jorn W Janneck. A machine model for dataflow actors and its applications. In *Signals, Systems and Computers (ASILOMAR), 2011 Conference Record of the Forty Fifth Asilomar Conference on*, pages 756–760. IEEE, 2011.

[JKRL09] Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato, and Yann LeCun. What is the best multi-stage architecture for object recognition ? In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 2146–2153. IEEE, 2009.

[Joa99] Thorsten Joachims. Svmlight : Support vector machine. 19(4), 1999.

[Joa02] Thorsten Joachims. *Learning to Classify Text Using Support Vector Machines : Methods, Theory and Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe : Convolutional architecture for fast feature embedding. *arXiv preprint arXiv :1408.5093*, 2014.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[KLK02] M. Krips, T. Lammert, and A. Kummert. Fpga implementation of a neural network for a real-time hand tracking system. In *Electronic Design, Test and Applications, 2002. Proceedings. The First IEEE International Workshop on*, pages 313–317, 2002.

[KS04] Yan Ke and Rahul Sukthankar. Pca-sift : A more distinctive representation for local image descriptors. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, CVPR'04, pages 506–513, Washington, DC, USA, 2004. IEEE Computer Society.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[Lap05] Ivan Laptev. On space-time interest points. *Int. J. Comput. Vision*, 64(2-3) :107–123, September 2005.

[LBBH98] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[LBOM12] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks : Tricks of the trade*, pages 9–48. Springer, 2012.

[LCTHS88] Yann Le Cun, D Touresky, G Hinton, and T Sejnowski. A theoretical framework for back-propagation. In *The Connectionist Models Summer School*, volume 1, pages 21–28, 1988.

[len15] Example de modèle lenet-5. http://deeplearning.net/tutorial/lenet.html, 2015.

[LGRN09] Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 609–616. ACM, 2009.

[Lip87] Richard P Lippmann. An introduction to computing with neural nets. *ASSP Magazine, IEEE*, 4(2) :4–22, 1987.

[LLZ09] SHEN Lin-Lin and JI Zhen. Gabor wavelet selection and svm classification for object recognition. *Acta Automatica Sinica*, 35(4) :350–355, 2009.

[LM+87] Edward Lee, David G Messerschmitt, et al. Synchronous data flow. *Proceedings of the IEEE*, 75(9) :1235–1245, 1987.

[LM02] Rainer Lienhart and Jochen Maydt. An extended set of haar-like features for rapid object detection. In *IEEE ICIP 2002*, pages 900–903, 2002.

[LMS13] S. Lee, K. Min, and T. Suh. Accelerating histograms of oriented gradients descriptor extraction for pedestrian recognition. *Computers and Electrical Engineering*, pages 1043 – 1048, 2013.

[LMSR08] Ivan Laptev, Marcin Marszałek, Cordelia Schmid, and Benjamin Rozenfeld. Learning realistic human actions from movies. In *Conference on Computer Vision & Pattern Recognition*, jun 2008.

[LMW+08] Christophe Lucarz, Marco Mattavelli, Matthieu Wipliez, Ghislain Roquier, Mickaël Raulet, Jörn W Janneck, Ian D Miller, and David B Parlour. Dataflow/actor-oriented language for the design of complex signal processing systems. In *Conference on Design and Architectures for Signal and Image Processing (DASIP 2008)*, pages 1–8, 2008.

[Low04] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2) :91–110, November 2004.

[LP95] E.A. Lee and T.M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5) :773–801, 1995.

[LSCM12] Seonyoung Lee, Haengseon Son, Jong Chan Choi, and Kyoungwon Min. Hog feature extractor circuit for real-time human and vehicle detection. In *TENCON 2012 - 2012 IEEE Region 10 Conference*, pages 1–5, Nov 2012.

[LZ05] Yi Liu and Yuan F Zheng. One-against-all multi-class svm classification using reliability measures. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, volume 2, pages 849–854. IEEE, 2005.

[MBB+15a] Luca Maggiani, Cedric Bourrasset, François Berry, Matteo Petracca, Paolo Pagano, and Claudio Salvadori. Hog-dot : a parallel kernel-based gradient extraction for embedded image processing. In *IEEE Signal Processing Letters*, 2015.

[MBB+15b] Luca Maggiani, Cedric Bourrasset, François Berry, Jocelyn Sérot, Matteo Petracca, Paolo Pagano, and Claudio Salvadori. Parallel image gradient extraction core for FPGA-based smart cameras. In *Proceedings of ACM/IEEE International Conference on Distributed Smart Cameras*, 2015.

[MBQ+16] Luca Maggiani, Cedric Bourrasset, Jean-Charles Quinton, François Berry, and Jocelyn Serot. Bio-inspired heterogeneous architecture for real-time pedestrian detection applications. In *manuscipt accepted with minor revisions, JRTIP*, 2016.

[Mer09] J. Mercer. Functions of positive and negative type, and their connection with the theory of integral equations. *Philosophical Transactions of the Royal Society of London A : Mathematical, Physical and Engineering Sciences*, 209(441-458) :415–446, 1909.

[MGE+11] Tomasz Malisiewicz, Abhinav Gupta, Alexei Efros, et al. Ensemble of exemplar-svms for object detection and beyond. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 89–96. IEEE, 2011.

[mit] Mitrion-C. http://forum.mitrionics.com/uploads/Mitrion_Users_Guide.pdf. Consultation : 2015-11-4.

[MJ01] LR Medsker and LC Jain. Recurrent neural networks. *Design and Applications*, 2001.

[MPS+09] Julien Mairal, Jean Ponce, Guillermo Sapiro, Andrew Zisserman, and Francis R Bach. Supervised dictionary learning. In *Advances in neural information processing systems*, pages 1033–1040, 2009.

[MRG+86] James L McClelland, David E Rumelhart, PDP Research Group, et al. Parallel distributed processing. *Explorations in the microstructure of cognition*, 2, 1986.

[MS05] Krystian Mikolajczyk and Cordelia Schmid. A performance evaluation of local descriptors. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 27(10) :1615–1630, 2005.

[MS09] G. Martin and G. Smith. High-level synthesis : Past, present, and future. *Design Test of Computers, IEEE*, 26(4) :18–25, July 2009.

[MSP+14] L. Maggiani, C. Salvadori, M. Petracca, P. Pagano, and R. Saletti. Reconfigurable architecture for computing histograms in real-time tailored to fpga-based smart camera. In *Industrial Electronics (ISIE), 2014 IEEE 23rd International Symposium on*, pages 1042–1046, June 2014.

[MTT+12] K. Mizuno, Y. Terachi, K. Takagi, S. Izumi, H. Kawaguchi, and M. Yoshimoto. Architectural study of hog feature extraction processor for real-time object detection. In *Signal Processing Systems (SiPS), 2012 IEEE Workshop on*, pages 197–202, Oct 2012.

[NBD+03] Walid Najjar, Wim Böhm, Bruce Draper, Jeff Hammes, Robert Rinker, J Ross Beveridge, Monica Chawathe, Charles Ross, et al. High-level language abstraction for reconfigurable computing. *Computer*, 36(8) :63–69, 2003.

[NJW+02] Andrew Y Ng, Michael I Jordan, Yair Weiss, et al. On spectral clustering : Analysis and an algorithm. *Advances in neural information processing systems*, 2 :849–856, 2002.

[NLG99] Walid A Najjar, Edward A Lee, and Guang R Gao. Advances in the dataflow computational model. *Parallel Computing*, 25(13–14) :1907 – 1929, 1999.

[NLM+09] Ashkan Hosseinzadeh Namin, Karl Leboeuf, Roberto Muscedere, Huapeng Wu, and Majid Ahmadi. Efficient hardware implementation of the hyperbolic tangent sigmoid function. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pages 2117–2120. IEEE, 2009.

[NOF12] Ryusuke Nosaka, Yasuhiro Ohkawa, and Kazuhiro Fukui. Feature extraction based on co-occurrence of adjacent local binary patterns. In *Advances in Image and Video Technology*, pages 82–91. Springer, 2012.

[ope] OpenCL Specification. https://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf. Consultation : 2015-11-6.

[OPH96] Timo Ojala, Matti Pietikäinen, and David Harwood. A comparative study of texture measures with classification based on featured distributions. *Pattern Recognition*, 29(1) :51–59, January 1996.

[OPM02] T. Ojala, M. Pietikainen, and T. Maenpaa. Multiresolution gray-scale and rotation invariant texture classification with local binary patterns. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(7) :971–987, Jul 2002.

[ORK+15] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2, 2015.

[Pea14] Judea Pearl. *Probabilistic reasoning in intelligent systems : networks of plausible inference*. Morgan Kaufmann, 2014.

[Pol86] D. Pollard. Convergence of stochastic processes. (springer series in statistics). springer-verlag, new york - berlin - heidelberg - tokyo 1984, 216 pp., 36 illustr., dm 82. *Biometrical Journal*, 28(5) :644–644, 1986.

[POP98] Constantine P. Papageorgiou, Michael Oren, and Tomaso Poggio. A general framework for object detection. In *Proceedings of the Sixth International Conference on Computer Vision*, ICCV '98, pages 555–, Washington, DC, USA, 1998. IEEE Computer Society.

[Por05] Fatih Porikli. Integral histogram : A fast way to extract histograms in cartesian spaces. In *in Proc. IEEE Conf. on Computer Vision and Pattern Recognition*, pages 829–836, 2005.

[Poz05] Daniel S Poznanovic. Application development on the src computers, inc. systems. In *null*, page 78a. IEEE, 2005.

[QG11] Jean-Charles Quinton and Bernard Girau. Predictive neural fields for improved tracking and attentional properties. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 1629–1636. IEEE, 2011.

[Ras99] Carl Edward Rasmussen. The infinite gaussian mixture model. In *NIPS*, volume 12, pages 554–560, 1999.

[Ros58] Frank Rosenblatt. The perceptron : a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6) :386, 1958.

[SB09] Sundararajan Sriram and Shuvra S Bhattacharyya. *Embedded multiprocessors : Scheduling and synchronization*. CRC press, 2009.

[SBB14] Jocelyn Sérot, François Berry, and Cédric Bourrasset. High-level dataflow programming for real-time image processing on smart cameras. *Journal of Real-Time Image Processing*, pages 1–13, 2014.

[Ser15] Serot. The Caph Programming Language home page. http://caph.univ-bpclermont.fr, 2015.

[SG91] Yoan Shin and Joydeep Ghosh. The pi-sigma network : An efficient higher-order neural network for pattern classification and function approximation. In *Neural Networks, 1991., IJCNN-91-Seattle International Joint Conference on*, volume 1, pages 13–18. IEEE, 1991.

[SH09] Ruslan Salakhutdinov and Geoffrey E Hinton. Deep boltzmann machines. In *International Conference on Artificial Intelligence and Statistics*, pages 448–455, 2009.

[Sha93] Jun Shao. Linear model selection by cross-validation. *Journal of the American statistical Association*, 88(422) :486–494, 1993.

[SI07] Eli Shechtman and Michal Irani. Matching local self-similarities across images and videos. In *Computer Vision and Pattern Recognition, 2007. CVPR'07. IEEE Conference on*, pages 1–8. IEEE, 2007.

[SJC+09] M. Sankaradas, V. Jakkula, S. Cadambi, S. Chakradhar, I. Durdanovic, E. Cosatto, and H.P. Graf. A massively parallel coprocessor for convolutional neural networks. In *Application-specific Systems, Architectures and Processors, 2009. ASAP 2009. 20th IEEE International Conference on*, pages 53–60, July 2009.

[SM07] P. Sabzmeydani and G. Mori. Detecting pedestrians by learning shapelet features. In *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, pages 1–8, June 2007.

[SMH07] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted boltzmann machines for collaborative filtering. In *Proceedings of the 24th international conference on Machine learning*, pages 791–798. ACM, 2007.

[SQZ93] Jocelyn Sérot, Georges Quénot, and Bertrand Zavidovique. Functional programming on a dataflow architecture : Applications in real-time image processing. *Machine Vision and Applications*, 7(1) :44–56, 1993.

[SQZ95] J. Sérot, G. M. Quénot, and B. Zavidovique. A visual dataflow programming environment for a real-time parallel vision machine. *Journal of Visual Languages and Computing*, 6 :327–347, 1995.

[str] stream it. http://groups.csail.mit.edu/cag/streamit/index.shtml. Consultation : 2015-11-10.

[Sut66] W. R. Sutherland. *The on-line graphical specification of computer procedures*. thesis, Massachusetts Institute of Technology, 1966.

[SWB+07] Thomas Serre, Lior Wolf, Stanley Bileschi, Maximilian Riesenhuber, and Tomaso Poggio. Robust object recognition with cortex-like mechanisms. *IEEE Trans. Pattern Anal. Mach. Intell.*, 29(3) :411–426, March 2007.

[TDP15] Hong-Phuc Trinh, Marc Duranton, and Michel Paindavoine. Efficient data encoding for convolutional neural network application. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4) :49, 2015.

[TK15] Vipin Tiwari and Nilay Khare. Hardware implementation of neural network with sigmoidal activation functions using cordic. *Microprocessors and Microsystems*, 39(6) :373–381, 2015.

[TLRW14] Stavros Tripakis, Rhishikesh Limaye, Kaliappa Ravindran, and Guoqiang Wang. On tokens and signals : Bridging the semantic gap between dataflow models and hardware implementations. In *Embedded Computer Systems : Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pages 51–58. IEEE, 2014.

