Les effets de l'environnement sur le développement et l'organisation d'architectures de traitement matériel auto-organisées.

Laurent Fiack

UNIVERSITÉ DE CERGY-PONTOISE
ÉCOLE DOCTORALE SCIENCES ET INGÉNIERIE

THÈSE

pour obtenir le titre de

Docteur en Sciences

Spécialité Sciences et Technologies
de l'Information et de la Communication

par

Laurent Fiack

Les effets de l'environnement sur le développement et l'organisation d'architectures de traitement matériel auto-organisées

Thèse dirigée par Benoît Miramond

préparée à ETIS dans l'équipe ASTRE

Soutenue le 2 Décembre 2015

Devant la Commission d'Examen

Jury

Rapporteur	:	B. Granado	-	LiP6
Rapporteur	:	M. Paindavoine	-	LEAD
Examinateur	:	S. Viollet	-	ISM
Examinateur	:	N. Cuperlier	-	ETIS
Examinateur	:	A. Upegui	-	InIt
Directeur	:	B. Miramond	-	ETIS

Remerciements

Je voudrais tout d'abord remercier MM. Bertrand Granado, Michel Paindavoine, Stéphane Viollet, Nicolas Cuperlier et Andres Upegui d'avoir accepté de me faire l'honneur de participer à mon jury de thèse. Leurs remarques pertinentes et leurs questions très intéressantes lors de la soutenance ont permis de valoriser les travaux présentés dans ce manuscrit. Je tiens également à exprimer de vifs et chaleureux remerciements envers Benoît Miramond, mon directeur de thèse, pour son aide, son soutien, sa disponibilité et la confiance qu'il m'a accordé.

Je remercie Inbar Fijalkow, directrice d'ETIS lors de mon arrivée et Mathias Quoy, actuel directeur, pour leur accueil au sein du laboratoire et pour leur soutien dans mes démarches administratives. Je remercie également Olivier Romain directeur de l'équipe ASTRE pour son soutien.

Je remercie la Communauté d'Agglomération de Cergy-Pontoise pour avoir financé cette thèse, sans quoi ces travaux n'auraient pas été possibles.

Un grand merci à Annick, Anthony, Astrid, Nelly, Sokhena et à toutes les personnes des services administratifs d'ETIS, de l'ENSEA et de l'UCP pour leur aide dans toutes les démarches administratives, pour leur efficacité permanente et leur disponibilité.

Je tiens tout particulièrement à remercier l'ensemble des membres du laboratoire ETIS et plus particulièrement les doctorants sans lesquels la vie au laboratoire aurait été différente. Commençons par Alexis avec qui j'ai partagé mon bureau, rapidement remplacé par Agathe. Continuons avec Laurent R. pour les longues conversations tant scientifiques que vidéo-ludiques. S'ajoute également Jérôme pour les discussions scientifiques, sociétales et les plans pour conquérir le monde. J'ai également une pensée très amicale pour Lounis, David P.Q., Laurent G., Amel, Ahcine, Marwen, Adrien, Yuhui et Liang qui ont contribué à rendre ces années inoubliables.

Un grand merci également à David, Olivier, Vero, Anne, Voisin et tout les autres avec qui j'ai pu partager les repas de midi, autours de discussions enflammés sur des sujets critiques.

Je remercie Anne et Voisin avec qui nous avons fondé l'association Nova-Robotics ainsi que Véro et Antoine qui nous ont rejoint par la suite.

J'exprime à présent mes plus vifs remerciements à ma famille qui m'a soutenu toutes ces années sans jamais douter de moi et qui a toujours été une source d'encouragement et de réconfort. Un grand merci à Cachou dont les ronronnements m'ont permis de faire face à l'adversité. Je tiens à remercier l'ensemble de mes amis, en particulier Lucas, Louis et Romain, avec qui je pouvais de temps en temps oublier ma condition de thésard. Pour tout cela, je vous dis un grand merci.

Table des matières

Table des figures

Liste des tableaux

Acronymes et notations

AER	Address Event Representation ... 75
ALU	Arithemetic/Logic Unit ... 88
AMP	Asymmetric Multi-Processing ... 20
API	Application Programming Interface ... 116
CABA	Cycle-Accurate Bit-Accurate ... 17
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit ... 17
DDR3	Double Data Rate ... 16
DMA	Direct Memory Access ... 16
DNF	Dynamic Neural Field ... 75
DoG	Difference Of Gaussians ... 42
DSP	Digital Signal Processor ... 46
FIFO	First-In First-Out ... 48
FPGA	Field Programmable Gate Array ... 19
GALS	Globally-Asynchronous Locally-Synchronous ... 113
GFLOPS	Giga FLoating Operation Per Second ... 25
GPGPU	General-Purpose Graphical Processing Unit ... 17
GPIO	General-Purpose Input/Output ... 113
GPU	Graphical Processing Unit ... 19
HDL	Hardware Design Language ... 19
HLS	High Level Synthesis ... 19
IP	Intellectual Property
ITRS	International Technology Roadmap for Semiconductors
JTAG	Joint Test Action Group ... 108
LUT	Look-Up Table ... 56
LVDS	Low-Voltage Differential Signaling ... 109
MAC	Medium Access Control ... 16
MACC	Multiplier-Accumulator ... 83
MIMD	Multiple Instruction Multiple Data ... 105
MISD	Multiple Instruction Single Data ... 105
MMU	Memory Management Unit ... 16
NoC	Network on Chip ... 15
NUMA	Non-Uniform Memory Access ... 17
PE	Processing Element ... 76
RAM	Random Access Memory ... 88
RISC	Reduced Instruction Set Computer ... 17
SIFT	Scale-Invariant Feature Transform ... 44
SIMD	Single Instruction Multiple Data ... 105
SISD	Single Instruction Single Data ... 105

SLAM Simultaneous Localization And Mapping 45
SMP Symmetric Multi-Processing 16
SoC System-On-Chip 19
SOM Self-Organizing Map 75
SRAM Static Random Memory Access 108
SURF Speeded Up Robust Features 45
UART Universal Asynchronous Receiver/Transmitter 118
VLIW Very Long Instruction Word 16
WCET Worst Case Execution Time 31

Résumé

Depuis l'invention du premier micro-processeur en 1971, la densité des transistors a doublé tous les deux ans. Cette augmentation exponentielle de la densité d'intégration a permis la complexification des architectures et a mené à une augmentation constante de la puissance de calcul. Jusqu'en 2004, cette amélioration de la puissance de calcul s'est traduite par l'augmentation de la fréquence de fonctionnement des processeurs. Depuis, cette course à la fréquence a été remplacée par la démocratisation du parallélisme. En conséquence de cette augmentation exponentielle, conjecturée par Gordon Moore dans la *Loi de Moore*, des micro-processeurs à plus de 5 milliards de transistors sont disponibles dans le commerce en 2015.

La diminution de la dimension des transistors a également permis de concevoir des circuits de moins en moins gourmands énergétiquement. Cela a permis d'apporter de plus en plus de puissance de calcul dans les systèmes embarqués. Cela a également bénéficié au domaine de l'intelligence artificielle, et des réseaux de neurones qui demandent beaucoup de puissance de calcul, ainsi qu'à la robotique mobile.

Le nombre de processeurs intégrables au sein d'une même puce augmente lui aussi. On a ainsi vu apparaître il y a quelques années de nombreux travaux sur les architectures *many-cœur*, dont l'ambition est d'intégrer plusieurs centaines de processeurs sur un seul circuit.

Malgré les efforts déployés à la fois sur les axes technologiques et architecturaux, de nombreuses difficultés et limitations demeurent. D'une part, l'arrivée du parallélisme de masse boulverse le mode de pensée des développeurs de logiciels, formés depuis l'apparition de l'informatique à décomposer les problèmes pour les traiter de manière séquentielle. D'autre part, la résolution de tâches cognitives, basées sur la vision ou l'audition par exemple, reste difficile à appréhender. La tendance actuelle pour permettre à ces tâches cognitive de s'exécuter sur des dispositifs embarqués consiste à déporter les calculs les plus lourds sur des serveurs distants.

Le domaine de la robotique mobile a en particulier de plus en plus besoin de faire appel à des tâches cognitives, telles la reconnaissance de lieu, de visage ou d'objets, l'interaction avec leur environnement ou avec des êtres humains par exemple. Les robots ont également besoin de faire preuve d'adaptation et d'autonomie puisqu'ils sont placés dans un environnement inconnu, non prédictible et dynamique. Ils doivent construire une représentation de cet environnement, puisque leur perception à un instant donné est limitée. Si le comportement du robot doit s'adapter en fonction de l'environnement, le calculateur responsable de la prise de décision doit s'adapter lui aussi.

L'objet de cette thèse est d'analyser les moyens à mettre en place pour qu'un tel calculateur puisse s'adapter, à la fois aux spécificités du robot, à sa morphologie, et également aux évolutions, à la dynamique de l'environnement. Pour cela, nous identifions plusieurs étapes.

Dans un premier temps, le contrôleur doit capter les informations contenues dans l'environnement. Le robot est muni de différents capteurs, permettant d'acquérir des informations parmi plusieurs modalités. Parmi ces différents flux d'information brute, le contrôleur doit extraire l'information pertinente. Il construit ainsi une carte de saillance de son environnement.

L'étape suivante consiste à organiser l'architecture interne du contrôleur en fonction de la richesse des différentes modalités. C'est cette étape qui permet au contrôleur de s'adapter, par le moyen de la plasticité matérielle, inspirée de la plasticité cérébrale.

Enfin, le contrôleur doit être programmable. Il doit disposer de puissance de calcul accessible par un utilisateur.

Introduction

Les moyens déployés ces dernières décénies par l'industrie et le monde de la recherche dans le domaine des semi-conducteurs ont mené, en 2015, à l'intégration de plusieurs milliards de transistors au sein d'une même puce.

L'évolution de cette intégration a permis, dans un premier temps, d'augmenter la fréquence de fonctionnement des circuits, notamment grâce à la mise en place de pipelines de plus en plus larges dans les processeurs. Cette course à la fréquence s'est arrêtée en 2004, à cause d'une trop grande dissipation thermique d'une part, et de performances médiocres, malgré un débit théorique élevé d'autre part.

Depuis l'arrêt de l'augmentation de leur fréquence de fonctionnement, la performance des processeurs n'a pourtant pas cessé d'augmenter. Cette fois l'amélioration s'est faite grâce au parallélisme. Les constructeurs ont ainsi intégré deux processeurs par puce, puis rapidement quatre, puis huit.

L'augmentation de l'intégration dépends de procédés technologiques qui se heurtent aux limites physiques des matériaux. Elle devrait toutefois se poursuivre dans les années à venir, l'ITRS prévoyant en effet l'intégration de 24 milliards de transistors sur une même puce à l'horizon 2024 [ITRS, 2013].

Du multi-cœur à l'émergence des many-cœurs

Pour exploiter cette quantité toujours grandissante de transistors, l'architecture des processeurs se complexifie. De plus en plus de logique est attribuée à l'optimisation du pipeline, avec l'exécution d'instructions dans le désordre et l'intégration d'algorithmes de prédiction de plus en plus complexes. Les architectures vectorielles permettent également d'augmenter la performance en exécutant des instructions sur plusieurs données à la fois.

La performance des processeurs augmente plus rapidement que la performance de la mémoire [Patterson et al, 1997]. De ce fait, le goulot d'étranglement créé par les accès mémoire devient de plus en plus important, et s'accroit d'autant plus que le nombre de cœurs de calcul augmente. Ainsi une grande partie de la surface des puces est occupée par une hierarchie de caches, et un effort particulier est mis en place pour occuper ces caches au mieux.

Outre l'augmentation de la complexité des cœurs de calcul, leur nombre s'accroit également. On a ainsi vu apparaître depuis quelques années plusieurs travaux sur les architectures *many-cœur*, dont l'ambition est d'intégrer plusieurs centaines voire milliers de processeurs sur un seul circuit. Le passage d'une dizaine de cœurs de calcul à plusieurs centaines pose de nombreuses difficultés. Il n'est pas possible de simplement démultiplier les architectures existantes pour multiplier les performances.

Tout d'abord les cœurs de calcul ne peuvent plus communiquer par le biais d'un bus, partagé ou hierarchique, pour des raisons de scalabilité [Guerrier and Greiner, 2000]. La bande passante est partagée entre les différents éléments, et la fréquence de fonctionnement du bus diminue car la charge capacitive augmente. Pour passer à l'échelle, les cœurs de calcul doivent être connectés par un Network on Chip (NoC).

Ensuite le modèle à mémoire partagé utilisé dans les architectures multi-processeurs classiques pose également des problèmes lors du passage à l'échelle. Dans un tel modèle, la

bande passante est une fois de plus partagée entre les différents cœurs de calcul. L'échange de données s'effectue par des moyens de synchronisation comme les sémaphores ou les boîtes aux lettres dont la gestion se complexifie également avec l'augmentation du nombre d'éléments. La cohérence entre les caches des cœurs de calcul doit être maintenue, ce qui augmente encore la consommation en bande passante. De plus, les performances de la gestion des caches est fortement impactée par la latence du réseau [Borkar, 2007].

L'alternative est le modèle à mémoire distribué et consiste à répartir la mémoire parmi les différents éléments. Chaque cœur à accès à sa propre mémoire privée et ne peut pas accéder au contenu de la mémoire d'un autre cœur. Les échanges de données s'effectuent à l'aide d'un protocole de passage de messages [Marchesan Almeida et al, 2009]. Ce modèle semble plus efficace pour les grands réseaux, comme ceux des many-cœurs, car seuls les protocoles de communication sont partagés par les cœurs.

Nous allons maintenant présenter plusieurs architectures many-cœurs, industrielles ou académiques, illustrant différentes approches.

Le MPPA-256 de Kalray

Le many-cœur MPPA-256 de Kalray [de Dinechin et al, 2013] dispose de 256 cœurs de calcul organisés en 16 clusters de 16 cœurs, et de 32 cœurs réservés au système, un par cluster et 16 répartis sur les 4 systèmes d'entrées/sorties. Ces 4 systèmes d'entrées/sorties disposent chacun de 4 cœurs Symmetric Multi-Processing (SMP) pouvant accéder à un port PCI-Express, à un Medium Access Control (MAC) Ethernet et à divers périphériques. Ils peuvent également accéder à 64 GiB de mémoire Double Data Rate (DDR3). Ils peuvent exécuter un système d'exploitation riche tel Linux ou un système d'exploitation temps-réel dédié.

Les 16 clusters sont connectés par deux NoC en grille 2D torique, un pour les données et un pour le contrôle. Le NoC est basé sur la commutation de paquets, avec un algorithme de routage de type *wormhole*. Chaque cluster possède 16 cœurs de calcul, un cœur de contrôle et une mémoire partagée de 2MiB, organisée en 16 banques accessibles en parallèle. Un Direct Memory Access (DMA) est également présent pour assurer les communications avec les NoC ainsi qu'au sein de la mémoire partagée.

Les cœurs de calcul ou de contrôle implémentent une architecture Very Long Instruction Word (VLIW) à 5 voies disposant de 2 unités arithmétique et logique. Ils ont également un cache de données et d'instructions de 8 kiB chacun et d'une Memory Management Unit (MMU) pour isoler les processus et fournir un mécanisme de virtualisation de la mémoire.

Les trois niveaux d'architecture, le système complet, un cluster et un cœur sont représentés en FIGURE 1.

La mémoire suit ainsi un modèle hybride [Diaz et al, 2012], localement partagée et globalement distribuée. Au sein d'un cluster, les cœurs de calcul peuvent communiquer entre eux par le moyen de variables partagées dans la mémoire. Entre les clusters, les informations s'échangent par passage de message grâce au NoC de données.

Le many-cœur peut être programmé de deux manières, soit en flot de données [Bilsen et al, 1996], soit de manière plus classique selon la norme POSIX. Le langage permettant de représenter les applications en flot de données est basé sur le C. Le compilateur se charge de distribuer les tâches sur les différentes mémoires et les différents cœurs. Un encodeur H.264 est implémenté sur le MPPA-256 en guise d'exemple et est comparé à une implémentation sur un processeur Core i7-3820 d'Intel. Les deux implémentations donnent des résultats semblables d'environ 50 images par secondes, mais on observe une consommation 20 fois moins élevée

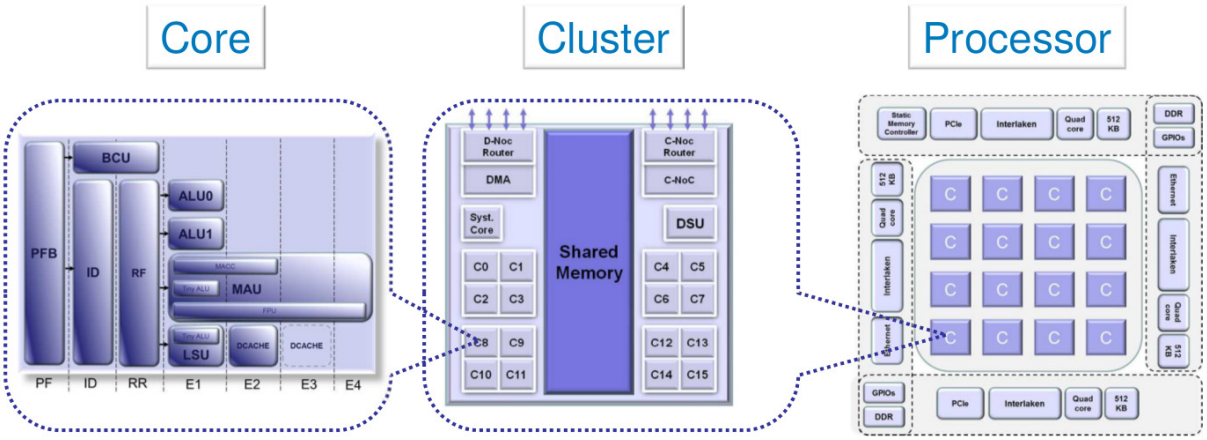


FIGURE 1 – **Architecture du MPPA-256.** À gauche, une représentation du pipeline d'un cœur, au centre l'organisation d'un cluster et à droite l'architecture du many-cœur complet.

avec l'implémentation sur le many-cœur.

Programmé en POSIX, les processeurs d'entrées/sorties déploient les processus sur les clusters. À l'intérieur d'un cluster, l'application est séparée en threads qui s'exécutent sur les cœurs selon un modèle classique à mémoire partagée. La communication inter-processus qui s'effectue par passage de message est suportée par le NoC. Une application de calcul financier, basée sur la méthode de Monté-Carlo illustre ce mode de programmation. Le MPPA-256 est ainsi comparé à un processeur Core i7-3820 d'Intel et un General-Purpose Graphical Processing Unit (GPGPU) Tesla C2075 d'NVIDIA. Les performances sur le many-cœur se situent à mi-chemin entre le GPGPU et le processeur. La consommation sur many-cœur est une fois encore 20 fois moins élevée par rapport à celle du Central Processing Unit (CPU) et 6 fois moins élevée comparée au GPGPU.

L'architecture TSAR

Le many-cœur TSAR [Greiner, 2009], développé au LIP6, est prototypé en langage SystemC et modélisé au niveau Cycle-Accurate Bit-Accurate (CABA). Il permet théoriquement d'intégrer jusqu'à 4096 cœurs de calcul, mais le prototype n'en contiendra que 128. De part sa nature simulée, de nombreuses caractéristiques de l'architecture sont paramétrables. Les valeurs énoncées ci-dessous font partie d'un cas d'utilisation particulier.

L'architecture TSAR est un ensemble de clusters interconnectés en grille 2D par le NoC DSPIN (Distributed, Scalable, Predictable Integrated Network) [Panades et al, 2006]. Chaque cluster est constitué de 4 cœurs 32 bits Reduced Instruction Set Computer (RISC) MIPS32, d'un interconnect local, d'un cache de 256 kiB, d'un accès au NoC et d'un DMA. Chaque cœur de calcul dispose de caches d'instructions et de données de 16 kiB chacun et d'une MMU. L'un des clusters, défini arbitrairement, est dédié à la gestion des entrées/sorties et peut être connecté à un terminal texte, une sortie vidéo ou encore à un disque dur externe. L'architecture TSAR est représentée en FIGURE 2.

Le many-cœur a été conçu pour accueillir des systèmes d'exploitations standards comme Linux [Almaless, 2014]. Chaque cluster contient une banque de mémoire sous la forme de cache, la mémoire est donc distribuée physiquement. Chaque processeur peut accéder à toutes les banques, la mémoire est donc partagée logiquement. Le temps d'accès dépend cependant de la distance à la banque mémoire concernée, TSAR est ainsi une architecture Non-Uniform Me-

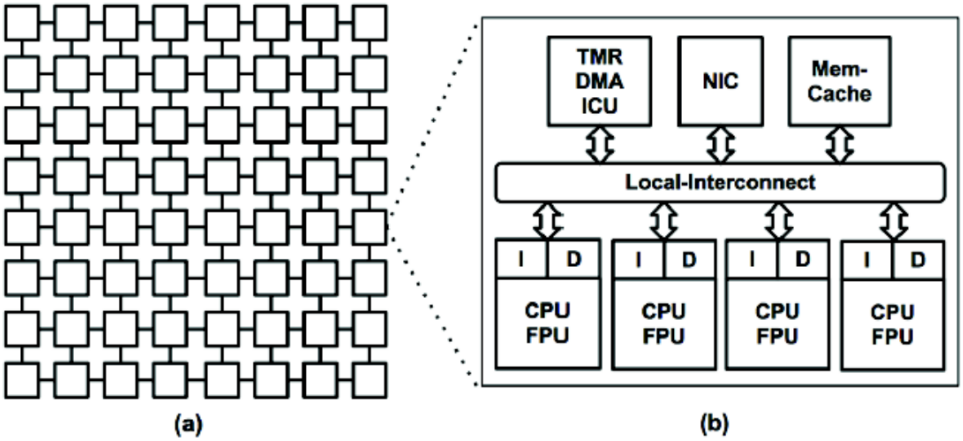


FIGURE 2 – **Architecture TSAR.** Une grille 2D de cluster de 4 cœurs permet d'intégrer jusqu'à 4096 cœurs de calcul.

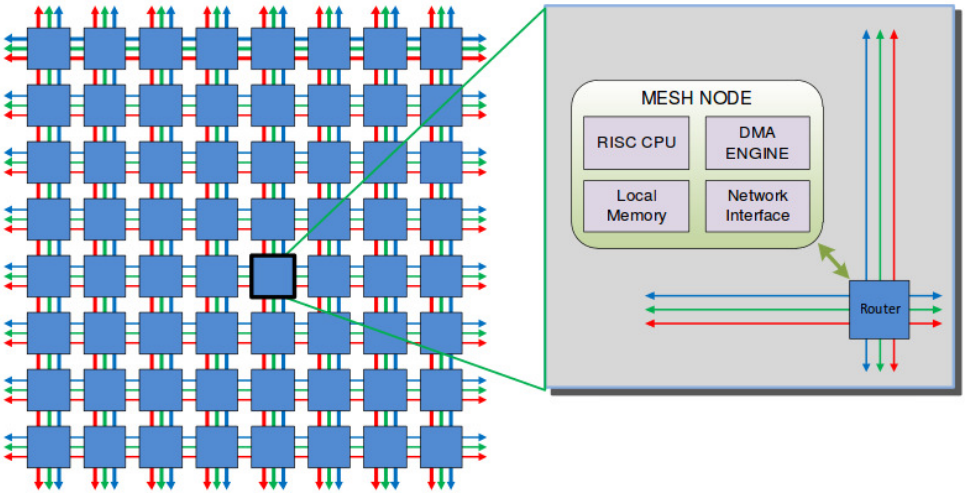


FIGURE 3 – **Architecture du many-cœur Epiphany**

mory Access (NUMA).

Le co-processeur Epiphany-IV d'Adapteva

Le many-cœur Epiphany-IV d'Adapteva [Olofsson et al, 2011] contient 64 cœurs de calcul, interconnectés par un NoC organisé en grille 2D. Ces cœurs, cadencés à 1GHz, sont composés de processeurs 32 bits RISC superscalaire, ils disposent d'une mémoire locale de 32 kiB, d'un accès au réseau et d'un DMA. L'architecture du many-cœur est représentée en FIGURE 3.

La société Adapteva annonce être en mesure de générer de manière automatique, en 24 heures, un circuit comprennant 1024 cœurs de calcul, grâce à la régularité de l'architecture et d'atteindre ainsi 70 GFLOPS. Comme la version à 64 cœurs occupe 2 mm^2 et consomme 2 W, une version à 1024 cœurs semble possible si l'on compare ces grandeurs à celles des processeurs actuels.

Cependant, le fait de multiplier simplement les cœurs de calcul en conservant l'architecture actuelle risque de ne pas répondre aux attentes de la firme. L'architecture n'a pas de systèmes

de caches, à la place les cœurs disposent d'une mémoire locale de taille réduite. Le stockage des données dans ces mémoires est alors laissé à la discretion du développeur de l'application, et doit donc être gérée de manière logicielle.

Les architectures reconfigurables

Grâce à l'augmentation de l'intégration, une autre technologie a émergé il y a quelques années, il s'agit des architectures reconfigurables. Ces architectures permettent d'émuler des circuits numériques, et donc de modifier ces circuits après leur déploiement.

La reconfiguration dynamique partielle

Plus récemment le mécanisme de reconfiguration dynamique partielle est apparu [Dye, 2010; Bourgeault, 2011]. Il permet de modifier une partie de l'architecture d'un circuit pendant l'exécution, sans affecter le fonctionnement du reste du circuit. Les flots de reconfiguration dynamique partielle, proposés par Xilinx et Altera souffrent de plusieurs limitations pour être efficacement utilisable. Tout d'abord, les langages de description matérielle historique dans le domaine des architectures reconfigurables, le VHDL et le Verilog, sont par nature statiques et ne permettent pas nativement de gérer la reconfiguration dynamique. Cette reconfiguration est ainsi gérée par des outils propriétaires.

Ces outils ne sont pas adaptés à des architectures massivement parallèles, ni à des applications fortement dynamiques. Les modèles proposés par ces flots de conception sont statiques et centralisés autour d'un soft-processeur. De plus, il n'y a qu'un port de reconfiguration, ne permettant pas la reconfiguration de plusieurs modules simultanéments.

Les Field Programmable Gate Array (FPGA)-System-On-Chip (SoC)

Une autre catégorie de circuits proposent un couplage étroit entre un processeur embarqué récent et une matrice FPGA. Ces FPGA-SoC sont disponibles chez les deux principaux constructeurs de FPGA, Xilinx [Xilinx, 2015] et Altera [Altera, 2013]. Les versions des deux constructeurs sont assez similaires. Les IP déployées sur la matrice FPGA peuvent être accédées *via* des registres *mappés* en mémoire, ou à travers un DMA. L'architecture d'un FPGA-SoC Zynq de Xilinx est montré en FIGURE 4. Les solutions proposées par Altera sont similaires. En dehors des possibilités de debug par une sonde JTAG, la matrice FPGA est programmée exclusivement à travers le processeur, elle peut en outre être reconfigurée dynamiquement, et partiellement, par le processeur maître. On observe les mêmes limites que précédemment en terme de dynamicité et de parallélisme massif.

Altera propose de décrire les architectures en OpenCL [Stone et al, 2010], langage conçu pour programmer des systèmes parallèles hétérogènes. Une application OpenCL a besoin d'un processeur hôte jouant le rôle de contrôleur, et d'autres périphériques, comme d'autres processeurs, des Graphical Processing Unit (GPU) ou ici un FPGA. Xilinx propose, grâce à sa suite Vivado High Level Synthesis (HLS), de synthétiser une architecture directement à partir du langage C [Santarini, 2012]. Dans les deux cas, le concepteur peut gagner entre 5 et 10 fois plus de temps par rapport à une conception avec les outils Hardware Design Language (HDL) traditionnels [Economakos et al, 2013]. En dépit des effors déployés par les deux firmes, la partie FPGA de ces circuits reste compliquée à appréhender pour les informaticiens. Le concepteur doit avoir de bonnes connaissances en conception matérielle. Un code C ou OpenCL donnant

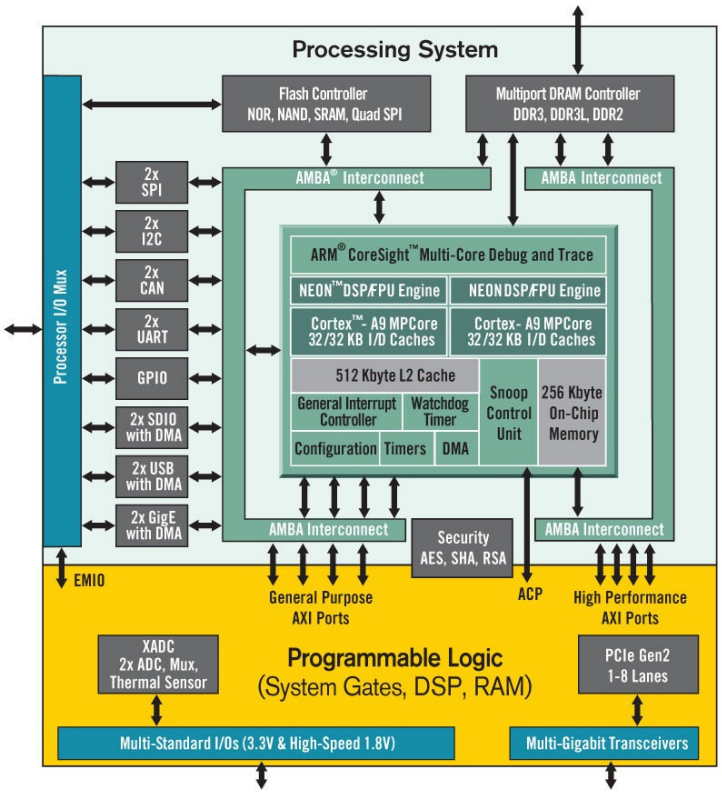


FIGURE 4 – **Architecture interne d'un FPGA-SoC Zynq de Xilinx.** Une zone de logique reconfigurable est accessible par de nombreux moyens depuis le processeur dual-cœur ARM.

de bons résultats après synthèse ressemblera plus à une description matérielle de haut niveau comme en SystemC qu'à un algorithme décrit dans un langage de programmation classique.

Les plateformes exploitant la reconfiguration dynamique partielle

Plusieurs plateformes ont été développées pour permettre de tirer parti de la reconfiguration dynamique partielle, de manière plus efficace que les solutions industrielles.

Parmi ces plateformes, nous parlerons d'abord du projet FOSFOR (Flexible Operating System FOr Reconfigurable platform).

Il s'agit d'une plateforme hétérogène, composée de plusieurs processeurs et de zones reconfigurables, permettant de gérer l'exécution de tâches logicielles et matérielles. La plateforme est prototypée sur FPGA, et se base sur les moyens en place pour la reconfiguration dynamique. Les soft-processeurs, ici des Leon3, exécutent le système d'exploitation Asymmetric Multi-Processing (AMP) RTEMS. Sur cette plateforme, la communication entre les processeurs se fait au travers d'une mémoire partagée. L'architecture de la plateforme FOSFOR est représentée en FIGURE 5.

Dans ces travaux, le système d'exploitation RTEMS a été porté en matériel, afin que les tâche matérielles accèdent aux mêmes services que les tâches logicielles [Gantel, 2014]. Les tâches matérielles communiquent par le biais d'un réseau sur puce. Ainsi, une tâche matérielle peut utiliser des services comme les sémaphores. De manière analogue à un changement de

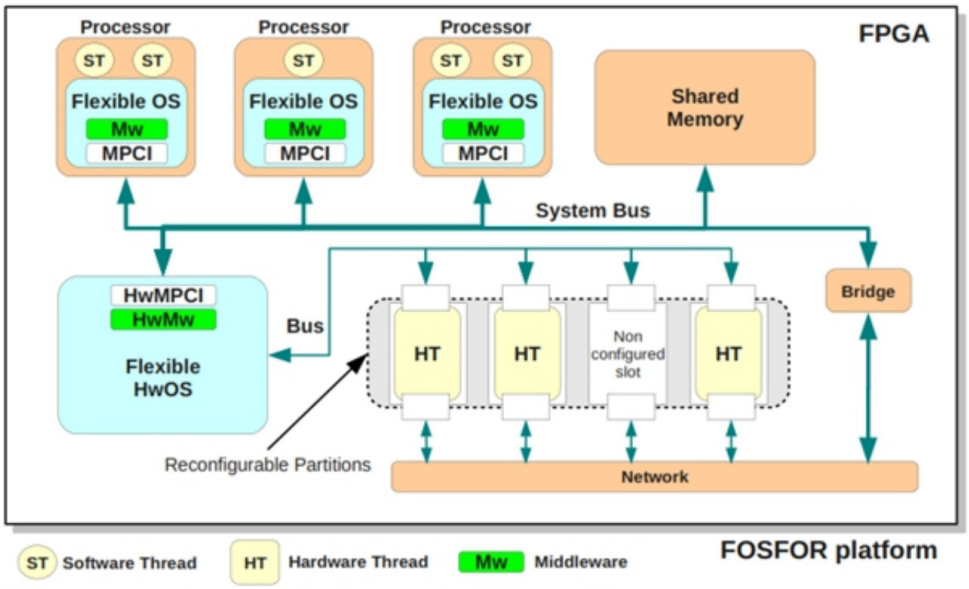


FIGURE 5 – **Architecture de la plateforme FOSFOR.**

contexte logiciel, la reconfiguration dynamique permet à une tâche matérielle d'être préemptée, et possiblement relogée dans une autre zone reconfigurable.

Cette plateforme met en évidence plusieurs limites. D'abord, l'usage d'une mémoire partagée limite fortement la scalabilité. La présence d'un seul port de reconfiguration limite également la scalabilité, puisqu'une seule zone peut être reconfigurée à la fois. Ensuite, les travaux menés sur cette plateforme sont fortement dépendants du FPGA ciblé, et ne peuvent pas aisément être portés sur un autre FPGA.

Cependant, la plateforme FOSFOR permet de bénéficier de la reconfiguration dynamique partielle pour augmenter virtuellement la surface d'un système sur puce.

Une autre plateforme a été créée dans le contexte du projet BORPH (Berkeley Operationg system for ReProgrammable Hardware). Cette plateforme propose une extension des commandes Unix aux architectures matérielles reconfigurables. La reconfiguration des tâches matérielles est déclenchée par un appel système depuis une application Unix. Les tâches matérielles et les processus logiciels communiquent à travers des appels d'entrées/sorties standards. La présence d'un système d'exploitation standard accélère le développement grâce à l'abstraction matérielle, et aux primitives déjà en place [So and Brodersen, 2008].

Les tâches matérielles sont exclusivement connectées au processeur hôte, ce qui limite une fois encore la scalabilité de la plateforme. Les travaux effectués sur cette plateforme pourraient particulièrement bien s'adapter sur un FPGA-SoC, en fournissant des mécanismes pour déployer de manière dynamique des accélérateurs matériels indépendants. Le manque d'intéractions entre les tâches matérielles ne permet pas de déployer une application parallèle complexe.

La plateforme MATIP (MPI Application Task Integration Platform) [GAMOM NGOUNOU EWO, 2015] est une plateforme parallèle reconfigurable implémentant la bibliothèque de passage de messages MPI. Elle est composée de trois couches, une couche d'interconnexions, une de communications, et enfin une couche d'application.

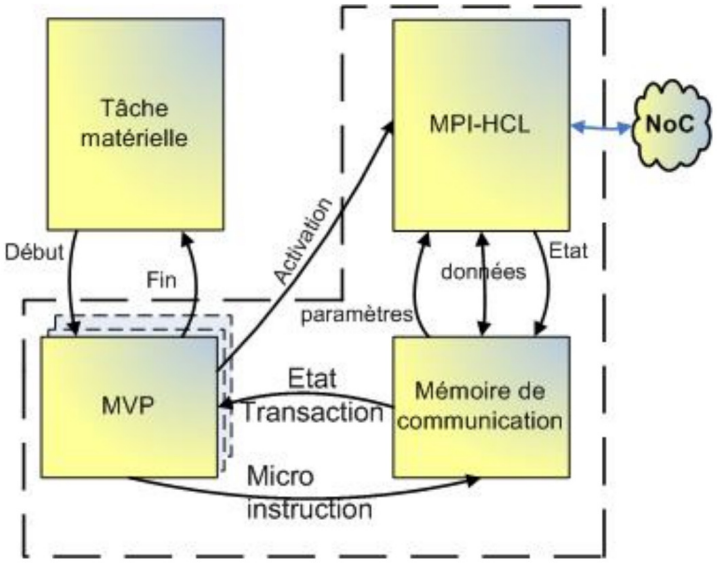


FIGURE 6 – **Architecture de la plateforme MATIP.** Vue centrée sur le module de communication.

La couche d'interconnexions implémente un réseau dynamique de type *crossbar*. Ce type de réseaux a une consommation en ressources proportionnelle au carré du nombre de nœuds, et est limité à 16 tâches dans l'état actuel. La séparation de l'architecture en couches permet cependant d'apporter des modifications au réseau d'interconnexions sans interférer sur les autres couches.

La couche de communications définit le mode des communications entre les tâches matérielles, supportées par la couche d'interconnexions. Elle implémente l'environnement de communication MPI sous la forme de processeurs spécialisés. Il y a un processeur de communications par tâche matérielle et par nœud d'interconnexion. L'implémentation matérielle de la bibliothèque MPI permet d'obtenir une latence plus basse que les solutions logicielles. Un processeur de communications est représenté en FIGURE 6.

La couche d'application contient les tâches matérielles définies par l'utilisateur de la plateforme. L'utilisateur dispose d'un *template* VHDL qui fournit une interface avec la couche de communications. Une tâche matérielle dispose d'une mémoire de communications pour invoquer les primitives MPI.

Parmi les primitives MPI, la primitive *spawn* permet d'invoquer un ou plusieurs processus. Cette primitive a également été implémentée et adaptée aux tâches matérielles, grâce à la reconfiguration dynamique [Ewo et al, 2014]. L'application peut ainsi créer dynamiquement des tâches en fonction des besoins.

En définitive, en étant totalement distribuée, la plateforme MATIP répond en partie à la problématique de la scalabilité. Le choix d'un réseau *crossbar* limite actuellement à 16 le nombre de tâches matérielles, mais peut être traité de manière indépendante du reste de la plateforme grâce à l'architecture en couches. Il est possible d'y déployer des applications dynamiques, grâce à l'implémentation de la primitive *spawn* et de son adaptation aux tâches matérielles.

L'architecture en elle même est portable, mais la reconfiguration dynamique dépends de solutions propriétaires et reste complexe à mettre en place et spécifique à un FPGA particulier. La plateforme MATIP ne propose pas de mécanismes de préemption ou d'ordonnancement lorsque le nombre de tâches matérielles dépasse le nombre d'emplacements prévus dans la

plateforme, à l'instar de la plateforme FOSFOR.