[TPM08] Oncel Tuzel, Fatih Porikli, and Peter Meer. Pedestrian detection via classification on riemannian manifolds. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(10) :1713–1727, October 2008.

[TT10] Xiaoyang Tan and Bill Triggs. Enhanced local texture feature sets for face recognition under difficult lighting conditions. *Trans. Img. Proc.*, 19(6) :1635–1650, June 2010.

[uAS07] Zain ul Abdin and B. Svensson. A study of design efficiency with a high-level language for fpgas. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–7, March 2007.

[Vap82] Vladimir Vapnik. *Estimation of Dependences Based on Empirical Data : Springer Series in Statistics (Springer Series in Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[viv] Vivado hls homepage. http://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. Consultation : 2015-11-10.

[VJ01] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 1, 2001.

[VJS03] P. Viola, M. J. Jones, and D. Snow. Detecting pedestrians using patterns of motion and appearance. *Computer Vision, IEEE International Conference on*, 2, 2003.

[VLL+10] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders : Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research*, 11 :3371–3408, 2010.

[VThG15] Benoît Chappet De Vangel, Cesar Torres-huitzil, and Bernard Girau. Randomly spiking dynamic neural fields. *J. Emerg. Technol. Comput. Syst.*, 11(4) :37 :1–37 :26, April 2015.

[W+94] Paul G Whiting et al. A history of data-flow languages. *Annals of the History of Computing, IEEE*, 16(4) :38–59, 1994.

[WD92] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4) :279–292, 1992.

[WH86] DE Rumelhart GE Hinton RJ Williams and GE Hinton. Learning representations by back-propagating errors. *Nature*, pages 323–533, 1986.

[WHY09] Xiaoyu Wang, Tony X Han, and Shuicheng Yan. An hog-lbp human detector with partial occlusion handling. In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 32–39. IEEE, 2009.

[WN05] Bo Wu and R. Nevatia. Detection of multiple, partially occluded humans in a single image by bayesian combination of edgelet part detectors. In *Computer Vision, 2005. ICCV 2005. Tenth IEEE International Conference on*, volume 1, pages 90–97 Vol. 1, Oct 2005.

[YLJ+13] Herve Yviquel, Antoine Lorence, Khaled Jerbi, Gildas Cocherel, Alexandre Sanchez, and Mickael Raulet. Orcc : Multimedia development made easy. In *Proceedings of the 21st ACM International Conference on Multimedia*, MM '13, pages 863–866, New York, NY, USA, 2013. ACM.

[YPW+15] Shihong Yao, Shaoming Pan, Tao Wang, Chunhou Zheng, Weiming Shen, and Yanwen Chong. A new pedestrian detection method based on combined hog and lss features. *Neurocomputing*, 151, Part 3(0) :1006 – 1014, 2015.

[ZCX+07] Lun Zhang, Rufeng Chu, Shiming Xiang, Shengcai Liao, and Stan Z Li. Face detection based on multi-block lbp representation. In *Advances in biometrics*, pages 11–18. Springer, 2007.

[ZF13] Matthew D Zeiler and Rob Fergus. Stochastic pooling for regularization of deep convolutional neural networks. *arXiv preprint arXiv :1301.3557*, 2013.

[ZHYT11] Junge Zhang, Kaiqi Huang, Yinan Yu, and Tieniu Tan. Boosted local structured hog-lbp for object localization. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 1393–1400, June 2011.

[ZL09] Wei Zheng and Luhong Liang. Fast car detection using image strip features. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2009.

[ZSHH10] Yongbin Zheng, Chunhua Shen, Richard Hartley, and Xinsheng Huang. Effective pedestrian detection using center-symmetric local binary/trinary patterns. *arXiv preprint arXiv :1009.0892*, 2010.

[ZYCA06] Qiang Zhu, Mei-Chen Yeh, Kwang-Ting Cheng, and Shai Avidan. Fast human detection using a cascade of histograms of oriented gradients. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2*, CVPR '06, pages 1491–1498, Washington, DC, USA, 2006. IEEE Computer Society.

[zyn] Zynq-7000, Technical Reference Manual. http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf. Consultation : 2015-11-4.

Publications

Publications avec comité de lecture dans des revues internationales

2015 – "*Bio-inspired Heterogeneous Architecture for Real-Time Pedestrian Detection Applications*", Journal of Real-Time Image Processing (JRTIP), L. Maggiani, **C. Bourrasset** ,J-C. Quinton, F. Berry et J.Sérot. *Manuscrit accepté avec révisions mineures*.

– "*A Dataflow Object Detection System for FPGA-based Smart Camera*", IET Circuits, Devices and Systems, **C. Bourrasset**, L.Maggiani, J. Serot, F. Berry , 2015.

– "*HOG-Dot :a Parallel Kernel-Based Gradient Extraction for Embedded Image Processing*", IEEE Signal Processing Letters, L.Maggiani, **C. Bourrasset**, M.Pettraca, F. Berry, P.Pagano, C.Salvadori, 2015.

2014 – "*High-level dataflow programming for real-time image processing on smart cameras*", Journal of Real-Time Image Processing (JRTIP), J. Serot, F. Berry, **C. Bourrasset**, 2014.

Publications avec comité de lecture dans des conférences internationales

2015 – "Parallel Image Gradient Extraction Core for FPGA-based Smart Cameras", ACM/IEEE International Conference on Distributed Smart Cameras, L.Maggiani, **C. Bourrasset**, F. Berry, J. Serot, 2015.

2013 – "FPGA-based Smart Camera Mote for Pervasive Wireless Network". Seventh ACM/IEEE International Conference on Distributed Smart Cameras, **C. Bourrasset**, J. Serot, F. Berry, ICDSC 2013, **Excellent Paper Award**.

– "DreamCAM : A FPGA-based platform for smart camera networks". Seventh ACM/IEEE International Conference on Distributed Smart Cameras, **C. Bourrasset**, L.Maggiani, J. Serot, F. Berry, P.Pagano, ICDSC 2013.

– "Distributed FPGA-based smart camera architecture for computer vision applications". Seventh ACM/IEEE International Conference on Distributed Smart Cameras, **C. Bourrasset**, L.Maggiani, J. Serot, F. Berry, P.Pagano, ICDSC 2013.

Annexe

A

Description des acteurs pour la détection de piétons

Dans cette annexe, l'ensemble de l'implémentation du système défini dans le chapitre 4 est présenté. Dans le cas des acteurs simples, seul le code CAPH est nécessaire à la compréhension du comportement de l'acteur. Pour ce qui est des acteurs plus élaborés, des représentations graphiques sous forme de machines d'états, correspondantes à la description CAPH sont fournies afin d'accompagner le lecteur dans l'analyse du code. Aussi, les schémas de connexion des entrées/sorties de certains acteurs utilisant des canaux de communication rebouclés seront dessinés.

On rappelle que la distinction entre les jetons de contrôle et de données est assurée par le type *variant* `dc` pour les entrées et sorties, défini comme :

```
type t dc = SoS | EoS | Data of t
```

où `SoS` (Start of Structure), `EoS` (End of Structure) et `Data` sont des *constructeurs* encodant respectivement les jetons de contrôle "<", ">" et les jetons de données. Il est possible d'utiliser une notation abrégée pour le type `dc`, où `SoS` est équivalent à `'<`, `EoS` à `'>` et `Data of t` à `'t`. On utilisera la notation abrégée pour les descriptions d'acteurs alors que la notation complète est donnée sur les machines d'état.

A.1 Méthode HOG-Dot

A.1.1 Convoleur centré

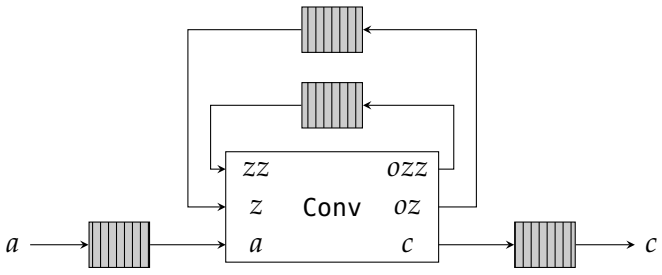


Fig. A.1: Schéma de connexion de l'acteur Convolution 3×3

Listing A.1: Listing CAPH de l'acteur de convolution centré `conv33`

```
actor conv (k:signed<19> array[9], n:unsigned<4>, v:signed<10>)
   in (a:unsigned<8> dc,  z:unsigned<8> dc,  zz:unsigned<8> dc)
  out (c: signed<10> dc, oz:unsigned<8> dc, ozz:unsigned<8> dc)

var s : {WaitSoF, WaitSoL1, BufL1, WaitSoL2, BufL2,WaitNewL, Bufpix1, Bufpix2, Conv, Dumpixel,
     Dumptok, Dumpline, DumpEoF} = WaitSoF

var x1 :unsigned<8>
var x2 :unsigned<8>
var x4 :unsigned<8>
var x5 :unsigned<8>
var x7 :unsigned<8>
var x8 :unsigned<8>

rules
| (s:WaitSoF , a:'<)                  -> (s:WaitSoL1, c:'<)
| (s:WaitSoL1, a:'<)                  -> (s:BufL1, oz:'<)
| (s:BufL1,    a:'p)                  -> (s:BufL1, oz:'p)
| (s:BufL1,    a:'>)                  -> (s:WaitSoL2, oz:'>)
| (s:WaitSoL2, a:'<, z:'<)            -> (s:BufL2, c:'<, ozz:'<, oz:'<)
| (s:BufL2,    a:'x0,z:'x3)           -> (s:BufL2, c:'v, ozz:'x3, oz:'x0)
| (s:BufL2,    a:'>, z:'>)            -> (s:WaitNewL, c:'>, ozz:'>, oz:'>)
| (s:WaitNewL, a:'<, z:'<, zz:'<)     -> (s:Bufpix1, c:'<, ozz:'<, oz:'<)
| (s:WaitNewL, a:'>)                  -> (s:Dumptok, ozz:'>, oz:'>)
| (s:Bufpix1,  a:'x0, z:'x3, zz:'x6) -> (s:Bufpix2, x7:x6, x4:x3, x1:x0, ozz:'x3, oz:'x0)
| (s:Bufpix2,  a:'x0, z:'x3, zz:'x6) -> (s:Conv, c:'v, x8:x7, x5:x4, x2:x1, x7:x6, x4:x3, x1:
    x0,
                                                ozz:'x3, oz:'x0)
| (s:Conv, a:'x0, z:'x3, zz:'x6)      -> (s:Conv, c:let cc = (k[0]:signed<19>)*(x0:signed<19>)+
                                                           (k[1]:signed<19>)*(x1:signed<19>)+
                                                           (k[2]:signed<19>)*(x2:signed<19>)
                                                           (k[3]:signed<19>)*(x3:signed<19>)
                                                           (k[4]:signed<19>)*(x4:signed<19>)
                                                           (k[5]:signed<19>)*(x5:signed<19>)
                                                           (k[6]:signed<19>)*(x6:signed<19>)
                                                           (k[7]:signed<19>)*(x7:signed<19>)
                                                           (k[8]:signed<19>)*(x8:signed<19>)
                                                           in '((cc >> n):signed<10>),
                                                           x8:x7, x5:x4, x2:x1, x7:x6,
                                                           x4:x3, x1:x0, ozz:'x3, oz:'x0)
| (s:Conv, a:'>, z:'>, zz:'>)         -> (s:Dumpixel, c:'v, ozz:'>, oz:'>)
| (s:Dumpixel )                       -> (s:WaitNewL, c:'>)
| (s:Dumptok, z:'<, zz:'< )           -> (s:Dumpline, c:'<);

-- mapping function for conv_hv
net conv_hv kernel norm pad i = let rec (o,z,zz) = conv (kernel,norm,pad) (i,z,zz) in o;
```

Listing A.2: Fonctions d'ordre supérieur pour les convolueurs

```
-- Poids de la méthode HOG-Dot
const coeff = [
    [0S, 512S,0S, 0S,0S, 0S, 0S, -512,0S],
    [0S, 473S,0S, 196S,0S,-196, 0S,-473,0S],
    [0S, 362S,0S, 362S,0S,-362, 0S, -362,0S],
    [0S, 196S,0S, 473S,0S,-473, 0S,-196,0S],
    [0S, 0S, 0S, 512S, 0S, -512, 0S, 0S,0S],
    [0S, -196,0S, 473S,0S, -473, 0S, 196S, 0S],
    [0S, -362S,0S,362S,0S,-362, 0S, 362S,0S],
    [0S, -473,0S, 196S,0S,-196, 0S, 473S,0S]
    ]:signed<19>array[8][9];

-- Mapping function convs parallel
net convs dup coef x =
    let ff i x =  conv_hv (coef[i]) 9 0 x in
        mapi ff (dup i);
```

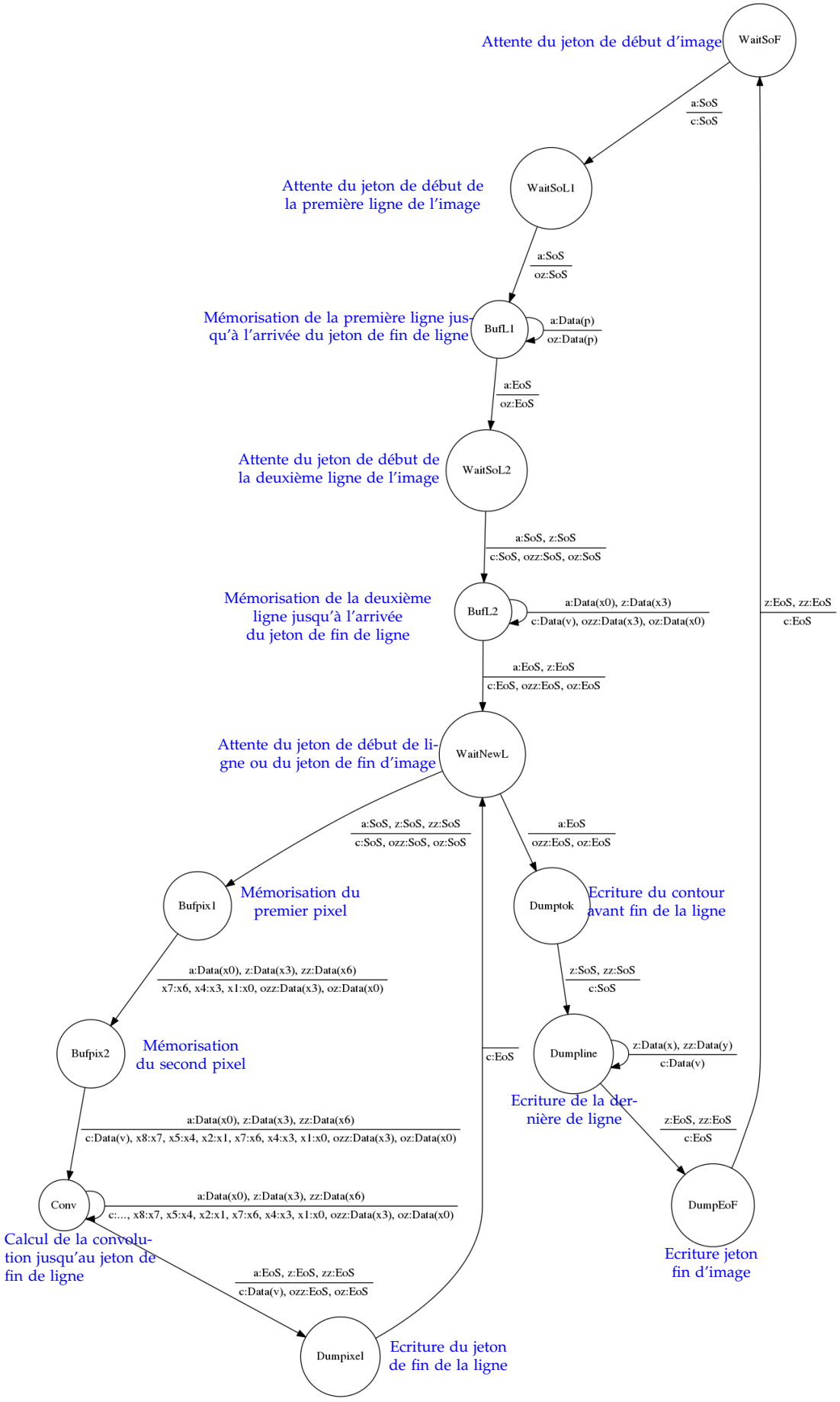


Fig. A.2: Machine d'états de l'acteur de convolution centré `Conv33`

A.1.2 Argmax

Listing A.3: Listing CAPH de la fonction d'Argmax

```
actor abs()
 in(a:signed<s> dc)
 out(s:unsigned<s> dc)
rules
| a:'< -> s:'<
| a:'> -> s:'>
| a:'i -> s: if i<0 then '(-i:unsigned<s>) else '(i:unsigned<s>)
;
```

Listing A.4: Listing CAPH de l'acteur `Comp`

```
actor comp ()
 in  ( i1:unsigned<10> dc,  i2:unsigned<10> dc, nb1: unsigned<3> dc, nb2: unsigned<3> dc)
 out (mag:unsigned<10> dc, bin:unsigned<3> dc)
rules
|(i1:'<, i2:'<, nb1:'<,nb2:'<) -> (mag:'<, bin:'<)
|(i1:'>, i2:'>, nb1:'>,nb2:'>) -> (mag:'>, bin:'>)
|(i1:'a, i2:'b, nb1:'n1,nb2:'n2) -> (mag: if a>b then 'a else 'b, bin: if a>b then 'n1 else '
    n2)
;
```

Listing A.5: Listing CAPH de l'acteur `Binning`

```
actor binning()
 in( mag:unsigned<10> dc, bin:unsigned<3> dc)
 out(h0:unsigned<10> dc, h1:unsigned<10> dc,  h2:unsigned<10> dc,  h3:unsigned<10> dc,
     h4:unsigned<10> dc, h5:unsigned<10> dc,  h6:unsigned<10> dc,  h7:unsigned<10> dc )
rules (mag,bin) -> (h0,h1,h2,h3,h4,h5,h6,h7)
| ('<,'<)    -> ('<,'<,'<,'<,'<,'<,'<,'<)
| ('>,'>)    -> ('>,'>,'>,'>,'>,'>,'>,'>)
| ('m,'b) when b=0 -> ('m,'0,'0,'0,'0,'0,'0,'0)
| ('m,'b) when b=1 -> ('0,'m,'0,'0,'0,'0,'0,'0)
| ('m,'b) when b=2 -> ('0,'0,'m,'0,'0,'0,'0,'0)
| ('m,'b) when b=3 -> ('0,'0,'0,'m,'0,'0,'0,'0)
| ('m,'b) when b=4 -> ('0,'0,'0,'0,'m,'0,'0,'0)
| ('m,'b) when b=5 -> ('0,'0,'0,'0,'0,'m,'0,'0)
| ('m,'b) when b=6 -> ('0,'0,'0,'0,'0,'0,'m,'0)
| ('m,'b) when b=7 -> ('0,'0,'0,'0,'0,'0,'0,'m);
```

Listing A.6: Listing CAPH de l'acteur `paramtodc`

```
actor paramtodc(k:$t2)
     in(istream: $t1 dc)
    out(ostream: $t2 dc)
rules
| istream:'< -> ostream:'<
| istream:'> -> ostream:'>
| istream:'x -> ostream:'k
;
```

Listing A.7: Listing CAPH des fonctions d'ordre supérieur pour le calcul de l'argmax

```
-- Fonction d'ordre supérieur pour le premier étage comp
net comp_init io i1 ko k1 = comp (io, i1, paramtodc ko (io), paramtodc k1 (i1));

-- Fonction d'ordre supérieur du réseau pour l'argmax
net argmax ii =
    let (io,i1,i2,i3,i4,i5,i6,i7) = map (abs) ii in
        let (mag01,bin01) = comp_init io i1 0 1 in
        let (mag23,bin23) = comp_init i2 i3 2 3 in
        let (mag45,bin45) = comp_init i4 i5 4 5 in
        let (mag67,bin67) = comp_init i6 i7 6 7 in
            let (mag0123,bin0123) = comp(mag01,mag23,bin01,bin23) in
            let (mag4567,bin4567) = comp(mag45,mag67,bin45,bin67) in
                let (mag,bin) = comp(mag0123, mag4567, bin0123, bin4567) in
                    binning (mag,bin);
```

A.2 Fonction Histogramme

Listing A.8: Fonctions d'ordre supérieur pour les histogrammes

```
-- Fonction pour création d'un histogramme
net hist n m  hx = vsum n (hsum m (hx));

-- Fonction pour l'application de l'histogramme sur l'ensemble des classes
net hists xs ys hx =
    map (hist xs ys) (hx);
```

Listing A.9: Listing CAPH de l'acteur `hsum`

```
actor hsum (k:unsigned<8>)
  in  (a:unsigned<s> dc)
  out (c:unsigned<s> dc)

var s : {WaitSoF, WaitSoL, Sum} = WaitSoF
var x_sum   : unsigned<s>
var j       : unsigned<8>

rules
| (s:WaitSoF, a:'<)              -> (s:WaitSoL, c:'<)                      --(1)
| (s:WaitSoL, a:'<)              -> (s:Sum,     c:'<, xsum:0, j:0)         --(2)
| (s:WaitSoL, a:'>)              -> (s:WaitSoF, c:'>)                      --(3)
| (s:Sum,     a:'x) when j<(k-1) -> (s:Sum,     xsum: xsum+x, j:j+1)       --(4)
| (s:Sum,     a:'x) when j=(k-1) -> (s:Sum,     c:'(xsum+x), xsum:0, j:0)  --(5)
| (s:Sum,     a:'>)              -> (s:WaitSoL, c:'>);                     --(6)
```

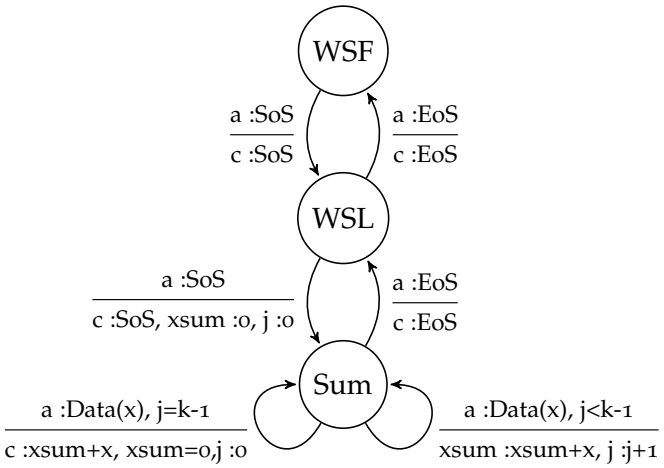
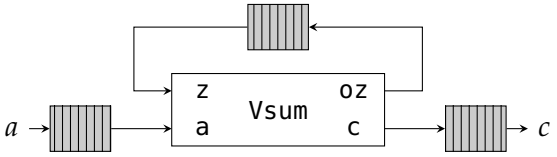


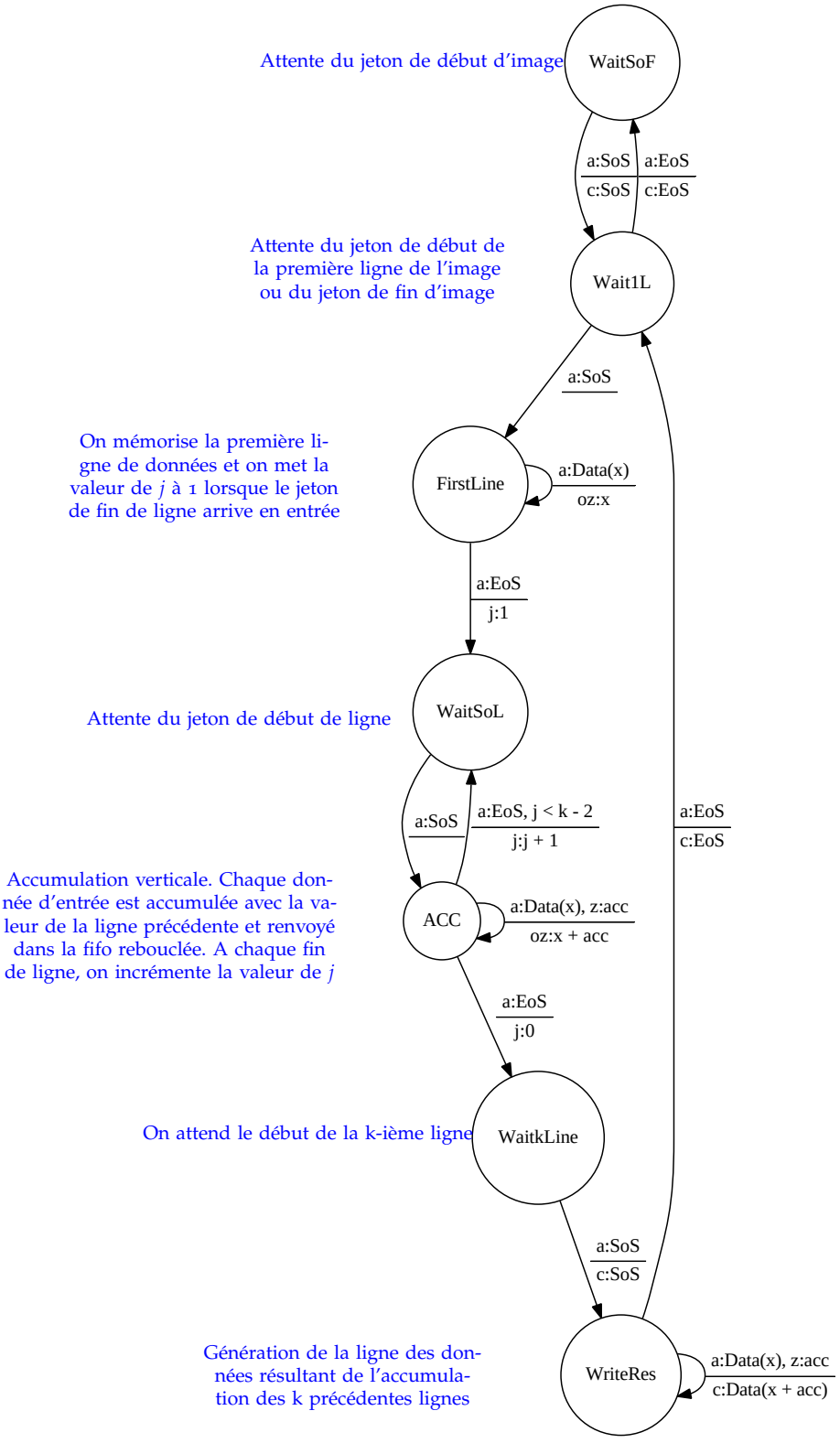
Fig. A.3: Implémentation de l'acteur `hsum`