CHAPITRE 1

Architecture auto-organisée

Sommaire

Pour faire face aux applications toujours plus gourmandes, les architectures de calcul se complexifient, et font de plus en plus appel au parallélisme. Il est possible en 2015, technologiquement, d'intégrer 1000 processeurs sur une même puce, ce qui permet d'atteindre, en théorie, 70 Giga FLoating Operation Per Second (GFLOPS) pour une consommation de quelques dizaines de Watts. En pratique, l'intégration de plusieurs centaines de cœurs de calcul au sein d'une même puce soulève de nombreuses questions, à la fois au niveau des modèles de programmation et des contraintes architecturales.

Le modèle à mémoire partagée est à proscrire pour des raisons de scalabilité. Le modèle concurrent, à mémoire distribué est plus adapté, mais nécessite un réseau sur puce performant. Un modèle hybride, localement partagé et globalement distribué, semble être un bon compromis, en particulier pour une architecture constituée de clusters.

Les exemples mis en avant pour montrer l'efficacité des many-cœurs consistent en des déroulages de boucles, ou à des applications décrites en flot de données. Les many-cœurs implémentant des systèmes de contrôle et incluant la notion de processus nécessitent un ou plusieurs éléments centralisés, ce qui limite la scalabilité à long terme. Ils font dans ce cas appel à un système d'exploitation standard, lui même centralisé.

D'autre part, les architectures reconfigurables permettent de déployer des architectures dédiées, qui ont un meilleur rapport entre la puissance de calcul délivrée et la consommation énergétique que les architectures généralistes, tout en gardant une certaine flexibilité. Certaines architectures peuvent être reconfigurées dynamiquement et partiellement, mais leur utilisation reste anecdotique, car elles restent compliquées à mettre en place, malgré leur potentiel. Le portage d'un algorithme sur une architecture reconfigurable, dynamique ou non, demeure compliqué, malgré les outils de synthèse depuis des langages de plus haut niveau.

Les solutions actuelles pour déployer des applications parallèles et dynamiques sur les architectures reconfigurables dynamiquement et partiellement sont peu scalables. L'une des limite viens de la présence d'un unique port de reconfiguration interne. Les plateformes présentées ci-dessus disposent de *slots* reconfigurables, qui ne peuvent être reconfigurés que

tour à tour. Ces plateformes sont en outre difficiles à maintenir, car les outils utilisés pour gérer la reconfiguration dynamique sont verrouillés, et dépendent de technologies en constante évolution. Enfin, l'ordonnancement entre les tâches matérielles est géré une fois de plus de manière centralisé, car inspiré de solutions logicielles.

1.1 Des architectures auto-organisées

Pour passer à l'échelle et permettre le déploiement d'applications dynamiques et requérant une puissance de calcul importante, sous une consommation énergétique réduite, les circuits de calcul doivent être capables d'organiser leur architecture interne de manière distribuée, pendant l'exécution de l'application.

La question des architectures de traitement matériel auto-organisées est relativement récente, et de ce fait on trouve assez peu de travaux dans ce domaine dans la littérature. On retrouve néanmoins plusieurs projets qui abordent cette question, sous différents aspects.

Parmi les projets précurseurs dans ce domaine, on peut citer le projet POEtic [Tyrrell et al, 2003] qui s'appuie sur trois mécaniques d'évolutions inspirées de la nature :

— La **phylogénèse**, c'est à dire l'évolution des espèces sur plusieurs générations,
— l'**ontogénèse**, le développement d'un individu depuis les informations contenues dans son code génétique,
— l'**épigénèse**, le développement en fonction de l'environnement à plus court terme par un mécanisme d'apprentissage.

D'autres travaux ont suivi cette approche bio-inspirée en s'appuyant sur un ou plusieurs axes d'évolutions définis dans le modèle POE. Directement inspiré par les travaux menés dans le cadre de POEtic, le projet PERPLEXUS [Sanchez et al, 2007] étend les mécanismes de développement sur les trois axes à un système multi-agents. Chaque agent est muni d'un cœur de calcul auto-organisé nommé Ubichip, capable d'évoluer sur du long terme, de développer des connexions synaptiques et d'apprendre sur du plus court terme.

Plus récemment, le concept de calcul invasif à été introduit dans [Teich et al, 2011]. Le calcul invasif prend place dans une architecture many-cœur et permet à un programme ou à l'instance d'une architecture reconfigurable de prendre possession d'un élément de calcul pour paralléliser son traitement. Cette prise de possession se déroule en trois phases :

— L'**invasion** vise à réserver la ressource à utiliser,
— l'**infection** consiste à programmer cette ressource,
— la **retraite** permet, à la fin de l'exécution parallèle, de libérer la ressource.

Les auteurs de [Chillet et al, 2011] ont conçu un ordonnanceur, basé sur un réseau de neurones inspiré des réseaux de Hopfield. Ce réseau est capable d'ordonnancer des tâches dans un système multi-processeurs hétérogènes. Ces travaux n'adressent pas le problème de l'auto-organisation.

Enfin, le dernier projet que nous citerons dans cette section a consisté à concevoir un filtre de convolution adaptatif pour des applications de traitement d'images sur FPGA [Mora et al, 2013]. L'adaptation suit l'axe phylogénétique en étant pilotée par un algorithme génétique, et consiste à faire évoluer les calculs réalisés lors de la convolution. Pour se faire, ils sont réalisés

sur une zone reconfigurable dont l'architecture change en fonction du traitement demandé.

L'inspiration biologique est une voie intéressante lors du développement d'architectures parallèles auto-organisées. Le monde du vivant, allant des aspects microscopique, comme les interactions entre les cellules ou l'organisation neuronale, jusqu'aux aspects macroscopiques, comme l'évolution d'une espèce, regorge de systèmes dit *holistiques*, où l'ensemble d'un système est supérieur à la somme de ses parties. Nous cherchons à déterminer s'il est possible d'adapter ce phénomène d'émergence à un ordonnancement distribué sur un grand many-cœur.

1.1.1 Modèle POE

L'objectif du projet POEtic est de définir une architecture capable d'implémenter une grande variété de mécanismes bio-inspirés. Cette architecture se définit comme un :

"*substrat de calcul flexible, inspiré par les phases d'évolution, de développement et d'apprentissage des systèmes biologiques.*"

Le modèle POE, pour **Phylogénèse**, **Ontogénèse** et **Épigénèse**, implémente ces trois phases d'adaptation.

La phylogénèse se définit comme l'évolution basée sur les modifications des informations génétiques d'une espèce, selon trois mécanismes, la reproduction sélective, le croisement et la mutation. Transposée aux domaines de l'informatique et des architectures numériques, il en découle les algorithmes génétiques qui simulent ces trois mécanismes. Ils sont généralement capables de trouver des solutions à des problèmes qui ne peuvent être résolus par des approches classiques.

L'ontogénèse est responsable du développement d'un individu multi-cellulaire parmi l'espèce. Ce développement s'effectue en deux phases, d'abord la croissance, c'est à dire la multiplication des cellules, puis leur différenciation. Chaque cellule contient initialement l'ensemble du génome et se spécialise pour une fonction particulière, en fonction de son entourage, et de sa position dans l'organisme.

L'épigénèse est le mécanisme d'adaptation ayant le plus été exploité en informatique par le biais de l'apprentissage, en particulier des réseaux de neurones. Ici, l'adaptation se fait au cours de la vie de l'individu, en fonction de ses interactions avec son environnement, de manière supervisée ou non.

L'adaptabilité s'appuie sur un *substrat moléculaire*, constitué d'une surface de logique reconfigurable. L'architecture POEtic est constituée d'une grille à deux dimensions de cellules, et est organisée en trois couches :

— La couche du génotype implémente l'axe phylogénétique, soit l'évolution de l'espèce,
— la couche de configuration forme l'axe ontogénétique qui spécifie le développement d'un individu,
— la couche du phénotype constitue l'axe épigénétique en permettant l'implémentation d'algorithmes d'apprentissage.

L'architecture d'une cellule est représentée en FIGURE 1.1.

La couche du génotype contient un ensemble d'opérateurs qui définit toutes les fonctions des cellules, il peut s'agir par exemple des opérateurs implémentés par un modèle neuronal. Une table de différentiation détermine quel opérateur est utilisé par chaque cellule. Chaque

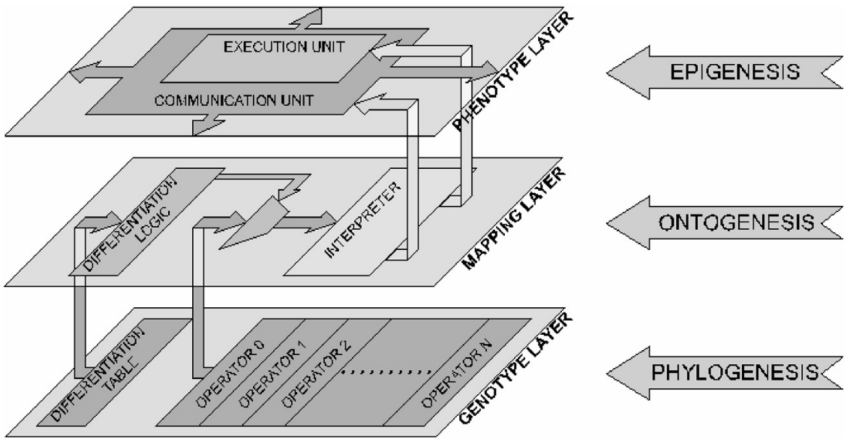


FIGURE 1.1 – **Architecture d'une cellule POEtic.**

cellule contient l'intégralité du génome, ce qui permet de définir la fonction d'une cellule après la fabrication et de la modifier en cours d'exécution.

La couche de configuration sélectionne l'opérateur à implémenter dans la cellule en identifiant la position de la cellule dans la grille.

Enfin, la couche du phénotype est constitué d'une unité d'exécution qui contient plusieurs ressources de calcul, ainsi que d'une unité de communication qui permet d'échanger des informations avec les autres cellules. Une utilisation classique de cette couche est d'implémenter un réseau de neurones. Ainsi, la troisième couche de chaque cellule peut implémenter un neurone.

1.1.2 Filtre adaptatif évolutionnaire

Les algorithmes évolutionnaires, qui forment une classe d'algorithmes d'optimisation, tirent leur inspiration de la biologie, en particulier du mécanisme de sélection naturelle. L'évolution de cette architecture suit donc l'axe phylogénétique. Le principe consiste à présenter à un système une entrée, lui faire calculer la sortie et la comparer à une vérité terrain. Le système est modifié en fonction des différences entre la sortie et la vérité terrain pour qu'à l'étape suivante la différence soit atténuée.

Pour cela, on commence par générer une *population* de solutions, c'est à dire un ensemble de solutions tirées aléatoirement. Parmi cette population, on repère les solutions qui s'approchent le plus du système désiré, à l'aide d'une fonction de *fitness*. Le choix de cette fonction de *fitness* représente la partie la plus compliquée dans l'implémentation d'un algorithme génétique. Les solutions les plus proches sont gardées pour la génération suivante. Elles sont alors *reproduites*, *croisées*, et *mutées*. Ces trois actions permettent de se rapprocher de l'optimal, tout en évitant de tomber dans un minimum local.

Le filtre adaptatif est composé des éléments suivants [Gallego et al, 2013]. Une zone de logique reconfigurable permet d'héberger le calcul du noyau de convolution du filtre. Il est constitué d'une grille d'éléments de calcul. Ensuite, l'algorithme évolutionnaire décide quand et comment modifier le filtre dans la zone reconfigurable. Cet algorithme est simplement exécuté par un soft-processeur Microblaze. Un autre élément est formé par le *moteur* de reconfiguration. Il est chargé de modifier la grille d'éléments de calcul du filtre en fonction des décisions prises par l'algorithme évolutionnaire, depuis le port de configuration interne du FPGA. Enfin, le module d'évaluation de la fonction de *fitness* calcule la somme des différences,

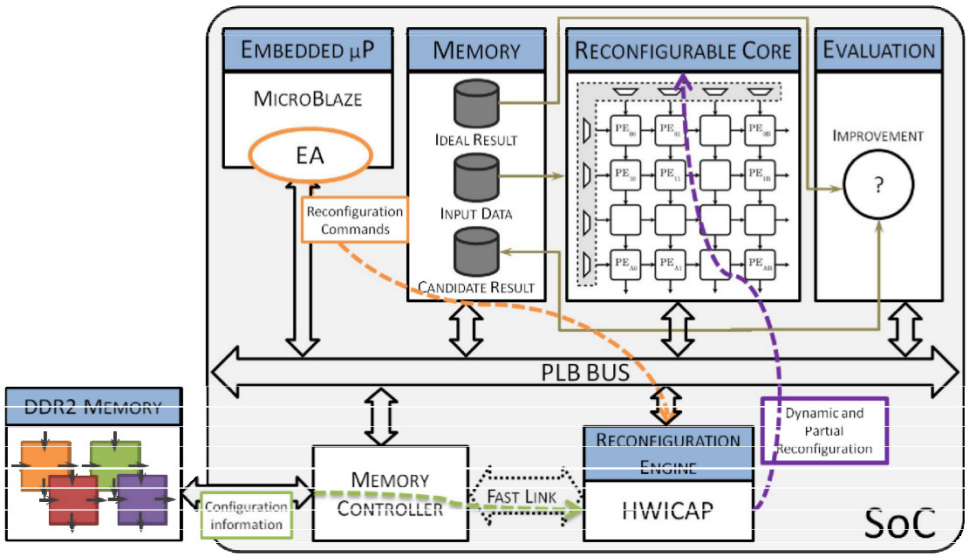


FIGURE 1.2 – **Architecture du filtre adaptif évolutionnaire.**

pixel à pixel, entre l'image résultat et l'image de référence, puis envoie cette valeur à l'algorithme évolutionnaire.

La zone de traitement du filtre est constituée d'une architecture systolique, composée d'une grille d'éléments de calcul. Chaque élément de calcul effectue une opération simple, comme une addition, une soustration ou la recopie d'une donnée par exemple. Les données d'entrées peuvent venir des directions nord et ouest, le résultat étant envoyé vers le sud et l'est. Le rôle de l'algorithme évolutionnaire est de choisir l'opération effectuée par les différents éléments de calcul. Il choisit également quels pixels, parmi le noyau de convolution, envoyer aux éléments de calcul. L'architecture complète du système est représentée en FIGURE 1.2.

Pour obtenir un filtre, la phase d'apprentissage a besoin de générer 800 000 filtres et de procéder à autant de reconfigurations et d'évaluations. Instancié sur un Virtex 5 de Xilinx, le système est capable de générer et de tester plusieurs milliers de filtres par secondes. L'évolutions complète de l'architecture d'un filtre dure ainsi environ 4 minutes. Le filtre est alors capable de traîter 200 millions de pixels par secondes, et peut donc être utilisé dans des applications temps-réel.

Cette architecture a également la capacité d'être tolérante aux pannes. La reconfiguration permet de rapidement réparer les pannes transitoires, et l'algorithme génétique peut trouver des solutions accéptables même si certains éléments de calcul sont victimes de fautes permanentes. Un mécanisme de détection de fautes a été implémenté, dans [Salvador et al, 2011], sur le processeur MicroBlaze. Cette solution ne protège que la zone reconfigurable, le reste de l'architecture doit être durci par d'autres moyens.

En définitive, le couplage d'un algorithme évolutionnaire à une architecture reconfigurable dynamiquement et parallèlement permet de générer un filtre sans connaissances *a priori* de l'environnement dans lequel il sera placé.

Cependant, il existe plusieurs limites par rapport aux ambitions que nous avons dans ces travaux. Tout d'abord, le temps nécessaire à la phase d'adaptation rend impossible l'adaptation en cours d'exécution. Dans un contexte de robotique mobile, l'environnement du robot peut être amené à changer rapidement, si le robot change de zone, où si un humain rentre

dans la pièce par exemple. Ensuite, l'adaptation s'effectue sur un micro-processeur et est par conséquent centralisée, ce qui limite la scalabilité de l'architecture.

1.1.3 Calcul invasif

Dans une architecture many-cœur, le calcul invasif permet à un programme qui s'exécute sur un processeur d'explorer les processeurs voisins et de se répliquer dans ces processeurs [Teich, 2008]. La zone ainsi envahie sert de substrat de calcul à l'exécution parallèle du programme, jusqu'à la fin de celui-ci. À ce moment là, le programme effectue une phase de retraite et reprend l'exécution séquentielle sur un seul processeur. Ici le terme programme prend un sens assez large puisqu'il peut également s'agir d'une architecture déployée sur une surface reconfigurable dynamiquement. L'adaptation dans ce projet suit plutôt l'axe ontogénétique dans le sens où elle provient des intéractions avec le voisinage.

La demande d'invasion de nouvelles ressources est faite par le programme exécuté sur un ou plusieurs processeurs, de manière analogue à la création d'un *thread* dans un système d'exploitation classique. De ce fait, le mécanisme d'invasion est totalement distribué. Il est également dynamique, puisque la demande d'invasion peut dépendre des données à traiter à un instant donné.

Le calcul invasif est bien adapté lorsqu'il s'agit de paralléliser l'exécutions de boucles imbriquées, puisqu'il s'agit du même code exécuté sur des données différentes. Un exemple d'invasion est représenté en FIGURE 1.3.

Le détecteur de Harris [Harris and Stephens, 1988] a été implémenté sur le many-cœur supportant le calcul invasif [Sousa et al, 2013]. En fonction de la quantité de ressources disponibles, l'application exploitant les résultats du détecteur peut agir sur deux paramètres, la qualité et le débit des résultats. Lorsque le nombre de cœurs accessibles est limité, l'algorithme peut sous-échantillonner l'image d'entrée pour accélérer les calculs. À l'opposé, lorsque plusieurs cœurs de calcul sont disponibles, l'algorithme peut envoyer plusieurs sous-blocs de l'image à différents cœurs pour augmenter le débit.

Ici, la reconfiguration peut être définie comme globalement distribuée, localement centralisée. Les processus formant une application sont en effet les seuls responsables de la reconfiguration. Les *threads* qui peuvent être utilisés au sein de ces processus sont entièrement gérés par le processeur hôte du processus.

Lorsqu'une ressource est déjà occupée par un autre programme, la primitive d'invasion renvoie un code d'erreur à l'utilisateur, il n'y a pas de mécanisme de compétition. Ainsi, le premier programme à être exécuté peut limiter l'exécution des suivants, même s'ils sont plus prioritaires.

L'utilisateur doit spécifier dans quelle direction l'invasion doit s'effectuer, dans une direction particulière, ou dans toutes les directions, et peut fixer un maximum. Cela demande une connaissance à priori de la position des programmes sur l'architecture et limite la dynamicité de l'ensemble.

1.1.4 Ordonnanceur par réseau de neurones

L'ordonnancement de plusieurs tâches sur plusieurs ressources est un problème complexe, difficile à résoudre avec les approches classiques. Les auteurs de [Cardeira and Mammeri, 1995] ont montré qu'il était possible de trouver une solution à ce problème avec un réseau de neu-

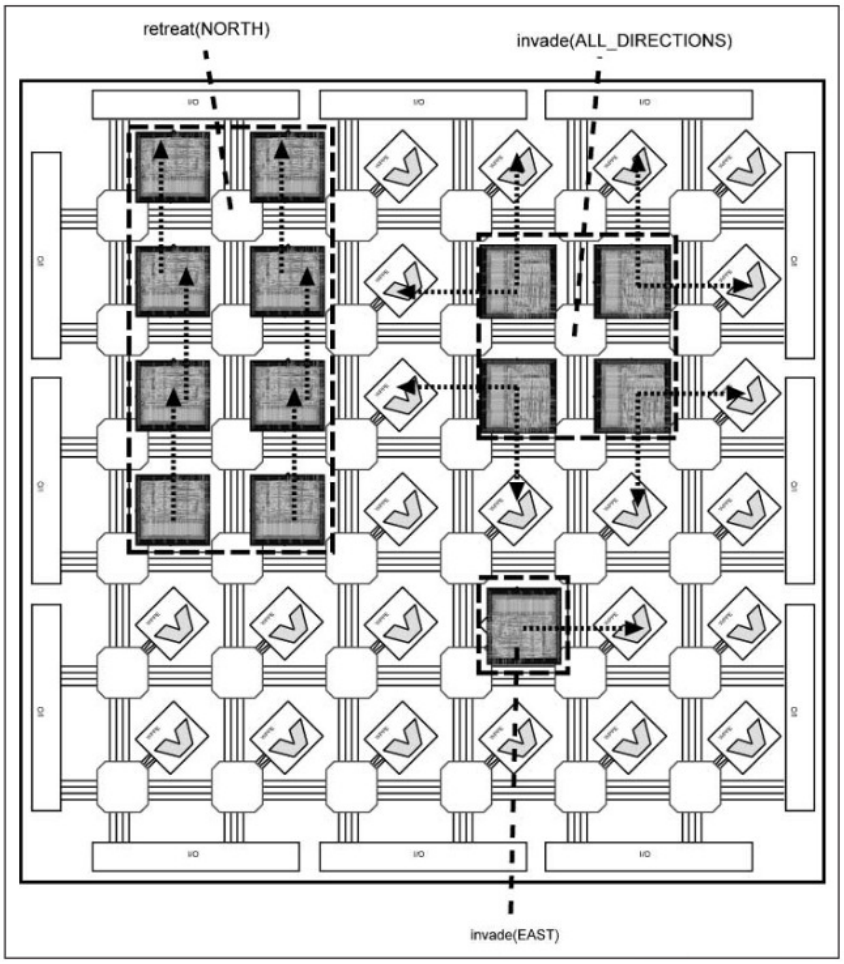


FIGURE 1.3 – **Invasion au sein d'une architecture many-cœur.** Représentation d'invasions uni- et multi-directionnelle et d'une phase de retraite.

rones artificiel de Hopfield [Hopfield, 1982]. Cependant, le nombre de neurones requis est important, et la convergence est difficile à atteindre.

Avant de poser la question de l'ordonnancement de plusieurs tâches sur plusieurs ressources, les auteurs ont d'abord résolu le problème sur une seule ressource. Pour se faire, un neurone est attribué à chaque tâche à ordonnancer, et un neurone supplémentaire est associé à une tâche fictive qui permet à la ressource de se mettre en pause. L'ordonnancement s'effectue selon le Worst Case Execution Time (WCET) défini par l'utilisateur. Finalement, le neurone actif indique quelle tâche doit être traitée.

L'extension à un ordonnancement sur plusieurs ressources consiste à dupliquer le réseau de neurones sur chaque ressource et à empêcher une tâche exécutée sur une ressource donnée de s'exécuter sur une autre ressource. Ainsi, chaque ressource a besoin d'autant de neurones qu'il y a de tâches à ordonnancer. Pour permettre la convergence, des neurones cachés doivent être ajoutés au système, dont la quantité dépend fortement de l'application, et augmente avec le WCET des tâches.

Les auteurs de [Chillet et al, 2011] proposent une amélioration visant à réduire la quantité de neurones requis, et d'améliorer la convergence. Pour cela, ils utilisent des neurones inhibiteurs

actifs lorsque leur ressource correspondante exécute une tâche donnée. Cette solution permet de réduire le nombre de neurones cachés, et de rendre leur quantité indépendante du WCET.

Les auteurs ont validé leur système sur une application composée de 10 tâches, à ordonnancer sur une architecture hétérogène disposant de 5 ressources. Le système a besoin d'un total de 10 neurones par tâche et par ressource, et nécessite donc un total de 500 neurones. Ce nombre est réduit par l'hétérogénéité de la plateforme, puisque toutes les tâches ne peuvent pas être exécutées par toutes les ressources. Ainsi, le nombre de neurones est réduit à 160 pour l'exemple considéré.

Ces travaux apportent la preuve qu'il est possible d'ordonnancer un grand nombre de tâches sur plusieurs ressources en suivant l'axe épigénétique, là ou les approches plus classiques peinent à trouver une solution. La quantité de neurones ainsi que leur connectivité élevée limite la scalabilité et le système ne poura pas être adapté dans un contexte many-cœur. De plus, le réseau de neurones dépend fortement de l'application et doit être repensé pour chaque application.

1.1.5 Synthèse

Nous avons vu plusieurs projets d'architectures de traitement matériel bio-inspirées. Selon le modèle POE, ces architectures peuvent être classées selon trois axes, l'axe Phylogénétique, l'axe Ontogénétique et l'axe Epigénétique.

L'axe phylogénétique se base sur les algorithmes évolutionnaires, qui sont issus des mécanismes d'évolutions et de sélection naturelle. Son utilisation vise principalement à fournir une architecture valide pour un problème ou un environnement donné, inconnu à l'avance. Son cycle d'adaptation est relativement long et pourra difficilement être utilisé dans un cadre d'auto-organisation dynamique. Les travaux récents sur cet axe ont permis de générer un filtre matériel sans connaissances de l'environnement. L'algorithme évolutionnaire est exécuté sur un soft-processeur ce qui le rend inutilisable sur de grands réseaux.

L'axe ontogénétique, qui décrit le développement d'un système multi-cellulaire, est le vecteur d'exécution de l'axe phylogénétique. Il permet, par le moyen de la colonisation et des interactions avec les cellules voisines, de configurer le substrat de calcul pour instancier l'architecture définie par l'axe phylogénétique. Le calcul invasif, qui se rapproche le plus de cet axe, permet de mettre en place une adaptation globalement distribuée et localement centralisée. La reconfiguration matérielle et la reprogrammation logicielle sont considérés de la même manière. Il ne propose en revanche pas de mécanisme de compétition, ce qui peut ammener à des situations où des tâches peu prioritaires bloquent des tâches qui le sont plus.

Enfin, l'axe épigénétique permet au système d'évoluer au rythme des variations de l'environnement et des objectifs à atteindre. Cette évolution s'effectue grâce à des mécanismes d'apprentissage. Les travaux sur cet axe montrent qu'il est possible d'ordonnancer des tâches sur plusieurs ressources à l'aide d'un réseau de neurones. Cependant les modèles proposés sont peu scalable et dépendent fortement de l'application. Ce système est donc peu automatisé et demande de bonnes connaissances dans le domaine des réseaux de neurones au développeur d'applications.

1.2 Contributions

En analysant les solutions actuelles pour adresser le problème des architectures massivement parallèles et potentiellement reconfigurable, nous avons pu observer plusieurs limites,

mais aussi quelques opportunités.

Les architectures parallèles sont capables d'accueillir plusieurs centaines de cœurs de calcul sur une seule puce. Le modèle de la mémoire évolue et passe d'un modèle à mémoire partagée, à un modèle à mémoire distribué, ce qui permet, à l'aide d'un mécanisme de passage de message soutenu par un réseau sur puce, de passer à l'échelle.

L'usage de ces architectures semble limité à des applications relativement bas niveau, comme des déroulages de boucles ou des applications décrites en flot de données. Les applications nécessitant plus de contrôle font appel à des solutions centralisées peu scalables.

La reconfiguration dynamique partielle permet d'adresser des applications à la fois gourmandes en puissance de calcul et dynamiques. Elle est rendue complexe par les outils actuels verrouillés, une technologie qui évolue rapidement et un manque d'inter-opérabilité. De plus, elle dépend de systèmes de reconfiguration centralisés. Lorsque l'on souhaite tester les mécanismes gérant la reconfiguration dynamique, cette dernière peut être remplacée par une reprogrammation logicielle pour s'affranchir de ces verrous.

Les architectures auto-organisées apportent des solutions à certaines de ces limites. Il est possible par exemple de générer un filtre matériel de manière automatique en présentant une image vérité terrain à un système et en le laissant évoluer vers une solution. Cependant, cette solution est relativement figée et reste une application de bas niveau.

Une autre architecture permet à un programme ou à une architecture matérielle de coloniser une zone d'un circuit pour y réaliser un traitement, puis de libérer les ressources utilisées à la fin du traitement. Cette solution peut néanmoins conduire à des situations de blocage à cause du manque de mécanisme d'ordonnancement ou de compétition.

L'ordonnancement d'un grand nombre de tâches sur plusieurs cœurs de calcul, possiblement hétérogènes, peut être réalisé à l'aide d'un réseau de neurones. Les solutions actuelles ne permettent pas de réaliser cet ordonnancement sur des systèmes massivement parallèles, et restent fortement dépendantes de l'application.

Dans cette thèse, nous cherchons donc à concevoir une plateforme parallèle auto-organisée, qui supporte le passage à l'échelle, et soit autonome. Elle fera appel à un modèle à mémoire distribuée, soutenue par un réseau sur puce. Elle devra être capable de supporter la reconfiguration matérielle dynamique partielle et parallèle, mais pourra être abstraite par une reprogrammation logicielle. L'ordonnancement des tâches, logicielles ou matérielles, sera réalisé par un mécanisme d'auto-organisation et de compétition. Cette plateforme est définie dans le cadre du projet SATURN (Self-Adaptive Technologies for Upgraded Reconfigurable Neural computing) [SATURN, 2010].

Ainsi cette architecture sera ciblée par des applications complexes. Cette architecture s'inscrit dans un contexte de robotique mobile, où le robot est considéré comme un système adaptatif. L'architecture de contrôle du robot ne peut plus être pensée seule, car elle fait partie intégrante du robot et de ses spécificités morphologiques : ses capteurs, ses actionneurs, son électronique. Nous nous appuyons dans ces travaux sur la théorie de l'*embodiment* [Clark, 1999; Wilson and Foglia, 2011].

Nous considérons ainsi le robot en temps qu'entité autonome, placé en immersion dans un environnement inconnu et dynamique. Dans cet environnement, il sera amené à naviguer, à manipuler des objets et à interagir avec des humains. Les capteurs du robot perçoivent sans interruption l'information contenue dans l'environnement et l'envoient, sous forme de stimuli, au contrôleur. Le contrôleur intègre ces informations provenant de différentes modalités et provoque une action sur l'environnement de la part du robot, créant ainsi une boucle sensori-

motrice. C'est dans le cadre de cette boucle d'interaction entre le robot et l'environnement que l'architecture de contrôle a besoin de s'adapter. L'adaptation se fait à la fois par rapport à la morphologie du robot et à la dynamique de l'environnement.

La nature des stimulations multimodales captées, à la fois qualitative et quantitative, dépendent de l'environnement, et des capteurs en eux-mêmes. L'adaptation au sein de l'architecture s'appuie sur l'apprentissage de classes d'information qui catégorisent la nature et la richesse de l'environnement. Le contrôleur construit ainsi une représentation interne de l'environnement du robot. De cette représentation interne émergent des aires de traitement, de manière à imiter la plasticité des aires corticales dans le cerveau des mammifères. La taille de ces zones sera proportionnelle à l'importance des classes de données associées.

L'apprentissage des classes d'informations est basé sur un réseau de neurones artificiel inspiré par les cartes auto-organisatrices de Kohonen [Kohonen, 2012] et par les champs neuronaux [Amari, 1977]. Ce réseau de neurones s'intègre dans la boucle sensori-motrice, constituée par le robot et ses interactions avec l'environnement, et y joue un rôle central.

L'intégration d'un réseau de neurones dans une architecture de contrôle modifie la manière dont un problème est résolu. À la place d'une approche classique consistant à séparer le problème en différentes sous-tâches, pour ensuite les programmer, le réseau de neurones effectue un apprentissage, à partir duquel le comportement attendu du système émerge.

Ce mode de conception apporte plus d'adaptabilité, grâce aux capacités de généralisation des réseaux de neurones. Cependant, en perdant la programmabilité du contrôleur, nous perdons également le déterminisme et la prédictibilité que garantissent les architectures classiques (tout du moins dans le cas monoprocesseur [Miramond, 2014]). Nous proposons dans ces travaux un compromis, en ajoutant une architecture many-cœur programmable au réseau de neurones adaptatif. Ainsi les aires de traitement énoncées plus haut, qui émergent de la nature des informations contenues dans l'environnement grâce au réseau de neurones, redeviennent programmables.

La contrainte principale lors de la conception de cette architecture est la scalabilité. Pour que l'architecture soit scalable, nous avons besoin que le réseau de neurones et que l'architecture many-cœur programmable soient distribués. De plus, l'information saillante doit être extraite de l'environnement avant d'être transmise au réseau de neurones. En effet, il ne pourra pas traiter l'information brute provenant des capteurs, car elle est bien trop dense et complexe. Pour réduire la charge de calcul et rendre la généralisation possible, les informations pertinentes de l'environnement doivent être extraites [Treisman and Gelade, 1980].

1.2.1 Principe général de l'architecture SATURN

Pour munir notre contrôleur des capacités d'adaptation et de programmabilité énoncées ci-dessus, nous avons séparé son organisation en plusieurs couches. Les différentes couches de l'architecture du contrôleur sont représentées en FIGURE 1.4, et décrites ci-dessous.

La première couche, la **couche d'acquisition** a pour objectif de capturer les diverses informations contenues dans l'environnement. Ces informations peuvent être de différentes nature et provenir de différentes sources. Une caméra, par exemple, peut fournir des images ou une quantité de mouvements, un micro peut donner des informations auditives et un système d'odométrie peut fournir des données proprioceptives. L'organisation de cette couche

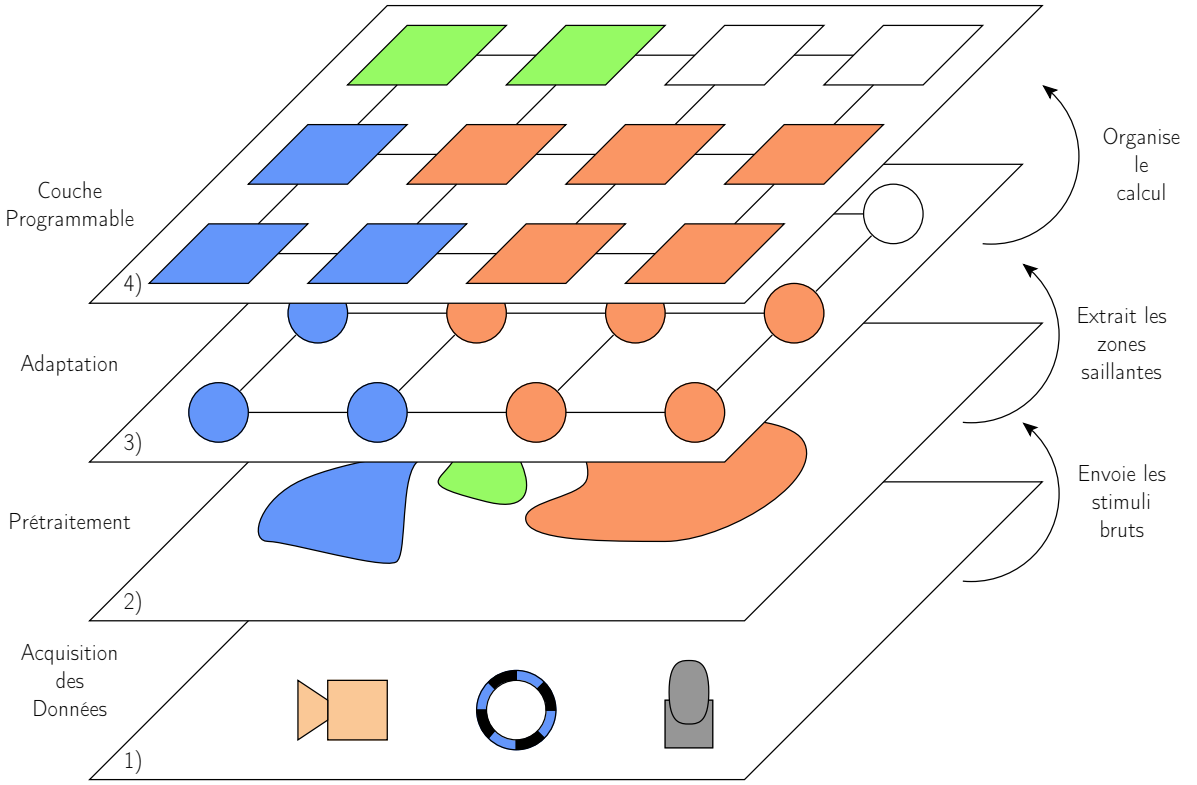


FIGURE 1.4 – **Vue en couche de l'architecture du contrôleur.** La **couche d'acquisition (1)** envoie les stimuli bruts à la **couche de pré-traitement (2)** qui extrait les zones saillantes pour permettre à la **couche d'adaptation (3)** d'organiser le calcul dans la **couche programmable (4)**.

dépend fortement de la morphologie du robot et de la nature des capteurs.

Les données brutes extraites de la couche d'acquisition sont propagées à la couche suivante, la **couche de pré-traitement**. Elle a pour rôle d'extraire l'information pertinente de l'environnement, pour à la fois réduire la quantité d'informations à traiter, tout en facilitant la généralisation et l'apprentissage des données. En d'autres termes, cette couche construit une carte de saillance de l'environnement. L'organisation de cette couche est intimement liée à celle de la couche d'acquisition et dépend de l'application visée. Elle impacte en outre l'adaptation qui a lieu dans les couches supérieures.

Cette couche est étudiée dans le Chapitre 2 de cette thèse par le biais d'un système de vision bio-inspiré. Ce système, développé dans [Fiack et al, 2014a], est la source principale de perception d'un robot mobile autonome effectuant des missions de navigation. Il permet à un réseau de neurones d'apprendre et de reconnaître des scènes visuelles, et permet ainsi de construire un bassin d'attraction en associant des cellules de lieu à un mouvement souhaité. Dans cette étude, nous obtenons du robot un comportement de retour au nid grâce à l'apprentissage de données saillantes, extraites par le système de vision.

L'information saillante extraite par la couche de pré-traiement est transmise à la **couche d'adaptation**, qui forme la troisième couche de cette architecture. Elle pilote directement l'organisation du calcul de haut niveau réalisé dans la dernière couche. Elle est constituée d'une carte auto-organisatrice matérielle distribuée, conçue pour être scalable.

Le Chapitre 3 couvre dans un premier temps le modèle de la carte auto-organisatrice, développé par Rodriguez dans [Rodriguez et al, 2014]. Dans un second temps, nous y discutons deux implémentations matérielles. La première constitue une preuve de concept développée parallèlement au modèle neuronal [Rodriguez et al, 2013b]. La seconde implémentation est formée par un réseau de processeurs simplifiés, spécialisés dans le traitement des équations des neurones [Fiack et al, 2015].

Ce réseau, stimulé soit par des données issues de manipulations robotiques soit par des données synthétiques, est capable de catégoriser différentes classes de données, selon leur densité statistique. Cette catégorisation consiste en la formation de clusters de neurones, chaque cluster représentant une classe de données.

La quatrième et dernière couche forme la **couche programmable**. Elle est constituée d'une matrice d'éléments de calcul reconfigurables, qui peuvent héberger soit un soft-processeur soit un accélérateur matériel. Il y a autant d'éléments de calcul dans cette couche que de neurones dans la carte auto-organisatrice, et chaque élément de calcul est relié à un neurone. Ainsi, quand un neurone appartient à un cluster, en d'autres termes quand il représente une classe de données particulière, l'élément de calcul correspondant est colonisé, à la manière de [Sanchez et al, 2007], pour exécuter une tâche en lien avec ce type de données. On voit ainsi émerger des aires de calcul dans cette couche.

Quand un neurone se "déplace" vers un autre cluster, suite à une modification dans l'environnement par exemple, l'élément de calcul associé change simplement de tâche. Ce changement peut aller de l'exécution d'une autre fonction à une reconfiguration complète de l'élément. Les dimensions des aires de calcul sont dictées par les données d'entrée du système et varient au cours des changements de l'environnement. On parle d'une adaptation *data-driven*. Le système matériel s'adapte donc aux variations de l'environnement perçues par le robot. Chaque aire de calcul fonctionne à la manière d'une architecture parallèle hétérogène, reconfigurable et de taille dynamique.

Dans le Chapitre 4 nous proposons une architecture composée de micro-processeurs openMSP430, mis en réseau par une grille de routeurs. Nous prototypons cette architecture sur la plateforme Confetti, développée par Vannel dans [Vannel, 2007]. Nous nous basons sur cette plateforme pour aborder la problématique de la reconfiguration dynamique parallèle, qui permet à l'architecture du contrôleur elle-même de s'auto-organiser de manière complètement distribuée [Fiack et al, 2014b].

1.2.2 L'émergence d'aires de traitement

Parmi les quatre couches qui composent notre architecture, les deux premières, l'acquisition et le pré-traitement, sont intimement liées et dépendent de la morphologie du robot. Les deux couches suivantes, l'auto-organisation et la couche programmable, sont également liées et sont par conception indépendantes du robot, de ses missions et de l'environnement.

Les deux premières couches sont constituées d'un ensemble de fonctionnalités qui visent à extraire la saillance de différentes modalités. Elles peuvent être implémentées sous la forme d'IP de traitement câblées, que l'on peut ou non activer en fonction des périphériques connectés. Cette solution limite le nombre et la nature des capteurs que l'on peut interfacer avec le contrôleur, mais propose une solution *clé en main*.

À l'opposé, il est envisageable de réserver une surface reconfigurable à ces deux premières couches. Ainsi, les IP qui correspondent aux périphériques utilisées peuvent être implantées sur cette surface. Cette solution se rapproche des familles de FPGA Zynq chez Xilinx [Xilinx,

2015] et SoC de chez Altera [Altera, 2013] qui proposent un micro-processeur et un certain nombre d'éléments câblés fortement couplés à une matrice reconfigurable. Cette solution permet plus de choix d'évolutivité quant aux capteurs interfaçables avec le contrôleur. Si cette solution demande à l'utilisateur final plus de compétences, notamment en description matérielle, un ensemble d'IP peut être fourni pour le cas des fonctions les plus communes.

Nous ne discuterons pas plus loin les choix technologiques du substrat de calcul des deux premières couches, préférant nous focaliser sur les aspects architecturaux et sur les couches suivantes.

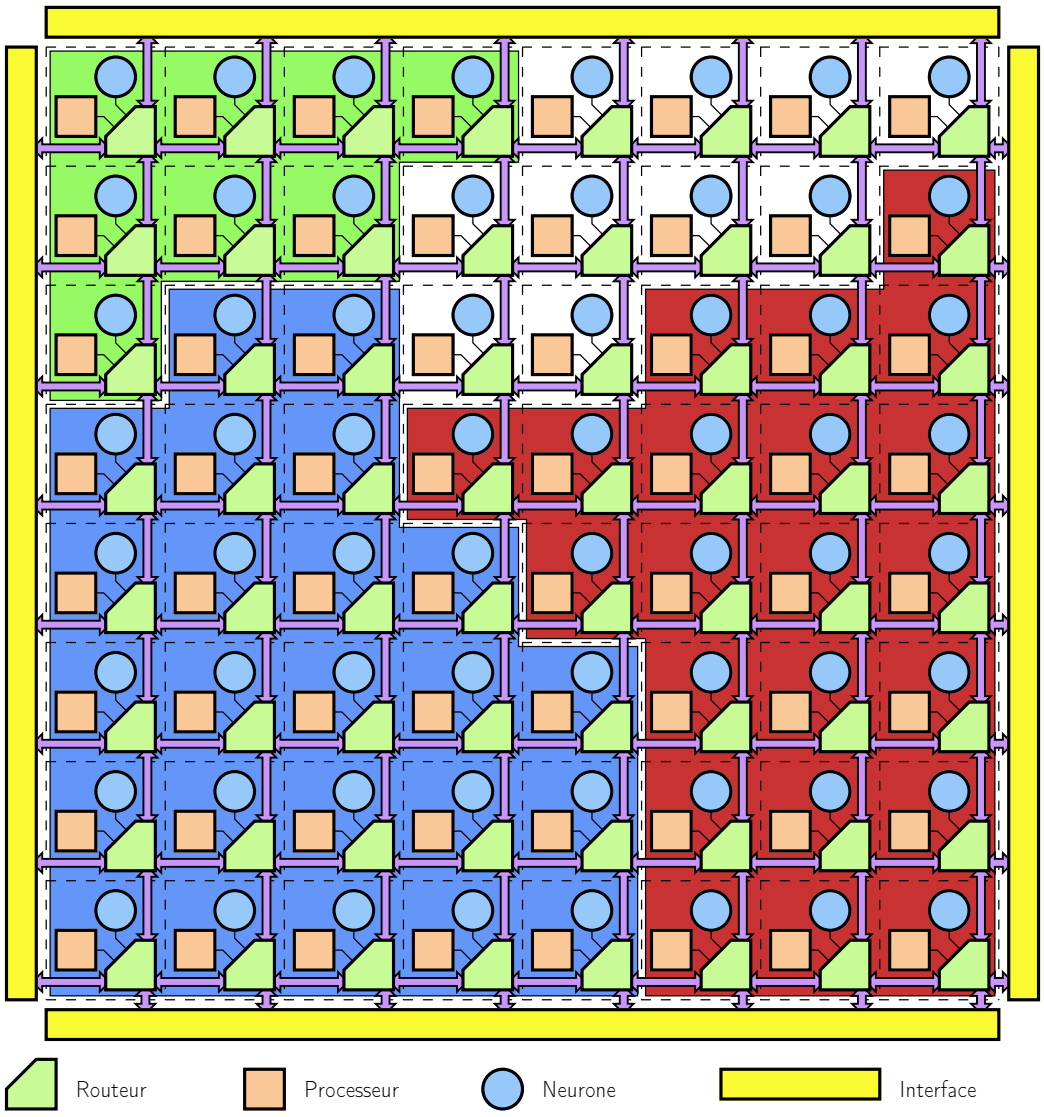


FIGURE 1.5 – **Exemple de grille de tuile** 8×8**.** Chaque tuile est composée d'un neurone, d'un élément de calcul et d'un routeur, et est connectée à ces quatre voisines. Une interface connecte la tuile au monde extérieur. À un instant donné, trois aires de calcul ont émergé. Le nombre de tuiles qui les composent dépend de la stimulation présente et passée de la modalité associée.

Les couches d'adaptation et de programmation ont mené au développement d'une architecture de type many-cœur auto-organisée. Elle est constituée d'une grille de **tuiles**, constituées

d'un neurone, d'un élément de calcul et d'un nœud de routage. Chaque tuile de la grille est connectée aux quatre tuiles voisines. Cette grille de tuiles de calcul reçoit des stimuli du monde extérieur grâce à une interface qui la connecte à la couche de pré-traitement.

Au sein d'une tuile, le neurone d'adaptation, stimulé par les couches les plus basses et communiquant localement avec ses voisins rejoint ou forme un cluster représentant une modalité de l'environnement. De ce fait, il décide d'une tâche à effectuer par l'élément de calcul reconfigurable. Si un neurone n'appartient à aucun cluster, l'élément de calcul pourra être mis en pause pour diminuer la consomation.

On voit ainsi apparaître des aires de calcul sur l'architecture. Un exemple de grille de tuiles de dimension 8×8 est représenté en FIGURE 1.5. Dans cet exemple, trois aires de calcul, représentées par trois couleurs, entrent en compétition.

Les aires rouge et bleue ont été stimulées de manière semblable et ont par conséquent recruté environ le même nombre de tuiles. L'aire verte quant à elle est plus réduite, ce qui signifie qu'elle a été moins stimulée.

Plusieurs tuiles ne sont pas recrutées et restent disponibles dans le cas où une des trois modalités venait à devenir plus active.

1.3 Organisation du mémoire

Dans cette thèse, nous nous intéressons aux problématiques architecturales d'un contrôleur matériel auto-organisé dans un contexte de robotique mobile bio-inspirée. Ce contrôleur est constitué d'une grille massivement parallèle d'éléments de calcul reconfigrables.

Des aires de traitement émergent dans cette grille grâce à une carte auto-organisatrice, stimulée par les données provenant des capteurs du robot. L'auto-organisation de l'architecture est donc pilotée par ces données d'entrée, dont la saillance est extraite par un système sensoriel.

L'acquisition des données, leur pré-traitement, la carte auto-organisatrice et la grille de calcul reconfigurable forment les quatre couches de l'architecture du contrôleur. Leur implémentation est traitée dans les trois chapitres constituant cette thèse. La couche d'acquisition ne nécessite pas un chapitre à part entière et est discutée en même temps que le pré-traitement.

Nous présentons dans le Chapitre 2 un système de vision correspondant à la couche d'acquisition et à la couche de pré-traitement. Il s'inspire du système visuel des mammifères, en mimant le mécanisme des saccades occulaires. Il est basé sur une approche multi-échelle pour construire une carte de saillance de l'environnement visuel à partir de laquelle il focalise son attention sur les points les plus saillants du champ visuel et extrait, autour de chaque point, un repère visuel.

Ce système visuel est validé de manière indépendante sur un robot mobile effectuant des missions de navigation. Un réseau de neurones embarqué dans le robot est chargé d'apprendre et de reconnaître des scènes visuelles à l'aide des repères générés par le système de vision. Ce réseau de neurones crée des cellules de lieux qui code une position particulière dans l'espace. Le robot peut ainsi associer un mouvement à effectuer à chaque position apprise pour pouvoir naviguer dans un environnement quelconque.

Ce système de vision est décrit en langage VHDL et porté sur un FPGA Zynq de Xilinx. Le micro-processeur du Zynq est utilisé pour gérer la pile TCP/IP nécessaire aux communications avec le reste du robot. Nous présentons les résultats d'implémentation, ainsi que des résultats comportementaux de missions robotiques de retour au nid.

Nous présentons ensuite dans le Chapitre 3 la couche adaptative. Elle est basée sur une carte auto-organisatrice inspirée des cartes de Kohonen et des champs neuronaux dynamiques. La principale contrainte lors de la conception de ce réseau de neurones est la scalabilité. Ensuite, de part la configuration en grille de la couche de calcul programmable, l'architecture de la carte doit être distribuée.

Les neurones de la carte s'organisent au sein de différents clusters en fonction de la nature et de la dynamique des données d'entrée. Ces clusters représentent les différentes modalités présentes dans l'environnement, détectées par les différents capteurs du robot.

Les travaux effectués dans le cadre de ce chapitre ont mené à deux implémentations matérielles, décrites en langage VHDL et portées sur FPGA Stratix V d'Altera. La première est une preuve de concept initiale, nous permettant de tester la viabilité d'une telle carte, et nous ayant permi de constater un certain nombre de difficultés. Nous avons pris en compte les différentes limites de cette première implémentation pour produire une nouvelle version plus légère et plus modulable. Nous présentons les résultats d'implémentation de ces deux architectures, ainsi que des résultats comportementaux obtenus avec des données dans un premier temps synthétiques, puis issues d'une vidéo tirée d'une mission de navigation robotique.

Enfin, dans Chapitre 4 nous présentons la couche programmable. Elle est constituée d'un ensemble d'éléments de calcul reconfigurables. Ces éléments, organisés en grille, sont reliés par un réseau de routage. Nous discutons l'architecture des nœuds de routage, puis d'une tuile de calcul construite autour d'un soft-processeur openMSP430.

Nous discutons du prototypage de cette couche sur la plateforme Confetti, constituée de plusieurs dizaines de FPGA dont la structure est particulièrement bien adaptée à notre projet. Nous y déployons une application simple et mesurons les capacités du réseau d'interconnexions.

CHAPITRE 2

Architecture de vision

Sommaire

Munir un robot mobile de capacités de vision semble être la voie la plus efficace pour que ces entités artificielles puissent intéragir de manière autonome dans un environnement toujours plus complexe où les intéractions avec l'Humain sont de plus en plus nombreuses. Lorsque l'on conçoit des robots devant intéragir avec le vivant, il semble naturel de s'inspirer du monde biologique, que ce soit pour percevoir, décider et agir.

Les algorithmes de vision bio-inspirés demandent généralement une puissance de calcul importante, ce qui rends difficile leur intégration dans les robots autonomes. D'abord, un tel système de vision doit traiter les informations en temps réel pour permettre au robot d'évoluer

dans un environnement dynamique. Ensuite, pour s'intégrer à un robot mobile, le système doit avoir une dimension modérée, et consommer peu.

Nous présentons dans ce chapitre un système sur puce de vision bio-inspiré, que nous prototypons sur un circuit FPGA. Ce système de vision est couplé à une architecture neuronale pour apprendre des scènes visuelles lors de tâches de navigation. Nous pouvons ainsi tester notre système dans un cas concrêt, avant de l'intégrer finalement à l'architecture SATURN pour étudier l'auto-organisation dirigée par les données discutée au Chapitre 1.

2.1 Processus attentionnel

Le système visuel présenté dans cette section est basé sur une approche multi-échelles pour extraire les primitives visuelles. Plus précisément, il fournit les caractéristiques locales des points d'intérêt détéctés parmi le flux de pixels fourni par une caméra. Ainsi, ce système permet d'envisager un large panel d'applications. Ces caractéristiques visuelles sont utilisées par un réseau de neurones capable d'associer des actions motrices à des informations visuelles. Ce réseau de neurones peut apprendre, par exemple, la direction du mouvement du robot en fonction de la reconnaissance d'un lieu. L'architecture neuronale de ce réseau est présentée en section 2.3.

Le système visuel étudié ici est divisé en deux modules principaux, décrits dans les sections suivantes :

— Un mécanisme multi-échelle pour l'extraction de points d'intérêt.
— Un mécanisme chargé d'extraire des caractéristiques locales pour chaque point d'intérêt.

2.1.1 La détection sur plusieurs échelles de points d'intérêt

L'approche multi-résolutions est de nos jour bien connue de la communauté de vision par ordinateur. Une grande variété de détecteurs de points d'intérêt peut être trouvée dans la littérature. Parmi eux, on trouve le détecteur de points d'intérêt de Lindeberg [Lindeberg, 1998], le détecteur de Lowe [Lowe, 2004], basé sur la détection de maxima locaux d'une image filtrée par une Difference Of Gaussians (DoG) ou encore le détecteur de Mikolajczyk [Mikolajczyk and Schmid, 2004], où les points d'intérêt correspondent à ceux fournis par le calcul d'une fonction de Harris bi-dimensionnelle et concordent avec les maxima locaux du Laplacien à travers les échelles. Le système visuel décrit ici est inspiré par la psychologie cognitive. Il extrait le voisinage des points d'intérêt, qui correspondent aux points anguleux de l'environnement visuel du robot. Plus précisément, les points d'intérêt correspondent aux maxima locaux d'une image de magnitude de gradient filtrée par une différence de gaussiennes. Ce détecteur est caractérisé par une bonne stabilité entre les échelles. Cette caractéristique est aussi appelée equivariance [Lindeberg, 1998]. À la suite du détecteur, le système fournit une liste de caractéristiques locales, triée par le biais d'une compétition entre les différents points d'intérêt.

Les points d'intérêt sont détectés dans un espace d'échelle échantillonné, basé sur une pyramide d'images. Les pyramides sont utilisés dans les méthodes multi-résolutions pour réduire le coût computationnel des opérations de filtrage. L'algorithme utilisé pour construire la pyramide est détaillé et évalué dans [Crowley and Riff, 2003]. La pyramide est basé sur un filtrage successif de l'image par des noyaux gaussiens bi-dimensionnels, normalisés par un facteur S.

Ces opérations réalisent un lissage successif de l'image. Deux lissages successifs sont effectués au moyen de deux noyaux gaussiens de variance $\sigma^2 = 1$ et $\sigma^2 = 2$. Le facteur d'échelle doublant, en d'autre termes une octave étant traitée, l'image peut être sous-échantillonnée par un facteur 2 sans perte d'information. Les mêmes noyaux gaussiens peuvent être réutilisés

pour continuer la construction de la pyramide. Une conséquence intéressante est que la taille des noyaux reste petite, permettant un calcul plus rapide de la pyramide. En effet, on constate qu'au-dela d'une demi-largeur et d'une demi-hauteur de noyau de 3σ, la précision machine est atteinte. Finalement, les images filtrées par les différences de gaussiennes peuvent être obtenues simplement en soustrayant deux images consécutives dans la pyramide. Une représentation de l'ensemble de l'algorithme est donné sur la FIGURE 2.1.

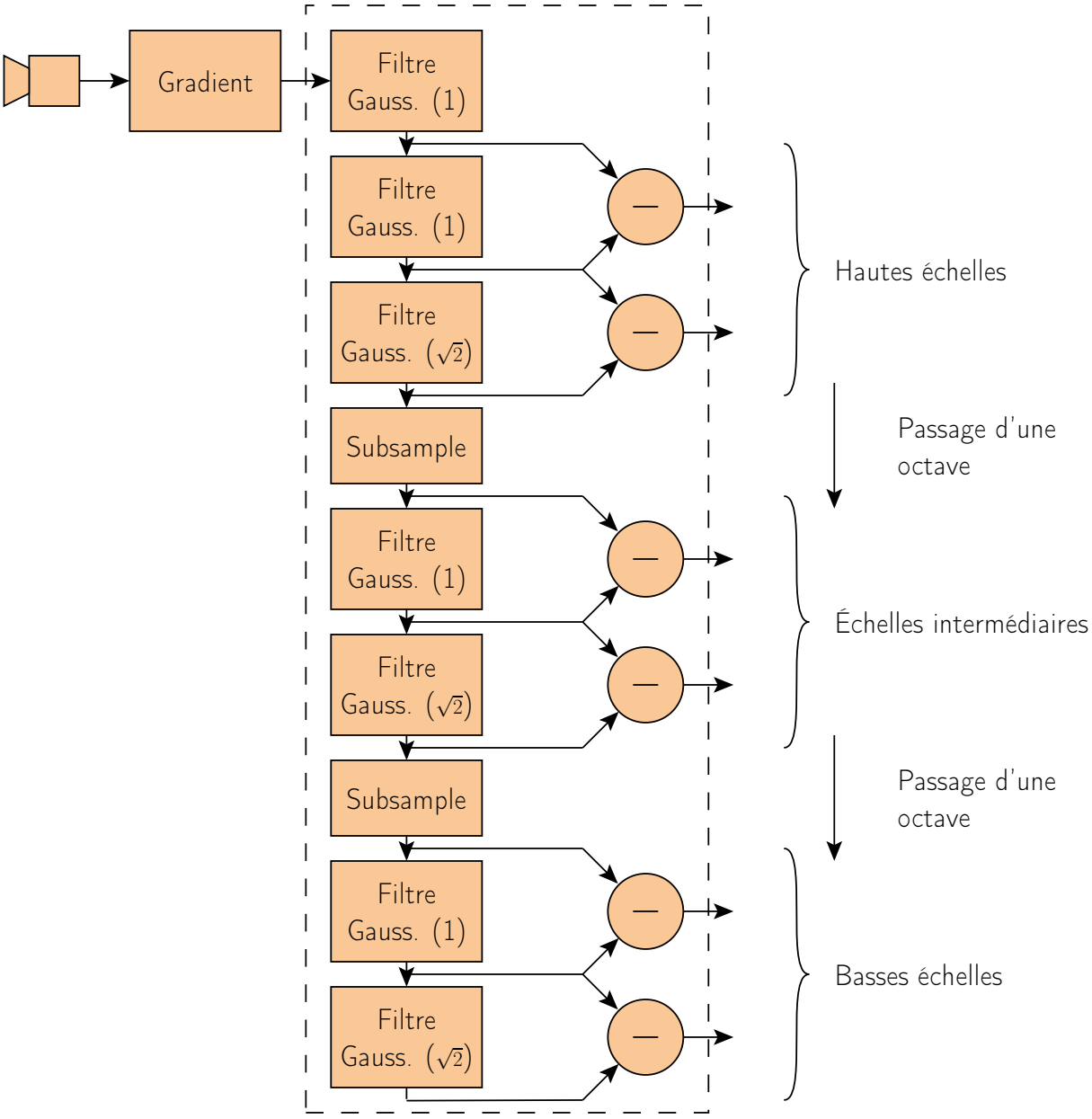


FIGURE 2.1 – **Représentation de la pyramide Gaussienne.**

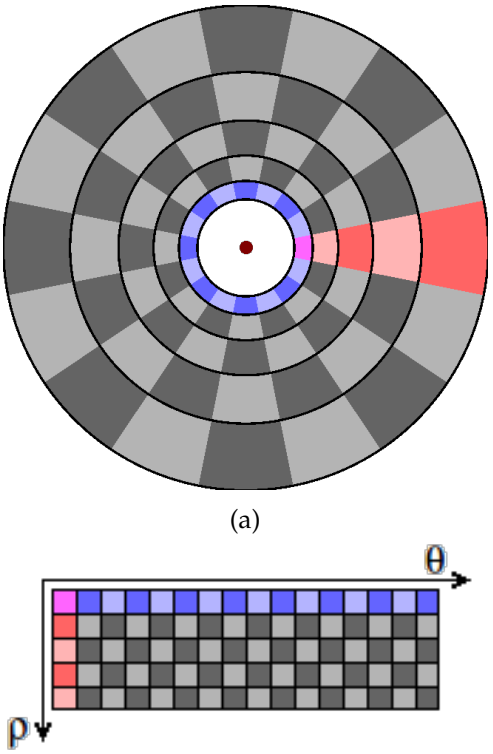
La détection des points d'intérêt s'effectue en recherchant dans chaque image de différences de gaussiennes les N maxima locaux les plus intenses. L'algorithme de recherche des points d'intérêt trie ainsi les maxima locaux suivant leur intensité et ne conserve que les N plus grands. Leurs coordonnées peuvent alors être extraites. La recherche d'un maximum local s'ef-

fectue dans un disque de rayon R, pouvant être paramétré. Le paramètre N corresponds au nombre maximal de détections. En effet, le robot peut explorer des environements plus ou moins riches en détails. Il peut évoluer par exemple en intérieur ou en extérieur, être confronté à des murs peu saillants ou en revanche à des objets plus complexes.

Un seuil de détection γ est mis en place pour éviter la détection de points non-saillants. Ce seuil est une valeur minimale à partir de laquelle un point peut être détecté. La présence de ce seuil est encore plus importante aux basses résolutions car l'information y est plus grossière. Cette particularité confère à l'algorithme un aspect dynamique, puisque le nombre de points d'intérêt – et par conséquent le nombre de caractéristiques locales – dépends de la scène visuelle qui n'est pas connue *a priori*.

2.1.2 L'extraction des imagettes caractéristiques

À ce niveau, le voisinage de chaque point d'intérêt doit être caractérisé dans le but d'être appris par le réseau de neurones. Les méthodes existantes pour caractériser des points d'intérêt sont nombreuses dans la littérature, comme Scale-Invariant Feature Transform (SIFT) par exemple. Dans notre application, nous utilisons une caractérisation inspirée de la vision des mammifères où le voisinage des points d'intérêt est représenté dans un espace log-polaire. Cette représentation a de bonnes propriétés en terme de robustesse au changement d'échelle et à la rotation. La caractéristique locale est ainsi une imagette qui résulte de la transformation log-polaire du voisinage (voir 2.2).



(a)

(b)

FIGURE 2.2 – **Représentation d'une transformée log-polaire.** L'image (a) représente un mapping log-polaire avec $\rho = 5$ et $\theta = 16$. L'image (b) est le résultat de la transformée dans l'espace (ρ, θ).

Les imagettes caractéristiques sont extraites à partir de l'image de la magnitude du gradient. Le voisinage de chaque point est constitué d'une couronne autour du point. Les rayons intérieurs et exterieurs (respectivement r_{int} et r_{ext}) de la couronne sont configurables. L'exclusion du disque intérieur permet d'éviter une sur-représentation des pixels centraux dans l'espace log-polaire. Le logarithme du rayon et l'angle sont échantillonnés en respectivement ρ et θ valeurs. Chaque caractéristique est ainsi une imagette de dimensions $\rho \times \theta$ pixels. Les dimensions des anneaux et des imagettes caractéristiques sont déterminées expérimentalement en analysant le taux de reconnaissance du réseau de neurones dans un environnement donné. Finalement, les imagettes log-polaires sont normalisées en vue d'être exploitées par le reste de l'architecture neuronale. En associant les données fournies par le système visuel avec des actions, le système global permet au robot de se comporter de manier cohérente dans son environnement [Maillard et al, 2005].

Cette chaîne algorithmique a longuement été étudiée [Maillard, 2007; Giovannangeli et al, 2006a]. Les résultats de ces études ont montré qu'elles permettaient de reconnaître des scènes visuelles avec un coût computationnel réduit. Ce chapitre se concentre donc sur l'implantation temps-réel de ce système visuel.

Le réseau de neurones intégré au robot est chargé d'apprendre à reconnaître plusieurs lieux particuliers. Ces apprentissages se font, pour chaque lieu, grâce à un jeu d'imagettes log-polaires associées à leur position dans l'espace. À chaque nouveau lieu reconnu, le robot apprend un mouvement à effectuer pour rejoindre son objectif. Ainsi, une fois l'apprentissage terminé, lorsque le robot reconnaît un lieu, il effectue le bon mouvement pour se déplacer jusqu'à son objectif.

La chaîne d'IP génère ainsi plusieurs résultats qui peuvent être relus par le logiciel s'exécutant sur le microprocesseur. Concrètement, il est possible d'accéder aux résultats suivants :

— Une image de différences de gaussiennes, ou n'importe quelle autre image intermédiaire, que l'on peut sélectionner grâce à un registre dédié.
— La liste des points d'intérêt extraits et triés à n'importe quelle échelle.
— La liste des imagettes log-polaires associées à chaque point d'intérêt.

2.2 Architecture matérielle de vision

Cette section couvre la première échelle du détecteur présenté dans la section 2.1 consacrée au processus attentionnel. Il existe plusieurs projets de systèmes de vision matériels dans la littérature.

Un détecteur de points SIFT temps réel et parallèle a été proposé dans [Bonato et al, 2008]. L'architecture proposée est capable de détecter des points caractéristiques à une cadence allant jusqu'à 30 images par secondes pour une taille de 320×240. Le système, complètement déployé sur FPGA, est configuré pour traiter trois octaves de cinq échelles, et travaille sur une imagette de taille 5×5 autour des points d'intérêt. Nous avons observé dans nos expérimentations que cela limitait la détection de points robustes. Ce système de vision a été appliqué à un mécanisme de Simultaneous Localization And Mapping (SLAM) sur un robot mobile, où ce dernier construit une représentation de son environnement à partir des imagettes extraites [Bonato et al, 2006].

Des architectures ont également été portés sur FPGA dans le cadre de l'algorithme Speeded Up Robust Features (SURF). Plusieurs approches [Schaeferling, 2010; Bouris et al, 2010; Battez-

zati et al, 2012] ont été validées et comparées sur différentes applications, comme la mesure de déformation d'objets ou le suivi de véhicule. Les résultats montrent que l'on peut développer une architecture efficace à partir de ce détecteur. Cependant, l'étape de mise en correspondance des caractéristiques est coûteuse et limite le comportement temps-réel de la méthode. Les résultats montrent des performances allant de 5ms à 340ms par image selon que l'on utilise ou non l'étape de mise en correspondance.

L'architecture proposée dans [Birem and Berry, 2012] est basé sur le détecteur de *Harris and Stephen* qui détecte, sélectionne et trie des points caractéristiques en temps réel. La méthode adaptative utilisée pour le seuillage des points est intéressante. La variation de la luminance induit une forte variation dans le nombre de points d'intérêts détectés. Les auteurs utilisent un seuil adaptatif pour assurer qu'un nombre suffisant de points soit détécté. Ils proposent également une architecture pour trier les points d'intérêts. Cette méthode nécessite que tous les points soient extraits et s'effectue donc après avoir traité l'image. L'architecture est capable de détecter les points à une cadence de 156 images par secondes pour une taille de 512×512.

Les auteurs de [Zhong et al, 2013] proposent une autre approche plus flexible. Elle est basée sur un système embarqué construit autour d'un FPGA et d'un Digital Signal Processor (DSP), ce qui permet d'allier la puissance brute fournie par le parallélisme du FPGA et la flexibilité apportée par la programmabilité du DSP. L'implémentation matérielle de la pyramide de Gaussiennes est similaire à celle que nous proposons. Nous ne nous focalisons cependant pas sur la flexibilité, mais tentons d'avoir une architecture avec l'empreinte la plus réduite.

Les rétines artificielles permettent de réaliser des traitements bas niveau de manière très économe en énergie, en plaçant des traitements analogiques au plus proche du capteur. Une caméra-sur-puce a été proposée dans [Musa et al, 2012] dans un contexte de filtrage d'images. Elle est constituée d'un capteur CMOS de 64×64 pixels et d'unité de calcul analogique capable d'extraire les minimum et maximum sur un voisinage 2×2. Le circuit complet montre une consommation d'une centaine de milli-watt. Une autre caméra-sur-puce a été proposée dans [Paindavoine et al, 2015] pour faciliter l'implémentation de l'algorithme HMAX. Le modèle HMAX est inspiré du système visuel ventral du macaque et s'applique à la reconaissance d'objets. Il consiste à filtrer les images avec des filtres directionnels, puis de trouver les maxima locaux dans ces images filtrées. Ces informations sont alors traitées par un réseau de neurones artificiel. La caméra-sur-puce effectue, sous une consommation de quelques milli-watt les deux première opérations, le filtrage et la détection des maxima locaux.

Le système de vision matériel bio-inspiré présenté dans ce chapitre est constitué d'une caméra standard dont les images sont traitées par une série de traitements. Ces traitements visent à mimer *a posteriori* le comportement des saccades occulaire observé chez les mammifères. Les rétines artificielles sont des capteurs dont la structure même est directement inspirée de la biologie. Par exemple, le capteur conçu dans [Berton et al, 2006] a une structure log-polaire, grâce à un effort mené sur la forme et l'organisation des pixels. Une résolution centrale plus grande qu'à la périphérie permet un bon compromis entre la précision et la densité des données de sortie. Pour une utilisation robotique, ce capteur pourrait être motorisé pour mimer plus finement le mécanismes de saccades occulaire.

Le capteur CurvACE proposé dans [Colonnier et al, 2015] est inspiré du fonctionnement de l'œil des mouches. Il dispose nativement d'une faible résolution mais est capable de reconstruire des images d'une grande résolution grâce à un mécanisme de vibrations. Grâce à une consommation réduite, ce capteur a pu être intégré sur des drones volant et a permis de mettre en œuvre des tâches d'odométrie visuelle.

L'organisation de notre architecture est représentée dans la FIGURE 2.3. Comme les blocs fonctionnels sont configurables en fonction des dimensions de l'image souhaitée, le reste de la chaîne visuelle correspond à de multiples instances de cette première échelle.

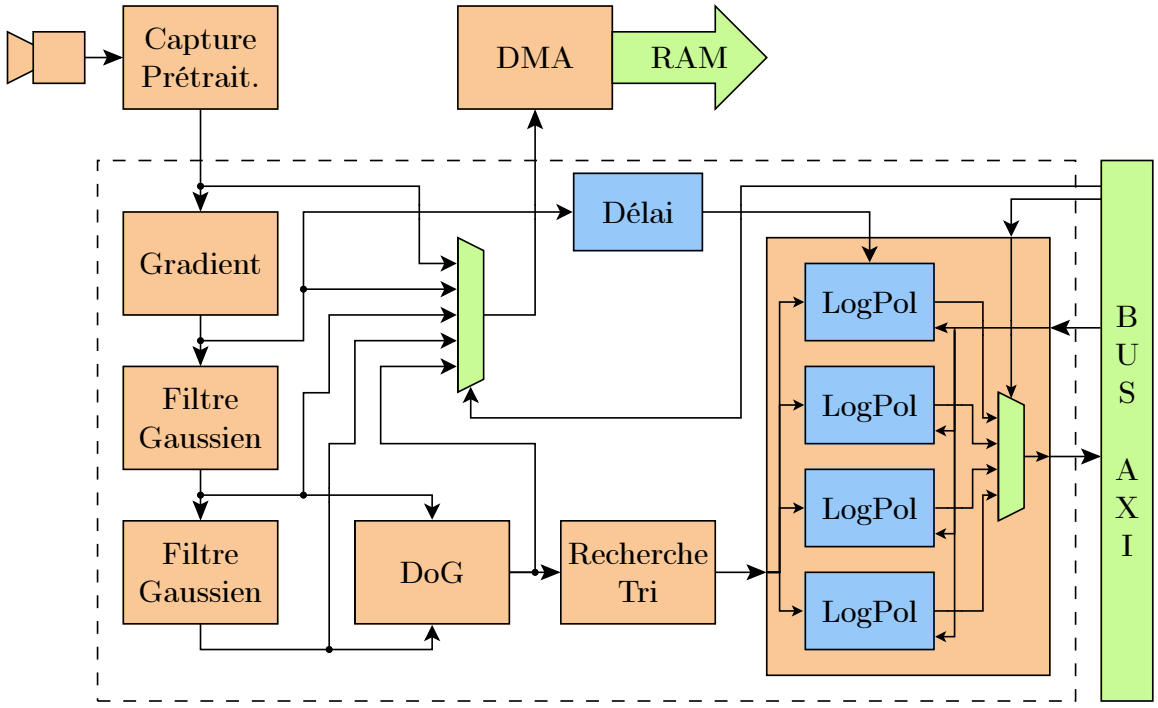


FIGURE 2.3 – **Vue globale de l'architecture de l'IP pour une échelle.** Le flux de pixels arrive de la caméra, circule à travers l'IP et va vers la mémoire du microprocesseur à travers un DMA. Une sortie intermédiaire peut être sélectionnée à l'aide d'un registre dédié. Un autre registre permet de sélectionner un point d'intérêt ainsi que son imagette. Enfin, ils peuvent être lus au moyen d'une interface *mappée* en mémoire.

L'architecture est composée d'un ensemble d'IP, décrites en VHDL. Elle prends pour entrée le flux de pixels provenant de la caméra, grâce à une interface de *streaming*. Dans notre prototype, il s'agit d'une interface `AXI Streaming` (voir [Xilinx, 2011]).

L'architecture de vision – conçue pour être associée à un microprocesseur – peut être configurée par le biais d'une interface *mappée* en mémoire.

La plupart des IP sont construites selon un modèle simplifié de type flot de donnée. Ils consomment et produisent des pixels en entrée et en sortie. Nous avons donc développé une interface standardisée pour connecter ces IP entre elles.

Les coordonnées des pixels sont nécessaires à chaque IP pour prendre en compte les bords des images. De plus, l'apprentissage des cellules de vues pour la navigation à besoin de connaître les coordonnées des points d'intérêts. Les coordonnées des pixels doivent ainsi être transférées d'une IP à l'autre. Pour prendre en compte les effets de la latence et garder la cohérence du pipeline, les coordonnées doivent, soit être retardées, soit être re-générées.

Compte tenu de ces contraintes, notre interface est composée des signaux suivants :

— La valeur de la luminance du pixel,
— ses coordonnées (x, y) dans l'image,
— un signal *enable* indicant qu'un pixel est valide.

2.2.1 Le gradient

La magnitude du gradient est calculée à partir d'une version simplifiée de l'opérateur de Sobel.

Soit I l'image d'entrée, G_x et G_y représentent respectivement les dérivées horizontales et verticales. Ces images intermédiaires sont construites par convolution avec les noyaux classiques de l'opérateur de Sobel (eq. 2.1).

$$G_x = I * \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix}, G_y = I * \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2.1}$$

La principale différence avec la méthode de Sobel classique réside dans le calcul de l'image de magnitude du gradient G. En effet, nous avons remplacé le calcul d'une racine carrée, coûteux en surface occupée sur notre circuit, par une somme de valeurs absolues (eq. 2.2).

$$G = abs(G_x) + abs(G_y) \tag{2.2}$$

L'architecture utilisée pour calculer la magnitude du gradient est représentée sur la FIGURE 2.4. Elle est composée de deux parties, la première étant responsable de la mémorisation des pixels d'entrée, la deuxième s'occupe du calcul en lui-même.

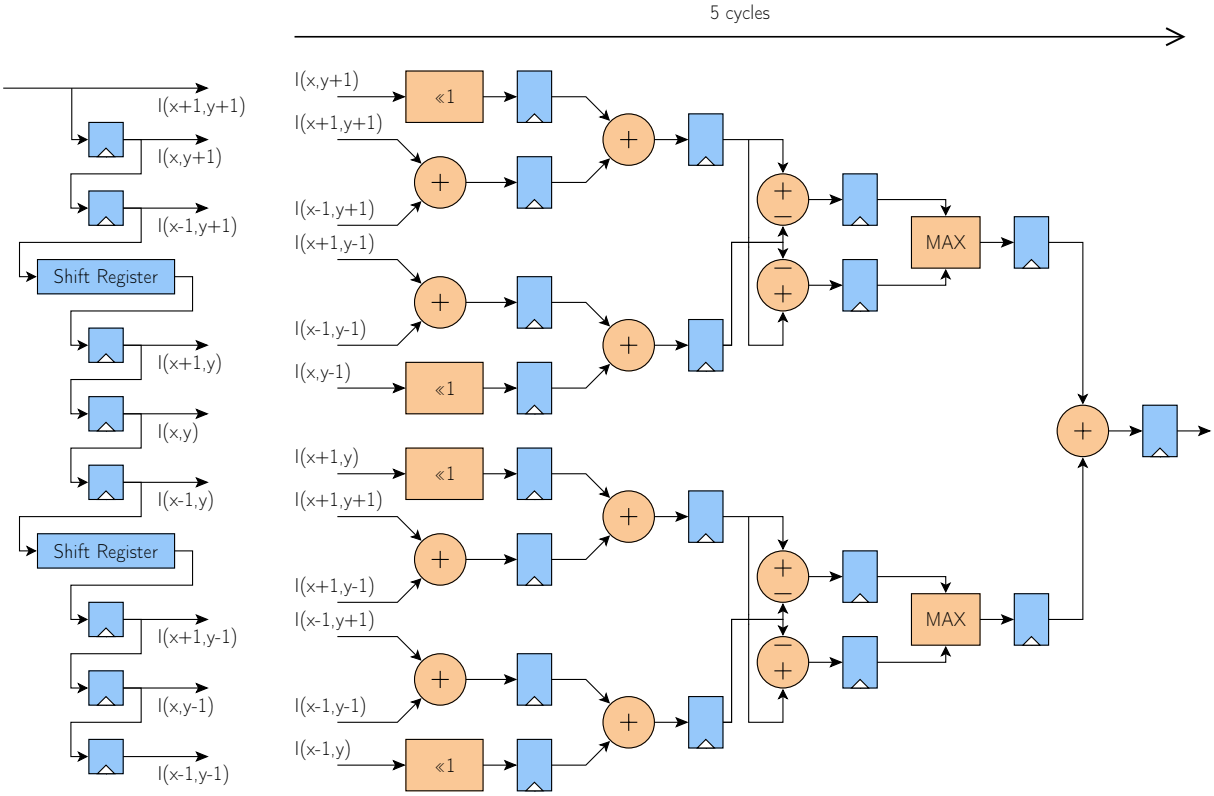


FIGURE 2.4 – **Architecture de l'IP de calcul de la magnitude du gradient.** La gestion des pixels d'entrée est montrée à gauche, le calcul en lui-même est représenté à droite.