Listing A.10: Listing CAPH de l'acteur `vsum`

```
actor vsum_act (k:unsigned<8>)
    in  (a:unsigned<16> dc,  z:unsigned<16>)
    out (c:unsigned<16> dc, oz:unsigned<16>)

var s : {WaitSoF,Wait1L, FirstLine, WaitSoL,ACC,WaitkLine,WriteRes} = WaitSoF
var j : unsigned <8>

rules
| (s:WaitSoF,   a:'<)              -> (s:Wait1L, c:'<)
| (s:Wait1L,    a:'>)              -> (s:WaitSoF,c:'>)
| (s:Wait1L,    a:'<)              -> (s:FirstLine)
| (s:FirstLine, a:'x)              -> (s:FirstLine, oz:x )
| (s:FirstLine, a:'>)              -> (s:WaitSoL, j:1)
| (s:WaitSoL,   a:'<)              -> (s:ACC)
| (s:ACC,       a:'x, z:acc)       -> (s:ACC,  oz:x+acc)
| (s:ACC,       a:'>) when j<(k-2) -> (s:WaitSoL, j:j+1)
| (s:ACC,       a:'>)              -> (s:WaitkLine, j: 0)
| (s:WaitkLine, a:'<)              -> (s:WriteRes, c:'< )
| (s:WriteRes,  a:'x, z:acc)       -> (s:WriteRes, c:'(x+acc) )
| (s:WriteRes,  a:'>)              -> (s:Wait1L, c:'> );

-- Mapping function
net vsum k ii = let rec (oo,z1) = vsum_act k (ii,z1) in oo;
```



(a) Schémas de connexions



(b) FSM de l'acteur

Fig. A.4: Implémentation de l'acteur `vsum`

A.3 Acteur Normalisation

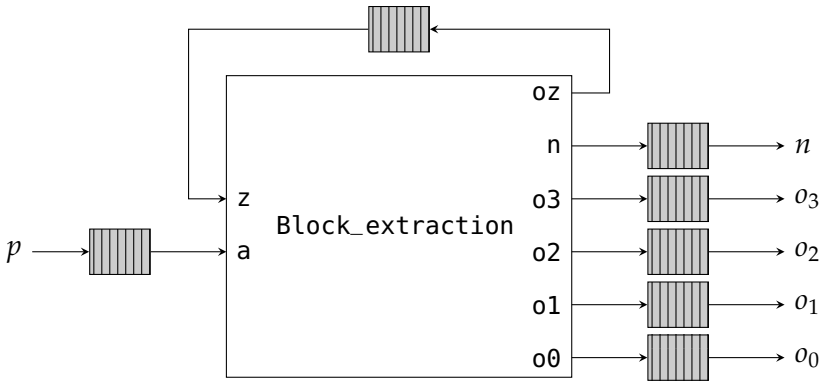


Fig. A.5: Schémas de connexion de l'acteur `block_extraction`

Listing A.11: Extraction block 2x2

```
actor block_extrac()
  in (a:unsigned<16> dc,     -- input
      z:unsigned<16> dc)        -- previous line (fed back through an external link)
  out (oo:unsigned<16> dc,    -- output
       o1:unsigned<16> dc,    -- output
       o2:unsigned<16> dc,    -- output
       o3:unsigned<16> dc,    -- output
        n:unsigned<18> dc,
      oz:unsigned<16> dc)       -- previous line (fed back through an external link)

var s : {WaitSoF,WaitSoL1,BufL1,WaitNewL,Bufpix1,Extract} = WaitSoF
var x1 : unsigned<16>
var x2 : unsigned<16>
var x3 : unsigned<16>
var j:unsigned <4>
var i:unsigned <8>
rules
| (s:WaitSoF, a:'<, z:'<)   -> (s:WaitSoL1, oo:'<, o1:'<, o2:'<, o3:'<,n:'<)
| (s:WaitSoF, a:'<)         -> (s:WaitSoL1, oo:'<, o1:'<, o2:'<, o3:'<,n:'<)
| (s:WaitSoL1, a:'<, z:'<)  -> (s:BufL1, oz:'<)
| (s:WaitSoL1, a:'<)        -> (s:BufL1, oz:'<)
| (s:BufL1, a:'p, z:'y)     -> (s:BufL1, oz:'p)
| (s:BufL1, a:'p)           -> (s:BufL1, oz:'p)
| (s:BufL1, a:'>, z:'>)     -> (s:WaitNewL, oz:'>)
| (s:BufL1, a:'>)           -> (s:WaitNewL, oz:'>)
| (s:WaitNewL, a:'<, z:'<)  -> (s:Bufpix1, oo:'<, o1:'<, o2:'<, o3:'<, oz:'<,n:'<)
| (s:Bufpix1, a:'x0, z:'x2) -> (s:Extract, x3:x2, x1:x0, oz:'x0)
| (s:Extract, a:'x0, z:'x2) -> (s:Extract, oo:'x3, o1:'x2, o2:'x1, o3:'x0,
n:'((x0:unsigned<18>)+(x1:unsigned<18>)+(x2:unsigned<18>)+(x3:unsigned<18>)), x1:x0, x3:x2, oz:'x0)
| (s:Extract, a:'>, z:'>)    -> (s:WaitNewL, oo:'>, o1:'>, o2:'>, o3:'>, oz:'>, n:'>)
| (s:WaitNewL, a:'>,z:'>)    -> (s:WaitSoF, oo:'>, o1:'>, o2:'>, o3:'>,n:'>)
| (s:WaitNewL, a:'>)         -> (s:WaitSoF, oo:'>, o1:'>, o2:'>, o3:'>,n:'>);

-- Fonction d'ordre supérieur pour connexion fifo récursive
net block2x2_extract i = let rec (a,b,c,e,n,l) = block_extrac(i,l) in (a,b,c,e,n);
```

Listing A.12: L1 Norme

```
actor norm()
  in (i:unsigned<16> dc, n:unsigned<22> dc)
 out (s:unsigned<8> dc)
rules
| (i:'<,n:'<)-> s:'<
| (i:'i,n:'n)-> s: if n=0 then 'o else '(   ((i:unsigned<24>) << 8) / (n:unsigned<24>):unsigned
    <8> )
| (i:'>,n:'>)-> s:'>;
```

Listing A.13: Norm factor

```
actor norm_factor()
  in (x1:unsigned<18> dc, x2:unsigned<18> dc, x3:unsigned<18> dc,x4:unsigned<18> dc,
      x5:unsigned<18> dc, x6:unsigned<18> dc, x7:unsigned<18> dc,x8:unsigned<18> dc)
  out(s:unsigned<22> dc)
rules
  (x1,x2,x3,x4,x5,x6,x7,x8) -> s
| ('<,'<,'<,'<,'<,'<,'<,'<) -> '<
| ('a,'b,'c,'d,'e,'f,'g,'h) -> '( (a:unsigned<22>)+(b:unsigned<22>)+(c:unsigned<22>)+(d:
    unsigned<22>)+
                                 (e:unsigned<22>)+(f:unsigned<22>)+(g:unsigned<22>)+(h:
                                     unsigned<22>))
| ('>,'>,'>,'>,'>,'>,'>,'>) -> '>
```

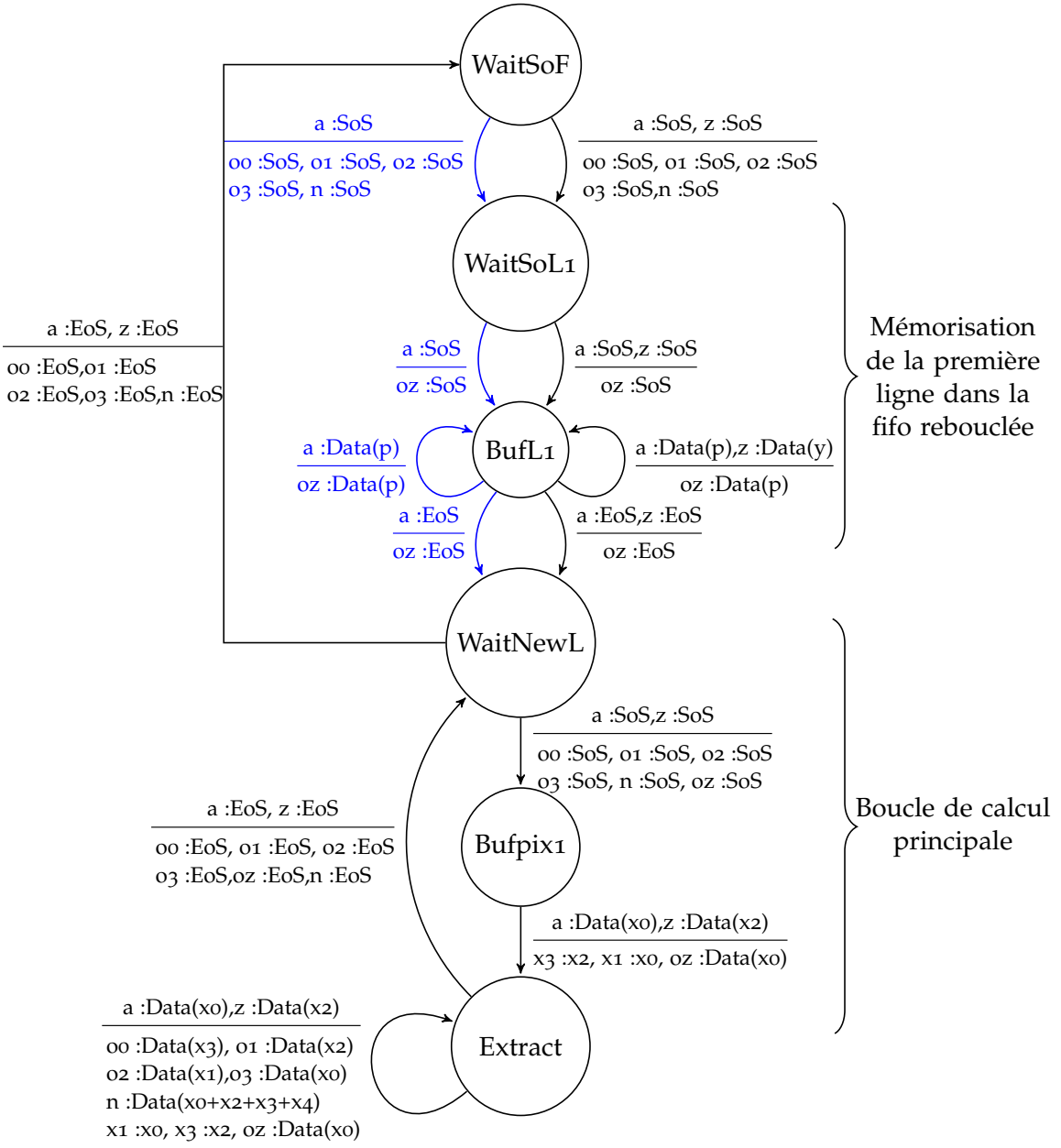


Fig. A.6: FSM de l'acteur `block_extraction`. Les chemins dessinés en bleu ne seront activés que lors de la première image. Ces chemins n'utilisent pas le canal d'entrée z vide au début de la première image, qui contiendra des données par la suite, données invalides que l'on se doit de lire pour éviter le débordement de la FIFO.

A.4 Acteur SkipData

Listing A.14: Skipdata

```
actor skipdata(n:unsigned<8>, m:unsigned<8>)
  in (a:unsigned<s> dc)
  out(o:unsigned<s> dc)

var s : {WSF,WSL,Extract,EndWindow,SkipLine} = WSF
var j:unsigned <8>
var i:unsigned <8>
rules
| (s:WSF, a:'<)                        -> (s:WSL, o:'<,i:0)
| (s:WSL, a:'<)                        -> (s:Extract, o:'<, j:0)
| (s:WSL, a:'>)                        -> (s:WSF, o:'>)
| (s:Extract, a:'xo) when j<m -> (s:Extract, o:'xo, j:j+1)
| (s:Extract, a:'xo)                   -> (s:Extract, j:0)
| (s:Extract, a:'>) when i < (n-1)  -> (s:WSL, o:'>, i:i+1)
| (s:Extract, a:'>)                    -> (s:EndWindow, o:'>)
| (s:EndWindow, a:'<)                -> (s:SkipLine, i:0)
| (s:EndWindow, a:'>)                -> (s:WSF, o:'>)
| (s:SkipLine, a:'xo)                -> (s:SkipLine)
| (s:SkipLine, a:'>)                 -> (s:WSL);
```

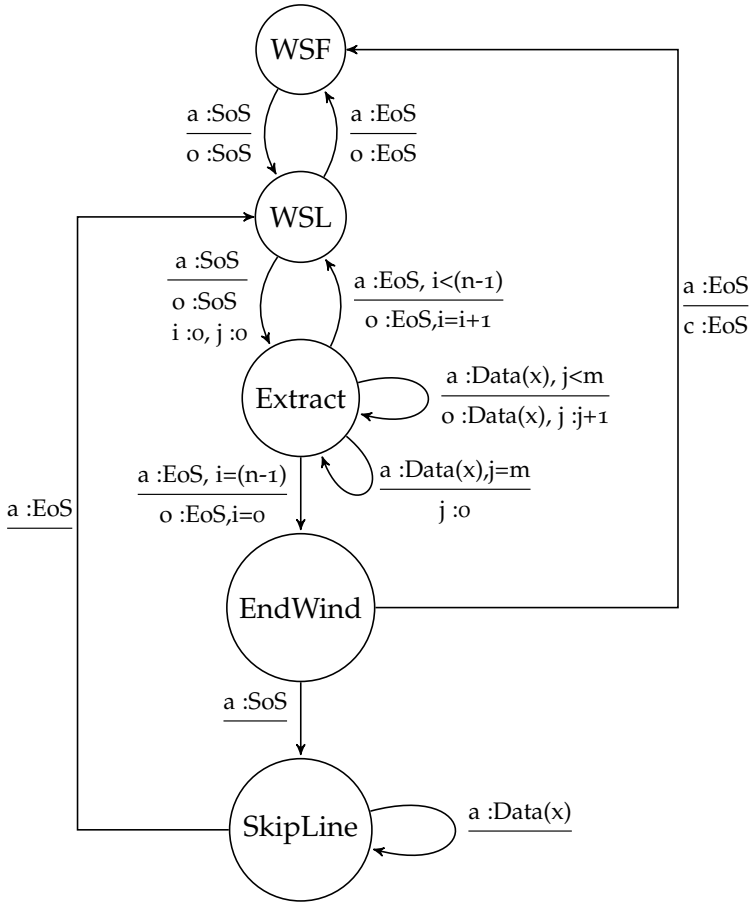
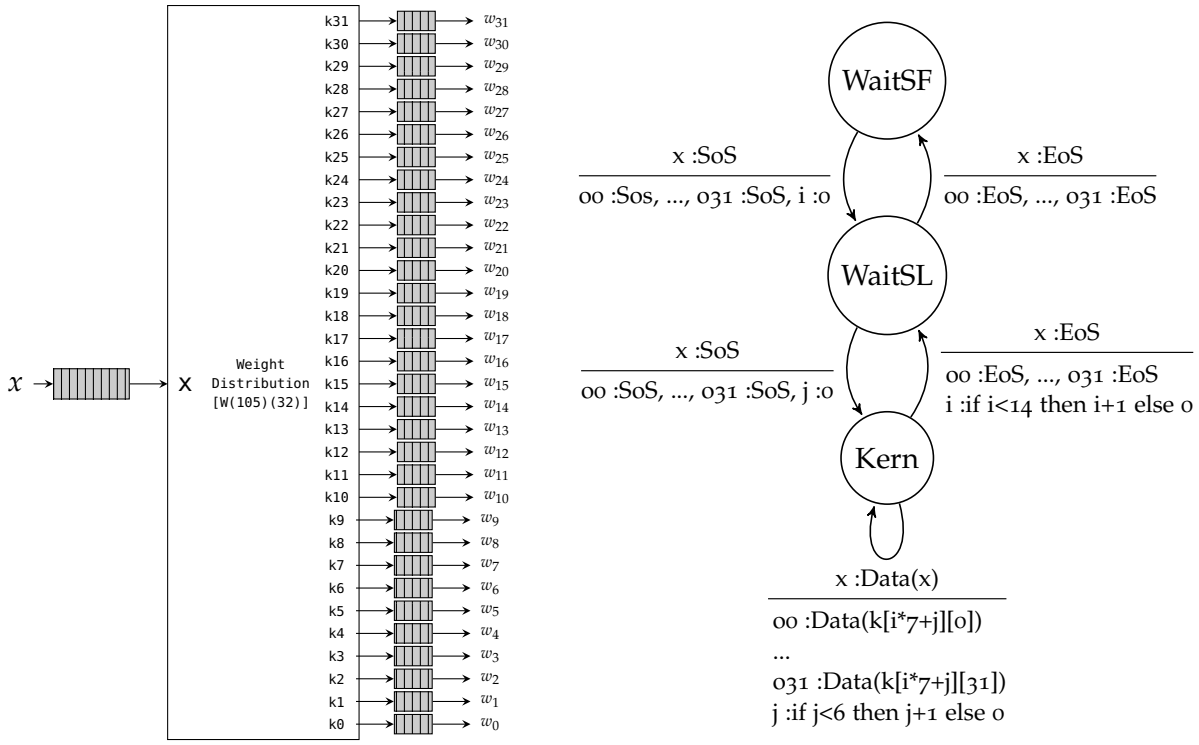


Fig. A.7: Implémentation de l'acteur `Skip`

A.5 Acteur Distribution de poids



(a) Schéma de connexions

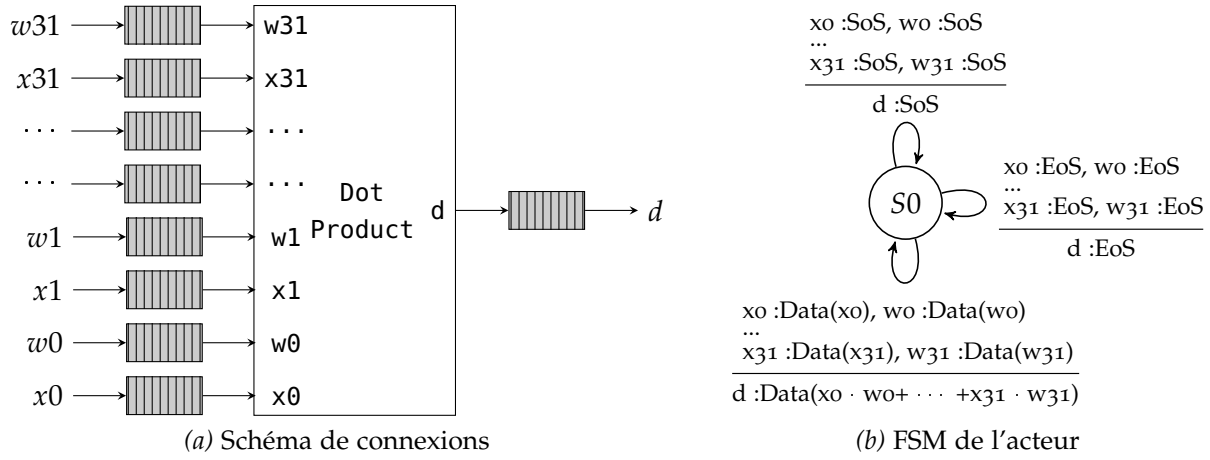
(b) FSM de l'acteur

Fig. A.8: Implémentation de l'acteur `Weight Distribution`

Listing A.15: Acteur de distribution de poids

```
actor kern (k:signed<9> array[105][32])
 in (ii: unsigned<8> dc)
 out(ko:signed<9> dc, k1:signed<9> dc,  k2:signed<9> dc,  k3:signed<9> dc,
     k4:signed<9> dc, k5:signed<9> dc,  k6:signed<9> dc,  k7:signed<9> dc,
     k8:signed<9> dc, k9:signed<9> dc, k10:signed<9> dc, k11:signed<9> dc,
    k12:signed<9> dc,k13:signed<9> dc, k14:signed<9> dc, k15:signed<9> dc,
    k16:signed<9> dc,k17:signed<9> dc, k18:signed<9> dc, k19:signed<9> dc,
    k20:signed<9> dc,k21:signed<9> dc, k22:signed<9> dc, k23:signed<9> dc,
    k24:signed<9> dc,k25:signed<9> dc, k26:signed<9> dc, k27:signed<9> dc,
    k28:signed<9> dc,k29:signed<9> dc, k30:signed<9> dc, k31:signed<9> dc)
var s : {WaitSF, WaitSL, Kern} = WaitSF
var i : unsigned<12>
var j : unsigned<12>
rules
|(s:WaitSF, ii:'<) -> (s:WaitSL, ko:'<, k1:'<, k2:'<, k3:'<, k4:'<, k5:'<, k6:'<, k7:'<,
                                 k8:'<, k9:'<,k10:'<,k11:'<,k12:'<,k13:'<,k14:'<,k15:'<,
                                k16:'<,k17:'<,k18:'<,k19:'<,k20:'<,k21:'<,k22:'<,k23:'<,
                                k24:'<,k25:'<,k26:'<,k27:'<,k28:'<,k29:'<,k30:'<,k31:'<,i:0)

|(s:WaitSL, ii:'>) -> (s:WaitSF, ko:'>, k1:'>, k2:'>, k3:'>, k4:'>, k5:'>, k6:'>, k7:'>,
                                 k8:'>, k9:'>,k10:'>,k11:'>,k12:'>,k13:'>,k14:'>,k15:'>,
                                k16:'>,k17:'>,k18:'>,k19:'>,k20:'>,k21:'>,k22:'>,k23:'>,
                                k24:'>,k25:'>,k26:'>,k27:'>,k28:'>,k29:'>,k30:'>,k31:'>)

|(s:WaitSL, ii:'<) -> (s:Kern, ko:'<, k1:'<, k2:'<, k3:'<, k4:'<, k5:'<, k6:'<, k7:'<,
                               k8:'<, k9:'<,k10:'<,k11:'<,k12:'<,k13:'<,k14:'<,k15:'<,
                              k16:'<,k17:'<,k18:'<,k19:'<,k20:'<,k21:'<,k22:'<,k23:'<,
                              k24:'<,k25:'<,k26:'<,k27:'<,k28:'<,k29:'<,k30:'<,k31:'<, j:0)

|(s:Kern, ii:'>) -> (s:WaitSL, ko:'>, k1:'>, k2:'>, k3:'>, k4:'>, k5:'>, k6:'>, k7:'>,
                               k8:'>, k9:'>,k10:'>,k11:'>,k12:'>,k13:'>,k14:'>,k15:'>,
                              k16:'>,k17:'>,k18:'>,k19:'>,k20:'>,k21:'>,k22:'>,k23:'>,
                              k24:'>,k25:'>,k26:'>,k27:'>,k28:'>,k29:'>,k30:'>,k31:'>, i: if i
                                  <14 then i+1 else 0)

|(s:Kern, ii:'x) -> (s:Kern,
       ko:'k[(i*7)+j][0],  k1:'k[(i*7)+j][1],  k2:'k[(i*7)+j][2],  k3:'k[(i*7)+j][3],
       k4:'k[(i*7)+j][4],  k5:'k[(i*7)+j][5],  k6:'k[(i*7)+j][6],  k7:'k[(i*7)+j][7],
       k8:'k[(i*7)+j][8],  k9:'k[(i*7)+j][9], k10:'k[(i*7)+j][10],k11:'k[(i*7)+j][11],
       k12:'k[(i*7)+j][12],k13:'k[(i*7)+j][13],k14:'k[(i*7)+j][14],k15:'k[(i*7)+j][15],
       k16:'k[(i*7)+j][16],k17:'k[(i*7)+j][17],k18:'k[(i*7)+j][18],k19:'k[(i*7)+j][19],
       k20:'k[(i*7)+j][20],k21:'k[(i*7)+j][21],k22:'k[(i*7)+j][22],k23:'k[(i*7)+j][23],
       k24:'k[(i*7)+j][24],k25:'k[(i*7)+j][25],k26:'k[(i*7)+j][26],k27:'k[(i*7)+j][27],
       k28:'k[(i*7)+j][28],k29:'k[(i*7)+j][29],k30:'k[(i*7)+j][30],k31:'k[(i*7)+j][31], j: if
            j<6 then j+1 else 0);
```