Les pixels d'entrée sont d'abord stockés dans une structure composée de mémoires de type First-In First-Out (FIFO) et dans des registres classiques. À chaque arrivée d'un pixel valide,

la structure de mémoire enregistre le pixel et restitue les 8 pixels requis pour la structure de calcul.

La structure de l'opérateur, représentant les calculs présentés aux eq. (2.1) et (2.2), est construite sous forme de pipeline. Ceci permet d'augmenter la fréquence du circuit et donc le débit de pixel en ajoutant quelques cycles de latence. On constate en effet que les 5 cycles supplémentaires apportés par le calcul du gradient ne sont pas significatifs lorsqu'on les compare à la latence imposée par la mémorisation des pixels d'entrée. Finalement, la latence totale se traduit, en nombre de cycles d'horloge, par la formule suivante $I_W + 6$, où I_W représente la largeur de l'image.

Une attention particulière doit être prise lors du traitement des bords de l'image. Pour que le pixel de sortie ne soit calculé qu'avec des pixels valides, nous utilisons une méthode classique consistant à remplacer, dans eq. (2.1), les pixels en dehors de l'image par la valeur zéro. La détection de ces pixels invalides s'effectue simplement par les coordonnées du pixel entrant et par la connaissance des paramètres génériques I_W et I_H (respectivement la largeur et la hauteur de l'image). Cette gestion des effets de bords atténue la magnitude du gradient à la bordure de l'image, mais n'affecte pas les autres pixels. Nous verrons par la suite que cette atténuation n'a que peu d'impact sur le reste de la chaîne.

2.2.2 Le filtre Gaussien

L'opération de filtrage Gaussien consiste à convoluer l'image par la fonction Gaussienne bidimensionnelle de l'equation 2.3 :

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.3}$$

Cette fonction est calculée à la synthèse et stockée dans un tableau de taille 9×9, car au-delà la fonction est nulle après quantification sur une précision de 16 bits.

L'architecture de l'IP est représentée sur la FIGURE 2.5 sur un noyau 3×3. Cette structure est capable de produire un pixel à chaque cycle d'horloge pour suivre le rythme imposé par la caméra.

La convolution bidimensionnelle peut être séparée en deux convolutions monodimensionelles, ce qui a pour effet d'économiser quelques multiplieurs. On peut voir dans la partie supérieure de la FIGURE 2.5a la convolution verticale, la convolution horizontale étant représentée juste en dessous.

Nous pouvons encore gagner quelques multiplieurs en tirant parti du fait que la fonction Gaussienne soit paire. Ainsi, les multiplieurs peuvent être *factorisés* comme dans la FIGURE 2.5b.

Comme dans le cas du gradient, il y a des précautions à prendre aux bordures de l'image. Cependant, ce point est plus problématique ici, compte tenu de la taille du noyau plus importante. Il existe plusieurs solutions pour prendre en compte les effets de bords.

Une solution pourrait être de réduire les dimensions de l'image de sortie et ainsi ne travailler qu'avec des pixels valides. Étant donnée la quantité de filtres Gaussiens utilisés dans la pyramide, la résolution serait trop faible, notamment dans les basses échelles.

La solution consiste donc à réduire la corruption introduite par les pixels non valides. Voyons les différentes approches :

— Les pixels invalides peuvent être mis à leur valeur maximale, ce qui rendrait les bords plus brillant.

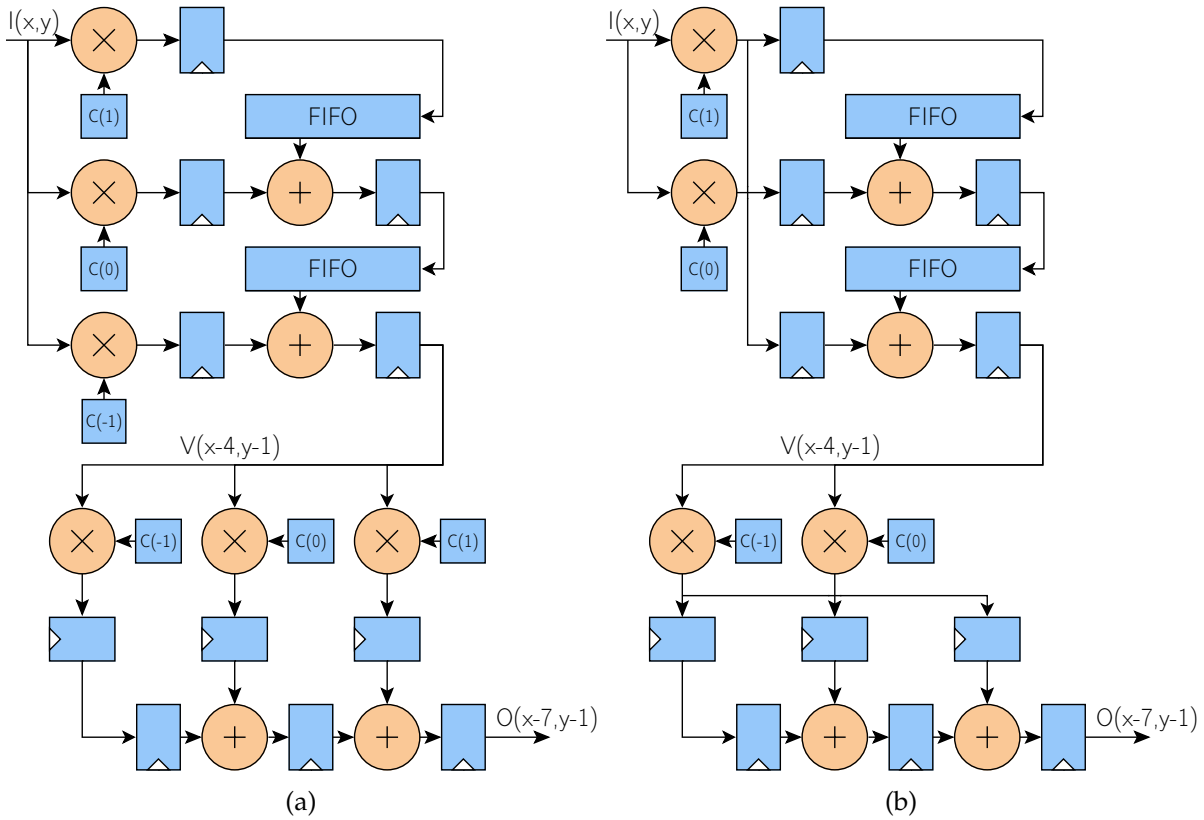


FIGURE 2.5 – **Architecture du filtre Gaussien.** En (a), la convolution bidimentionnelle est séparée en deux convolutions monodimensionnelles. En (b), les multiplieurs redondants son supprimés.

— Ils peuvent être mis à la valeur minimale, ce qui rendrait évidemment les bords plus sombres.
— Ils peuvent être mis à la valeur médiane, ce qui limiterait leur impact.
— Une dernière solution consisterait à effectuer une opération mirroir sur les bordures, mais cela compliquerait drastiquement l'IP, la rendant plus lourde.

Pour prendre une decision, nous devons prendre en compte les particularités de l'application. Avant tout, le premier filtre Gaussien prend pour entrée l'image de magnitude de gradient. Ce type d'images est habituellement sombre, sauf pour un environnement visuel riche en points saillants. Nous devons garder à l'esprit que les points saillants seront triés par intensité et que les points les plus faibles ne seront pas conservés. Ainsi, une bordure plus lumineuse risque de mener à la détection de faux points d'intérêt. La solution que nous retenons donc est d'assombrir artificiellement les bordures.

2.2.3 La différence de Gaussiennes

La différence de Gaussiennes est le module le plus simple de cette chaine. Il est simplement constitué d'un soustracteur. Il faut quand même prendre en compte la latence d'un filtre Gaussien, puisque la différence de Gaussiennes prend en entrée la sortie de deux filtres Gaussiens successifs.

Nous avons donc rajouté une mémoire FIFO pour synchroniser les deux flux de pixels en-

trants. Le système est représenté en FIGURE 2.6.

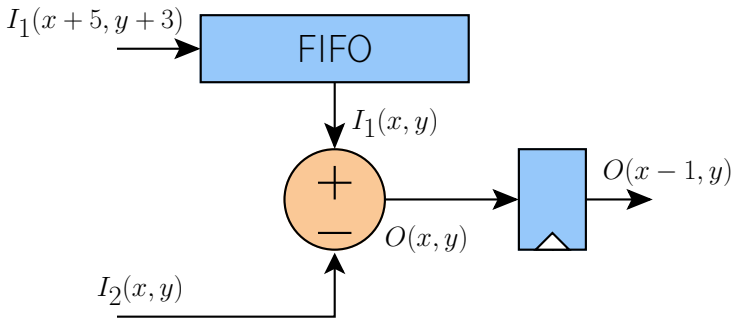


FIGURE 2.6 – **Architecture de l'IP de calcul de différence de Gaussiennes.** Une mémoire FIFO assure la synchronisation des deux flux de pixels entrants. La différence est calculée à partir d'un simple module soustracteur.

2.2.4 La recherche des points d'intérêt

L'approche classique

L'algorithme de recherche des points d'intérêt consiste à trouver les maxima locaux dans les images de différences de Gaussiennes. Pour chaque pixel, l'algorithme vérifie, dans un disque de rayon R, si le pixel est supérieur aux autres dans le but de déterminer s'il est le maximum dans cette zone.

L'IP travaille de façon similaire à un opérateur de convolution classique, mais avec un noyau circulaire. Pour respecter le rythme de la caméra, un pixel doit être traité à chaque cycle. On peut voir l'architecture de l'IP en FIGURE 2.7.

Un pixel doit satisfaire à quatre critères pour être identifié comme point d'interêt :

— Il doit être maximum local, tel que décrit ci-dessus ;

— Il ne doit pas y avoir d'autres points d'intérêt dans le disque de détection. Comme le demi-disque situé au dessus du pixel à traiter est déjà testé, il est possible de s'assurer que cette contrainte est respectée (voir FIGURE 2.8). Ce bloc permet également de choisir différents rayons pour la détection et l'inhibition.

— La valeur d'un point d'intérêt potentiel doit également être supérieure à un seuil de bruit γ. Ceci empêche des points d'intérêt d'être détectés dans des zones monotones.

— Le point doit être suffisament éloigné d'un bord de l'image afin que tous les pixels du disque soient valides. Les coordonnées (x, y) des pixels à traiter peuvent être utilisées à cette fin (voir FIGURE 2.9).

Une approche optimisée

On pourra voir plus loin, en section 2.2.7, que l'architecture classique présentée ci-dessus est un frein à la scalabilité à cause du disque de comparateurs dont la consommation en ressources est proportionnelle au carré de son rayon.

Pour éviter cette limite de scalabilité, une solution consiste à chercher d'abord les maxima horizontaux, puis à chercher verticalement le maximum parmi ces maxima. Cette solution considère cependant une fenêtre de recherche carrée, ce qui ne semble pas affecter la reconnaissance en aval.

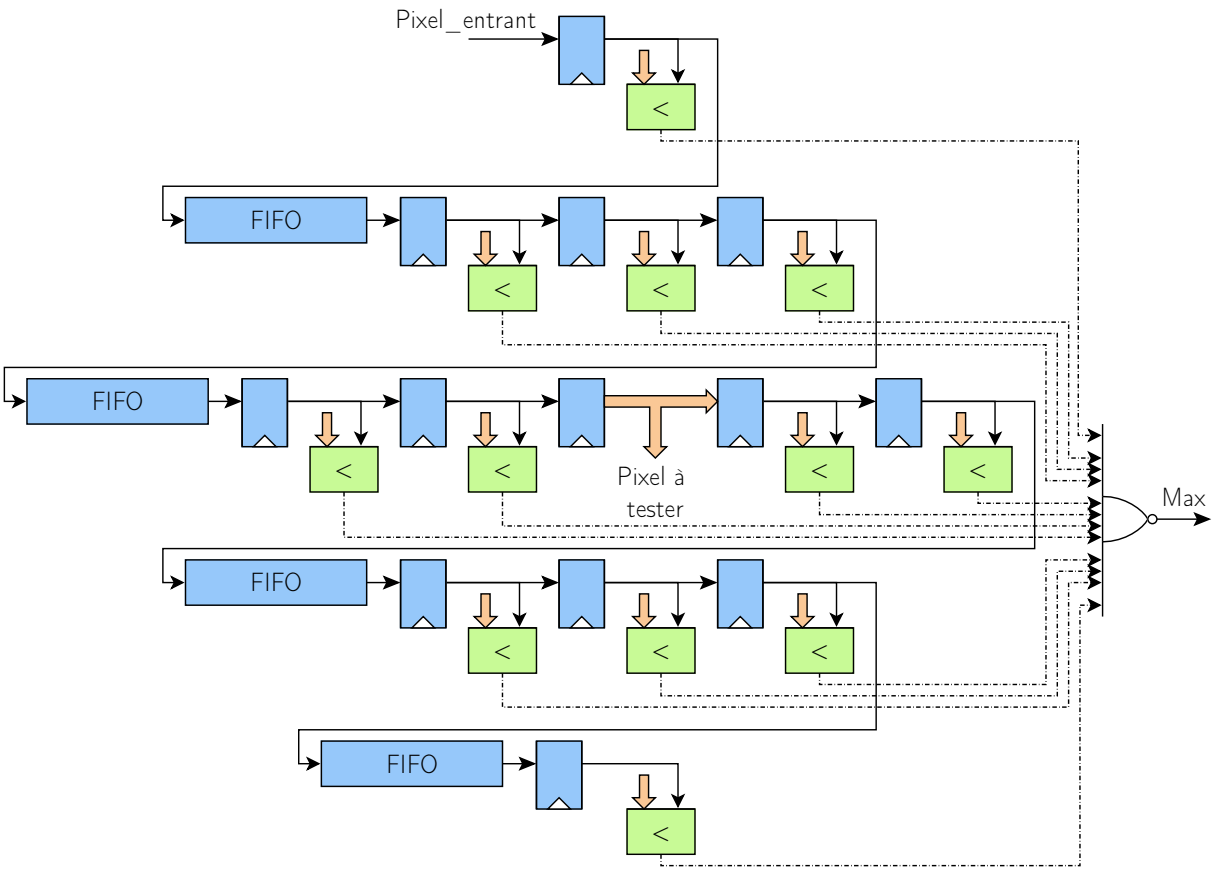


FIGURE 2.7 – **Architecture de l'IP de recherche des points d'intérêt :** détection des maxima locaux illustré pour un rayon de taille 3.

L'algorithme de recherche horizontale est implémenté sous la forme d'un arbre de comparateurs (voir FIGURE 2.10a). Les maxima sont alors stockés dans une mémoire FIFO. La coordonnée x de ces maxima doit également être mémorisée.

La recherche horizontale et la recherche verticale ne sont pas conçues de la même manière. La recherche horizontale doit déterminer la valeur et la position du maximum sur la ligne. La recherche verticale, quant à elle, indique si la ligne contenant le plus grand maximum horizontal est au centre. L'architecture verticale est représentée en FIGURE 2.10b.

La détection des maxima locaux ayant été optimisée, les trois autres critères, à savoir la compétition entre les maxima, la gestion des effets de bords, et le seuil de bruit, doivent toujours être satisfaits. Nous utilisons sans modifications les architectectures présentées dans la section précédente.

2.2.5 Le tri des points d'intérêt

En continuant à suivre la chaîne de traitement de la FIGURE 2.3, l'IP de tri prends pour entrée les points d'intérêt détectés par l'IP de recherche. Ils sont représentés à l'intérieur du module de tri par une structure contenant les coordonnées (x, y) ainsi que de leur intensité. Un index est rajouté à cette structure pour prendre en compte la transformation log-polaire. L'objet de cet index sera détaillé plus loin en section 2.2.6. Ces structures sont triées selon l'intensité des points d'intérêt.

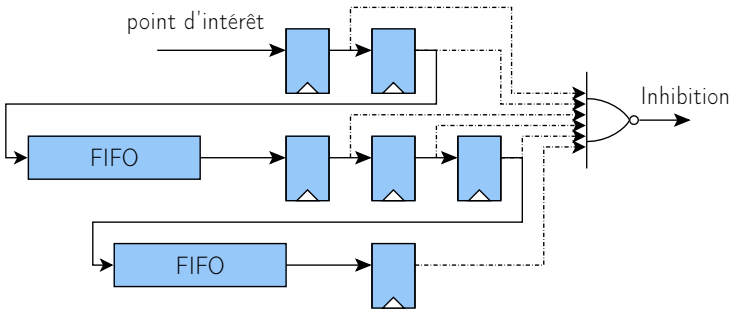


FIGURE 2.8 – **Architecture de l'IP de recherche des points d'intérêt :** inhibition des points voisins illustrée sur un noyau 3×3.

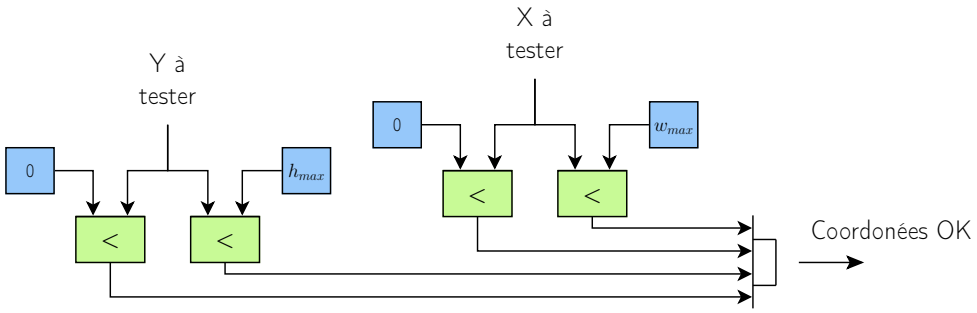


FIGURE 2.9 – **Architecture de l'IP de recherche des points d'intérêt :** test des effets de bords.

Une fois de plus, l'IP doit respecter la cadence imposée par le capteur. Nous considérons donc qu'un seul pixel puisse être produit par cycle d'horloge.

Considérons qu'à tout instant, la liste des points d'intérêt – représentée par un ensemble de structures décrites précédemment, stockées dans des registres – soit triée. Lorsqu'un nouveau point entre dans le module, son intensité est comparée avec celles des points d'intérêt contenus dans la liste triée. On sait alors dans quel registre insérer le nouveau point d'intérêt. Les points inférieurs sont alors décalés vers le bas de la liste, et le dernier est supprimé.

Pour plus de détails, voir la FIGURE 2.11.

2.2.6 La transformée log-polaire

Le module de transformée log-polaire est constitué de N sous-blocs, N étant le nombre de points d'intérêt maximum détectable dans une image. Chaque sous-bloc est responsable de l'extraction et de la transformée d'une imagette. Ce module peut donc traîter plusieurs imagettes en parallèle. Lorsqu'un point est abandonné au niveau de l'IP de tri des points d'intérêt, le sous-bloc correspondant s'interrompt, et commence à traiter l'imagette correspondant au nouveau point. L'index introduit précédemment est utilisé pour déterminer quel sous-bloc de transformation correspond à quel indice dans la liste triée.

Chaque sous-bloc de transformée log-polaire est décomposé en deux sous-modules : le générateur d'adresses et la transformée en elle-même.

Le générateur d'adresses (voir FIGURE 2.12) convertit les coordonnées cartésiennes en coordonnées polaires. Un test s'assure que le flux d'entrée fait parti du disque à extraire, puis ses coordonnées sont soustraites à celles du flux d'entrée. Cette opération a pour but de replacer l'origine du repère local par rapport à l'imagette à extraire. Les coordonnées log-polaires sont

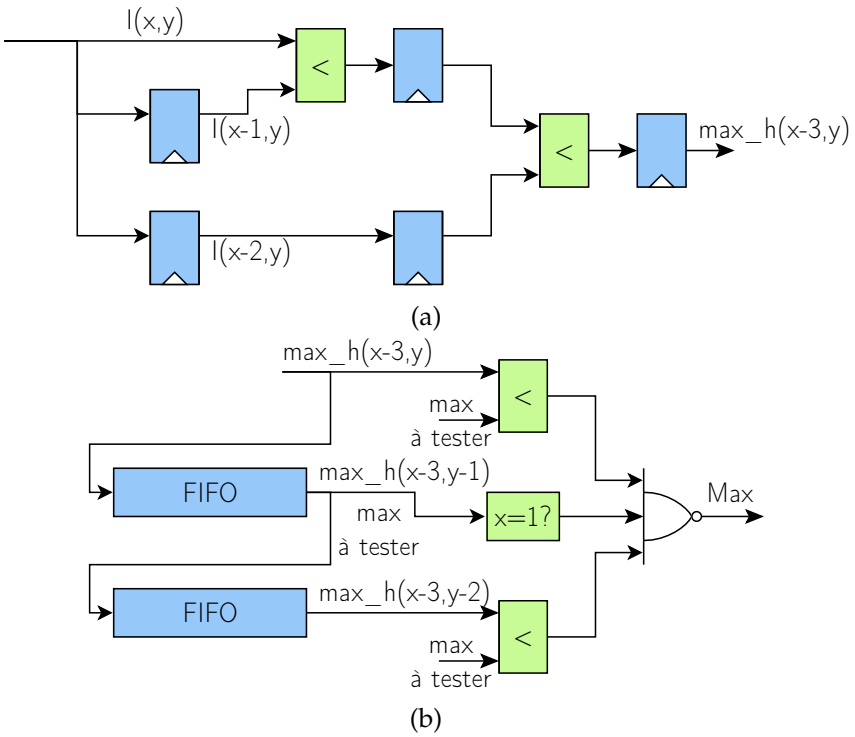


FIGURE 2.10 – **Architecture optimisée de l'IP de recherche des maxima locaux.** La version représentée ici travaille sur une fenêtre de taille 3×3. En (a), le maximum horizontal est trouvé, puis envoyé, avec sa position à (b). En (b), le maximum central est testé. S'il est maximal et si sa position correspond au centre de la fenêtre, il est proposé comme point d'intérêt candidat.

alors générées par une table de vérité.

Le module de transformation (FIGURE 2.13) reçoit quant à lui le flux de pixels issu du gradient ainsi que les coordonnées log-polaires du générateur d'adresses. Comme on peut le voir en FIGURE 2.3, le flux de pixels du gradient est synchronisé aux adresses générées au moyen d'une mémoire FIFO. Les pixels sont stockés dans une mémoire à la case correspondant à l'adresse représentée par les coordonnées log-polaires. Un pixel dans l'espace log-polaire peut être construit à partir de la moyenne de plusieurs pixels de l'espace cartésien. Un accumulateur est construit autour de la mémoire en tirant partie du double port des mémoires des FPGA modernes, la division étant effectuée à la lecture de l'imagette. Le module de transformation est constitué de deux mémoires et d'un jeu de multiplexeurs pour permettre le travail dans une mémoire pendant la relecture de l'autre. Un banc de registres mémorise l'intensité des points d'intérêt, ses coordonnées ainsi que la relation entre son rang dans la liste et l'index de son module de transformation.

L'architecture complète du module d'extraction et de transformée log-polaire est détaillée en FIGURE 2.14.

Finalement, un processeur peut lire les points d'intérêt et leur imagette associée à travers une interface mappée en mémoire, compatible AXI. Un registre d'index permet de choisir quel points d'intérêt relire. Un jeu de registres permet d'accéder à la valeur et aux coordonnées du point d'intérêt sélectionné. La suite de la plage d'adresse est directement reliée à la mémoire contenant l'imagette liée au point d'intérêt indiqué par l'index.

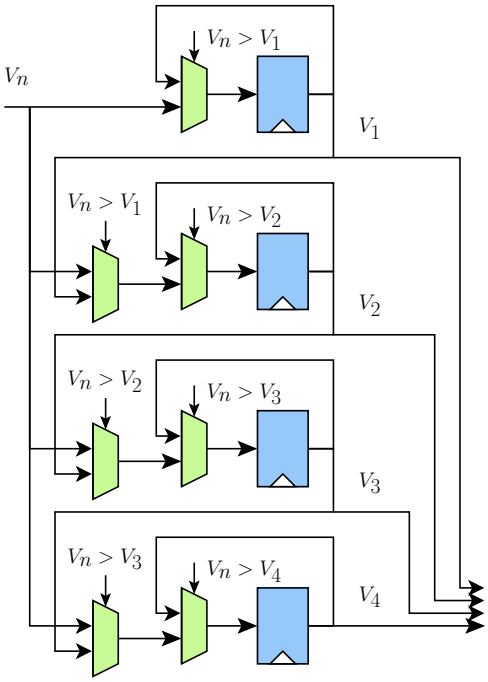


FIGURE 2.11 – **Architecture de l'IP de tri des points d'intérêt.** Une liste de quatre structures {x, y, index, intensité} est triée selon les intensités.

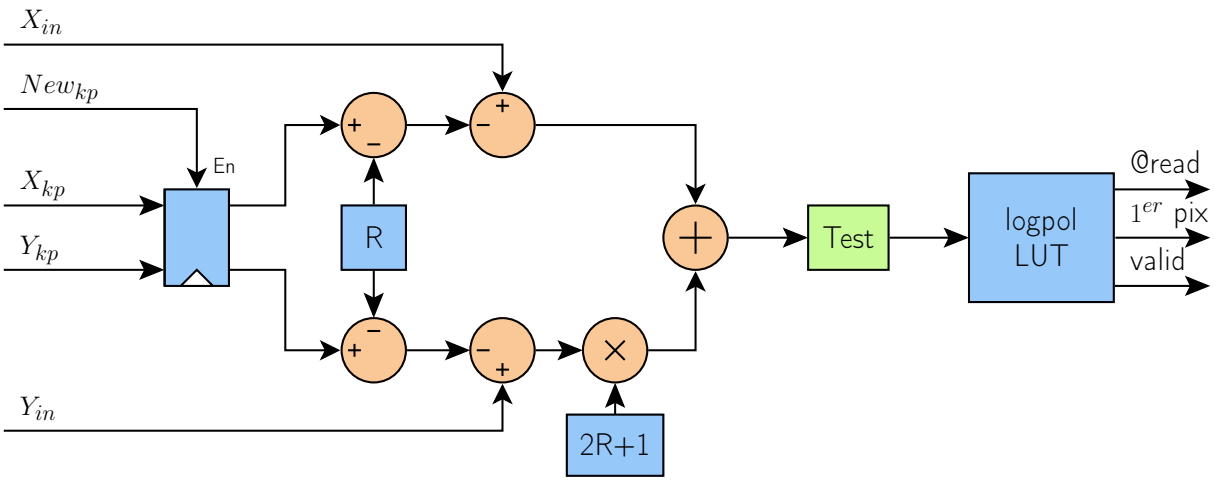


FIGURE 2.12 – **Architecture de l'IP de transformation log-polaire :** le générateur d'adresses.

2.2.7 Résultats d'implémentation

L'implémentation matérielle de l'échelle la plus basse a mené à une architecture fonctionnelle. Cette architecture est capable de traiter les images provenant du capteur à un rythme de 60 images par seconde. Un exemple de sortie de la chaine de traitement est donné en FIGURE 2.15. Les résultats détaillés de cette implémentation sont décrits dans les paragraphes suivants.

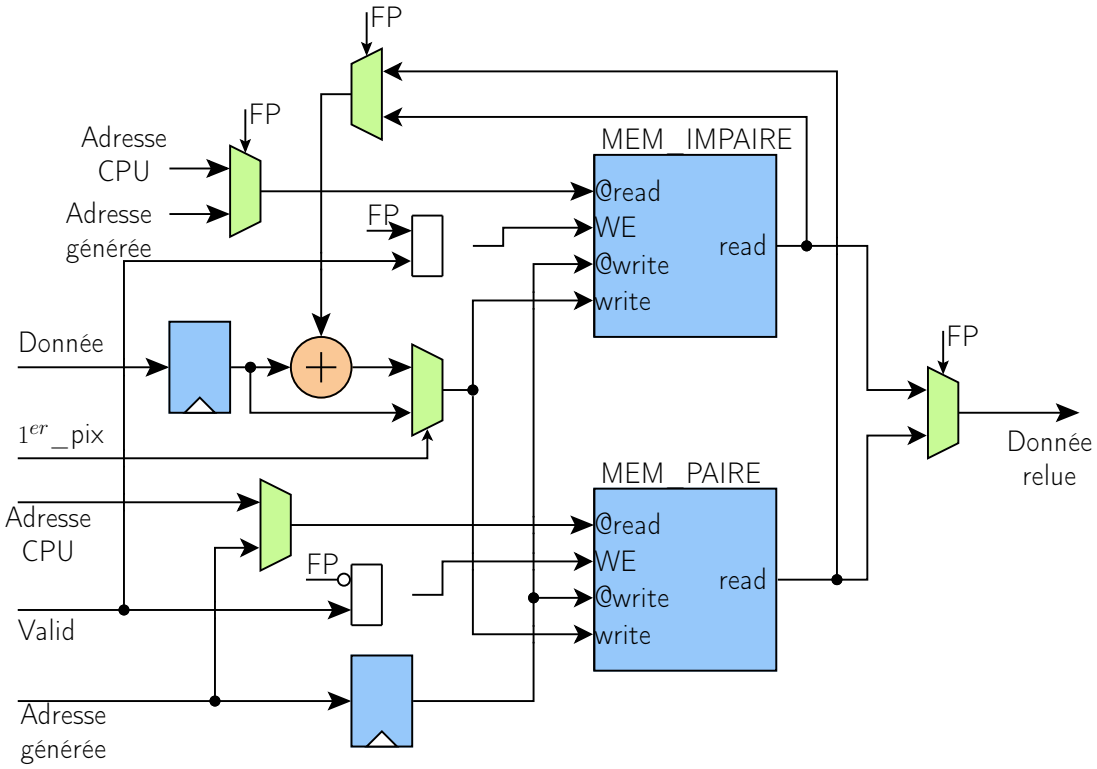


FIGURE 2.13 – **Architecture de l'IP de transformation log-polaire :** la transformation en elle-même.

Consommation en ressources matérielles

Nous explorons dans cette section l'influence des paramètres de l'architecture de vision sur la consommation en ressources matérielles. Cette étude nous permet de vérifier la scalabilité de l'architecture et de savoir quels sous-composants optimiser. On pourra également estimer la faisabilité d'un passage d'une seule échelle à un système multi-échelle.

Nous avons mené cette étude sur le FPGA Zynq 7020 de Xilinx qui est constitué d'un processeur dual-core ARM9 couplé à une matrice FPGA, constituée de 106 k registres, 53 k Look-Up Table (LUT), 560 kiB de mémoire distribuée et de 220 blocs DSP. Nous avons effectué une série de synthèses en faisant varier différents paramètres, comme la résolution, le nombre de points d'intérêt à détecter et le rayon de recherche des points. Les résultats présentés ci-dessous sont obtenus après synthèse et avant la phase de placement-routage. En outre, nous ne montrons que les ressources occupées par l'architecture de vision, le prétraitement des pixels et l'interconnexion de l'IP avec le processeur n'apparaissent pas.

Nous avons fait varier le rayon de recherche et le nombre de points d'intérêt pour les résolutions 1920×1080, 960×540 et 480×270 correspondant à la résolution FullHD et à ses sous-échantillonnages. Nous avons également comparé les deux méthodes de recherche des points d'intérêt discutées en section 2.2.4.

L'impact du nombre de points d'intérêt est montré en FIGURE 2.16. Le rayon d'extraction, est fixé à 12, ce qui semble être la limite en utilisant l'architecture de recherche classique. On peut voir que l'augmentation du nombre de points d'intérêt a une influence linéaire sur la consommation en ressources, même avec la méthode de recherche classique.

L'impact du rayon de recherche est montré en FIGURE 2.17. On constate sur la FIGURE 2.17a,

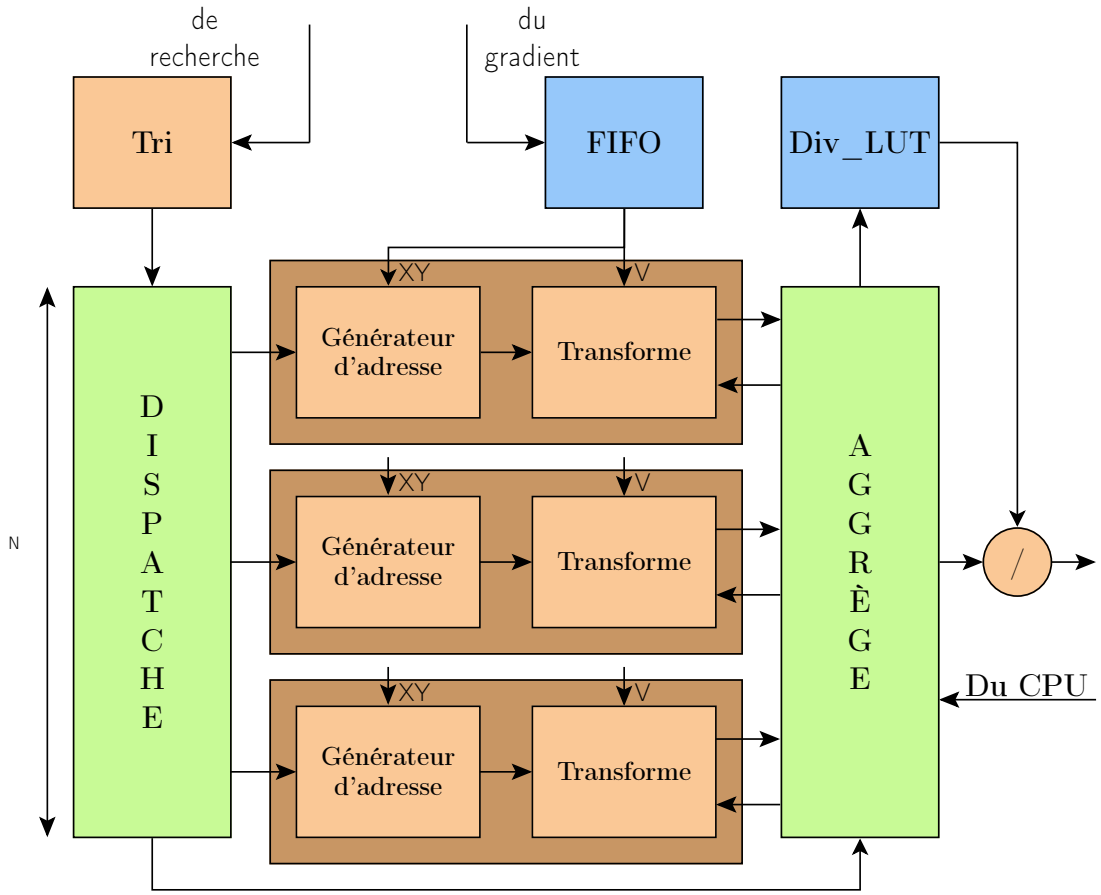


FIGURE 2.14 – **Architecture de l'IP de tranformation log-polaire.** N imagettes sont extraites et transformées en parallèle.

qu'avec la méthode classique la consommation augmente de manière quadratique, ce qui s'explique par la présence d'un disque de comparateur. En revanche, l'approche optimisée permet une consommation linéaire, comme montré en FIGURE 2.17b.

L'impact de la longueur rayon sur l'IP de recherche est montré en FIGURE 2.18 où l'on peut comparer les deux approches. On constate que la consommation de l'IP optimisée reste linéaire quel que soit le rayon de recherche.

On peut remarquer que sur toutes ces figures, seule l'utilisation mémoire est affectée par la résolution des images. En effet, la taille des différents noyaux reste identique quelle que soit la dimension de l'image. En revanche, les lignes de pixels à stocker sont plus longues et demandent ainsi plus de mémoire.

Finalement, nous avons tracé en FIGURE 2.19 la consommation de chaque sous-IP pour déterminer lesquelles étaient les plus gourmandes. Les paramètres sont fixés de la manière suivante :

— Résolution 470×270,
— 16 points d'intérêt,
— rayon d'extraction de 12 pixels.

Tout d'abord, notons que la colonne *gauss* correspond aux deux filtres Gaussiens. On peut remarquer une fois encore le gain obtenu par l'IP de recherche optimisée.

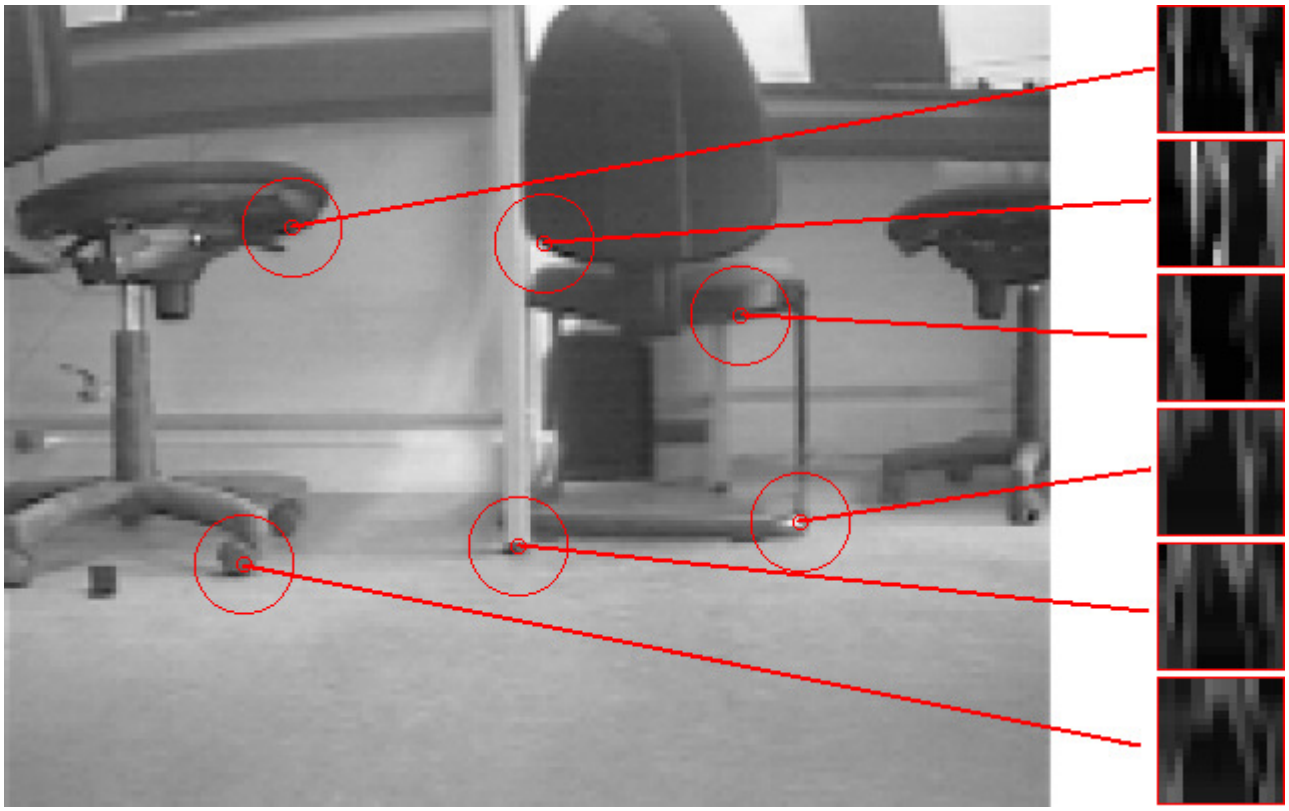


FIGURE 2.15 – **Détection et traîtement de six points d'intérêts.**

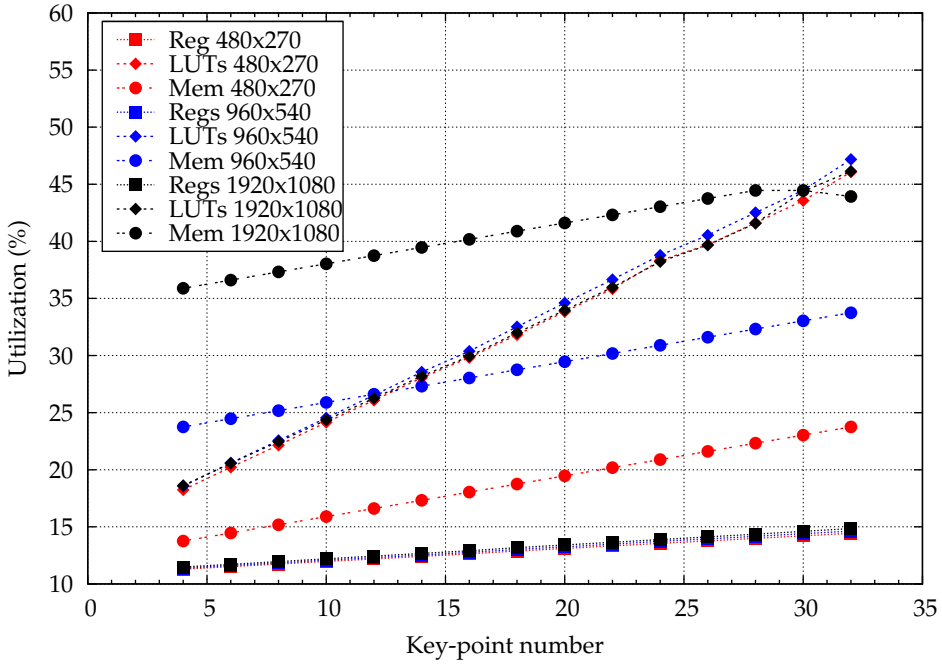


FIGURE 2.16 – **Consommation matérielle en fonction du nombre de points d'intérêt** pour l'ensemble de la chaîne. Le rayon de recherche des points est fixé à 12 pixels.