A.6 Acteur de produit scalaire



(a) Schéma de connexions *(b)* FSM de l'acteur

Fig. A.9: Implémentation de l'acteur `Dot product`

Listing A.16: Acteur Dot product

```
actor dot_product()
 in  (  x0:signed<9> dc,  x1:signed<9> dc,  x2:signed<9> dc,  x3:signed<9> dc,
        x4:signed<9> dc,  x5:signed<9> dc,  x6:signed<9> dc,  x7:signed<9> dc,
        x8:signed<9> dc,  x9:signed<9> dc, x0:signed<9> dc, x1:signed<9> dc,
       x2:signed<9> dc, x3:signed<9> dc, x4:signed<9> dc, x5:signed<9> dc,
       x6:signed<9> dc, x7:signed<9> dc, x8:signed<9> dc, x9:signed<9> dc,
       x0:signed<9> dc, x1:signed<9> dc, x2:signed<9> dc, x3:signed<9> dc,
       x4:signed<9> dc, x5:signed<9> dc, x6:signed<9> dc, x7:signed<9> dc,
       x8:signed<9> dc, x9:signed<9> dc, x0:signed<9> dc, x1:signed<9> dc,
        w0:signed<9> dc,  w1:signed<9> dc,  w2:signed<9> dc,  w3:signed<9> dc,
        w4:signed<9> dc,  w5:signed<9> dc,  w6:signed<9> dc,  w7:signed<9> dc,
        w8:signed<9> dc,  w9:signed<9> dc, w10:signed<9> dc, w11:signed<9> dc,
       w12:signed<9> dc, w13:signed<9> dc, w14:signed<9> dc, w15:signed<9> dc,
       w16:signed<9> dc, w17:signed<9> dc, w18:signed<9> dc, w19:signed<9> dc,
       w20:signed<9> dc, w21:signed<9> dc, w22:signed<9> dc, w23:signed<9> dc,
       w24:signed<9> dc, w25:signed<9> dc, w26:signed<9> dc, w27:signed<9> dc,
       w28:signed<9> dc, w29:signed<9> dc, w30:signed<9> dc, w31:signed<9> dc
)
 out ( dot: signed<24> dc )

rules (x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x0, x1, x2, x3, x4, x5, x6, x7, x8, x9, x0, x1,
     x2, x3, x4, x5, x6, x7, x8, x9, x0, x1,w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11,
    w12, w13, w14, w15, w16, w17, w18, w19, w20, w21, w22, w23, w24, w25, w26, w27, w28, w29,
    w30, w31) -> dot
|('<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<,
    '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<,
    '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<, '<) -> '<
|('>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>,
    '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>,
    '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>, '>)-> '>
|('x0, 'x1, 'x2, 'x3, 'x4, 'x5, 'x6, 'x7, 'x8, 'x9, 'x0, 'x1, 'x2, 'x3, 'x4, 'x5,
  'x6,'x7,'x8,'x9,'x0,'x1,'x2,'x3, x4,'x5, 'x6, 'x7, 'x8, 'x9, 'x0, 'x1,
   'w0, 'w1, 'w2, 'w3, 'w4, 'w5, 'w6, 'w7, 'w8, 'w9, 'w10, 'w11, 'w12, 'w13, 'w14, 'w15,
  'w16,'w17,'w18,'w19,'w20,'w21,'w22,'w23,'w24,'w25, 'w26, 'w27, 'w28, 'w29, 'w30, 'w31 )->
    let c = (x0:signed<24>)* (w0:signed<24>) + (x1:signed<24>)* (w1:signed<24>) +
            (x2:signed<24>)* (w2:signed<24>) + (x3:signed<24>)* (w3:signed<24>) +
            (x4:signed<24>)* (w4:signed<24>) + (x5:signed<24>)* (w5:signed<24>) +
            (x6:signed<24>)* (w6:signed<24>) + (x7:signed<24>)* (w7:signed<24>) +
            (x8:signed<24>)* (w8:signed<24>) + (x9:signed<24>)* (w9:signed<24>) +
           (x0:signed<24>)*(w10:signed<24>) + (x1:signed<24>)*(w11:signed<24>) +
           (x2:signed<24>)*(w12:signed<24>) + (x3:signed<24>)*(w13:signed<24>) +
           (x4:signed<24>)*(w14:signed<24>) + (x5:signed<24>)*(w15:signed<24>) +
           (x6:signed<24>)*(w16:signed<24>) + (x7:signed<24>)*(w17:signed<24>) +
           (x8:signed<24>)*(w18:signed<24>) + (x9:signed<24>)*(w19:signed<24>) +
           (x0:signed<24>)*(w20:signed<24>) + (x1:signed<24>)*(w21:signed<24>) +
           (x2:signed<24>)*(w22:signed<24>) + (x3:signed<24>)*(w23:signed<24>) +
           (x4:signed<24>)*(w24:signed<24>) + (x5:signed<24>)*(w25:signed<24>) +
           (x6:signed<24>)*(w26:signed<24>) + (x7:signed<24>)*(w27:signed<24>) +
           (x8:signed<24>)*(w28:signed<24>) + (x9:signed<24>)*(w29:signed<24>) +
           (x0:signed<24>)*(w30:signed<24>) + (x1:signed<24>)*(w31:signed<24>) in 'c;
```

A.7 Acteur Fenêtrage

Listing A.17: Acteur Fenetrage

```
-- Fenêtrage statique: utilisation des acteurs xsum et hsum
net windowing n m x = vsum n (hsum m (hx));
```

A.8 Acteur Biais

Listing A.18: Acteur Biais

```
actor decision(bias:signed<32>)
 in (i:signed<32> dc )
 out (o:unsigned<1> dc)
rules
| i:'< -> o:'<
| i:'> -> o:'>
| i:'p -> o: if p+bias < oS then 'o else '1
;
```

B

Impact du Filtrage DNF sur la séquence complète

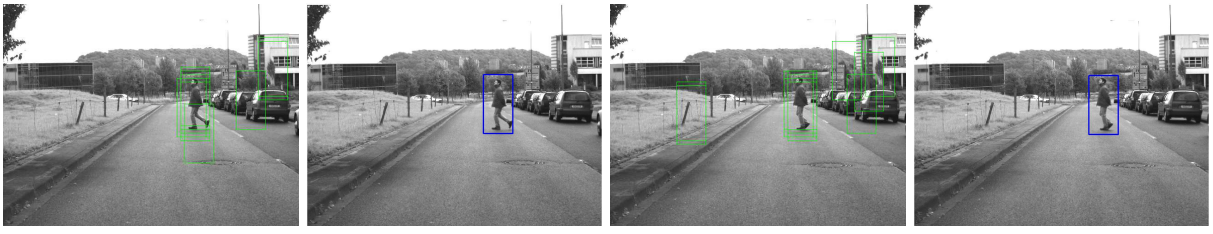
Dans cette annexe, les résultats de détection sur l'intégralité de séquence Daimler sélectionnée sont présentés. Pour chaque image, les résultats procurés par notre implémentation logicielle ILR (émulant le comportement du système matériel) et les résultats après le filtrage par un système DNF.

1-ILR 1-DNF 2-ILR 2-DNF

3-ILR 3-DNF 4-ILR 4-DNF

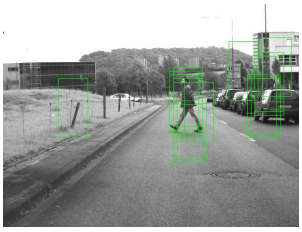
5-ILR 5-DNF 6-ILR 6-DNF

7-ILR 7-DNF 8-ILR 8-DNF

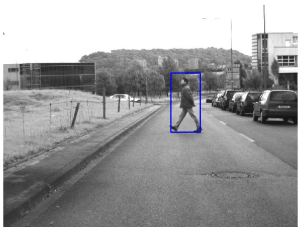


9-ILR 9-DNF 10-ILR 10-DNF

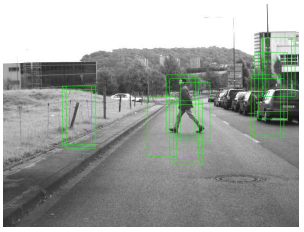
11-ILR 11-DNF 12-ILR 12-DNF



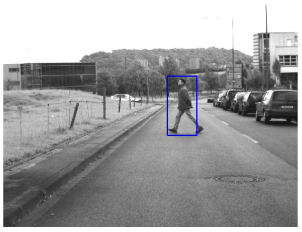
13-ILR



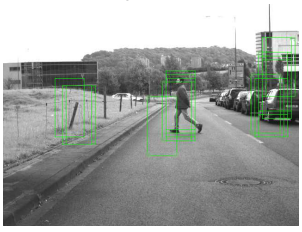
13-DNF



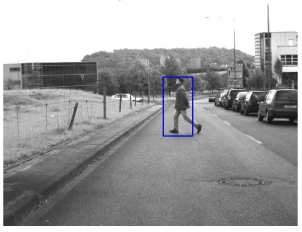
14-ILR



14-DNF

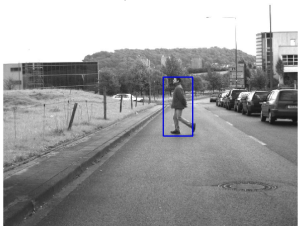


15-ILR

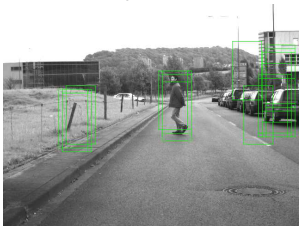


15-DNF

16-ILR

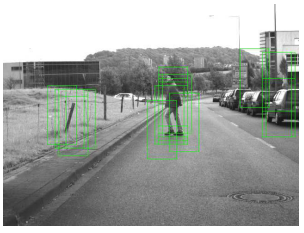


16-DNF



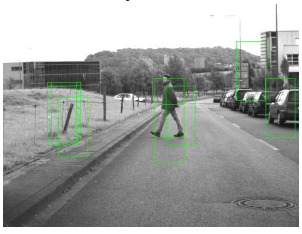
17-ILR

17-DNF

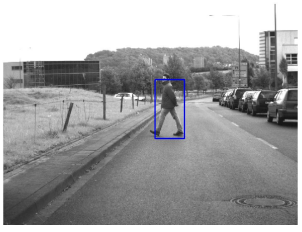


18-ILR

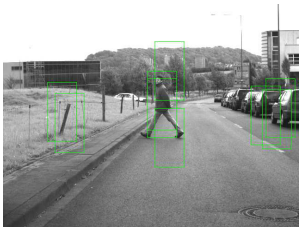
18-DNF



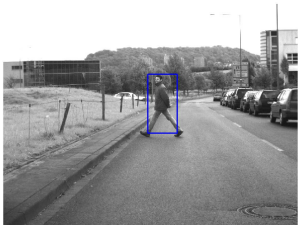
19-ILR



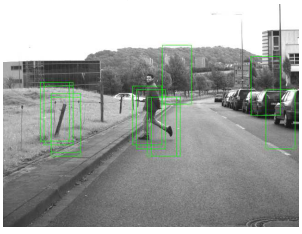
19-DNF



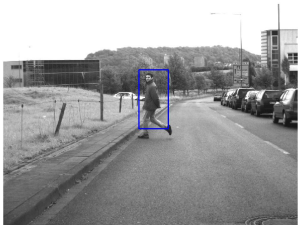
20-ILR



20-DNF

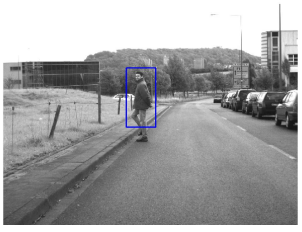


21-ILR

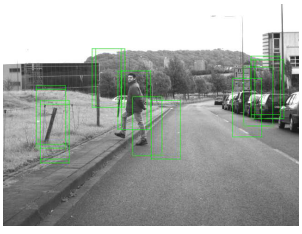


21-DNF

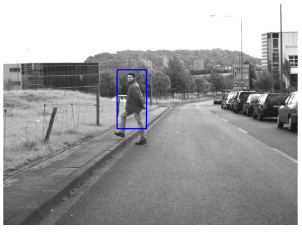
22-ILR



22-DNF

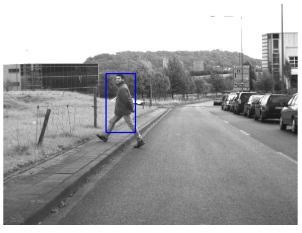


23-ILR

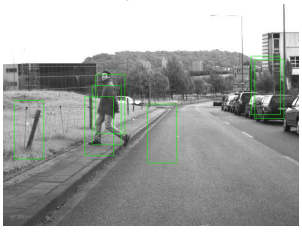


23-DNF

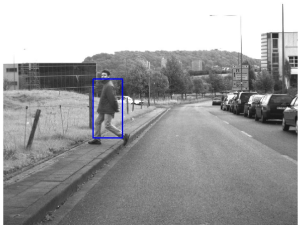
24-ILR



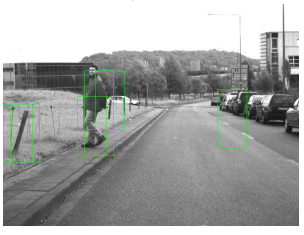
24-DNF



25-ILR

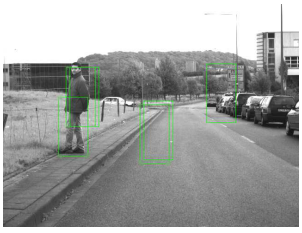


25-DNF

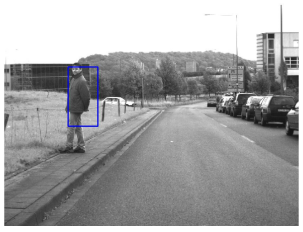


26-ILR

26-DNF



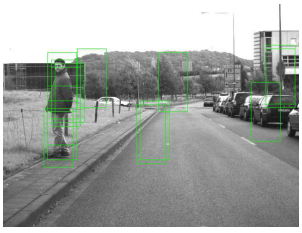
27-ILR



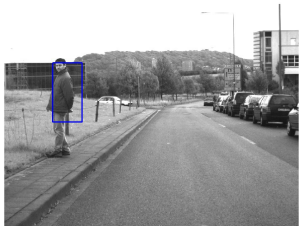
27-DNF

28-ILR

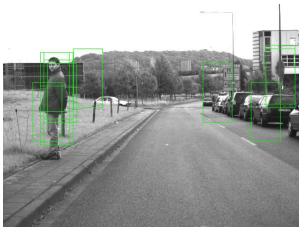
28-DNF



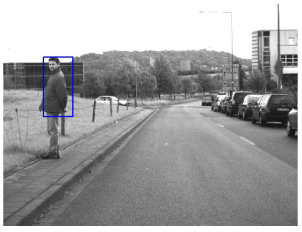
29-ILR



29-DNF



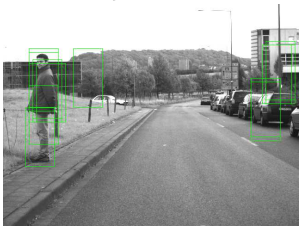
30-ILR



30-DNF

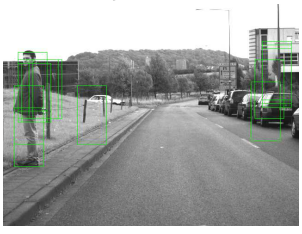
31-ILR

31-DNF



32-ILR

32-DNF



33-ILR

33-DNF

34-ILR

34-DNF

C

Description des acteurs pour les réseaux convolutionnels

C.1 Fonctions d'ordre supérieur pour les couches de convolution

Listing C.1: Listing CAPH des couches de convolution d'un CNN

```
-- Fonction convs: Couche C1 : ensemble de convolueurs à partir d'un seul flux
-- ============================================================
-- Paramètres de la fonction
-- actor_conv: nom de l'acteur à utiliser (ex:conv233c_wb_opt)
-- dup_fct: fonction de replication de entrees 1 to n => fixe le nombre de convolueurs à instancier
-- kernels_weights: tableau de poids
-- shift: facteur de normalisation des convolutions
-- biais_weights: biais ajouté en sortie.
-- x: flux d'entrée
-- ====================================================================
net convs actor_conv dup_fct kernels_weights shift biais_weights  x =
    let ff i x =  biais (biais_weights[i]) (actor_conv (kernels_weights[i]) shift x ) in
        mapi ff (dup_fct x);

-- fonctin Neurone avec acteur d'activation intégré
--====================================================================
-- x: tuple de flux d'entrée
-- actor_conv: nom de l'acteur à utiliser pour la convolution
-- kernels_weights: tableau de poids
-- shift: facteur de normalisation des convolutions
-- biais_weight: 1 biais spar neurone.
-- sumx: acteur de somme des conv
-- actor_activation: acteur d'activation
-- ====================================================================
net neurone_act actor_conv kernels_weights shift biais_weight sumx actor_activation x =
    let ff i x =  actor_conv (kernels_weights[i]) shift x in
        actor_activation(biais biais_weight (sumx (mapi ff (x)))) ;

-- Fonction convlayer: Couche de neurones
--====================================================================
-- conv_act: nom de l'acteur à utiliser pour la convolution
-- weights: tableau de poids 3D
-- shift: facteur de normalisation des convolutions
-- biais_weight: tableau de biais: 1 Biais par neurone
-- sum_act: acteur de somme des conv (depend du nombre de conv dans un neurone
-- fact : actur d'activation
-- ttx: tuple of tuple! chaque tuple correspond aux connections d'un neurone
-- ====================================================================
net convlayer conv_act weights shift biais sum_act fact ttx =
    let ff i ttx = neurone_act conv_act (weights[i]) o (biais[i]) sum_act fact ttx in
        mapi ff (ttx);
```