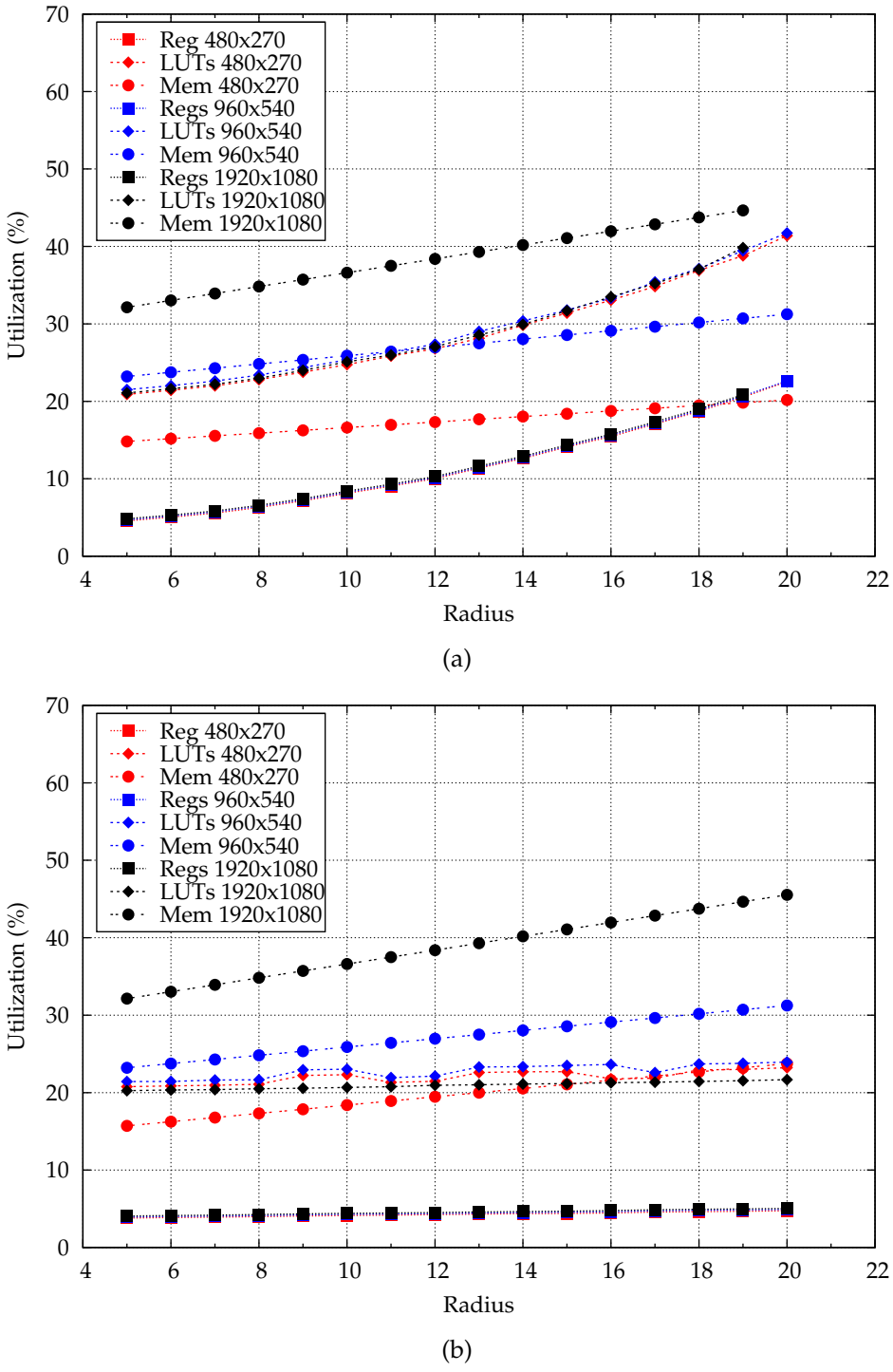


FIGURE 2.17 – **Consommation matérielle en fonction du rayon de recherche** pour l'ensemble de la chaîne. Les résultats pour la méthode classique sont montrés en (a) et les résultats de la méthode optimisée en (b). Le nombre de points d'intérêt est fixé à 16.

Enfin, on constate que l'IP de transformée log-polaire est la plus consommatrice en LUT et en mémoire. L'utilisation mémoire s'explique par le stockage des imagettes en parallèle, et

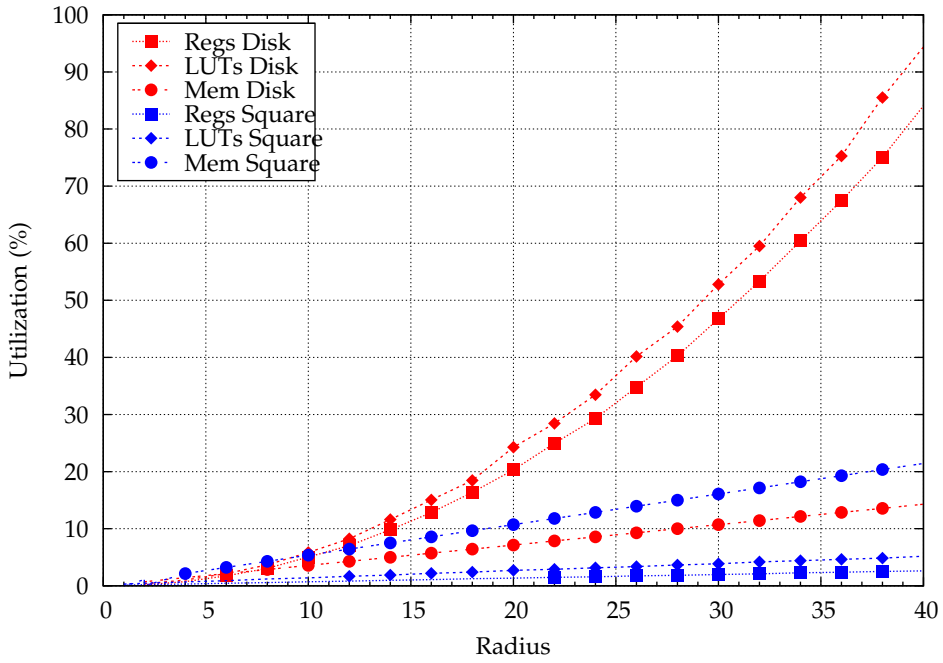


FIGURE 2.18 – **Comparaison des consommations des IP de recherche.** L'approche classique est représentée en rouge et l'approche optimisée en bleu.

l'utilisation en LUT par le fait que l'IP soit dupliquée autant de fois qu'il y a de points à détecter.

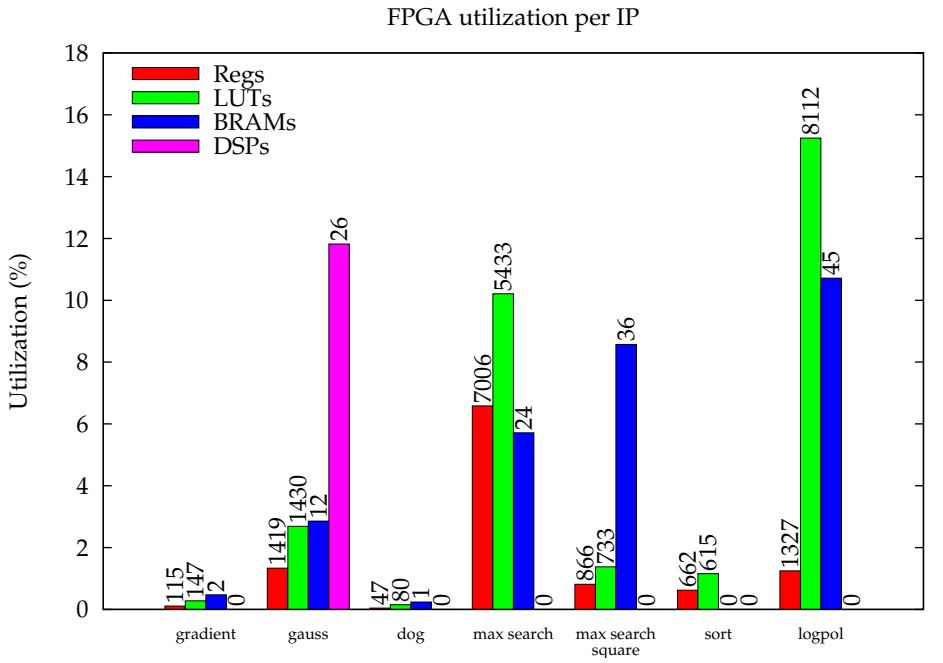


FIGURE 2.19 – **Consomation matérielle par sous-IP.** La quantité d'éléments consommés est affichée au dessus de chaque colonne.

L'architecture multi-échelle

Nous avons pu constater ci-dessus que malgré certaines optimisations, il était difficilement envisageable de traiter toutes les échelles de la pyramide sur le Zynq 7020. Cette limite est due d'une part à la manière parallèle d'extraire les imagettes log-polaires, qui consomme beaucoup de LUT. D'autre part, la mémoire nécessaire au stockage des pixels pour les différents noyaux est également importante.

Bien que certaines optimisations puissent être menées sur l'IP d'extraction et de transformée log-polaire, l'utilisation en mémoire reste incompressible. Nous envisageons d'utiliser un FPGA plus large, le Zynq 7045 disposant d'assez de mémoire pour héberger l'architecture complète.

Le taux de compression

L'un des principaux intérêt de ce système de vision est de diminuer la quantité de données à transmettre, notamment dans notre cas, au réseau de neurones. Une image de résolution 480×270 composée de pixels codés sur 10 bit est stockée sur 1266 kibit. L'architecture mono-échelle génère jusqu'à 16 imagettes de 16×16 pixels codés sur 16 bit, ce qui nécessite 64 kibit.

Nous calculons le taux de compression τ selon l'équation 2.4 :

$$\tau = \frac{\text{Taille du jeu d'imagettes}}{\text{Taille de l'image}} = 0.05 \tag{2.4}$$

La quantité de données transmises en sortie de la *smart*-caméra est réduite de manière significative. Nous pouvons extrapoler le taux de compression obtenu par l'architecture multi-échelle en multipliant ce résultat par le nombre d'échelles. Ainsi, pour une pyramide à 6 échelles, le taux de compression est de 30 %.

2.3 Navigation robotique basée sur la vision

Cette section couvre l'architecture neuronale qui contrôle le comportement de la plateforme robotique, à partir de l'information visuelle provenant de notre smart-caméra. L'architecture neuronale est basée sur des boucles *Perception-Action* (PerAc) pour l'association multimodale [P. Gaussier, 1995]. L'architecture PerAc modélise la perception par un processus dynamique liant la sensation avec l'action – dans notre cas la vision et le mouvement. Dans cette architecture, la perception du robot et son comportement résultent d'un couplage étroit entre les entrées visuelles et les actions motrices. Une boucle PerAc est la combinaison d'un chemin dit *réflexe* et d'un chemin dit de *catégorisation*.

En suivant une approche constructiviste, plusieurs architectures neuromimétiques de contrôle ont été proposées pour la navigation d'un robot mobile. Ces architectures, de complexité croissante, incluent plusieurs boucles PerAc [Maillard et al, 2005; Cuperlier et al, 2007]. Ces modèles sont inspirés de mécanismes d'auto-localisation et de navigation, retrouvés chez certains insectes tels que les abeilles, ou chez certains mammifères tels les rats ou les singes.

Ces architectures neuromimétiques s'appuyent sur les données provenant d'un système vision, très consommateurs de puissance de calcul, à tel point qu'il est difficile de les faire exécuter sur l'ordinateur embarqué d'un robot mobile. Ainsi, le robot doit accéder par une communication sans fil à un calculateur externe pour prendre en charge ces modèles complexes. Ce lien externe limite naturellement l'autonomie du robot.

Pour lever cette limitation, notre système sur puce de vision vient s'intégrer à la plateforme robotique, entre la caméra et l'ordinateur embarqué qui exécute le simulateur neuronal, Promethe (voir [P. Gaussier, 1995; Lagarde et al, 2008]). Pour valider notre SoC de vision, nous avons choisi d'évaluer la plateforme robotique avec une architecture de navigation PerAc simplifiée, basée sur une seule boucle sensori-motrice.

2.3.1 Le simulateur neuronal

Les réseaux de neurones décrits par la suite sont exécutés à l'aide du simuateur Promethe. Les neurones sont basés sur un modèle à fréquence de décharge. Les activités des neurones sont ainsi codés par des valeurs à virgule flottante, normalisées entre zéro et un. Les réseaux de neurones sont simulés selon un pas de temps t discret. À chaque pas de simulation, l'activité de tous les neurones est calculée, puis une phase d'apprentissage vient modifier le poids synaptique des connexions. Un système complet peut reposer sur plusieurs échelles temporelles, qui se comportent comme des boucles imbriquées. Ainsi, plusieurs pas à l'échelle de temps la plus basse s'effectuent pour chaque pas de l'échelle supérieure.

Dans cette architecture neuronale, l'apprentissage, réalisé en une fois, est dirigé par un signal spécifique appelé vigilance. Ce signal binaire global, représenté en FIGURE 2.20, est distribué à la quasi-totalité de nos réseaux de neurones. Quand ce signal vaut zéro, la valeur du poids des connexions synaptiques ne change pas. Quand il passe à un, il déclenche d'abord le recrutement d'un nouveau neurone dédié à l'apprentissage de l'entrée courante, puis autorise l'apprentisage. Le recrutement est réalisé en prenant séquentiellement le prochain neurone inutilisé du réseau. Les autres neurones ne modifient pas leurs poids synaptiques. L'équation d'apprentissage de chaque neurone suit le terme binaire V_k^{NN} dans le but de modéliser l'effet du filtrage qui controle quel neurone doit apprendre. Par exemple, dans chaque réseau de neurones, quand le signal de vigilance global vaut `1`, V_k^{NN} est aussi égal à `1` si et seulement si k est l'index du neurone recruté dans le réseau NN.

2.3.2 L'architecture de contrôle neuronale du robot

L'architecture présentée ici a pour but de permettre au robot d'effectuer de simples missions de navigation basées sur la vision, comme adopter un comportement de retour au nid. L'entrée visuelle est fournie par une caméra montée sur un servo-moteur, capable de prendre un panorama de l'environnement du robot.

Les neurones de cette couche modélisent des cellules de lieu découvertes dans le cerveau des mammifères. Les cellules de lieu ont un motif de décharge neuronale fortement corrélé à un lieu particulier et sont plus silencieuses lorsque l'animal est ailleurs [O'Keefe and Nadel, 1978]. Dans notre modèle, la couche des cellules de lieu est ainsi capable de caractériser et de reconnaître différents lieux dans l'environnement. Si le robot est à la position exacte où la cellule de lieu a été apprise, son activité est maximale. À l'inverse, quand le robot s'éloigne de cette position, l'activité de la cellule de lieu diminue en fonction de la distance entre la position apprise et la position courante. Une cellule de lieu garde ainsi une certaine quantité d'activité autour de la position apprise et correspond au champ de lieu de la cellule de lieu. Un *champ de lieu* est la projection dans l'environnement des lieux où une cellule de lieu donnée est active. Les unités sensori-motrices ont la capacité de généraliser, ce qui est une propriété intéressante de ce modèle, et peut être exploitée par des stratégies de navigation. Un mécanisme de compétition, le *Winner Takes All*, est utilisé pour sélectionner la cellule de lieu qui reconnait le mieux la position courante.

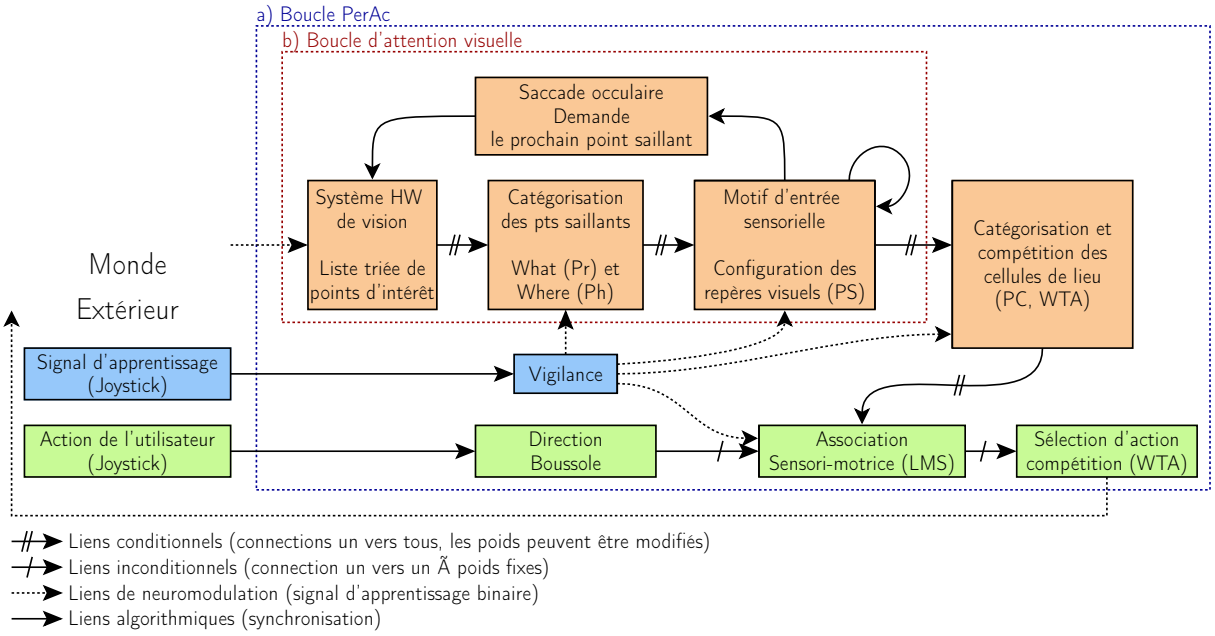


FIGURE 2.20 – **L'architecture PerAc de contrôle du robot.**
a) Boucle PerAc d'apprentissage sensori-moteur des associations entre le chemin moteur reflexe (en vert) et le chemin de catégorisation (en orange).
b) Boucle d'attention visuelle. Le calcul des repères visuels est effectué à une échelle temporelle plus fine que l'apprentissage neuronal. Cette échelle temporelle correspond à la cadence du système de vision matériel.

Notre modèle suit le concept de l'architecture PerAc, détaillé en FIGURE 2.20. L'architecture PerAc permet l'apprentissage en ligne d'associations sensori-motrices entre le chemin moteur et le chemin de catégorisation, c'est à dire la localisation. Ces éléments sensori-moteurs sont les briques de bases permettant de construire le comportement du robot. Ils codent les actions à accomplir dans un lieu – caractérisé par une situation perceptive – donné. La localisation courante résulte de l'analyse active de la scène visuelle, au moyen d'un mécanisme d'attention visuelle, représenté en FIGURE 2.20b. Ce mécanisme extrait et catégorise les caractéristiques saillantes – formées par les repères visuels – des images capturées. Cette étape d'apprentissage utilise plusieurs réseaux de neurones le long du chemin de caractérisation de notre modèle. Un réseau de neurones nommé *PS*, chargé de fusionner les repères visuels et leur azimuth dans la scène, est alimenté par ce mécanisme d'attention visuelle. L'activité du réseau de neurones *PS* forme une entrée sensorielle visuelle spécifique à une localisation donnée du robot dans son environement.

Le chemin de catégorisation de notre modèle se termine par un réseau de neurones, composé de cellules de lieu, qui apprend cette entrée visuelle. Ainsi, les cellules de lieu ont une activité corrélée avec la position spatiale du robot dans son environement. Finalement, le bloc d'association sensori-motrice pilote le comportement du robot. Ce réseau de neurones code les informations de localisation qui seront liées aux actions motrices attendues par le chemin réflexe.

Dans ce modèle simplifié, l'apprentissage du robot est supervisé par l'utilisateur. Les éléments sensori-moteurs pilotent le comportement du robot à l'aide d'opérations de conditionnement classiques, liant les actions désirées – maintenir une direction par exemple – à une position du robot spécifique. L'utilisateur doit alors, dans un premier lieu, positionner le robot à un endroit donné. Le robot est alors arrêté à la bonne orientation, ce qui a pour effet de coder l'action désirée. Le chemin réflexe lie de manière inconditionnelle l'orientation du robot au réseau de neurones sensori-moteur.

L'utilisateur active un signal de vigilance pour un pas de simulation neuronale, ce qui permet au modèle d'apprendre une nouvelle entrée sensorielle, correspondant à la position courante. Un nouveau neurone est recruté dans le réseau de neurones des cellules de lieu. Il catégorise la configuration spatiale des repères visuels détectés dans le panorama complet traité par le système visuel matériel. Ensuite, les neurones sensori-moteurs associent la cellule de lieu nouvellement recrutée avec l'action demandée par l'utilisateur, codée par l'orientation du robot.

Dans la phase de validation, il n'y a plus d'apprentissage, le signal de vigilance est désactivé, et les motifs des entrées sensorielles activent directement les cellules de lieu précédemment apprises en fonction de l'environnement perçu. À l'opposé de la phase d'apprentissage, le panorama de la scène visuelle est pris pendant le déplacement du robot. La cellule de lieu la mieux reconnue, celle qui gagne la phase de compétition, prédit le mieux la position courante. Elle déclenche ainsi les activités précédemment apprises par le réseau de neurones sensori-moteur. Le chemin de catégorisation peut alors prendre le contrôle des actions du robot et inhibe le chemin réflexe. Cette inhibition se déroule dans un réseau de neurones de sélection d'action à l'aide d'une compétition *Winner Takes All* (WTA).

Le mécanisme d'attention visuelle

Le chemin de catégorisation de cette architecture PerAc mène à la création d'entrées visuelles inspirées du traitement visuel des mammifères. En effet, les observations du système visuel des mammifères a mené à l'identification de deux chemins principaux : le *What ?* et le

Where ? [Goodale and Milner, 1992]. Le premier permet d'identifier les points caractéristiques, les repères visuels, trouvés par la rétine. Le second donne l'information quant à la position dans ces images, il s'agit de l'azimuth. En nous basant sur ces découvertes, et en suivant l'approche PerAc, nous avons défini dans notre modèle une entrée sensorielle provenant de la fusion de ces deux types d'informations.

Notre SoC de vision fournit une liste triée de points saillants aux réseaux de neurones qui les traitent l'un après l'autre au travers d'un mécanisme attentionnel. Ce mécanisme attentionnel catégorise à la fois l'identité des repères visuels (le *What ?*) et leur position angulaire par rapport au nord magnetique (le *Where ?*). Plus de détails sont donnés en FIGURE 2.20b. Chaque repère visuel traité pendant ce processus est accumulé dans une *product space matrix* (PS) codant la configuration des repères visuels extraits de la scène visuelle.

Ce mécanisme attentionnel se concentre alors sur les points saillants extraits par le système de vision. Pour chacun de ces points, deux processus se produisent en parallèle :

— Une catégorisation permettant de coder le repère visuel correspondant,
— Une position angulaire par rapport au nord, donnée par une boussole, est calculée pour ce point. Cet angle est codé par une population neuronale et une diffusion gausienne permet la généralisation.

La fusion de ces deux flux d'informations permet de coder la configuration spatiale des repères *via* une constellation de repères et de leur azimuth. Un signal de rétro-action inhibiteur permet de sélectionner le prochain point saillant, en formant une saccade occulaire.

Ce processus se déroule à une échelle de temps plus fine que le reste du réseau de neurones, puisque le mécanisme attentionnel doit itérer sur toutes les images correspondant à un même panorama.

Le *What ?* : la couche d'identité des repères visuels (Pr)

La couche des repères visuels est composée de neurones qui codent chacun pour une imagette autour d'un point saillant. Cette couche modélise l'information du *What ?* peut être codé dans le cortex perirhinal (Pr), ou dans d'autres zones du chemin visuel ventral du cortex temporal [Kolb and Tees, 1990]. Les neurones *Pr* sont tous connectés aux imagettes locales fournies par le système visuel matériel à travers une connexion *un vers tous* alterable. Un neurone de cette couche est connecté à tous les pixels d'une imagette. La modification du poids du lien d'un neurone k de cette couche est calculé selon l'équation 2.5 :

$$\Delta W_{k,ij}^{Pr} = I_{ij}(t) \cdot V_k^{Pr} \tag{2.5}$$

où $W_{k,ij}^{Pr}(t)$ est le poids du lien du pixel i,j au $k^{\text{ème}}$ neurone Pr. À l'initialisation, $W_{k,ij}^{Pr} = 0$. La quantité $I_{ij}(t)$ code la valeur du pixel (i,j) de l'imagette I à l'instant t. V_k^{Pr} représente le signal de vigilance décrit précédemment, et déclenche le recrutement d'un neurone. Ce nouveau neurone est le seul qui peut apprendre dans le réseau, ainsi $V_k^{Pr} = 1$ quand k est l'index du neurone recruté, sinon, $V_k^{Pr} = 0$.

La valeur de l'activité $X_k^{Pr}(t)$ du $k^{\text{ème}}$ neurone Pr à un instant t est calculé selon l'équation 2.6 :

$$X_k^{Pr}(t) = f^{RT}\left(1 - \frac{1}{N_I \cdot M_I} \sum_{i,j=1}^{N_I,M_I} \|W_{k,ij}^{Pr}(t) - I_{ij}(t)\|\right) \tag{2.6}$$

avec N_I et M_I respectivement les nombres de pixels sur les axes x et y d'une imagette. $W_{k,ij}^{Pr}(t)$ est le poids du lien entre le pixel i,j et la $k^{\text{ème}}$ cellule Pr. I_{ij} est la valeur du $ij^{\text{ème}}$ pixel de

l'imagette. Enfin, $f^{RT}(x) = \frac{1}{1-RT}[x - RT]^+$ est la fonction d'activation qui étend la dynamique de la sortie où RT est un seuil de reconnaissance. $[x]^+ = x$ si $x \geq 0$ et 0 sinon. D'avantage de détails sur l'impact de cette compétition peuvent être trouvés dans [Giovannangeli et al, 2006b].

Le *Where ?* : la couche d'azimuth des repères visuels (Ph)

La couche parahippocampale (Ph) utilise une population neuronale pour coder l'azimuth d'un point de focalisation, c'est à dire la direction absolue venant de la boussole magnétique de notre robot. Cette couche ne réalise pas d'apprentissage, puisqu'elle met en relation la valeur numérique du compas avec la population neuronale. Plus précisément, l'azimuth des repères visuels, $\theta(t)$, est calculé en applicant la valeur de la boussole magnétique au centre de l'image, et en décalant cette valeur en fonction de la position x du repère visuel dans l'image et de l'angle d'ouverture horizontal de la caméra, θ_c, suivant l'équation 2.7 :

$$\theta(t) = \theta_{boussole}(t) - \left(\frac{X_{max}}{2} - x\right) \cdot \frac{\theta_c}{X_{max}} \tag{2.7}$$

avec X_{max} la résolution horizontale de l'image utilisée.

Chaque neurone N_{Ph} de la couche Ph, ayant pour activité X_i^{Ph}, a une direction préférée pour laquelle sa fréquence de décharge est maximale. Cette fréquence diminue de manière monotone de un à zéro avec la distance angulaire entre sa direction préférée et la direction $\theta(t)$ du point de focalisation courant. L'activté d'un neurone Ph est donnée par l'équation 2.8 :

$$X_i^{Ph}(t) = f\left(\|2\pi \cdot \frac{i}{N_{Ph}} - \theta(t)\|\right) \tag{2.8}$$

avec f une fonction de diffusion latérale Gaussienne centrée autour du neurone N_i^{Ph} ayant pour effet de diminuer l'activité pour les angles proches. Le paramètre σ de la fonction gaussienne fixe la taille de la diffusion latérale.

L'entrée sensorielle visuelle : la couche *Product Space* (PS)

La fusion du *What ?*, porté par les repères visuels, et du *Where ?*, donné par l'azimuth, est réalisée par un réseau de neurones *sigma-pi* appelé *Product Spacer* (PS) Les neurones du PS sont actifs tant que toutes les imagettes des points saillants n'ont pas été explorées. Ils sont organisés sous forme de matrice dont le nombre de lignes est égale à N_{Pr}. Les neurones partagent ainsi le même index i. Le nombre de neurones dans Pr et le nombre de colonnes sont fixés à cinq, ce qui implique qu'il peut y avoir 5 orientations différentes pour un repère visuel donné, repérés par un index l. L'activité des neurones de cette matrice est notée $X_{il}^{PS}(t)$.

La matrice PS est connectée aux neurones Pr *via* une connexion *un vers voisinage* ayant des poids égaux à un. Un neurone actif dans la couche des repères visuels pré-active ainsi cinq neurones dans PS, correspondant au cinq azimuth possibles sous lesquels le robot peut voir le repère. La matrice PS est connectée aux neurones Ph à travers une connexion *un vers tous*.

L'activité des neurones PS est calculée en trois étapes. D'abord, l'activité maximale est recherchée parmi les neurones Pr ainsi que parmi les neurones Ph. Ensuite, le produit P_{il} de ces deux activités est calculé par l'équation 2.9 :

$$P_{il}(t) = \left(\max_{i \in N_{Pr}} X_i^{Pr}(t) \cdot W_{il,i}^{Pr-PS}\right) \cdot \left(\max_{j \in N_{Ph}} X_j^{Ph}(t) \cdot W_{il,j}^{Ph-PS}\right) \tag{2.9}$$

Enfin, le produit P_{il} est accumulé avec l'ancien produit provenant du précédant repère visuel de la même scène visuelle, selon l'équation 2.10 :

$$X_{il}^{PS}(t+1) = \left[X_{il}^{PS}(t) + P_{il}(t)\right]^{+} \tag{2.10}$$

Un neurone PS apprends à s'activer quand un repère visuel est reconu sous un certain angle. Cette activité décroit graduellement pour des angles proches. L'activité de tous le neurones PS est forcée à zéro au démarrage de la boucle attentionnelle lorsqu'un nouveau panorama est traité.

L'apprentissage est seulement réalisé sur les poids des liaisons reliant Ph à PS. Le poids est maximal pour l'angle sous lequel le repère visuel a été appris. L'apprentissage des poids suit l'équation 2.11 :

$$\begin{array}{l} W_{il,j}^{Ph-PS}(t) = X_i^{Pr}(t) \cdot X_j^{Ph}(t) \cdot V_{il}^{PS}(t) \\ i = argmax_{p \in N_{Pr}} \left(X_p^{Pr}(t)\right) \\ j = argmax_{q \in N_{Ph}} \left(X_q^{Ph}(t)\right) \end{array} \tag{2.11}$$

À l'initialisation, le poids $W_{il,j}^{Ph-PS}$ vaut zéro. Lors de la reconnaissance d'un repère visuel i, un nouveau neurone d'index il est recruté séquentiellement selon l'index l de la colonne de la matrice PS. Une fois de plus, l'apprentissage de $W_{il,j}^{Ph-PS}$ et le recrutement d'un nouveau neurone n'est effectué que lorsque le signal de vigilance V_{il}^{PS} vaut `1`.

La constellation des repères visuels sur PS, résultant du traitement de l'entrée visuelle, caractérise un lieu. Nous utilisons un réseau de neurones modélisant des *cellules de lieu* pour apprendre les motifs de l'activité au sein de la matrice PS.

La localisation du robot : la couche des cellules de lieu (PC)

Dans notre modèle, une cellule de lieu apprends à catégoriser un motif particulier d'activité de la matrice PS codant un lieu particulier. Chaque cellule de lieu est liée à tous les neurones de la matrice PS *via* des connexions *un vers tous* qui suivent une loi d'apprentissage Hebbienne, caractérisée par l'équation 2.12 :

$$\frac{dW_{k,il}^{PS-PC}(t)}{dt} = V_k^{PC}(t) \cdot X_{il}^{PS}(t) \cdot X_k^{PC}(t) \tag{2.12}$$

Le recrutement d'un nouveau neurone pour encoder un nouveau lieu se produit quand le signal de vigilance est mis à un. L'activité de ce neurone recruté à l'instant t est mise au maximum, c'est à dire $X_k^{PC}(t) = 1$.

L'activité de la $k^{\text{ème}}$ cellule de lieu résulte du calcul de la distance entre le motif de l'activité apprise par la matrice PS et son activité courante. Elle est exprimée selon l'équation 2.13 :

$$X_k^{PC}(t) = \frac{1}{W_j}\left(\sum_{il}^{N_{PS}} W_{k,il}^{PS-PC} \cdot X_{il}^{PS}(t)\right) \tag{2.13}$$

avec $W_j = \sum_{il}^{N_{PS}} W_{j,il}^{PS-PC}$.

2.3.3 La couche d'association sensori-motrice (SM)

En nous basant sur l'architecture PerAc [P. Gaussier, 1995; Maillard et al, 2005] nous avons défini la représentation de l'objectif du robot comme un processus dynamique liant ces sensations, c'est à dire la configuration des repères visuels codés par les cellules de lieu, et les actions

correspondantes. Dans notre modèle, les neurones d'association sensori-motrice apprennent à associer la cellule de lieu gagnante à une action, c'est à dire, dans notre cas, un cap à maintenir. Les éléments sensori-moteurs ainsi définis, peuvent être utilisés pour adopter un comportement robuste de retour au nid.

Chaque neurone sensori-moteur est lié à la fois au réseau de cellules de lieu *via* les connexions apprises et au réseau de neurones d'orientation à travers des connexions constantes. Le poids de ces connexions constantes est fixé à une valeur plus faible que ceux provenant des réseaux de neurones de catégorisation, après l'apprentissage. Les activités provenant de ces derniers peut ainsi outrepasser celles des réseaux d'orientation, une fois l'association sensori-motrice apprise.

L'activité $X_j^{SM}(t)$ du $j^{\text{ème}}$ neurone de ce réseau à l'instant t est calculé par l'équation 2.14 :

$$\begin{aligned} X_j^{SM}(t) &= \left(1 - V_j^{SM}(t)\right) \cdot M_j^{SM}(t) + V_j^{SM}(t) \cdot S_j^{SM}(t) \\ S_j^{SM}(t) &= X_k^{PC}(t) \cdot W_{j,k}^{PC-SM} \\ M_j^{SM}(t) &= X_j^{Orientation}(t) \cdot W_{j,j}^{Orientation-SM} \end{aligned} \tag{2.14}$$

L'apprentissage dans la couche sensori-motrice suit une loi d'apprentissage Hebbienne, donnée dans l'équation 2.15 :

$$\frac{dW_{j,k}^{PC-SM}(t)}{dt} = V_j^{SM} \cdot X_k^{PC}(t) \cdot X_j^{SM}(t) \tag{2.15}$$

Lors d'un comportement de retour au nid, l'action associée est caractérisée par la direction à suivre pour atteindre l'objectif, depuis la position courante. L'apprentissage de quatre associations sensori-motrices pointant vers l'objectif est alors suffisant pour permettre au robot de naviguer jusqu'à l'objectif. Ces quatre associations sensori-motrices forment un bassin d'attraction centré sur l'objectif.

Même si cette stratégie de navigation repose sur des apprentissages sensori-moteurs assez basiques, il permet néanmoins un comportement robuste en environnement intérieur ou exterieur [Gaussier et al, 2000, 2002].

Malgré tout, quand les tâches de navigation doivent être effectuées dans des environnements plus complexes, comme de multiples pièces et des corridors, ce modèle peut facilement être étendu pour s'appuyer sur une séquence d'unités sensori-motrices liant les cellules de lieu et les actions [Giovannangeli et al, 2006b]. Ce modèle peut aussi s'appuyer sur une planification en reliant des unités sensori-motrices pour décrire des chemins [Cuperlier et al, 2007; Hirel et al, 2011]. Cette approche de perception basée sur un bassin d'attraction sensori-moteur dynamique a également été appliqué à des problèmes de reconnaissance d'objets [Maillard et al, 2005].

La couche de sélection d'actions (WTA)

Ce réseau de neurones est composé d'un simple *Winner Takes All* qui sélectionne une seule action parmi celles proposées par la couche sensori-motrices à travers des connexions *un vers un*. Comme dans le réseau de neurones sensori-moteur, chaque neurone représente une action qui correspond à une orientation absolue que le robot doit adopter, à une vitesse constante.

2.3.4 Résultats comportementaux

Dans cette section, nous présenterons d'abord le setup expérimental, puis nous discuterons des résultats.

Setup expérimental pour la navigation en intérieur

Nous utilisons une plateforme robotique basée sur un Robulab de la société Robosoft, muni de 8 capteurs ultra-son, d'un PC embarqué implémentant les réseaux de neurones discutés en Section 2.3, de notre système de vision, et d'une bousolle magnétique indiquant l'orientation. Une représentation de la plateforme est donnée en FIGURE 2.21.

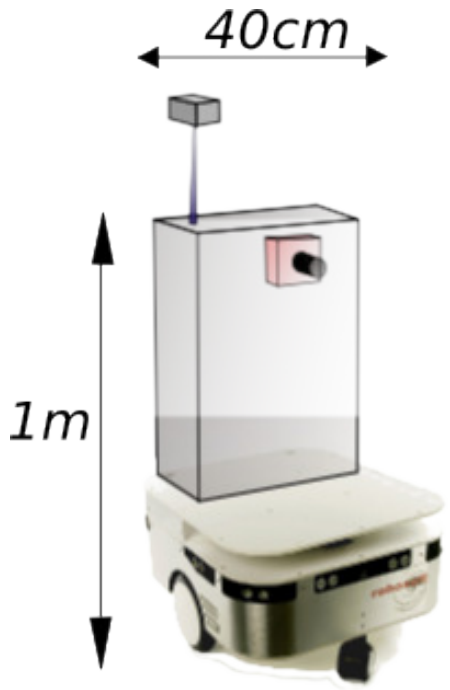


FIGURE 2.21 – **Plateforme robotique.** Nous utilisons un Robulab de la société Robosoft, intégrant notre système de vision matériel, une bousolle magnétique et un PC embarqué. Le robot mesure $40\ \mathrm{cm}$ de large et $1\ \mathrm{m}$ de haut.

Nous avons évalué le système de vision en l'intégrant à une architecture de contrôle chargée d'effectuer une mission de retour au nid. L'expérimentation est effectuée en deux phases, d'abord le robot apprends quatre cellules de lieu autour de son objectif. Chaque cellule de lieu est espacée de $2.4\ \mathrm{m}$ de ses deux voisines, et de $1.7\ \mathrm{m}$ de l'objectif. L'expérimentateur place le robot, à l'aide d'un joystick, sur l'un des lieux à apprendre, et l'oriente vers l'objectif. L'apprentissage des repères visuels et de l'association sensori-motrice est alors déclenchée par l'utilisateur. À chaque apprentissage des repères visuels, le robot prends un panorama complet de l'environnement.

Ensuite, le robot est placé à différents endroits dans l'environnement et est laissé en autonomie. Pendant cette phase de reconnaissance, les panoramas sont effectuées avec moins d'images que pendant la phase d'apprentissage. Ainsi, le système de vision du robot prends plusieurs images de l'environnement dans lesquelles il extrait les imagettes log-polaire des 16 points les plus saillants. Ces repères, associés à leurs orientations absolues, sont traités séquentiellement et intégrés par l'architecture neuronale. Ceci a pour effet de plus ou moins activer les neurones de la couche des cellules de lieu. Le degré d'activité de ces cellules code le taux de reconnaissance de la scène courante. Un mécanisme de compétition permet de

sélectionner la cellule de lieu qui code le mieux la position du robot, et de déclencher le mouvement correspondant. De ce fait, le comportement du robot, c'est à dire la direction qu'il prend, est uniquement dirigée par l'unité sensori-motrice la mieux reconnue.

Pour tester les performances de notre système, nous avons enregistré les trajectoires du robot depuis 8 points de départ autour de l'objectif. D'abord, nous testons si l'association sensori-motrice est correctement apprise en plaçant le robot environ 60 cm derrière chacune des 4 cellules de lieu apprises. Ensuite, nous testons les propriétés de généralisation des cellules de lieu en plaçant le robot sur 4 autres points situés entre les cellules de lieu.

Résultats

Les tracés obtenus lors de la première expérimentation sont donnés en FIGURE 2.22. L'association sensori-motrice est correctement apprise et permet au robot d'atteindre l'objectif quand il est placé proche d'une cellule de lieu apprise. Durant les trajectoires des trois premières cellules de lieu (PC1 à PC3), une seule cellule a été reconnue. Pendant la trajectoire correspondant à PC4, la cellule PC3 a été brièvement reconnue. Cette reconnaissance peut être expliquée par le fait que la quatrième cellule a été apprise près d'un bureau et était basée sur des repères visuels proche. Ces points provoquent de grandes variations pour de faibles mouvements, ce qui impacte les propriétés de généralisation des cellules de lieu.

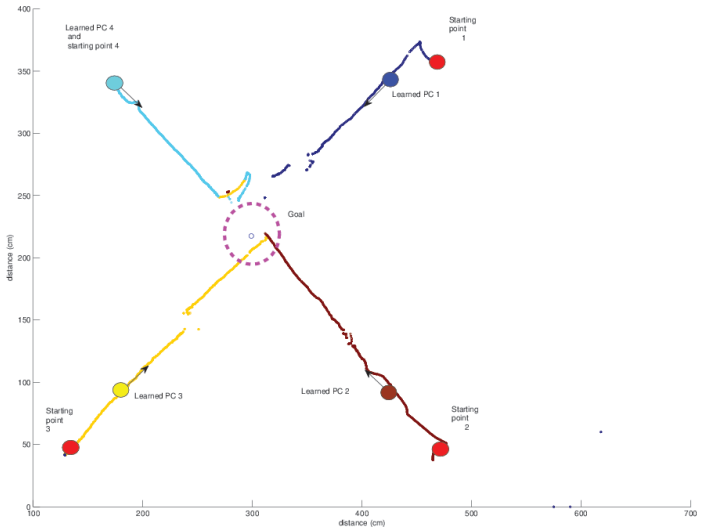
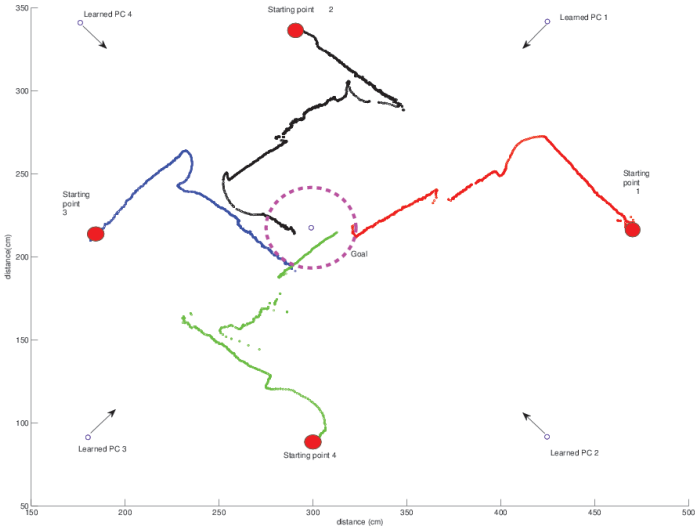


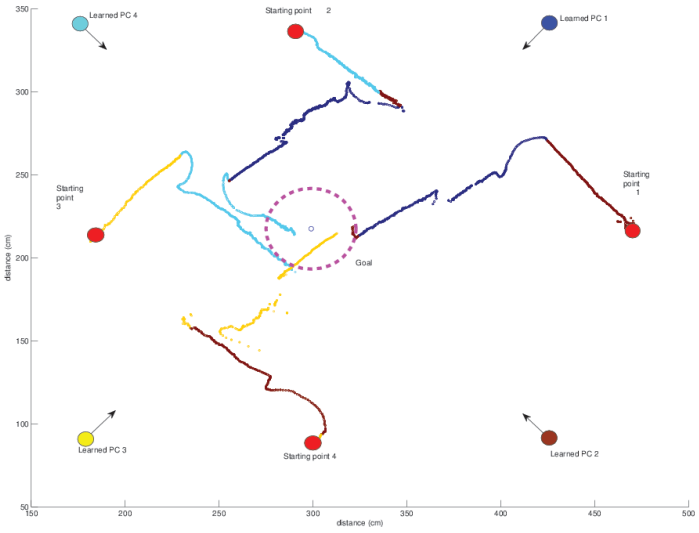
FIGURE 2.22 – **Résultat du retour au nid depuis quatre points derrière les cellules de lieu.** Les positions d'apprentissage des cellules de lieu sont représentés par un cercle de couleur, et les points de départs sont représentés par un cercle rouge. La couleur des cellules de lieu est superposée au tracé de la trajectoire du robot.

La FIGURE 2.23 montre la trajectoire du robot depuis 4 différentes positions proches des frontières entre les champs de lieu. Les trajectoires sont composées de différentes phases au cours des quelles le robot suit la direction de la cellule de lieu la mieux reconnue. À un moment donné, la cellule la plus active change ce qui provoque un changement de direction de la part du robot.

Ces expérimentations mettent en évidence les mécanismes de généralisation des cellules de



(a)



(b)

FIGURE 2.23 – **Résultats du retour au nid depuis quatre points proche des frontières entre les champs de lieu.** En (a), les trajectoires sont représentées en une seule couleur pour plus de lisibilité. En (b), la couleur de la cellule de lieu est superposée au tracé de la trajectoire.

lieu. Au cours de ces manipulations, le rayon de chaque champ de lieu est d'environ 1.2 m. Le robot n'a besoin que de seulement 4 cellules de lieu pour naviguer dans une zone de plus de 16 m^2.

2.4 Discussions et perspectives

Dans ce chapitre, nous avons présenté un système de vision matériel bio-inspiré. Grâce à un travail d'optimisation, il a été implanté sur une carte FPGA ZC702 et traite en temps réel le flux d'image provenant d'une caméra en résolution 480×270. Ce système de vision est conçu pour extraire l'information saillante dans l'environnement visuel d'un robot mobile autonome. Le système de vision couplé au réseau de neurones forme un modèle d'attention visuel et lie la sensation à l'action.

Cette validation sur robot mobile, avec une architecture neuronale déjà en place est la première étape d'une intégration à l'architecture SATURN. Le système de vision jouera ainsi le rôle d'un des pré-traitements pour la couche d'auto-adaptation.

Pour permettre à ce système de vision d'adresser un plus large éventail d'applications, nous envisageons d'implémenter la pyramide complète et ainsi de travailler sur plusieurs échelles. Pour se faire, nous travaillons sur une carte à FPGA plus puissante, la ZC706. Un système de contrainte entre les basses et les hautes échelles est à l'étude pour réaliser un traitement plus efficace.

CHAPITRE 3

Carte auto-organisatrice matérielle

Sommaire

3.1 L'auto-organisation au sein de l'architecture

3.1.1 Fonctionnement d'une carte auto-organisatrice

Les cartes auto-organisatrices sont une classe particulière de réseaux de neurones non-supervisés. Elles sont utilisées pour caractériser un environnement, c'est à dire mettre en évidence la répartition statistique des données dans un espace.

Une carte auto-organisatrice se présente sous la forme d'une grille, comme montré en FIGURE 3.1a. Les nœuds représentent les neurones, et les arrêtes représentent les connexions entre les neurones : les synapses.

Un neurone est caractérisé par ses poids internes qui varient avec le temps, en fonction des données d'entrée. Les neurones d'une carte auto-organisatrice sont souvent représentés

dans l'espace des entrées, c'est à dire que leurs coordonnées dans l'espace sont codées par la valeur de leurs poids. Les poids de chaque neurone sont initialisés de manière aléatoire, comme illustré dans la FIGURE 3.1b.

Pour un stimuli d'entrée donné, les neurones sont plus ou moins activés en fonction de la distance de leurs poids au vecteur d'entrée. Le neurone dont les poids sont les plus proches du vecteur d'entrée est considéré comme vainqueur et se rapproche d'autant plus du vecteur d'entrée. Il attire alors les neurones proches, dans l'espace topologique, vers lui.

Ceci a pour effet de regrouper les neurones pour former des clusters autour des stimuli les plus fréquents. Dans le cas de l'architecture SATURN, les clusters ainsi formés représentent différentes modalités perçues dans l'environnement, comme la vidéo, l'audio, ou des informations proprioceptive par exemple. La formation des clusters est représentée en FIGURE 3.1c.

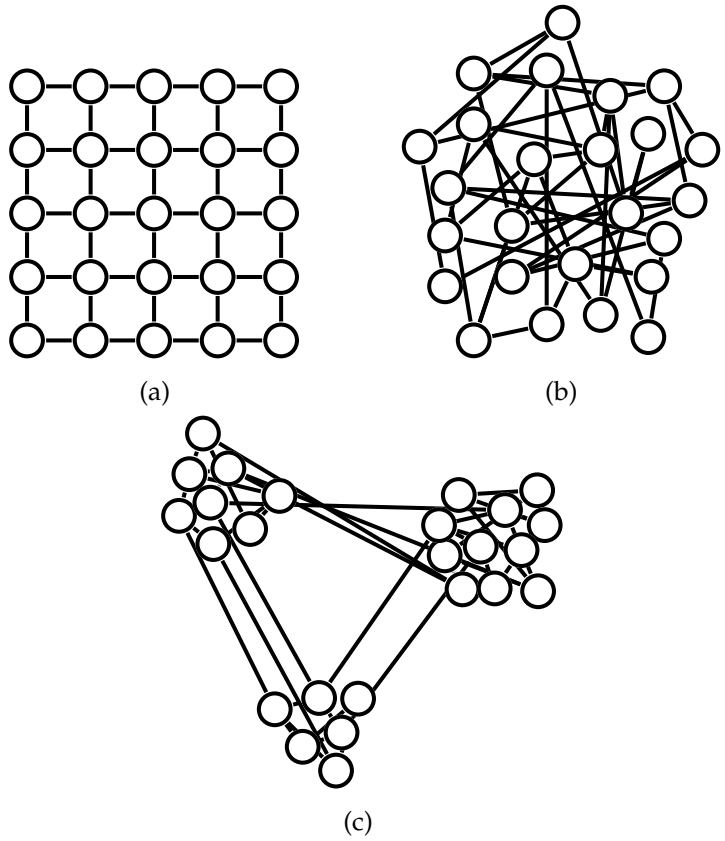


FIGURE 3.1 – **Représentations d'une carte auto-organisatrice.** a) Représentation topographique d'une grille 2D de neurones. b) Représentation dans l'espace des entrées d'un réseau de neurones à l'initialisation. c) Après plusieurs itérations, on observe l'apparition de clusters de neurones, qui représentent la densité des données.

Dans notre cas d'étude, chaque élément de calcul de la couche programmable de l'architecture est associé à un neurone de la carte auto-organisatrice. Ainsi, l'appartenance d'un neurone à un cluster conditionne la modalité à traiter par son élément de calcul. Il définit de ce fait le choix de la tâche, matérielle ou logicielle, à effectuer.

Les caractéristiques des cartes auto-organisatrices énoncées ci-dessus, associées à des mécanismes de reconfiguration dynamique parallèle, apportent de la plasticité matérielle à

l'architecture.

3.1.2 Contraintes de conception et d'implémentation

Il existe plusieurs solutions dans la littérature pour résoudre le problème de l'estimation de densité de probabilité. On retrouve en particulier des méthodes basées sur des réseaux de neurones, telles les Self-Organizing Map (SOM) et les Dynamic Neural Field (DNF). Ces solutions peuvent être implémentées en matériel, comme dans [Appiah et al, 2009] par exemple, où les poids sont représentés sur des bus trois-états. Un réseau de 60 neurones est implémenté sur FPGA pour une tâche de détection de caractères sur des images binarisées.

On trouve également beaucoup de travaux sur le bus Address Event Representation (AER) [Emery et al, 2009], utilisé pour permettre la transmission de spikes entre neurones. Le projet décrit dans [Zamarreño-Ramos et al, 2013] utilise le réseau AER appliqué aux ConvNets pour simuler plusieurs centaines de milliers de neurones sur un FPGA Virtex 6 de Xilinx. Les auteurs de [Carrillo et al, 2013] proposent un réseau hierarchique permettant d'exécuter environ mille neurones à une cadence de plusieurs milliers d'itérations par seconde.

D'autre part, plusieurs projets cherchent à construire des cartes neuronales de plus en plus grandes pour s'approcher de la complexité du cerveau Humain. Parmi ces projets, on citera le projet SpiNNaker [Furber and Temple, 2007] constitué de plusieurs milliers de processeurs ARM, connectés sur un réseau AER, et qui simulent chacun environ 1000 neurones. D'autres projets se basent sur des supercalculateurs, comme le *Blue Brain Project* [Markram, 2006] qui simule de manière précise une colonne corticale de rat, soit environ 10 000 neurones, ou le projet C2S2 SyNAPSE [Modha et al, 2011; Merolla et al, 2011] qui simule de manière moins précise le cortex du chat, soit environ 1,6 milliard de neurones ! Enfin le projet FACETS [Schemmel et al, 2010] simule 180 000 neurones de manière analogique sur un *wafer* complet.

L'ambition de ces projets est bien au-delà de la notre, et tranche avec notre besoin d'embarquabilité dans le cadre de la robotique mobile. Cependant, ils représentent un bon indicateur de la limite maximale actuelle en terme de complexité neuronale.

Notre approche adresse en effet un défi plus spécifique. D'abord le modèle neuronal est défini de telle sorte qu'il soit capable d'ordonancer des tâches sur une architecture many-cœur. Ensuite, notre architecture doit être complètement décentralisée pour pouvoir associer chaque neurone à un élément de calcul. Partant des limitations actuelles des technologies 2D-CMOS pour faire face à la complexité croissante des architectures massivement parallèles, nous cherchons à approximer ces méthodes au moyen d'un modèle neuronal à faible connectivité doté d'une règle d'apprentissage distribuée.

La carte auto-organisatrice doit suivre un certain nombre de contraintes, à la fois dans sa conception et dans son implémentation. De part la nature de l'architecture many-cœur de la couche programmable, elle doit être distribuée, puisque chaque élément de calcul dispose de son neurone. Il n'y a donc pas de contrôleur centralisé, chaque neurone participe localement au fonctionnement de la carte. Le fait que la carte soit distribuée pose directement le problème de la communication. En effet, il n'est pas envisageable de connecter tous les neurones entre eux. La carte auto-organisatrice doit donc être capable de fonctionner avec une connectivité la plus restreinte possible. Ensuite, le comportement de la carte doit faire preuve de dynamicité par rapport à l'environnement. Elle doit être capable d'absorber les modifications rapides, mais doit s'adapter aux modifications lentes de l'environnement. Enfin, on souhaite garder une indépendance entre les éléments de calcul programmable de l'architecture et les neurones de la carte auto-organisatrice. Les éléments de calculs étant reconfigurables, il n'est pas souhaitable

que les neurones partagent des ressources avec ces éléments.

Le mécanisme de fonctionnement des cartes auto-organisatrices, des cartes de Kohonen aux champs neuronaux, est basé sur la sélection du neurone dont la distance au stimulus d'entrée courant est la plus courte. Dans les travaux existants, le calcul de cette distance est réalisé au moyen d'un contrôleur centralisé, ou d'un graphe de connexions complet.

Ainsi, la couche d'auto-organisation de notre architecture est constituée d'une grille de neurones localement connectés. Un Processing Element (PE), pouvant être un processeur ou une surface reconfigurable, ainsi qu'un routeur permettant de diagloguer avec les autres PE sont associés à chaque neurone de la carte. L'ensemble de ces trois éléments est appelé une tuile.

L'allocation d'une tâche à un PE au sein d'une tuile est ainsi contrôlée par son neurone, en fonction de l'apprentissage distribué de ses poids d'entrée. Pendant la durée de la mission du robot, l'effet de l'apprentissage distribué sur l'architecture est de piloter l'évolution du nombre de neurones, et donc de PE, associé à chaque modalité.

3.2 Définition d'un modèle

D'une part, l'utilisation d'une carte de Kohonen [Kohonen, 1990] n'est pas envisageable dans notre cas, puisque elle repose sur l'élection d'un neurone gagnant. Cette élection, réalisée de manière centralisée, est incompatible avec les contraintes de scalabilité des architectures many-cœur.

D'autre part, il existe dans la littérature différents modèles, tels les champs neuronaux dynamiques [Amari, 1977], qui permettent d'éviter l'utilisation d'un contrôleur centralisé. Dans ces modèles, l'élection du neurone gagnant est un phénomène émergeant des intéractions latérales entre les neurones, mais nécessite une connectivité importante.

Au cours de ses travaux sur le sujet, Laurent Rodriguez a proposé dans [Rodriguez et al, 2014] un modèle de carte auto-organisatrice faiblement connectée, d'où émergent des propriétés d'auto-organisation dans un contexte multimodal.

3.2.1 Description du modèle

Les travaux menés sur les modèles cités à la section précédente ont conduit à la définition d'un modèle doté de propriétés d'auto-organisation avec une connectivité la plus restreinte possible. Ce modèle, appelé *Distributed Multiplicative Activity-Dependant Self Organizing Map* (DMAD-SOM) est composé de deux couches. Une couche d'entrée fournit les données provenant d'un pré-traitement, comme celui défini au chapitre 2, à la deuxième couche, composée des neurones d'apprentissage.

La topologie du DMAD-SOM est constituée d'une grille de neurones à deux dimensions, où chaque neurone est connecté à ses quatre voisins cardinaux. À cause des contraintes architecturales énoncées précédemment le modèle doit satisfaire à plusieurs conditions.

Premièrement, la carte doit représenter une fonction de densité qui évolue en fonction du temps. En outre, l'apparition d'un nouveau cluster ne doit pas complètement effacer un cluster plus ancien. En d'autres termes, elle doit s'adapter aux modifications de l'environnement, tout en conservant les expériences passées. Ce phénomène, que l'on nomme l'*élasticité* de la carte, correspond aux mécanismes d'apprentissage au sein de la carte et est piloté par l'activité des neurones. Cette activité est issue à la fois de la stimulation du vecteur d'entrée, et de l'influence des neurones voisins.

Deuxièmement, comme chaque neurone n'est connecté qu'à ses quatre voisins, une règle additionnelle doit substituer l'élection d'un neurone gagnant. L'activité est ainsi propagée

au voisinage, et permet d'organiser les neurones au sein de chaque cluster. Cette propriété correspond à l'apprentissage *distribué* du DMAD-SOM.

Les régles d'apprentissage ont été définies au moyen d'une approche expérimentale. Une analyse des solutions classiques existantes a permis de mettre en évidence deux principes agissant comme des conditions nécessaires à la formation de cluster de neurones dont la taille dépend de la fonction de densité des stimuli d'entrée.

Tout d'abord, si un neurone est trop proche du vecteur d'entrée, il n'a pas besoin d'apprendre. Les autres neurones quant à eux doivent continuer à apprendre, pour pouvoir former des clusters de taille suffisante pour bien représenter la donnée d'entrée. De manière analogue, si un neurone est proche d'un de ses voisins, il ne doit pas apprendre plus.

La plasticité afférente

L'apprentissage sur les synapses afférentes, c'est à dire les synapses d'entrée, a pour but de minimiser la distance entre le vecteur de stimulus et le vecteur des poids d'entrée. Ce premier principe peut être énoncé de la manière suivante :

La plasticité afférente d'un neurone doit être inversement proportionnelle à son taux d'activité.

Le potentiel d'action $X(t)$ est considéré comme la probabilité que le neurone représente un stimulus d'entrée à l'instant t. Pour un vecteur d'entrée à N dimensions, $X(t)$ est calculé selon l'eq. 3.1 :

$$X(t) = \prod_{k=0}^{N} P_k(t) \tag{3.1}$$

avec $P_k(t)$ le potentiel, normalisé dans l'intervalle $[0, 1]$, induit par la $k^{\text{ème}}$ synapse. De manière plus formelle, $P_k(t)$ est une fonction de correlation entre le poids $W_{a_k}(t)$ et la $k^{\text{ème}}$ composante du vecteur d'entrée $s_k(t)$. Il doit satisfaire les conditions suivantes :

1. $P_k(t) \to 1$ quand $|s_k(t) - W_{a_k}(t)| \to 0$
2. $P_k(t) \to 0$ quand $|s_k(t) - W_{a_k}(t)| \to \infty$
3. $P_k(t)$ est monotone pour $|s_k(t) - W_{a_k}(t)| < 0$ et pour $|s_k(t) - W_{a_k}(t)| > 0$

La fonction Gaussienne, respectant ces propriétés, est utilisée dans cette étude (eq. 3.2) :

$$P_k(t) = e^{\frac{-(s_k(t) - W_{a_k}(t))^2}{2\sigma^2}} \tag{3.2}$$

le paramètre σ agissant comme une *distance d'écoute* d'un neurone.

La plasticité afférente correspond à la règle d'apprentissage des poids d'entrée W_{a_k}, définie selon l'eq. 3.3 :

$$\Delta W_{a_k}(t) = Lr_a \times M(t) \times (s_k(t) - W_{a_k}(t)) \tag{3.3}$$

avec Lr_a le taux d'apprentissage sur les synapses afférentes, et $M(t)$ une fonction de modulation. La fonction de modulation $M(t)$, conditionnant le comportement global du réseau, est défini selon l'eq. 3.4 :

$$M(t) = U(t) \times \sum_{\forall j} W_{l_j}(t) \tag{3.4}$$

où W_{l_i} est le poids latéral lié au $i^{\text{ème}}$ voisin. Enfin, $U(t)$ est l'activité calculée par chaque neurone suivant l'eq. 3.5 :

$$U(t) = f(X(t) + Y(t) + S(t)) \tag{3.5}$$

avec $Y(t)$ le potentiel latéral, $S(t)$ une fonction d'auto-excitation et f une fonction de transition.

L'apprentissage latéral

Les intéractions latérales ont pour effet de permettre aux neurones de généraliser, et de mieux représenter la densité de probablité des données d'entrée. En d'autres termes, elles empèchent les neurones de s'agglomérer au centre de gravité des clusters, en les forçant à couvrir l'intégralité des classes de données.

Le mécanisme d'intéractions latérales fonctionne comme un taux d'attraction qu'un neurone exerce sur son voisinage. Pour cela, on définit une distance de taux d'activité $D_{ij}(t)$ entre deux neurones n_i et n_j selon l'équation suivante :

$$D_{ij}(t) = X_i(t) - X_j(t) \tag{3.6}$$

avec $X_i(t)$ et $X_j(t)$ respectivement l'activité des neurones n_i et n_j.

On dit que les neurones n_i et n_j sont équilibrés si $D_{ij}(t) = 0$. Sinon, l'un des deux neurones représente mieux la donnée que l'autre et va avoir tendance à l'attirer. La capacité d'un neurone à transmettre une part d'activité à son voisinage est dirigée par ses poids latéraux. Ainsi, si $X_j(t) > X_i(t)$, alors le poids latéral $W_{l_{ij}}$ liant n_i à n_j augmente, sinon il diminue. Ce poids latéral est mis à jour selon l'équation 3.7 :

$$\Delta W_{l_{ij}}(t) = l_{r_{lat}} \times (X_i(t) \times \alpha - X_j(t)) \tag{3.7}$$

avec $l_{r_{lat}}$ le taux d'apprentissage appliqué aux synapses latérales et α un paramètre d'échelle.

Pour plus de détails sur ce modèle, le lecteur peut se référer aux travaux de Rodriguez Rodriguez [2015].

3.2.2 Expérimentations

Plusieurs expérimentations ont été menées pour valider le modèle et ajuster les différents paramètres. Pour cela, un réseau de taille 10×10 a été entrainé sur trois jeux de données à deux dimensions. Le réseau est initialisé de manière ordonnée sur l'intervalle $[0, 1]^2$. La métrique utilisée pour mesurer la qualité de l'apprentissage est l'erreur quadratique moyenne entre les stimuli d'entrée et les positions des neurones.

Le premier jeu de données est constitué d'un ensemble de vecteurs tirés aléatoirement selon une loi normale 2D. Dans le second, les vecteurs sont tirés séquentiellements dans quatre zones distinctes et séparables. Le tirage dans une zone se fait selon une loi uniforme. Le troisième jeu de données est directement issu de données enregistrées sur une mission robotique. Il s'agit d'une vidéo prise par la caméra du robot, présentant deux parties. Dans la première partie, le robot est immobile, et l'environnement présente de nombreux détails. Dans la deuxième partie, le robot est en mouvement, dans un environnement moins riche. Il y a donc moins de détails, mais ces détails sont en mouvement. Le pré-traitement de cette vidéo brute consiste à calculer la quantité de contour ainsi que la quantité de mouvement. Ces deux quantités, normalisées dans l'intervalle $[0, 1]$, constituent le vecteur d'entrée du réseau.

Au cours de ces expérimentations, les paramètres du réseau sont réglés de la manière suivante :

— Valeur maximale des poids latéraux $W_{l_{max}} = 0.2$
— Paramètre Gaussien $\sigma = 0.1$
— Taux d'apprentissage afférent $L_{r_a} = 0.1$
— Taux d'apprentissage latéral $L_{r_l} = 0.2$

Le paramètre d'échelle α est variable et prends les valeurs 0.2, 0.4, 0.6 et 0.8.

Les résultats de ces trois expérimentations sont montrés respectivement en FIGURE 3.2, 3.3 et 3.4. Pour chacune de ces figures sont représentés les résultats pour α allant de 0.2 à gauche à 0.8 à droite. Chaque résultat est constitué dans un premier temps de la courbe de l'évolution de l'erreur quadratique moyenne en fonction du temps. Dans un second temps, nous montrons une capture de l'état du réseau après convergence, superposée à l'historique des stimuli (en bleu sur les figures).

Ces trois expérimentations indiquent que le paramètre d'échelle α règle la capacité qu'on les neurones à se repousser les uns les autres. Pour de petites valeurs, la plasticité afférente n'arrive pas à s'effectuer, et les neurones restent statiques. À l'inverse, pour une valeur trop grande, les neurones se spécialisent, ne représentent pas bien la richesse des données et dans le cas de la vidéo finissent dans un seul et unique cluster.

Les valeurs intermédiaires montrer un résultat plus satisfaisant, d'abord visuellement, où l'on peu voir graphiquement la position des neurones par rapport à l'historique des stimuli. Ensuite nous observons de manière quantitative que l'erreur quadratique moyenne est plus faible.

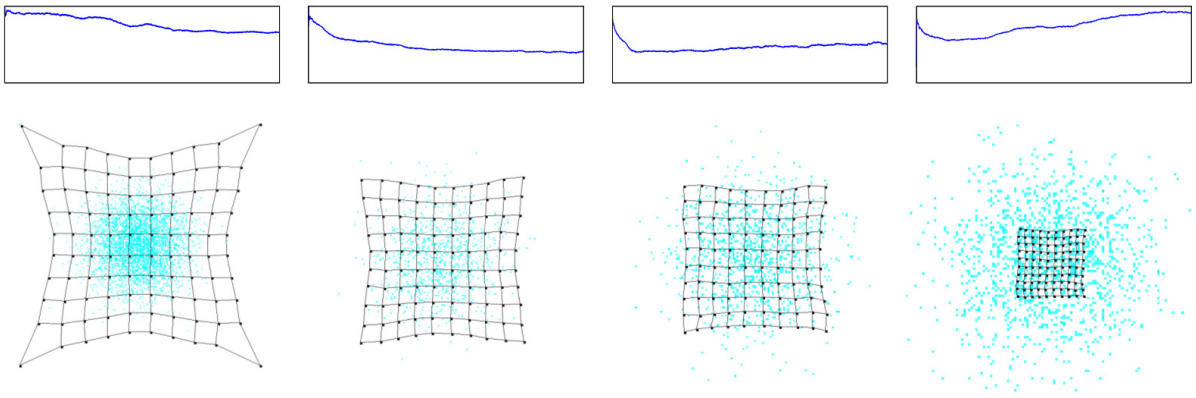


FIGURE 3.2 – **Résultats du réseau stimulé par une loi normale 2D.**

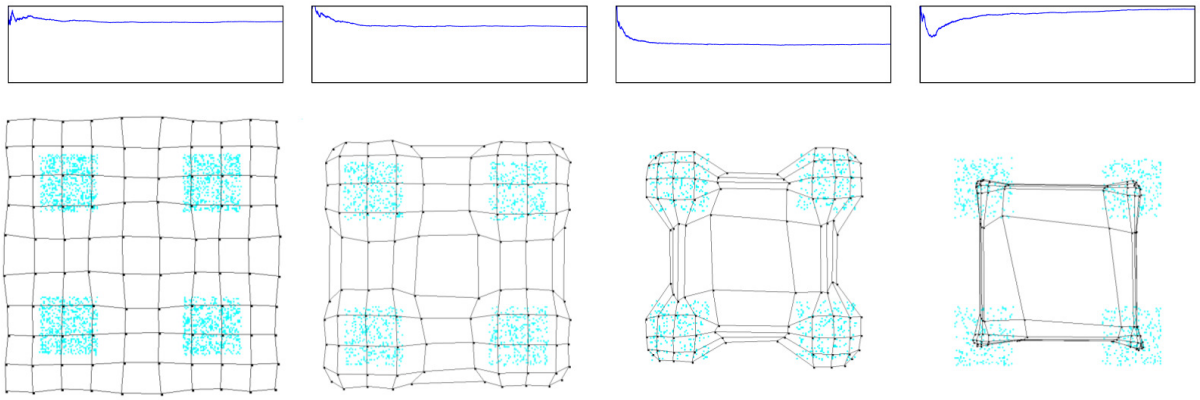


FIGURE 3.3 – **Résultats du réseau stimulé par quatre lois uniformes tirées séquentiellement.**

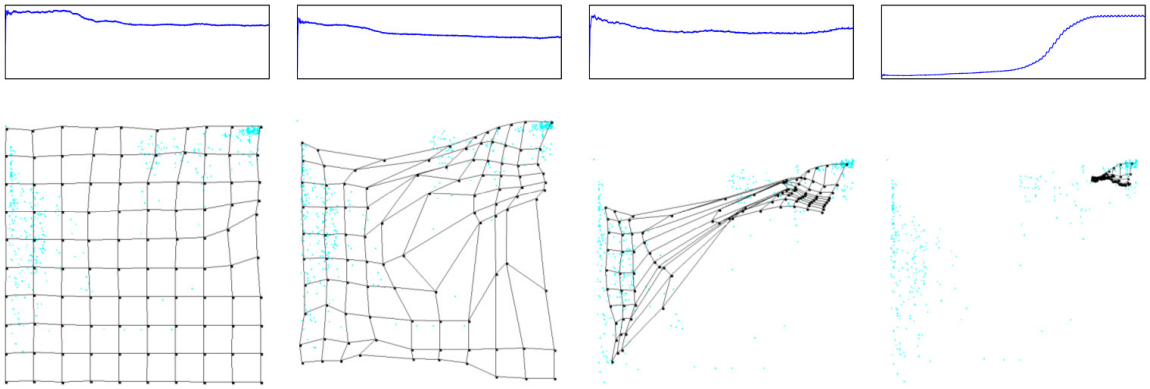


FIGURE 3.4 – **Résultats du réseau stimulé par une vidéo prise par un robot en mission de navigation.** L'apparition de deux clusters est liée aux deux phases de la vidéo : peu de mouvement et beaucoup de détails dans un premier temps, puis beaucoup de mouvement et moins de détails.

3.3 Une implémentation matérielle

Le modèle de la carte auto-organisatrice facilite son implémentation matérielle, de par son aspect distribué et sa faible connectivité. Il reste cependant plusieurs défis à relever pour une implémentation efficace.

Dans un premier temps, l'injection des données conditionne les performances globales du réseau. Les données proviennent d'un module unique, le prétraitement, et sont distribués à tous les neurones du réseau. De manière analogue, la lecture des résultats demande une attention particulière. Dans un cas concret d'utilisation, les résultats d'un neurone sont lus uniquement par leur élément de calcul propre. Cependant, pour des besoin de tests et de vérifications, les résultats doivent être récupérés par un organe centralisé. Il faut donc veiller à ce que la densité des connexions n'augmente pas de manière prohibitive pour de grandes tailles de réseaux.

Dans un second temps, l'implémentation matérielle nécessite de passer à une représentation des nombres en virgule fixe. La largeur des bus de donnée des neurones matériels doit être paramétrable pour permettre d'étudier l'impact de la précision des nombres sur le comportement de la carte.

Ensuite, l'un des obstacles récurrents dans l'implémentation de réseaux de neurones réside dans l'utilisation de fonctions non-linéaires. Ici, il s'agit de la fonction exponentielle de l'équation 3.2. Plusieurs solutions sont couramments utilisées, comme s'appuyer sur des approximations, ou encore pré-calculer la fonction et stocker ses résultats. Il est également possible de faire le calcul pendant l'exécution à l'aide d'algorithmes itératifs.

Enfin, de manière générale, les neurones matériels doivent être le plus légers possibles pour permettre l'implémentation de grands réseaux. La présence d'un grand nombre d'opérateurs – en particulier de multiplicateurs – limitera la quantité de neurones instanciables sur une surface donnée. En plus d'être légère, l'implémentation matérielle doit être la plus générique possible, pour faciliter de futures évolutions.

Certaines problématiques d'implémentations, comme la consommation et la dissipation thermique ont été écartées pour le moment, dans le but de se concentrer sur les aspects fonctionnels et comportementaux.

3.3.1 Description architecturale

L'architecture de la carte

Une version simplifiée de la carte auto-organisatrice présentée en section 3.2 a été portée en matériel. Dans cette version, l'influence de l'apprentissage latéral a été écartée dans le but de valider plus rapidement les concepts énoncés.

La carte auto-organisatrice matérielle est composée d'une grille de neurones physiques. Chaque neurone prends en entrée les activités de ses quatre voisins latéraux directs, ainsi que le vecteur afférent. La sortie des neurones est uniquement constituée de leur activité, dirigée vers leurs voisins. Un exemple de carte, configurée en taille 3×3, est montré en FIGURE 3.5a.

La carte peut changer de mode, comme illustré en FIGURE 3.5b, pour à la fois initialiser et relire les registres internes des neurones. En passant du mode *calcul* au mode *synchronisation*, les neurones sont vus comme de simples registres à décalage, reliés entre eux par les connexions d'activité latérale. Ce chemin de synchronisation permet ainsi d'initialiser les poids afférents et latéraux, et de donner une activité initiale. Il permet également, pendant les phases de validation, de relire l'évolution des poids internes et de l'activité des neurones.