C.2 Fonctions d'ordre supérieur pour les couches complètement connectée

Listing C.2: Listing CAPH des couches complètement connectée d'un CNN

```
-- Fonctions fc_act et fclayer spécifiques
-- ========================================================
-- Paramètres de la fonction fc_act
-- distr_act: nom de l'acteur de distribution de poids
-- weights: tableau de poids
-- n: nombre d'éléments vecticaux dans une carte entrante
-- m: nombre d'éléments horizontaux dans une carte entrante
-- x: flux utilisé pour l'activation
-- x0,...,x3: flux d'entrées
-- ========================================================================
net fcc_opt distr_act weights n m b x x0 x1 x2 x3 =
    let (w0,w1,w2,w3)= distr_act (weights,n,m) (x) in
    let yy = dot8(x0,x1,x2,x3,w0,w1,w2,w3) in
    biais b (vsum n (hsum m (yy)));

-- Paramètres de la fonction fclayer
-- distr_act: nom de l'acteur de distribution de poids
-- weights: tableau de poids
-- n: nombre d'éléments vecticaux dans une carte entrante
-- m: nombre d'éléments horizontaux dans une carte entrante
-- x: flux utilisé pour l'activation
-- ttx: t-uplet d'entrées
-- x0, x3 : élements du t-uplet ttx
-- ========================================================================
net fclayer_opt distr_act weights nx ny biais x ttx x0 x1 x2 x3 =
    let phi i ttx = fcc_opt distr_act (weights[i]) nx ny (biais[i]) x x0 x1 x2 x3 in
    nappi 4 phi ttx;

-- Fonctions fc_act et fclayer génériques avec produit scalaire partiel
-- ============================================================
net fcc2 distr_act weights n m b x nx=
    let w = distr_act (weights,n,m) (x) in
    let yy = foldl sum2 ( map2 (dot2) (w,nx)) in
    biais b (vsum n (hsum m (yy)));

net fclayer distr weights nx ny biais x ttx =
    let phi i ttx = fcc2 distr (weights[i]) nx ny (biais[i]) x ttx in
    nappi 4 phi ttx;
```

C.3 Acteurs pour les couches de convolution

C.3.1 Acteur de convolution avec suppression des bords

Listing C.3: Listing CAPH de l'acteur de convolution centré `conv33`

```
-----------------------------------
-- CONV 33 avec suppression contour (sans blanking)
-----------------------------------

actor conv33 (k:int<s,m> array[9], n:unsigned<4>)
  in (a:int<s,m> dc,  z:int<s,m> dc, zz:int<s,m> dc)
 out (c:int<s,m> dc, oz:int<s,m> dc,ozz:int<s,m> dc)

var s : {WaitSoF,WaitSoL1,BufL1,WaitSoL2,BufL2,WaitNewL,Bufpix1,Bufpix2,Conv} = WaitSoF

var x1 :int<s,m>
var x2 :int<s,m>
var x4 :int<s,m>
var x5 :int<s,m>
var x7 :int<s,m>
var x8 :int<s,m>

rules
| (s:WaitSoF , a:'<)                -> (s:WaitSoL1, c:'<)
| (s:WaitSoL1, a:'<, z:'<, zz:'<)   -> (s:BufL1, oz:'<) -- cleaning fifo rules after first image
| (s:WaitSoL1, a:'<)                -> (s:BufL1, oz:'<)
| (s:BufL1, a:'p,   z:'x, zz:'y)    -> (s:BufL1, oz:'p)-- cleaning fifo rules after first image
| (s:BufL1, a:'p)                   -> (s:BufL1, oz:'p)
| (s:BufL1, a:'>,   z:'>,zz:'>)     -> (s:WaitSoL2, oz:'>)-- cleaning fifo rules after first image
| (s:BufL1, a:'>)                   -> (s:WaitSoL2, oz:'>)
| (s:WaitSoL2, a:'<, z:'<)          -> (s:BufL2, ozz:'<, oz:'<)
| (s:BufL2, a:'x0,z:'x3)            -> (s:BufL2, ozz:'x3, oz:'x0)
| (s:BufL2, a:'>, z:'>)             -> (s:WaitNewL, ozz:'>, oz:'>)
| (s:WaitNewL, a:'<, z:'<, zz:'<)   -> (s:Bufpix1, c:'<, ozz:'<, oz:'<)
| (s:WaitNewL, a:'>)                -> (s:WaitSoF,c:'>)
| (s:Bufpix1, a:'x0, z:'x3, zz:'x6) -> (s:Bufpix2, x7:x6, x4:x3, x1:x0, ozz:'x3, oz:'x0)
| (s:Bufpix2, a:'x0, z:'x3, zz:'x6) -> (s:Conv, x8:x7, x5:x4, x2:x1, x7:x6, x4:x3, x1:x0, ozz
    :'x3, oz:'x0)
| (s:Conv, a:'x0, z:'x3, zz:'x6)    ->
    (s:Conv,
     c: let  cc = k[0]*x0+k[1]*x1+k[2]*x2+k[3]*x3+k[4]*x4+k[5]*x5+k[6]*x6+k[7]*x7+k[8]*x8
        in 'cc>>n, x8:x7, x5:x4, x2:x1, x7:x6, x4:x3, x1:x0, ozz:'x3, oz:'x0)
| (s:Conv, a:'>, z:'>, zz:'>)         -> (s:WaitNewL, c:'>, ozz:'>, oz:'>)
;
net conv233c_wb_opt kernel norm i = let rec (o,z,zz) = conv33 (kernel,norm) (i,z,zz) in o;
```

C.3.2 Acteur de Somme et de biais

C.3.3 Acteur ReLU

Listing C.4: Listing CAPH de l'acteur de biais et de ReLU

```
actor biais(b:signed<16>)
 in (a:int<s,m> dc)
 out(c:int<s,m> dc)
rules
|a:'< -> c:'<
|a:'> -> c:'>
|a:'x -> c:'(x+b);

actor sum3
 in (i1: int<s,m> dc, i2: int<s,m> dc, i3: int<s,m> dc)
 out (o: int<s,m> dc)
rules
| (i1:'<, i2:'<, i3:'<) -> o:'<
| (i1:'>, i2:'>, i3:'>) -> o:'>
| (i1:'a, i2:'b, i3:'c) -> o:'(a+b+c);
```

Listing C.5: Listing CAPH de l'acteur ReLU

```
actor relu
 in (a:int<s,m> dc)
 out(c:int<s,m> dc)
rules
|a:'< -> c:'<
|a:'> -> c:'>
|a:'x -> c:if x > 0 then 'x else '0;
```

C.4 Acteurs pour les couches de sous-échantillonnage

C.4.1 Acteur maxpool

Listing C.6: Listing CAPH des acteurs de sous-échantillonnage `pool_h` et `pool_v`

```
function sum (x,y) = x+y;
function max(x,y) = if x>y then x else y;

-- Horizontal operation
actor poolh (k:unsigned<8>)
  in (a:int<s,m> dc )
  out(c:int<s,m> dc )

var s : {WaitSoF,WaitSoL,Sum} = WaitSoF
var xsum  : int<s,m>
var j : unsigned <8>

rules
| (s:WaitSoF, a:'<) -> (s:WaitSoL, c:'<)
| (s:WaitSoL, a:'<) -> (s:Sum, c:'<, xsum:0, j:0)
| (s:WaitSoL, a:'>) -> (s:WaitSoF, c:'>)
| (s:Sum, a:'x) when j<(k-1) -> (s:Sum, xsum: max(xsum,x), j:j+1)
| (s:Sum, a:'x) when j=(k-1) -> (s:Sum, c:'(max(xsum,x)), xsum:0, j:0)
| (s:Sum, a:'>) -> (s:WaitSoL,c:'>)
;


-- Vertical Operation
actor poolv (k:unsigned<8>)
    in (a:int<s,m> dc, z:int<s,m>)
    out (c:int<s,m> dc, oz:int<s,m> )

  var s : {WaitSoF,Wait1L, FirstLine, WaitSoL,ACC,WaitkLine,WriteRes} = WaitSoF
  var j : unsigned <8>

 rules
| (s:WaitSoF, a:'<)       -> (s:Wait1L, c:'<)
| (s:Wait1L, a:'>)        -> (s:WaitSoF,c:'>)
| (s:Wait1L, a:'<)        -> (s:FirstLine)
| (s:FirstLine, a:'x)     -> (s:FirstLine, oz:x )
| (s:FirstLine, a:'>) when k>2 -> (s:WaitSoL, j:1)
| (s:FirstLine, a:'>) -> (s:WaitkLine, j:1)          -- si k=2 go direct en WriteRes
| (s:WaitSoL, a:'<)       -> (s:ACC)
| (s:ACC, a:'x, z:acc)    -> (s:ACC,  oz:max(x,acc))
| (s:ACC, a:'>) when j<(k-2)   -> (s:WaitSoL, j:j+1)
| (s:ACC, a:'>)              -> (s:WaitkLine, j: 0)
| (s:WaitkLine, a:'<)        -> (s:WriteRes, c:'< )
| (s:WriteRes, a:'x, z:acc)      -> (s:WriteRes, c:'(max(x,acc)) )
| (s:WriteRes, a:'>)         -> (s:Wait1L, c:'> )
;

net poolv_m k ii = let rec (oo,z1) = poolv k (ii,z1) in oo;

-- a surcharger avec des fonctions differentes suivant operation effectuée (ici max)
net pool n m  x = poolv_m n (poolh m (x));
```

C.5 Acteurs pour les couches complètement connectée

C.5.1 Exemple d'acteur de distribution de poids

C.5.2 Acteur de produit scalaire

Listing C.7: Listing CAPH de l'acteur de distribution de poids

```
actor distr4(wi:signed<16> array[4][16], n:unsigned<8>,m:unsigned<8>)
 in (ii:signed<16> dc)
 out(wo:signed<16> dc,w1:signed<16> dc,w2:signed<16> dc,w3:signed<16> dc)
 var s : {WaitSF, WaitSL, Kern} = WaitSF
 var i : unsigned<8>
 var j : unsigned<8>
 var k : unsigned<8>
rules
|(s:WaitSF, ii:'<) -> (s:WaitSL, wo:'<,w1:'<,w2:'<,w3:'<)
|(s:WaitSL, ii:'>) -> (s:WaitSF, wo:'>,w1:'>,w2:'>,w3:'>)
|(s:WaitSL, ii:'<) -> (s:Kern, wo:'<,w1:'<,w2:'<,w3:'<)
|(s:Kern, ii:'x) -> (s:Kern, wo:'wi[0][(i*m)+j],w1:'wi[1][(i*m)+j],w2:'wi[2][(i*m)+j],w3:'wi
    [3][(i*m)+j],j: if j<(m-1) then j+1 else 0)
|(s:Kern, ii:'>) -> (s:WaitSL, wo:'>,w1:'>,w2:'>,w3:'>,i: if i<(n-1) then i+1 else 0);
```

Listing C.8: Listing CAPH de l'acteur de distribution de poids

```
actor distr4(wi:signed<16> array[4][16], n:unsigned<8>,m:unsigned<8>)
 in (ii:signed<16> dc)
 out(wo:signed<16> dc,w1:signed<16> dc,w2:signed<16> dc,w3:signed<16> dc)
 var s : {WaitSF, WaitSL, Kern} = WaitSF
 var i : unsigned<8>
 var j : unsigned<8>
 var k : unsigned<8>
rules
|(s:WaitSF, ii:'<) -> (s:WaitSL, wo:'<,w1:'<,w2:'<,w3:'<)
|(s:WaitSL, ii:'>) -> (s:WaitSF, wo:'>,w1:'>,w2:'>,w3:'>)
|(s:WaitSL, ii:'<) -> (s:Kern, wo:'<,w1:'<,w2:'<,w3:'<)
|(s:Kern, ii:'x) -> (s:Kern, wo:'wi[0][(i*m)+j],w1:'wi[1][(i*m)+j],w2:'wi[2][(i*m)+j],w3:'wi
    [3][(i*m)+j],j: if j<(m-1) then j+1 else 0)
|(s:Kern, ii:'>) -> (s:WaitSL, wo:'>,w1:'>,w2:'>,w3:'>,i: if i<(n-1) then i+1 else 0);
```