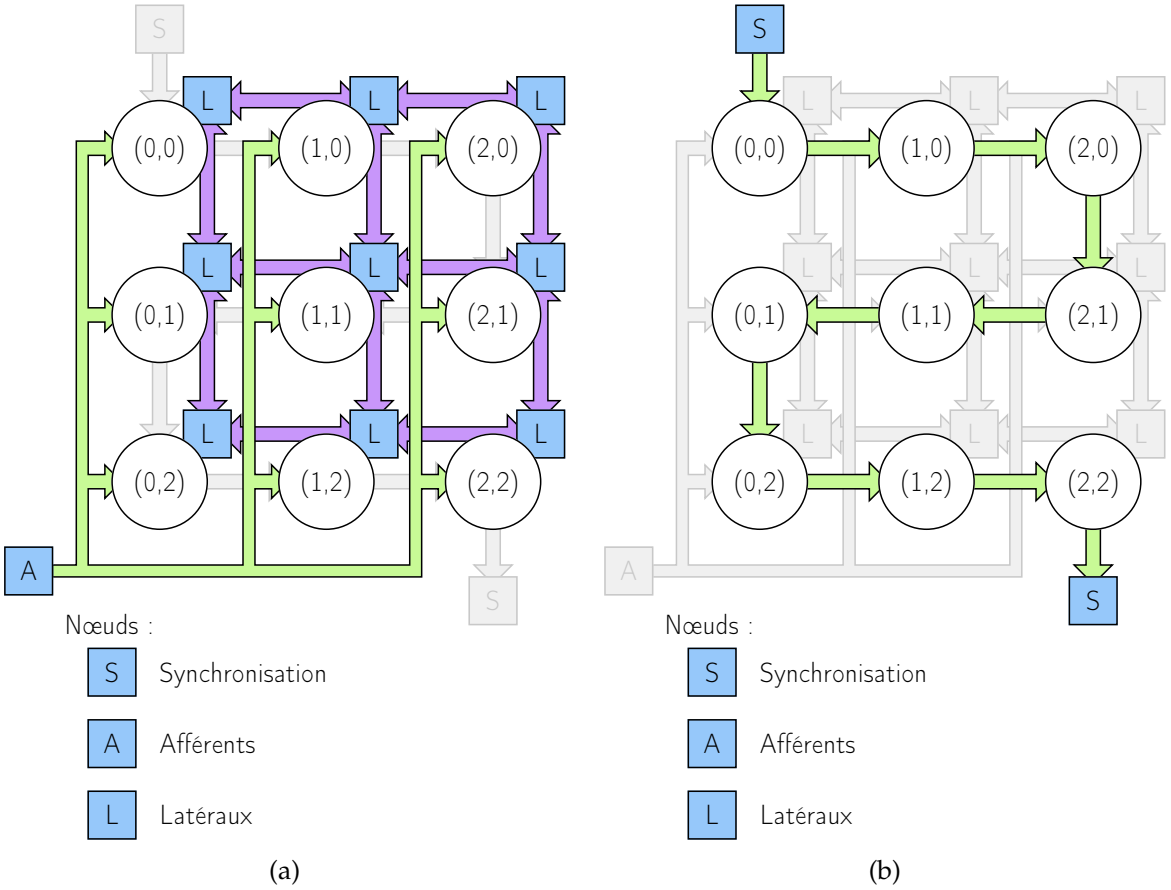


FIGURE 3.5 – **La carte auto-organistratice** est composée d'une grille de neurones connectés latéralement (a). Elle peut passer en mode *synchronisation* pour permettre l'accès à certains registres internes (b).

L'architecture interne des neurones

Les équations implémentées dans ces neurones sont une version simplifiée de celles présentées en section 3.2. L'activité $U(t)$ de chaque neurone est le résultat du passage du potentiel d'action $P(t)$ dans une fonction d'activation f, calculée selon l'eq. 3.8 :

$$U(t) = f(P(t)) = f(X(t) + Y(t)) \tag{3.8}$$

avec $X(t)$ le potentiel induit par le vecteur afférent, défini selon l'équation 3.9, et $Y(t)$ l'excitation latérale due au voisinage, définie par l'équation 3.10 :

$$X(t) = \prod_{i=0}^{N} e^{-\frac{(w_i(t)-x_i(t)^2)}{2\sigma^2}} \tag{3.9}$$

$$Y(t) = \sum_{j=0}^{L} (w_j(t)U_j(t)) \tag{3.10}$$

avec N la dimension du vecteur d'entrée, $w_i(t)$ le poids correspondant à la composante i du vecteur d'entrée x_i et w_j le poids correspondant à l'activité du voisin j, $U(j)$. L'apprentissage des poids afférents est defini par l'équation 3.11 :

$$\Delta w_i(t) = l \times U(t)\,[x_i(t) - w_i(t)] \tag{3.11}$$

avec l le taux d'apprentissage, constant et configurable.

Les neurones ont un signal d'entrée par lequel transite les valeurs du vecteur afférent. Lors du traitement d'une itération neuronale, les composantes du vecteur sont présentées en entrée des neurones les unes après les autres. La largeur du bus d'entrée est donc celle d'une composante du vecteur d'entrée. Ce bus est synchronisé à l'aide d'un signal `Enable`, permettant de valider les valeurs entrantes. Les neurones disposent également de quatre entrées latérales – `Nord`, `Sud`, `Est` et `Ouest` – permettant de lire l'activité des neurones voisins. Enfin, les neurones ont une sortie, l'activité, qui est envoyée aux quatre neurones voisins.

Les entrées/sorties des neurones ainsi que la plupart des signaux internes sont normalisés sur l'intervalle $[0, 1]$. La représentation utilisée pour ces signaux est de type `Q0.f` où `f` est un paramètre générique. La largeur de certains signaux internes peuvent être configurés pour être plus large pour permettre une meilleure précision.

Le chemin de calcul d'un neurone, dont le schéma interne est donné en FIGURE 3.6, est composé de trois branches : la branche d'entrée, la branche latérale, et la branche de sortie, correspondant respectivement aux équations 3.9, 3.10 et 3.8.

La branche d'entrée est chargée du calcul du potentiel afférent $X(t)$ selon l'équation 3.9. C'est aussi dans cette branche que les poids d'entrée sont stockés et mis à jours, comme décrit par l'équation 3.11. Les poids sont stockés dans une mémoire FIFO où les données sont décalées à chaque fois qu'une donnée entrante est présente, ce qui a pour effet de garder une synchronisation entre la composante d'entrée et le poids associé. En sortie de cette FIFO, le poids est mis à jour par le module d'apprentissage, pour à la fois être ré-injecté dans la FIFO, et être utilisé dans la suite du calcul. L'entrée afférente est alors soustraite à ce poids et le résultat est envoyé dans le module *Gauss*.

Le module *Gauss* est constitué d'une table qui stocke une fonction Gaussienne $G(x)$ précalculée selon l'équation $G(x) = e^{-\frac{x^2}{2\sigma^2}}$. La fonction étant paire, seule la partie positive est

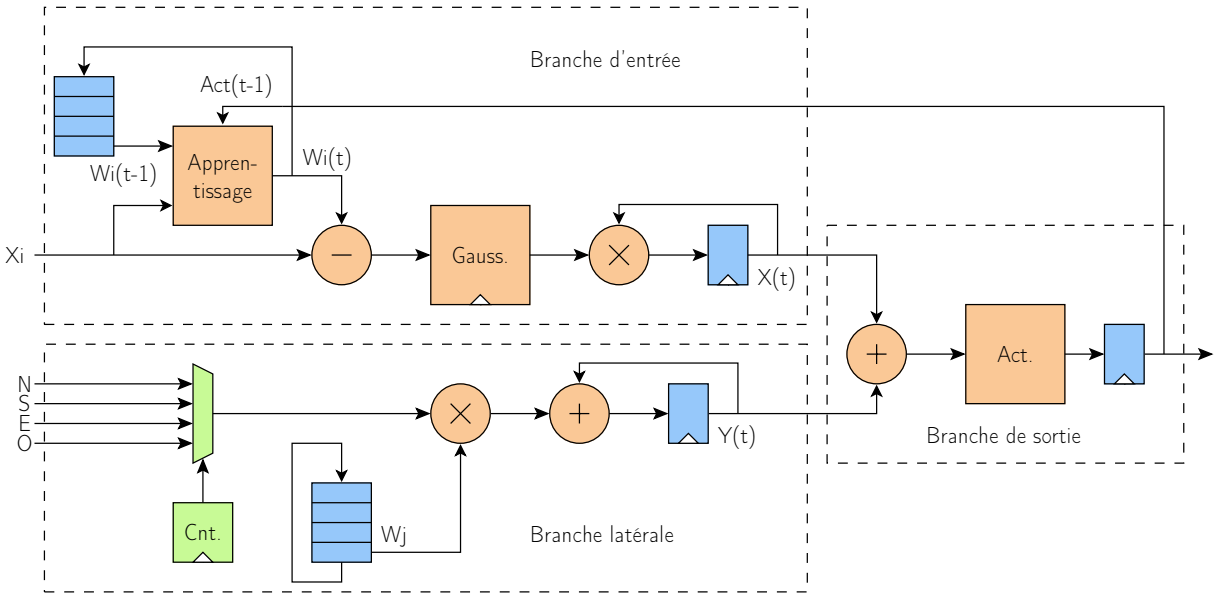


FIGURE 3.6 – **Vue interne du chemin de calcul d'un neurone matériel.** Son architecture est organisée en trois branches : la branche d'entrée, la branche latérale et la branche de sortie.

stockée. La taille de cette table peut néanmoins devenir conséquente pour de grandes largeurs de bus de données et, dans le cas d'une intégration sur un FPGA, cela peut nécessiter plusieurs mémoires embarquées. Dans ce cas, le pas d'échantillonage peut être variable, laissant une plus grande résolution au niveau du lobe central qu'à l'extérieur. Le découpage du pas d'échantillonnage est effectué automatiquement à la synthèse en fonction de la largeur du bus de la gaussienne et de la taille des blocs mémoire.

Enfin, le produit des N Gaussiennes est réalisé au moyen d'un multiplieur et d'un registre accumulateur. La largeur des données peut localement être augmentée pour ne pas perdre en précision.

La branche latérale réalise le calcul de l'excitation latérale $Y(t)$ décrit par l'équation 3.10. Les poids latéraux sont à nouveau stockés dans une FIFO. L'apprentissage latéral, qui n'avait pas encore été testé au moment de la conception de l'architecture matérielle n'a pas été mis en place dans cette version. Il peut néanmoins facilement être inséré entre la sortie et l'entrée de la FIFO, comme pour la branche d'entrée. L'activité des quatre voisins est lue de manière séquentielle, de manière synchronisée par rapport aux poids latéraux. Le calcul de l'excitation latérale est réalisé par le biais d'une structure Multiplier-Accumulator (MACC).

La branche de sortie, finalement, est simplement composée d'un additionneur, de la fonction d'activation et d'un registre. Dans notre cas, la fonction d'activation est un gain unitaire, suivi d'une saturation, ce qui permet de forcer l'activité dans l'intervalle $[0, 1]$.

Soient N la dimension du vecteur d'entrée et L le nombre de voisins latéraux. Le nombre de cycles d'horloge nécessaires à l'exécution d'une itération neuronale est imposée par la dimension du vecteur d'entrée si $L < N$, sinon il est imposé par le nombre de voisins. Un exemple de ces deux cas est montré en FIGURE 3.7. Dans ces chronogrames, la ligne X_i représente les données du vecteur d'entrée, la ligne X_l indique quel voisin est lu, et la ligne Act montre la sortie d'activité.

La FIGURE 3.7a montre une itération neuronale pour une dimension $N = 2$. On peut remarquer que les activités des neurones voisins sont traitées durant les quatre premiers cycles d'horloge, et que l'activité est traitée pendant le cinquième cycle, pour être disponible au sixième. Le délai induit par la lecture de la table contenant la Gaussienne n'est pas visible, puisque le traitement de la branche d'entrée est plus court que le traitement de la branche latérale. L'itération neuronale s'exécute donc dans ce cas en $L + 1 = 5$ cycles.

La FIGURE 3.7b montre une itération neuronale pour une dimension $N = 4$. Ici, cinq cycles sont nécessaires au traitement de la branche d'entrée à cause de la lecture de la table Gaussienne. Le sixième cycle est toujours requis pour calculer l'activité. Ainsi, l'activité est disponible au septième cycle, ce qui marque le début d'une nouvelle itération. L'itération dure alors $N + 2 = 6$ cycles. Le temps d'exécution global d'une itération neuronale peut donc être donné par l'équation 3.12.

$$T_{cycle} = \max(L + 1, N + 2) \tag{3.12}$$

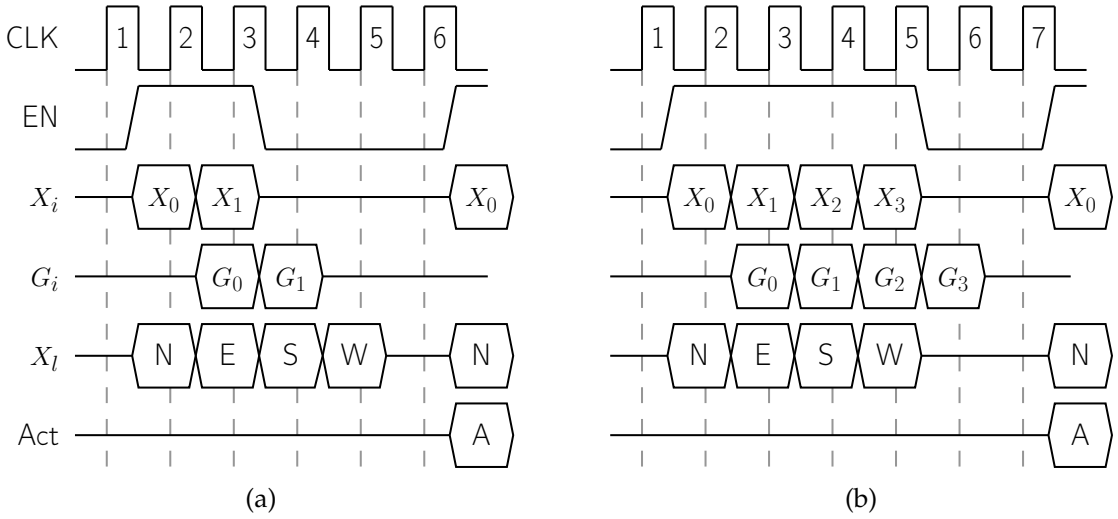


FIGURE 3.7 – **Chronogrammes d'itérations neuronales** pour $N = 2$ (a) et $N = 4$ (b).

Durant le mode *synchronisation*, les FIFO et le registre d'activité sont reliés entre eux par un jeu de multiplexeurs. Le chemin de synchronisation ainsi formé est représenté en FIGURE 3.8.

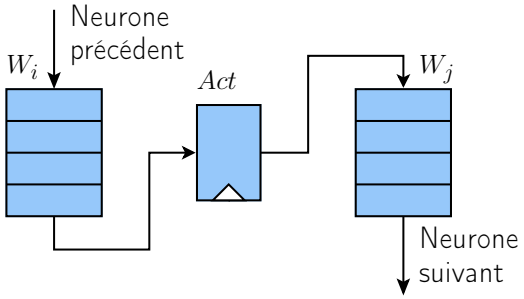


FIGURE 3.8 – **Chemin de synchronisation interne d'un neurone.**

3.3.2 Résultats

Les modèles du neurone et de la carte auto-organisatrice discutés précédemment ont été décrits en VHDL. Nous avons choisi de n'utiliser aucun composant spécifique à un constructeur pour être le moins dépendant possible d'une technologie particulière. Nous avons effectué une première validation en simulation, durant laquelle nous avons pu mesurer les effets des différents paramètres sur le comportement de la carte. Nous avons ensuite réalisé un portage sur une carte FPGA pour à la fois valider le fonctionnement dans un cas réel et tester la scalabilité de l'architecture.

Pour ces deux tests, un jeu de données a été extrait du simulateur logiciel, pour être injecté dans l'architecture. Pour les tests sur FPGA, les données ont été injectées par le biais d'un soft-processeur.

Résultats comportementaux

Pour comparer le comportement du réseau de neurones matériel au modèle logiciel, nous les avons stimulés avec un jeu de données artificel. Ce jeu de données est généré en tirant aléatoirement le vecteur d'entrée dans quatre zones séparables, comme dans les expérimentations discutées en section 3.2.2.

À chaque itération neuronale, nous avons mesuré la quantité de neurones appartenant à chacun des quatre clusters, codant l'allocation des éléments de calculs correspondant à l'une des quatre tâches. Nous avons tracé en FIGURE 3.9 cette évolution pour le modèle logiciel et pour le modèle matériel. On peut constater des variations entre les modèles matériels et logiciels, notamment les écarts entre les différentes tâches qui sont moins importants pour le modèle matériel. Cependant les allures restent semblables, et ces variations peuvent être atténuées en prêtant plus d'attention au réglage des paramètres, en particulier du taux d'apprentissage. Ce réglage aurait néanmoins pris beaucoup de temps. Nous avons préféré nous focaliser sur l'évolution de l'architecture que sur le réglage des paramètres.

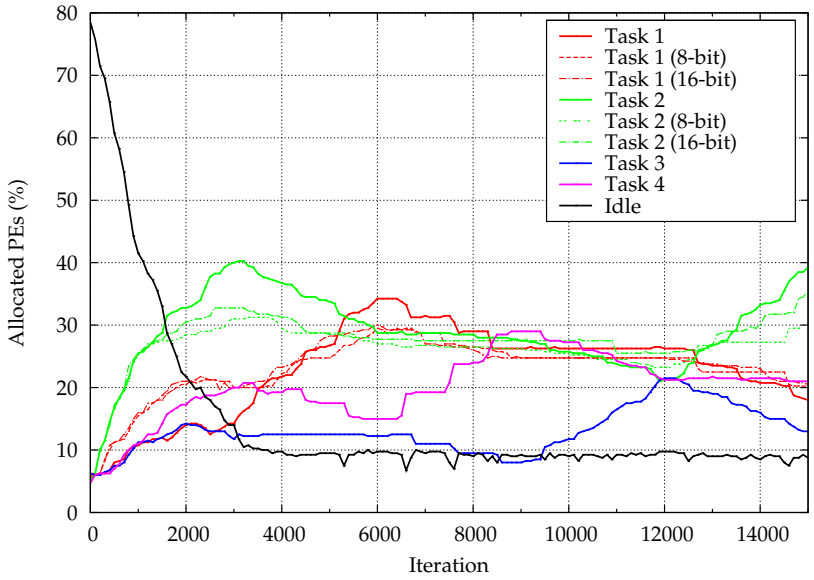


FIGURE 3.9 – **Évolution du nombre d'éléments de calcul alloués à chaque tâche en fonction du temps.** Les PE non alloués sont représentés par la tâche *Idle*.

Implémentation

L'architecture, décrite en VHDL a été portée sur le FPGA Stratix V 5SGSMD8N3F45 d'Altera. Ce FPGA, composé de 512 k *Adaptive Look-Up Tables* (ALUT), de 50 Mbit et de 1963 blocs DSP, était l'un des FPGA offrant la plus grande densité au moment où ces mesures ont été prises.

Pour étudier la scalabilité de l'architecture, nous avons fait varier les dimensions de la carte pour deux jeux de paramètres, et nous avons observé la consommation en ressources matérielles. Pour le premier jeu de paramètres, nous avons choisi une largeur de bus de 8 bit avec une extension de 16 bit pour la largeur du multiplieur et un vecteur de dimension 4. Pour le deuxième jeu de paramètres, la largeur de bus est doublée, et passe donc à 16 bit, les autres paramètres restant identiques.

La largeur du bus multiplieur a été fixée à 16 bit car une largeur plus faible aurait dégradé les performances du réseau et une largeur plus élevée aurait fait exploser la consommation en ressource des neurones. La dimension du vecteur d'entrée a été fixée arbitrairement, car elle a très peu d'influence sur la consommation du réseau.

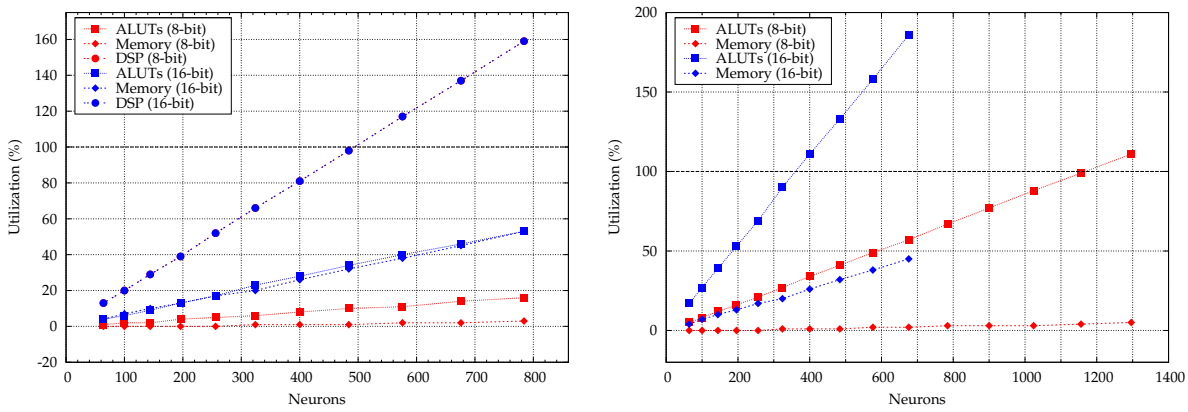
La consommation en ALUT, en mémoire et en blocs DSP est tracée en FIGURE 3.10a. On peut constater que la consommation augmente de manière linéaire pour chaque ressource. La consommation en ALUT et en mémoire est évidemment plus important pour la version 16 bit de l'architecture. Cependant, la limite est atteinte par l'utilisation en blocs DSP, pour un réseau de 484 neurones. Les courbes des deux jeux de paramètres sont confondues, car les blocs DSP du Stratix V ont une largeur de 18 bits, et sont consommés intégralement même pour une largeur de 8 bit.

Pour mieux pouvoir estimer la différence entre les deux jeux de paramètres, nous avons relancé les synthèses en interdisant au synthétiseur d'utiliser les blocs DSP. Les résultats de cette synthèse sont montrés en FIGURE 3.10b. Pour le premier jeu de paramètres, un réseau de taille 1156 neurones a pu être instancié. En revanche, la taille du réseau tombe à 324 neurones pour le second.

La fréquence de fonctionnement du circuit se situe aux alentours de 50 MHz pour des tailles de cartes approchant le maximum et plafonnent à 100 MHz pour des réseaux de taille plus modestes, inférieurs à 100 neurones.

Ces résultats, résumés en FIGURE 3.17 et en TABLE 3.1, nous montrent certaines limites quant à la scalabilité de l'architecture. Même si la consommation reste linéaire, les neurones, dans leur version 16 bit restent très consommateurs de ressources matérielles. De plus le système d'injection et de relecture de données, conçu dans une optique de prototypage, est très peu modulable et semble peu adapté au couplage à un élément de calcul. La fréquence de fonctionnement du circuit diminue fortement avec l'augmentation de la taille de la carte, à cause à la fois de la complexité interne des neurones, mais surtout de la connectivité *un vers tous* du système d'injection. Toutefois, considérant une fréquence de fonctionnement de l'ordre de 50 MHz et une durée d'itération neuronale inférieure à une dizaine de cycles d'horloge, le débit neuronal atteint 5 Mit/s, surpassant largements nos exigences.

En dépit de ces limites, ces travaux ont apporté une preuve de concept et ont montré la faisabilité d'une carte auto-organisatrice matérielle, décentralisée et faiblement connectée.



(a) (b)

FIGURE 3.10 – **Résultats de synthèse du réseau de neurones matériel.** Taux d'utilisation de différentes ressources en fonction de la taille du réseau. En (a), le synthétiseur est configuré par défaut, en (b) il n'utilise pas de blocs DSP.

Paramètre	**Set 1**	**Set 2**
Largeur du Bus (bit)	8	16
Largeur du Multiplieur (bit)	16	16
Nombre d'entrées	4	4
Résultat	**Set 1**	**Set 2**
Nombre de neurones avec DSP	484	484
Nombre de neurones sans DSP	1156	324

TABLE 3.1 – Récapitulatif des mesures

3.4 Le NPU : Un processeur neuronal

Au vu du constat énoncé ci-dessus, nous avons cherché à mieux comprendre les limites de cette architecture, pour pouvoir définir un système plus léger et plus modulable en profitant de la marge disponible au niveau du débit d'itérations neuronales.

La consommation élevée en ressources matérielles du neurone est principalement due au nombre d'opérateurs – plus particulièrement de multiplieurs – nécessaire au traitement des équations. Le stockage de la fonction Gaussienne représente également une forte part de la consommation. La quantité de mémoire nécessaire a été grandement diminuée par l'échantillonnage à pas variable, au détriment d'un surplus de logique.

Les neurones sont par ailleurs difficilement interfaçable, notamment en ce qui concerne la lecture des poids par un élément de calcul, qui n'était pas défini à ce moment du projet. De plus les neurones doivent être synchronisés, de la lecture du vecteur d'entrée à la lecture des activités latérales. De manière analogue, les neurones sont peu modulaires, dans un projet où les modèles sont en évolution constante. Compte tenu du temps de développement long des architectures matérielles, subir le flot de conception à chaque modification du modèle est prohibitif.

La fréquence globale du circuit diminue rapidement lorsque l'on considère des grandes cartes neuronales. Ceci est dû en partie au degré de pipeline assez restreint du chemin de calcul des neurones, et pourrait facilement être corrigé. L'autre raison réside dans la manière

dont le vecteur d'entrée est injecté dans l'architecture. Comme les neurones doivent recevoir les données d'entrée de manière synchronisée, le bus d'entrée est connecté au système qui génère les données. Le routage de ce bus au sein du FPGA augmente de manière significative le chemin critique et réduit de ce fait la fréquence maximale atteignable.

Ainsi, nous profitons de la place disponible au niveau du débit neuronal pour séquentialiser les opérations au sein d'un neurone et ainsi factoriser les opérateurs. Cette nouvelle architecture est alors construite autour d'une unité de calcul, capable d'effectuer les opérations de base nécessaires au traitement des équations du DMAD-SOM. Partant de ce principe, nous construisons ce nouveau système à la manière d'un micro-processeur spécifique léger, modulaire, et programmable, que nous appelons Neural Processing Unit (NPU).

3.4.1 L'architecture du NPU

Le NPU a été conçu de manière itérative où de nombreux aller-retours ont été effectués entre les différentes phases du projet. Par exemple, un choix au niveau architectural va impacter le jeu d'instruction, ce qui produira à son tour un effet sur l'architecture. Quel que soit le niveau considéré, chaque choix a visé à réduire l'empreinte du composant, sans sacrifier de manière trop significative le débit du neurone.

Le NPU est constitué d'une Arithemetic/Logic Unit (ALU), d'une mémoire Random Access Memory (RAM) embarquée double-port (DPRAM) et d'un contrôleur instancié sous la forme d'une machine à états finis. L'organisation interne du cœur du NPU est représenté en FIGURE 3.11.

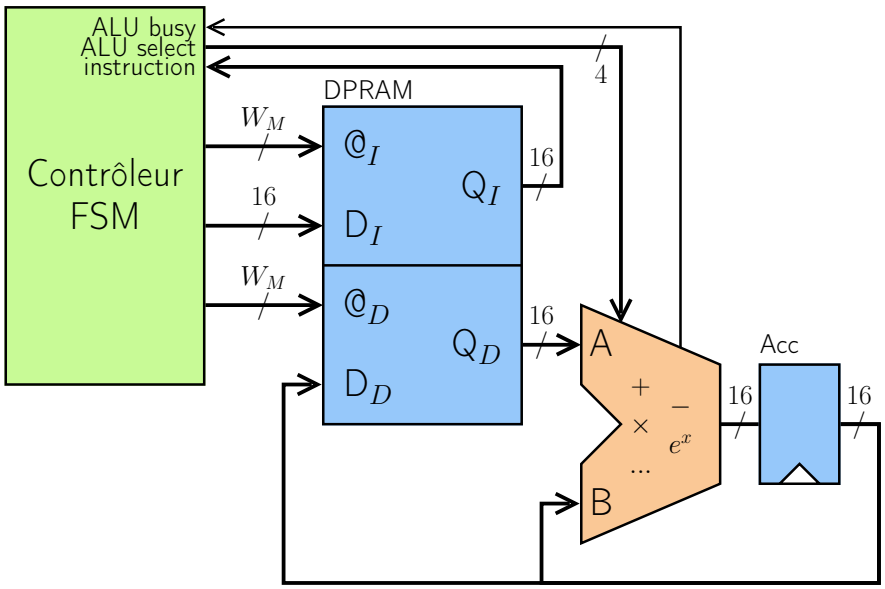


FIGURE 3.11 – **Organisation interne du NPU.** Il est constitué d'une mémoire double port, d'une ALU à accumulateur et d'un contrôleur.

Les deux ports de la mémoire permettent d'accéder indépendamment aux instructions et aux données stockées, ce qui rapproche le NPU d'une architecture de type Harvard. Ce choix, en plus d'augmenter la bande passante du composant permet de simplifier le contrôleur du neurone. Compte tenu des équations présentées précédemment, l'ALU est munie d'opérateurs basiques : l'addition, la soustraction et la multiplication. Les opérations logiques n'ont pas été

implémentées dans un premier temps, puisqu'elles n'étaient pas requises pour le traitement des équations neuronales. Cependant, il est aisé de rajouter ces opérateurs s'ils deviennent nécessaires. Le signal `ALU select`, de largeur 4 bit permet de sélectionner l'opération à effectuer par l'ALU. Ainsi, jusqu'à 16 opérations peuvent être instanciées dans l'ALU.

Pour pouvoir traiter rapidement la fonction Gaussienne, un opérateur de calcul d'exponentielles a été rajouté à cette ALU. Son implémentation sera discutée plus en détails par la suite. Pour simplifier le format de l'instruction et par conséquent son décodage, l'ALU a été conçue comme une structure à accumulateur, et ne prend qu'une opérande en entrée. Cette entrée d'opérande est directement connectée à la sortie du port de données de la mémoire. Les opérations classiques s'effectuent en un cycle d'horloge, mais l'opérateur d'exponentielle peut nécessiter plusieurs cycles pour être complété. Pour cette raison, l'ALU est capable, *via* le signal `ALU busy`, d'interrompre le reste du NPU le temps du calcul.

Le contrôleur lit les instructions stockées dans la mémoire par son port d'instruction, les décode, et en déduit quelle donnée lire, et quel calcul effectuer. Outre les opérations de calcul, il peut charger une donnée dans l'accumulateur, ou écrire son contenu dans la mémoire.

Ces trois composants forment un pipeline à trois étages, représenté en FIGURE 3.12, et organisés de la manière suivante :

1. Lecture d'une instruction,
2. Lecture ou écriture d'une donnée en mémoire,
3. Calcul et accumulation du résultat.

Cette organisation du pipeline est valable pour les instructions de calcul. Pour les instructions de chargement/rangement, le fonctionnement du pipeline est modifié. Les différents modes de fonctionnement du pipeline seront détaillés par la suite.

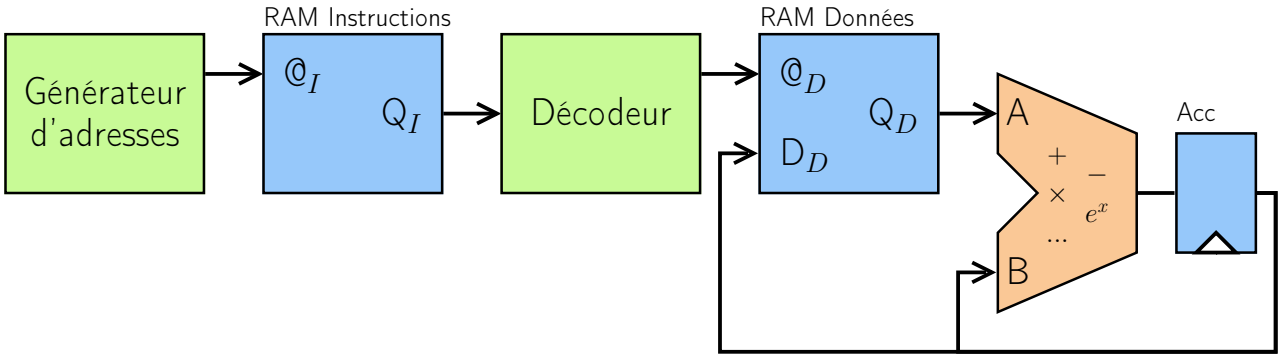


FIGURE 3.12 – **Vue en pipeline du NPU.** Le contrôleur et la mémoire sont séparés en deux blocs pour plus de lisibilité.

Le chemin de calcul du NPU a une largeur de 16 bit ce qui, selon la Section 3.3.2 semble être un bon compromis entre la qualité des résultats et l'utilisation de ressources matérielles. La largeur du bus d'instructions est la même que celle du bus de données, puisque les deux sont liées à la même mémoire embarquée. La taille de la mémoire, quand à elle, est configurable et impacte la largeur des bus d'adresse, notés W_M sur la FIGURE 3.11.

L'interfaçage

Jusqu'à présent, nous n'avons pas vu comment injecter le vecteur d'entrée dans ce NPU, ni comment lire l'activité produite et les poids internes. L'injection et la relecture des données

sont réalisés au moyen d'un module de `Communication`. Ce module n'a volontairement pas été spécifié en même temps que le reste de l'architecture, car son implémentation dépend du cas d'utilisation de la carte auto-organisatrice.

Dans le cas où la carte est testée seule, dans le cadre d'une validation par exemple, le module de `Communication` dispose de communications bi-directionnelles avec les quatre voisins latéraux, ainsi que de connexions pour le vecteur afférent. Il se présente alors comme un contrôleur chargé de lire l'activité des voisins pour l'injecter dans le neurone, de lire l'activité du neurone pour la transmettre aux voisins et de lire les données issues du vecteur d'entrée, et de le transmettre à l'un des voisins. Ce cas est représenté en FIGURE 3.13 où l'on peut voir les interactions entre le NPU tel que présenté précédement (a), et le module de communication (b). Une carte neuronale de taille 3×3 est montrée en exemple en FIGURE 3.14.

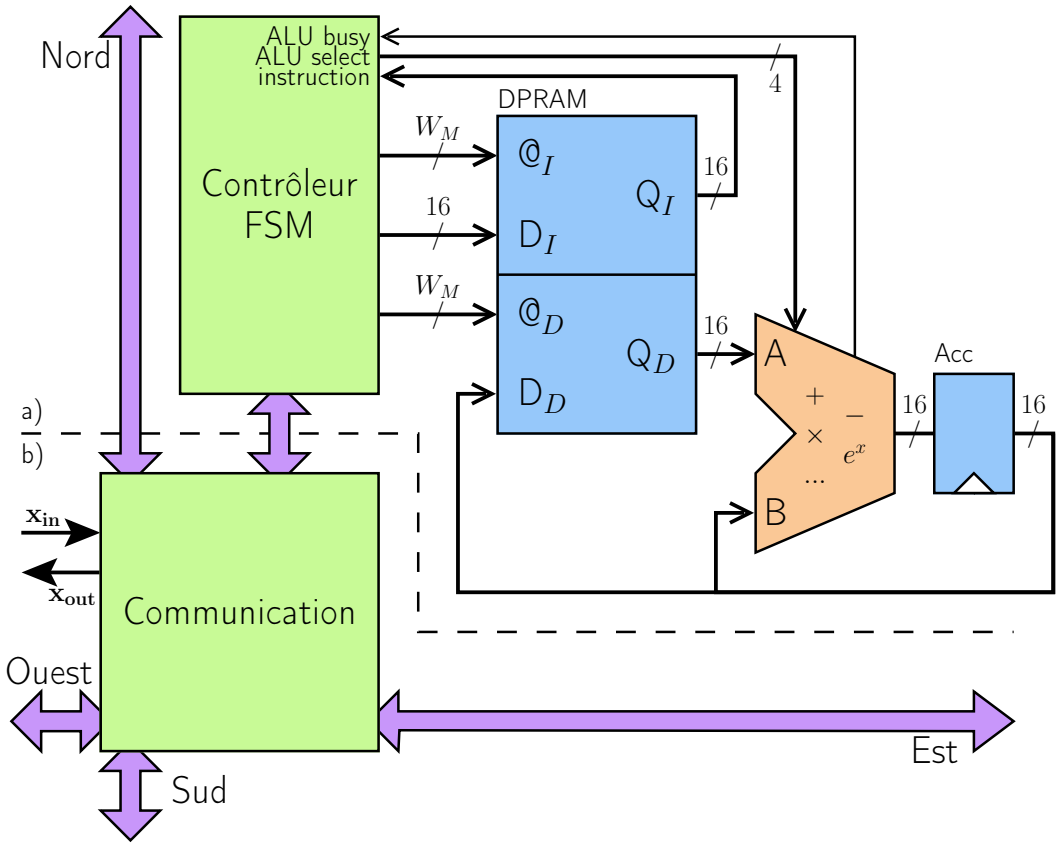


FIGURE 3.13 – **Organisation interne du NPU a)** connecté à un module de communication b).

Dans le cas étudié dans cette thèse, où les NPU sont associés à des éléments de calculs, un module de `Communication` différent est nécessaire. Il sera discuté au chapitre suivant.

Pour pouvoir injecter des données ou relire les résultats du NPU, le module de *communication* interrompt le fonctionnement normal du NPU, pour pouvoir accéder à sa mémoire. Il en découle deux modes de fonctionnement, un mode de calcul et un mode de communication. Ainsi, à l'initialisation de la carte, les NPU sont en mode communication jusqu'à ce qu'ils reçoivent leur première donnée. Ils passent ensuite en mode calcul, pour produire une activité et faire évoluer leurs poids. Ensuite, ils repassent en mode communication pour transmettre leur activité au voisinage, et éventuellement transmettre leurs poids à l'élément de calcul. Ils

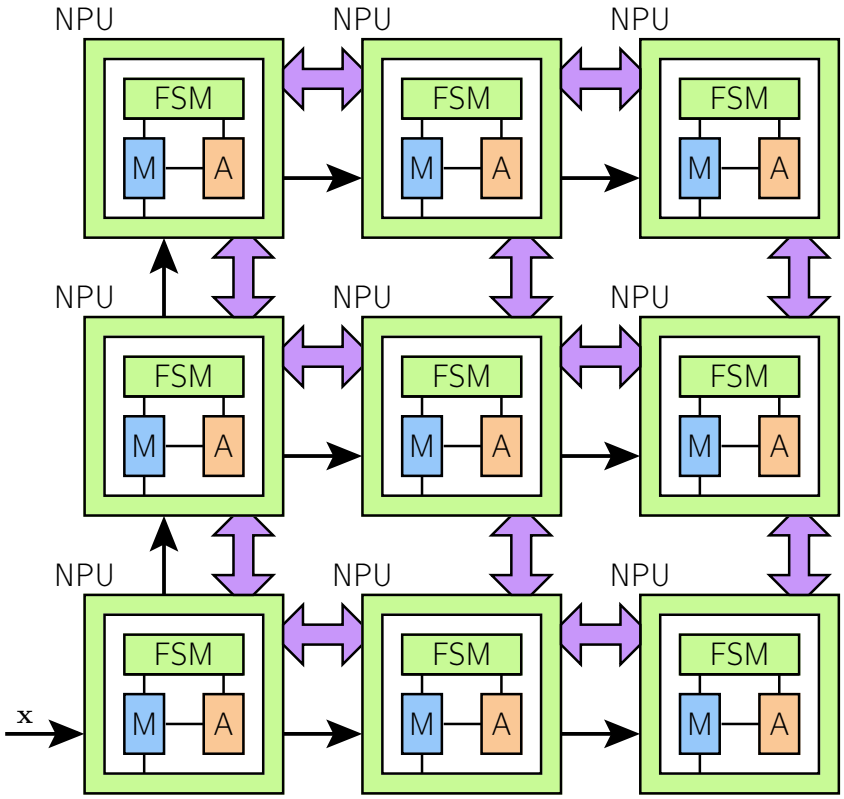


FIGURE 3.14 – **Carte neuronale à base de NPU** de taille 3×3. Chaque neurone est connecté à ses quatre voisins, et le vecteur d'entrée est transmis à tous les NPU par le biais d'un arbre de *broadcast*.

reçoivent ensuite un nouveau jeu de données et itèrent à nouveau.