C.6 Acteurs pour les couches de classification connectée

Listing C.9: Listing CAPH des acteurs pour le calcul de la distance euclidienne

```
 actor sub (w:int<s,m>)
 in (i1: int<s,m> dc)
 out (o: int<s,m> dc)
rules
| i1:'< -> o:'<
| i1:'> -> o:'>
| i1:'a -> o:'(a-w);

actor square
 in (i1: int<s,m> dc)
 out (o: int<s,m> dc)
rules
| i1:'< -> o:'<
| i1:'> -> o:'>
| i1:'a -> o:'(a*a);

actor sum4()
in (io:int<s,m> dc,i1:int<s,m> dc,i2:int<s,m> dc,i3:int<s,m> dc)
out(o:int<s,m> dc)
rules
|(io:'<,i1:'<,i2:'<,i3:'<) -> (o:'<)
|(io:'>,i1:'>,i2:'>,i3:'>) -> (o:'>)
|(io:'x0,i1:'x1,i2:'x2,i3:'x3)-> (o:'(x0+x1+x2+x3))
;
```

D

Architecture matérielle des FIFOS

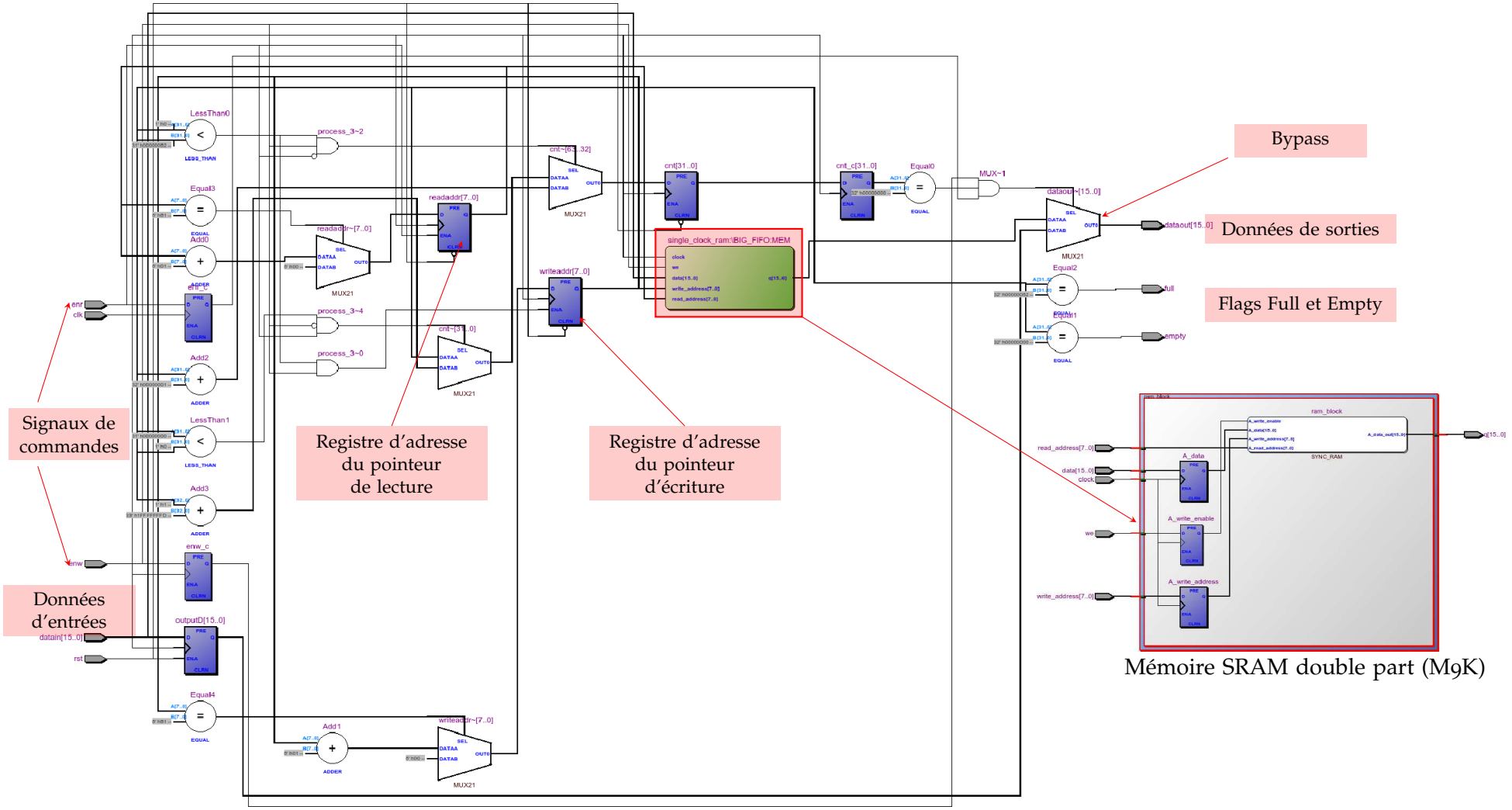


Fig. D.1: Exemple d'implémentation d'une FIFO circulaire instanciant de la mémoire SRAM double ports à partir du modèle de CAPH après le processus de synthèse sur un FPGA Cyclone III. On retrouve un bloc mémoire M9K avec les deux pointeurs associés, un système de bypass du bloc mémoire pour gérer le cas particulier de la lecture et de l'écriture simultanées sur la fifo lorsqu'elle est vide.

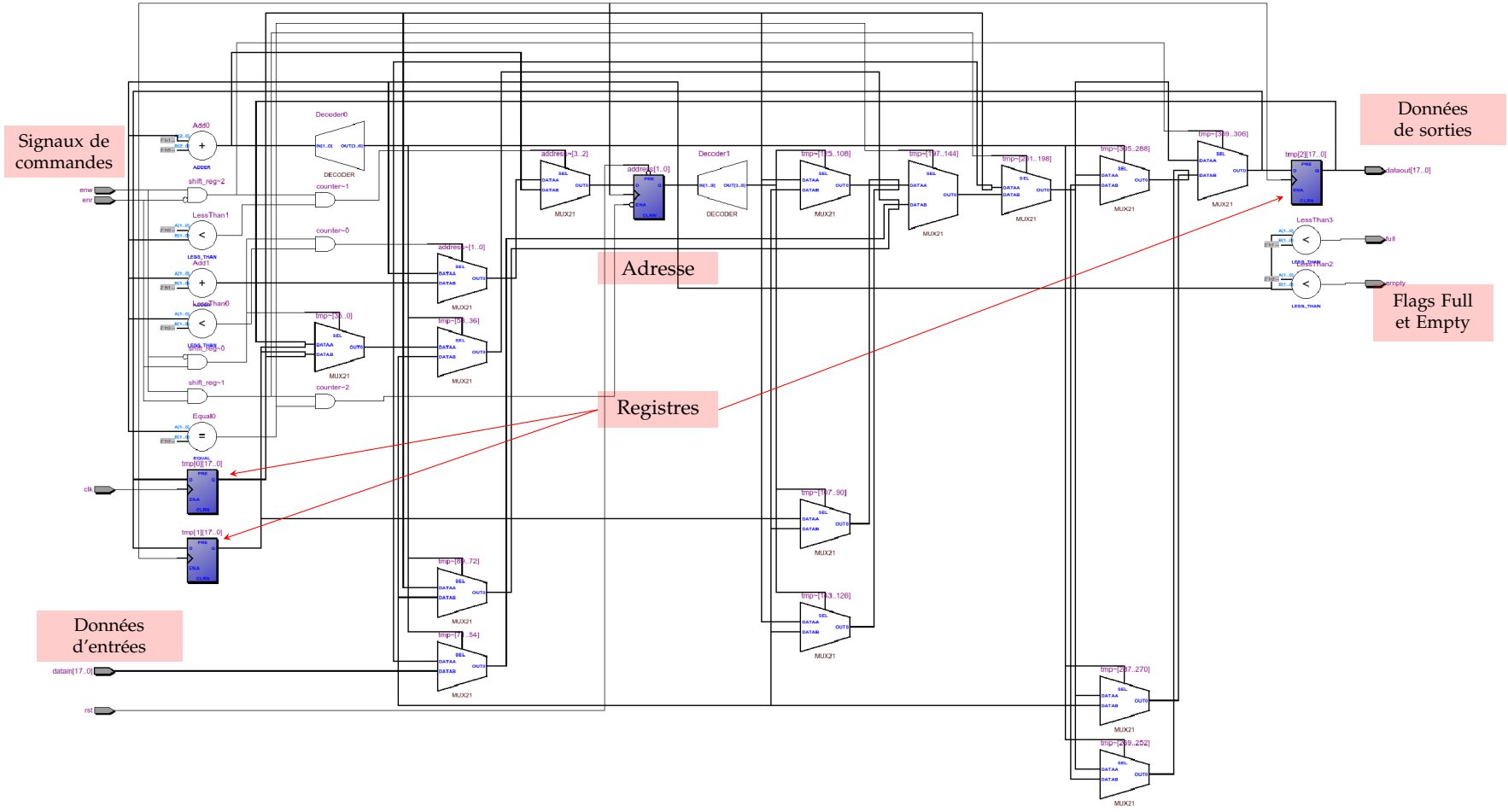


Fig. D.2: FIFO Registres trois places après le processus de synthèse sur un FPGA Cyclone III. Ce modèle de FIFO, basé sur un ensemble de registres à décalages, est utilisé pour les FIFOs peu profondes. On retrouve donc les trois registres, chacun composé de 18 bascules sur cet exemple, plus les signaux de contrôles (*full* et *empty*) et de commandes (*enw enr*) afin d'être conforme à l'interface standard des FIFOs.

E

Représentation intermédiaire d'un acteur CAPH

Listing E.1: Listing CAPH de l'acteur `hsum`

```
actor hsum (k:unsigned<8>)
  in  (a:unsigned<s> dc)
  out (c:unsigned<s> dc)

var s : {WaitSoF, WaitSoL, Sum} = WaitSoF
var x_sum  : unsigned<s>
var j      : unsigned<8>

rules
| (s:WaitSoF, a:'<)              -> (s:WaitSoL, c:'<)                    --(1)
| (s:WaitSoL, a:'<)              -> (s:Sum,     c:'<, xsum:0, j:0)       --(2)
| (s:WaitSoL, a:'>)              -> (s:WaitSoF, c:'>)                    --(3)
| (s:Sum,     a:'x) when j<(k-1) -> (s:Sum,     xsum: xsum+x, j:j+1)     --(4)
| (s:Sum,     a:'x) when j=(k-1) -> (s:Sum,     c:'(xsum+x), xsum:0, j:0) --(5)
| (s:Sum,     a:'>)              -> (s:WaitSoL, c:'>);                   --(6)
```

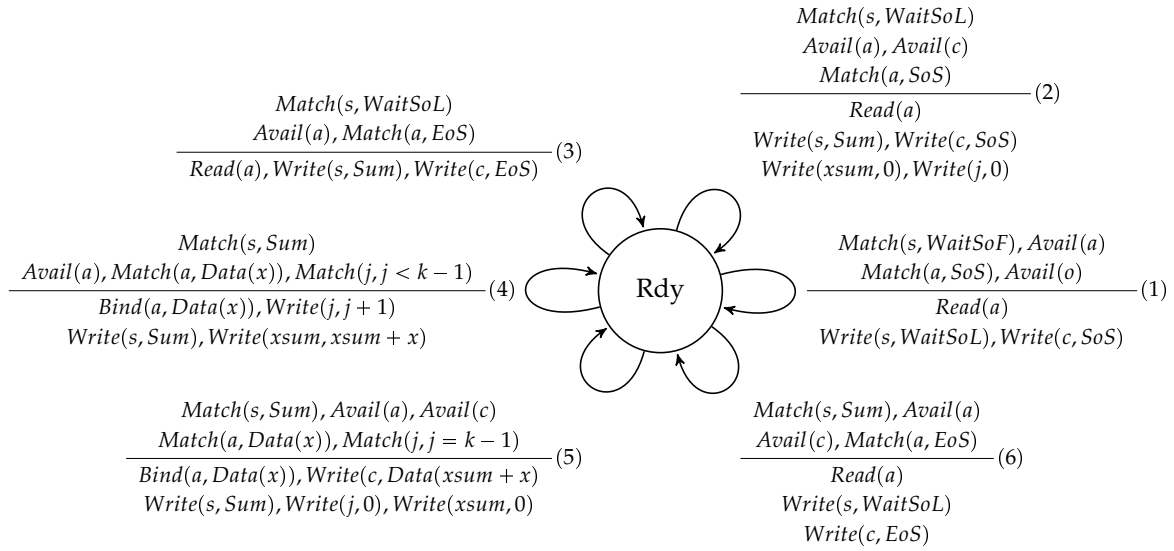


Fig. E.1: Représentation intermédiaire de l'acteur `hsum`. Les numéros entre parenthèses correspondent à ceux des règles d'activation du listing E.1

.