Le module de communication accède à la mémoire interne du NPU par le biais de son port d'instruction. Durant le mode de communication, tout l'espace mémoire du NPU est accessible par le module de communication, ce qui permet évidemment d'injecter des données et de relire les résultats, mais également de capturer des variables internes et de modifier des constantes, voire de changer des portions de code.

Le jeu d'instructions

L'analyse des équations du DMAD-SOM nous ont permi de définir un jeu d'instruction extrèmement réduit, facilitant ainsi le décodage. Les opérateurs rencontrés sont l'addition (`add`) et la soustraction (`sub`), la multiplication (`mul`) et la mise au carré (`squ`), et l'exponentielle (`exp`). À ces opérations, se rajoutent des instructions de chargement (`load`) et de rangement (`store`) permettant d'initialiser l'accumulateur ou de le sauvegarder en mémoire. Ces instructions sont résumées dans la TABLE 3.2.

Le portage des équations mène naturellement à un code très linéaire. Bien que les différentes sommes et produits donnent du code répétitif et régulier, facilement implémentable sous formes de boucles, nous n'avons implémenté ni structures conditionnelles, ni structures de saut. Comme les bornes de ces sommes et produits sont connues à l'avance, les boucles peuvent être déroulées, au détriment de la taille finale du programme.

Mnémonique	Opérandes	Opération
`nop`	-	-
`load`	src	`Acc` $\leftarrow$ src
`store`	dst	dst $\leftarrow$ `Acc`
`add`	op	`Acc` $\leftarrow$ `Acc` $+$ op
`sub`	op	`Acc` $\leftarrow$ `Acc` $-$ op
`mul`	op	`Acc` $\leftarrow$ `Acc` $\times$ op
`squ`	-	`Acc` $\leftarrow$ `Acc` * `Acc`
`exp`	op	`Acc` $\leftarrow exp($`Acc` $\times$ op$)$

TABLE 3.2 – Jeu d'instruction du NPU. `Acc` indique le registre accumulateur.

Ce choix peut sembler discutable au premier abord, mais il permet de simplifier grandement l'architecture du NPU, et n'impacte pas le temps d'exécution. Le seul effet négatif réside dans l'augmentation de la taille du code. Il faut néanmoins garder à l'esprit que le code issu des équations du DMAD-SOM ne dépasse pas la centaine d'instructions. La taille du code reste négligeable devant la dimension des mémoires embarquées disponibles au sein des FPGA modernes.

L'absence de structures conditionnelles et de saut simplifie également le codage des instructions, et fatalement leur décodage. Le choix de baser le NPU sur une architecture à accumulateur discuté précédemment nous donne naturellement des instructions à une opérande. Ainsi, les instructions sont découpées en trois champs, représentés en FIGURE 3.15. Le premier champ, `L/S` pour `load/store` permet de choisir si le NPU lit ou écrit une donnée en mémoire. Le champ `Code Op.` contient l'opération à effectuer par l'ALU. Enfin, le dernier champ, `Adresse` contient l'adresse de la donnée à manipuler.

Ainsi, lors d'une opération arithmétique ou logique, le bit `L/S` indiquera de lire une donnée en mémoire, le `Code Op.` précisera quelle opération effectuer et le champ `Adresse` permet de savoir quelle variable utiliser pour le calcul. Une opération `load` consistant à charger une variable dans le registre accumulateur fonctionne comme une opération arithmétique ou logique, pour laquelle l'ALU est configurée en mode *bypass*. Lors d'une opération `store` durant laquelle le NPU va stocker la valeur contenue dans l'accumulateur en mémoire, le bit `L/S` déclenche une écriture en mémoire et inhibe l'écriture dans le registre accumulateur. Le résultat de l'ALU est alors simplement ignoré.

Ce format d'instruction permet un décodage immédiat, puisque le bit `L/S` et l'`Adresse` sont directement connectés à la mémoire et au registre accumulateur, le `Code Op.` est quand à lui envoyé sans traitement supplémentaire à l'ALU.

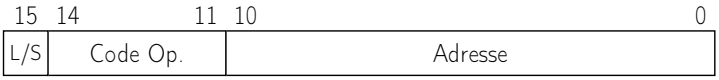


FIGURE 3.15 – **Format d'une instruction** composé de trois champs : `load/store`, `Code Op.`, et `Adresse`.

La fonction Gaussienne

Dans la première implémentation matérielle du DMAD-SOM, la fonction Gaussienne était pré-calculée et stockée dans une table. Ici, nous cherchons une autre solution pour laisser le

maximum de mémoire disponible au programme et à ses données. La difficulté du calcul de la Gaussienne réside dans le calcul de l'exponentielle. Ce problème est résolu par un algorithme itératif, inspiré de l'algorithme CORDIC (*COordinate Rotation DIgital Computer*, calcul numérique par rotation de coordonnées).

L'objectif de cet algorithme est de trouver une valeur $\hat{y}$ qui estime y tel que :

$$y = \exp(x) \tag{3.13}$$

Pour cela, on introduit le terme α tel que :

$$\begin{aligned} \hat{y} \times \exp(\alpha) &= \exp(x) \\ \hat{y} &= \exp(x) \times \exp(-\alpha) \end{aligned} \tag{3.14}$$

Cette égalité est vraie dans deux cas particulierement intéressants :

1. $(\alpha = 0, \hat{y} = y)$: ce cas représente la solution que l'on cherche à atteindre ;
2. $(\alpha = x, \hat{y} = 1)$: ce cas est facile à construire et constitue le point de départ de l'algorithme.

L'algorithme va alors générer une séquence de couples $(\alpha, \hat{y})$ vérifiant l'égalité 3.14, en commençant par $(\alpha = x, \hat{y} = 1)$. À l'itération i, le couple $(\alpha_i, \hat{y}_i)$ est mis à jour de la manière suivante :

$$\alpha_{i+1} = \begin{cases} \alpha_i - k_i & \text{si } (\alpha_i - k_i > 0) \\ \alpha_i & \text{sinon} \end{cases} \tag{3.15}$$

$$\hat{y}_{i+1} = \begin{cases} \exp(x) \times \exp(-\alpha_{i+1}) & \text{si } (\alpha_i - k_i > 0) \\ \hat{y}_i & \text{sinon} \end{cases} \tag{3.16}$$

En substituant α_{i+1} par $\alpha_i - k_i$ dans 3.16, on obtient :

$$\hat{y}_{i+1} = \hat{y}_i \times \exp(k_i) \tag{3.17}$$

À chaque itération, on choisit une valeur de k_i telle que la multiplication de $\hat{y}_i$ par le terme $\exp(k_i)$ soit décomposable en une somme et un décalage. Les expérimentations ont montré qu'un total de 12 itérations suffisait à atteindre une précision suffisante par rapport à la largeur du bus de données du NPU. Les valeurs de k_i pour ces 12 itérations sont données dans la TABLE 3.3.

Cet algorithme peut être facilement porté en matériel. On peut ainsi construire un module calculant les nouveaux états $(\alpha_{i+1}, \hat{y}_{i+1})$ en fonction des états précédents $(\alpha_i, \hat{y}_i)$ avec des composants simples. Chacun de ces éléments est alors responsable du traitement d'une itération particulière et peut être chainé avec d'autres éléments pour réaliser le calcul complet. Ces éléments peuvent prendre deux formes différentes, pour les itérations de 0 à 3 d'une part, et de 4 à 11 d'autre part. Ils sont représentés respectivement en FIGURE 3.16a et 3.16b. Cette architecture permet de calculer rapidement des exponentielles, mais nécessite de dupliquer 12 fois cet élément.

Une autre stratégie consiste à utiliser un module plus générique, permettant de satisfaire à toutes les itérations. Un compteur d'itérations pilotant un jeu de multiplexeurs permet de calculer l'exponentielle de manière itérative, en 12 cycles d'horloge. L'architecture de ce module est montrée en FIGURE 3.17. Comme elle est plus légère, nous l'avons utilisé dans l'ALU du NPU.

i	k_i	$\exp(k_i)$
0	5.5452	1/256
1	2.7726	1/16
2	1.3863	1/4
3	0.6931	1/2
4	0.2877	3/4
5	0.2877	3/4
6	0.1335	7/8
7	0.0645	15/16
8	0.0317	31/32
9	0.0157	63/64
10	0.0078	127/128
11	0.0039	255/256

TABLE 3.3 – Coefficients k_i pour la plage de données considérée.

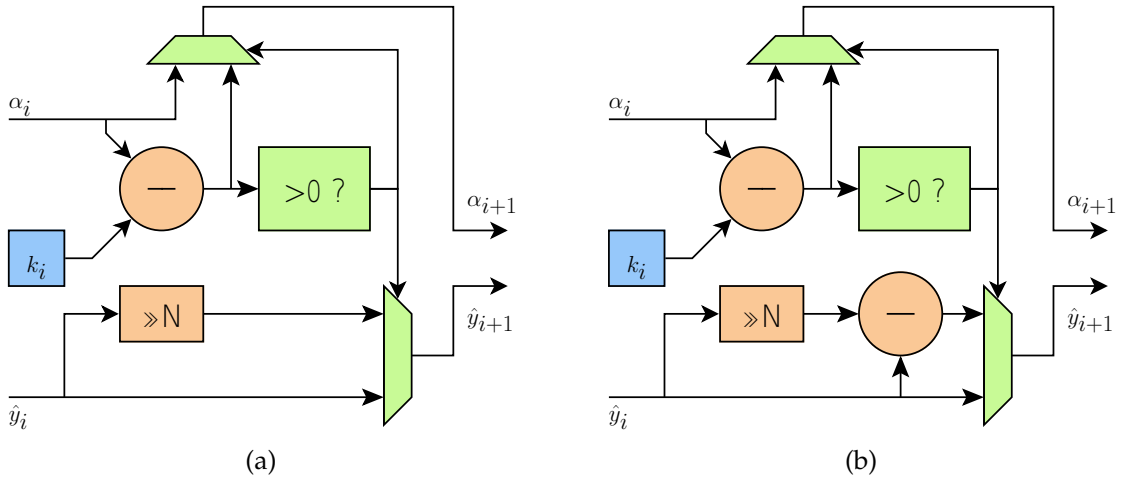


FIGURE 3.16 – **Élément de calcul rapide d'exponentielle.** Le bloc a) peut traiter les itérations de 0 à 3 et le bloc b) peut traiter les itérations de 4 à 11.

3.4.2 Résultats

L'architecture du NPU a été décrite en VHDL, testée en simulation et portée sur carte FPGA. Les équations ont été implémentées dans un langage assembleur, spécialement développé pour le NPU. Le comportement de la carte est le même que celui obtenu dans la section 3.3.2 pour l'architecture 16 bit. Ce résultat n'est pas surprenant, compte tenu du fait que les calculs sont les mêmes, avec la même représentation des nombres. Le calcul de la Gaussienne, donnant des résultats comparables dans les deux cas, n'affecte pas l'évolution de la carte.

Le NPU a été porté sur le FPGA Stratix V 5SGXEA7N2F45C2ES d'Altera équipant la carte DE5-NET de Terasic. Ce FPGA dispose de 234 k ALUT, 50 Mbit et de 256 blocs DSP. Comme pour la première implémentation matérielle, nous avons testé la scalabilité du réseau en augmentant progressivement les tailles de réseaux et en mesurant la quantité de ressources utilisées.

La consommation en ALUT, en registres, en mémoire et en blocs DSP, a été tracée en FIGURE 3.18. Les algorithmes de synthèse d'Altera ayant évolué entre ces deux mesures, et

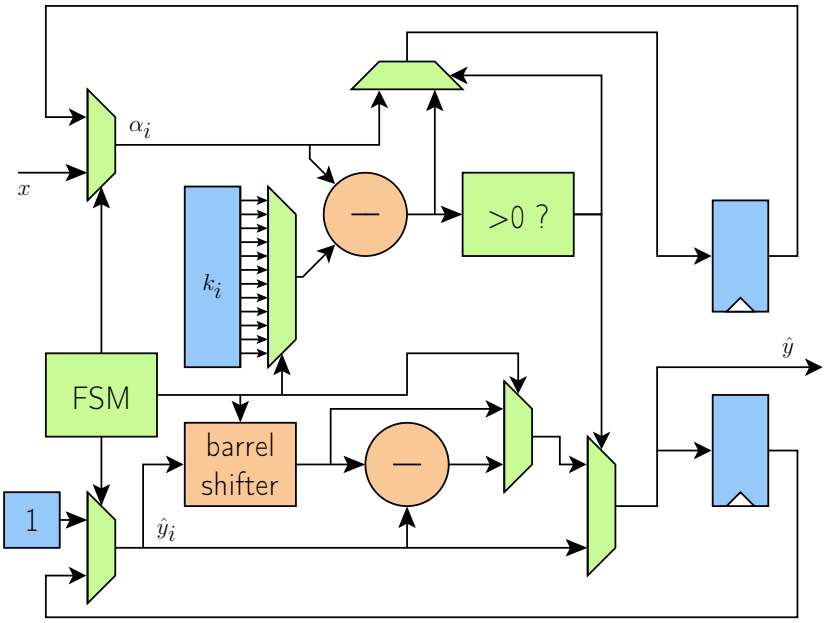


FIGURE 3.17 – **Module de calcul léger d'explonentielle.** Toutes les itérations sont traitées séquentiellement.

comme nous avons changé de cible entre les deux manipulations, la première architecture matérielle, configurée en 16 bit a été re-synthétisée. Les résultats obtenus sont reportées sur cette même figure.

On constate tout d'abord que le NPU est beaucoup moins gourmand en ressources que le neurone matériel, puisque nous pouvons instancier jusqu'à 684 NPU sur notre FPGA. Bien que le synthétiseur donne des résultats surprenants en terme d'utilisation de blocs DSP, nous pouvons instancier jusqu'à 256 neurones matériels. La limite de DSP pour les réseaux de NPU est atteinte pour une taille de 256, car chaque NPU utilise un bloc DSP. Le point d'inflexion sur la courbe d'utilisation d'ALUT que l'on observe aux alentours de 500 NPU est liée à l'utilisation des blocs DSP. En effet, chaque bloc DSP peut accueillir deux multiplieurs 16 bits, mais le synthétiseur choisit d'utiliser en priorité les blocs DSP vides. Ainsi, à partir de 512 NPU, les multiplieurs sont construits avec les ALUT du FPGA.

Nous avons regardé plus précisément la consommation des différents éléments internes aux NPU pour d'éventuelles futures optimisations. Dans une carte de taille 26×26, nous avons sélectionné deux neurones particuliers. Pour le premier, le multiplieur de l'ALU a été inféré par un bloc DSP et pour le deuxième directement par les ALUT du FPGA. Nous avons regardé la consommation en ALUT et en registres du module de communication, du contrôleur FSM, de l'ALU et du module de calcul d'exponentielles. Ces résultats sont montrés en FIGURE 3.19.

On constate dans un premier temps que les différents sous-éléments ont une consommation relativement homogène, si l'on excepte le multiplieur câblé de l'ALU. Il est intéressant de noter, que le module de calcul d'exponentielles est plus léger en termes d'ALUT que le multiplieur. Enfin, on constate que l'ensemble de la consommation en éléments logiques du NPU correspond à un peu plus de deux multiplieurs câblés. Ceci nous permet d'apprécier la légèreté de l'architecture du NPU, et d'anticiper la place occupée sur les technologies actuelles et futures.

Enfin, la fréquence de fonctionnement du circuit est plus stable par rapport à la taille du réseau, et reste supérieure à 100 MHz.

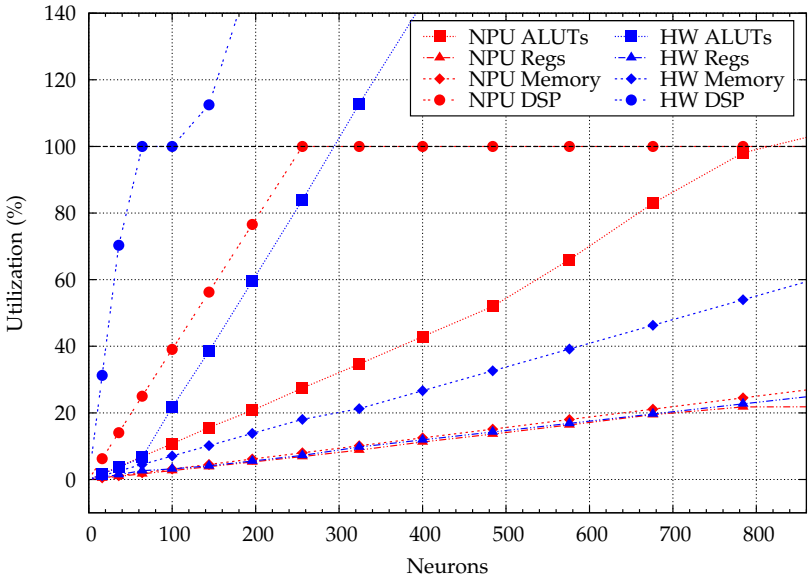


FIGURE 3.18 – **Résultats de synthèse du réseau de NPU.** Taux d'utilisation de différentes ressources en fonction de la taille du réseau.

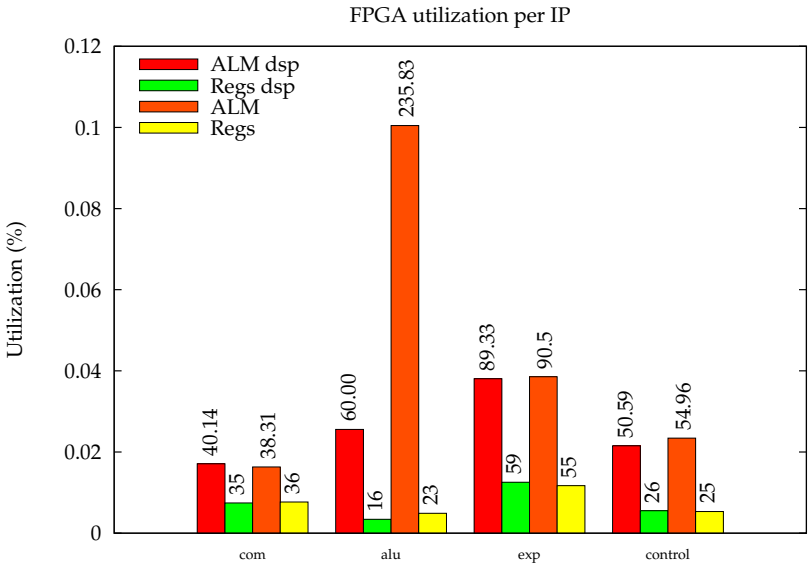


FIGURE 3.19 – **Consommation en ressources des différents sous-éléments du NPU.** Les résultats sont donnés pour un réseau de taille 26×26. Deux NPU différents sont comparés, le premier a son multiplieur inféré par un bloc DSP, le deuxième par les ALUT du FPGA.

L'un des atouts du NPU est sa programmabilité, qui permet de tester rapidement de nouveaux modèles neuronaux distribués et faiblement connectés. Les équations sont alors programmées dans un langage assembleur spécifique à ces processeurs neuronaux. Nous avons porté le même modèle que celui implémenté en matériel en section 3.3. Un exemple de code, correspondant au calcul d'une des itérations de la somme de Gaussiennes liée au potentiel afférent, est donné en FIGURE 3.20

```
; Variable definition
def x, 0x0C, 0          ; Input branch result
def y, 0x0D, 0          ; Lateral branch result
def param, 0x0E, -200   ; Gaussian Parameter
def act, 0x0F, 0        ; Activity
def l, 0x10, 0x0010     ; Learning rate

; Code sample
load x1                 ; Load stimuli
sub w1                  ; Sub related weight
squ                     ; Square acc
exp param               ; Exp(acc*param)
mul x                   ; Product of Exp
store x                 ; Store partial product
```

FIGURE 3.20 – Exemple de code pour le NPU

L'analyse de ce code nous permet de connaître la quantité de mémoire utilisée par le programme et par les données, ainsi que le nombre de cycles nécessaires au traitement d'une itération neuronale. Nous avons testé deux codes différents, le premier implémentant un réseau à deux entrées, et le deuxième à quatre entrées. Au vu de la linéarité inhérente du code, nous pouvons déduire l'utilisation mémoire et le temps de traitement d'un vecteur d'entrée de taille N. Les résultats obtenus sont présentés dans la TABLE 3.4.

On constate que la quantité de mémoire nécessaire pour stocker le programme et ses données est relativement faible, de l'ordre de quelques centaines d'octets. Ce résultat peut notamment être mis en rapport avec la taille des mémoires embarquées dans notre FPGA qui contiennent chacune 2 kiB. Le débit d'itérations neuronales a diminué par rapport à la première version matérielle, mais reste aux alentours du million d'itérations par secondes. Ce résultat reste toujours plus que satisfaisant par rapport à nos éxigences.

	2 entrées	4 entrées	N entrées
Programme (mots)	41	65	$17 + 12 \times N$
Données (mots)	17	21	$13 + 2 \times N$
Mémoire (octets)	116	172	$60 + 28 \times N$
Durée (cycles)	63	109	$17 + 23 \times N$
Durée (µs)	$0,63$	$1,09$	$f_{\text{circuit}} \times (17 + 23 \times N)$
Débit (Mit/s)	$1,6$	$0,9$	$\frac{1}{f_{\text{circuit}} \times (17+23 \times N)}$

TABLE 3.4 – Analyse de programmes neuronaux.

En somme, nous avons réussi à créer un modèle de carte auto-organisatrice plus léger, plus scalable, et nous permettant d'intégrer plus de neurones. Le débit neuronal est toujours supérieur de plusieurs ordres de grandeur par rapport à nos éxigences. La quantité de mémoire nécessaire pour stocker un programme neuronal et ses données est plus que raisonnable.

La suite logique pour intégrer des cartes toujours plus grandes est de multiplexer temporellement les NPU pour leur permettre d'exécuter plusieurs neurones logiciels.

3.4.3 Multiplexage temporel

Partant du constat que le budget temporel restant est conséquent, et que la mémoire disponible est peu utilisée, une évolution possible est de faire exécuter à chaque NPU une sous-grille de neurones logiciels, pour permettre l'implémentation de cartes plus grandes. Un exemple d'une carte neuronale 9×9 exécutée sur un réseau de NPU de taille 3×3 est montré en FIGURE 3.21. Nous avons voulu tester les limites de cette technique, à la fois en terme de mémoire, de temps d'exécution et de temps de communications.

Le multiplexage temporel impose d'apporter certaines évolutions à l'architecture. Dans un premier temps, nous devons considérer l'ajout de structures de sauts, car il n'est pas envisageable de dupliquer le code à l'ajout de neurones logiciels. Comme l'exécution de la sous-grille de neurones est régulière et prédictible, un mécanisme de boucle câblée est souhaitable.

Ensuite, les communications doivent être repensées. Il y a maintenant deux types de communications différentes à prendre en compte : les communications entre les neurones logiciels au sein d'un même NPU, et les communications entre les NPU. Les communications au sein d'un NPU peuvent se faire par le biais de variables partagées. L'activité calculée par un neurone logiciel peut facilement être relue par le neurone logiciel voisin. Les transmissions de données entre les NPU nécessitent de complexifier le module de communication. Au lieu de ne transmettre qu'une variable d'activité à ces voisins, ce module devra transmettre l'activité de tous les neurones de la bordure d'une sous-grille.

Enfin, le format des instructions tel que défini dans la sous-section 3.4.1, disposant d'un champ d'adresse de 11 bits, ne permet d'adresser que 4 kiB de mémoire. Pour pouvoir accéder à des mémoires de taille supérieure, la solution la moins lourde consiste implémenter un système simplifié de segmentation de mémoire. Ainsi un registre stockerait les bits de poids fort de l'adresse, alors que les bits de poids faible seraient contenu dans l'instruction.

Limites d'implémentation

Nous avons vu précédement que la limite d'implémentation de réseaux de NPU est due à la consommation en éléments logiques, en particulier de l'ALU. Ici, la limite d'implémentation de cartes de neurones logiciels est imposée par la quantité de mémoire totale disponible dans le FPGA. En faisant varier la quantité de mémoire disponible pour chaque NPU, on modifie le taux de neurones logiciels par NPU, noté N_{SW}/NPU. Ainsi, diminuer le ratio N_{SW}/NPU, permet de rendre l'exécution plus rapide, alors que l'augmenter réduit la consommation en éléments logique. Pour chaque configuration, on pourra donc estimer le temps nécessaire au traitement de la carte neuronale complète et vérifier si le budget temporel n'est pas dépassé.

Considérons le FPGA Stratix V 5SGXEA7N2F45C2ES utilisé précédement, bon représentant des technologies disponibles au moment de la rédaction de cette thèse. Sa mémoire est constituée de blocs `M20k` qui, dans notre cas, sont configurés avec une largeur de 16 bit pour une profondeur de 2 kiB. Ce FPGA dispose de 2560 blocs `M20k`, ce qui constitue un total de 5 MiB exploitable. Il est possible de cumuler plusieurs blocs `M20k` par NPU, avec une limite à un maximum 64 blocs par l'architecture 16 bit, ce qui nous donne une plage théorique allant de 2560 NPU à 2 kiB à 40 NPU à 128 kiB. En pratique, cette plage est limitée par les ALUT du FPGA à un maximum d'environ 700 NPU.

Nous avons mesuré la mémoire utilisée par le programme qui exécute les neurones, ainsi que par les variables de chaque neurone. Nous pouvons en déduire, pour chaque taille de mémoire, la quantité de neurones logiciels que peut exécuter un NPU. Comme nous pouvons savoir combien de NPU peuvent être implémentés sur notre FPGA en fonction de la quantité de mémoire attribuée, nous pouvons en déduire la taille de la carte totale. Pour

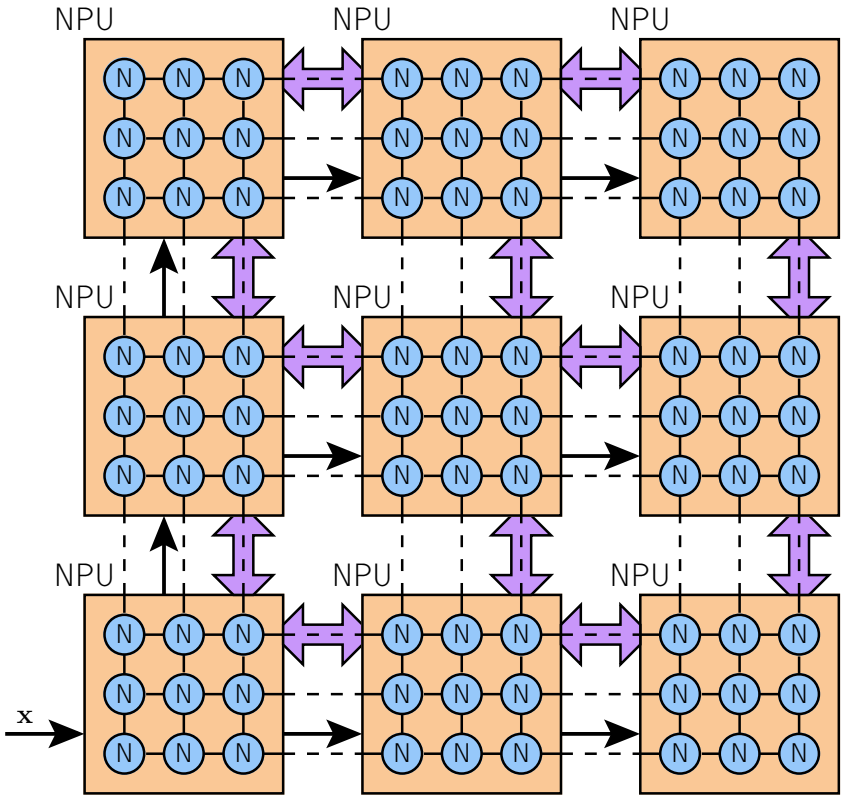


FIGURE 3.21 – **Exemple d'une carte neuronale** 9×9 **exécutée sur un réseau** 3×3**.**

faciliter l'implémentation, nous nous restreignons à l'utilisation de grilles carrées, à la fois pour le réseau de NPU et pour les sous-cartes de neurones logiciels. Les résultats obtenus sont résumés en TABLE 3.5. Les deux premières lignes sont données pour indication, et ne sont pas implémentables dans notre FPGA, à cause de la taille de la grille de NPU.

Mémoire	Grille NPU	Carrée	Sous-grille	Carrée	Grille totale
2 kiB	2560	2500	35	25	62 k
4 kiB	1280	1225	74	64	78 k
8 kiB	640	674	153	144	97 k
16 kiB	320	289	311	289	83 k
32 kiB	160	144	626	625	90 k
64 kiB	80	64	1256	1225	78 k
128 kiB	40	36	2516	2500	90 k

TABLE 3.5 – Taille du réseau de NPU, des sous-cartes de neurones logiciels et du réseau complet en fonction de la taille mémoire allouée aux NPU.

Limites temporelles

L'analyse menée jusqu'à présent nous a permis de savoir en quoi les ressources matérielles du FPGA limitaient la taille du réseau de NPU. Nous avons en outre pu augmenter la taille

de la carte neuronale en multiplexant temporellement les NPU pour leur permettre d'exécuter une sous-carte de neurones logiciels. La taille de la carte neuronale complète est ainsi limitée par la quantité de mémoire distribuée disponible dans notre FPGA.

Nous cherchons maintenant dans quelle mesure le multiplexage temporel affecte le débit neuronal de la carte complète. Nous devons d'abord mieux cerner les besoins dans un cas d'application concret. Prenons comme exemple le traitement des imagettes log-polaires discuté au chapitre 2. D'après les expérimentations que nous avons effectué, on estime qu'il peut y avoir un maximum de 20 points d'intérêt par échelle, menant à un maximum de 120 imagettes par image. Le débit en sortie de caméra est adaptable en fonction de la vitesse du robot, et peut atteindre jusqu'à 60 images par secondes. Nous avons tracé, en FIGURE 3.22, le débit neuronal nécessaire pour traiter toutes les imagettes, en fonction du débit de la caméra, pour des images ayant entre 60 et 120 points d'intérêt. Ces exigences varient donc entre 1200 et 7200 itérations neuronales par seconde.

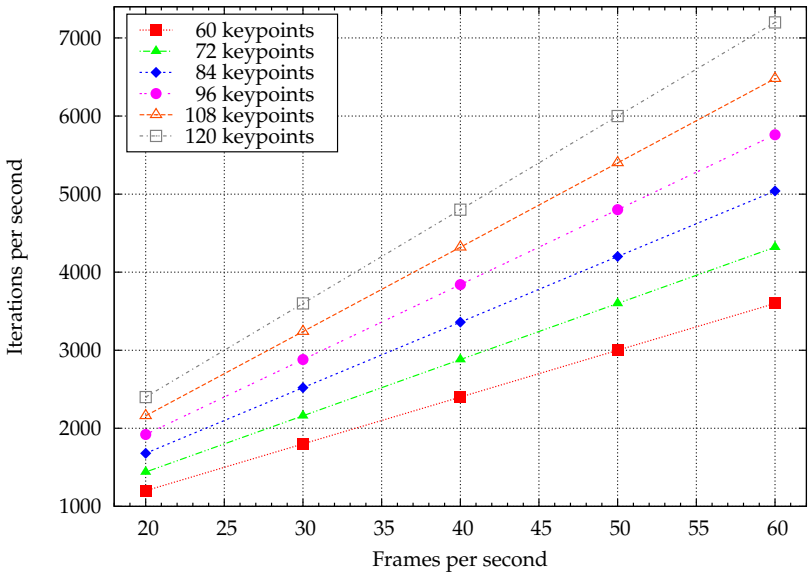


FIGURE 3.22 – **Exigences pour l'apprentissage temps-réel basé sur la vision.**

Pour mesurer le débit neuronal de la carte, nous mesurons le temps de traitement d'une itération, en comptant l'injection des données ainsi que la transmission de l'activité des neurones aux NPU voisins. Nous avons effectué ces mesures pour différents réseaux de NPU et pour différentes cartes de neurones logiciels. Les résultats sont montrés en FIGURE 3.23, avec la limite de 7200 itérations par seconde indiquée en pointillés horizontaux.

La limite imposée en terme de surface occupée par les neurones est similaire à la limite temporelle pour les exigences les plus fortes. Parmis le large panel de solutions, on peut choisir un jeu de paramètres qui maximise la taille de la carte neuronale ou qui minimise la quantité de logique consommée.

3.5 Discutions et perspectives

Après avoir présenté un modèle de carte auto-organisatrice distribuée et composée de neurones faiblement connectés, nous avons discuté de son implémentation matérielle. Nous en avons effectué une première implémentation fonctionnelle, mais difficilement exploitable,

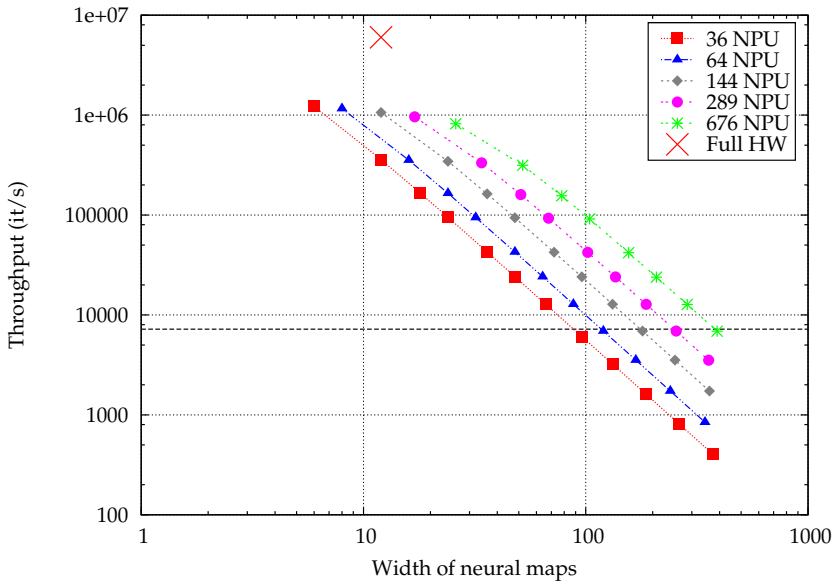


FIGURE 3.23 – **Débit de la carte en fonction du nombre de neurones logiciels** pour différentes tailles de réseaux de NPU. Le débit de la première implémentation est représentée par une croix rouge pour comparaison.

car complexe et consommant beaucoup de ressources matérielles. Cependant, au vu des bons résultats de cette architecture, et de la rapidité d'exécution, nous avons envisagé un nouveau modèle.

Ce modèle s'appuie sur des processeurs neuronaux spécifiques pour séquentialiser les traitements et économiser des opérateurs. Cette nouvelle architecture programmable nous permet d'apporter plus facilement des modifications aux modèles neuronaux.

Une perspective à court terme est d'intégrer le multiplexage temporel des neurones pour exécuter des cartes neuronales toujours plus importantes. Ensuite, le module de communication pourrait être étendu pour supporter d'autres modèles, notamment des neurones à spike.

CHAPITRE 4

Architecture de la couche programmable

Sommaire

4.1 Les architectures parallèles

Bien que les architectures parallèles aient été utilisées depuis de nombreuses années, dans les calculateurs haute performance notamment, sa démocratisation parmi le grand public ne s'est faite qu'assez récemment. Les architectures séquentielles ont en effet atteint des limites à

la fois physiques, en terme de fréquence, de consommation et de dissipation, mais aussi architecturales, les architectures pipeline et superscalaires n'étant pas scalable. L'année 2004 marque l'arrêt de la *course à la fréquence* et des architectures purement séquentielles avec l'annulation du processeur Pentium 5 d'Intel à cause d'une dissipation thermique trop importante, ainsi que du manque de performance par rapport aux versions précédentes, malgré une fréquence annoncée de 7GHz.

Les besoins computationnels ne cessant d'augmenter dans tous les domaines, scientifiques, commerciaux ou encore grand public, les architectures parallèles se sont imposées, sous la forme de processeurs multi-cœurs principalement.

4.1.1 Les niveaux de parallélisme

Pour améliorer la performance des architectures de calcul, le parallélisme a été introduit à différents niveaux. Cette sous-section couvre différents niveaux de parallélisme, mais il est important de noter qu'un processeur moderne bénéficie en général de parallélisme sur plusieurs de ces niveaux.

Le premier est le parallélisme au niveau bit. Il consiste simplement à augmenter la largeur des mots que le processeur manipule. Ceci diminue le nombre d'instruction à exécuter pour réaliser une opération sur des variables dont la taille dépasse la largeur du mot. Les processeurs sont ainsi passé rapidement d'architectures 4-bit à des architectures 32-bit, puis plus récemment aux architectures 64-bit.

Le second est le parallélisme au niveau instruction. Les instructions qui composent un programme peuvent être regroupées voire ré-ordonnées pour être exécutées en parallèle, sans que le résultat final n'en soit altéré. La forme la plus commune de parallélisme d'instruction est le pipeline, qui consiste à découper une instruction en N sous-actions. Les différentes sous-actions sont traitées en parallèle sur des instructions différentes. Ainsi, N instructions sont présentes en même temps dans le pipeline et sont traitées par une sous-action différente.

Certains processeurs peuvent voir leur pipeline, ou une partie de ce pipeline, dupliqué. On parle alors d'architecture superscalaire. Pour pouvoir être regroupées, les données manipulées par les instructions doivent être décorrélées.

Certains mécanismes peuvent améliorer les performances en présence de données corrélées. Parmi les plus courant on trouve l'exécution dans le désordre, l'exécution spéculative, et la prédiction de branchement.

Le troisième niveau est le parallélisme de donnée. Pour certains types de calculs, notamment le calcul vectoriel ou matriciel, les mêmes opérations sont effectuées sur plusieurs données différentes. On peut alors s'appuyer sur le parallélisme intrinsèque d'un jeu de donnée particulier pour accélérer les calculs. Ce niveau de parallélisme se retrouve notamment dans les architectures vectorielles et dans les GPU.

Enfin, le quatrième niveau discuté ici est le parallélisme au niveau tâche. Ce type de parallélisme consiste à faire exécuter des programmes différents sur des jeux de données identiques ou différents. Le programmeur doit alors séparer son problème en différentes sous-tâches.

4.1.2 La mémoire

L'une des problématiques majeures des architectures parallèles est liée à la mémoire. La mémoire peut être soit partagée entre les différents éléments de calcul, soit distribuée sur l'architecture. Dans le cas d'une mémoire partagée, les éléments de calcul ont accès à un seul et unique espace d'adressage, et peuvent donc s'échanger des informations *via* la mémoire. À l'opposé, dans le cas d'une mémoire distribuée, les éléments de calcul ont chacun leur propre espace d'adressage et doivent faire appel à un mécanisme de passage de message pour communiquer. La communication entre les éléments peut se faire à travers un réseau d'interconnexions. Chaque élément fonctionne de manière asynchrone ce qui impose un mécanisme de synchronisation entre les tâches.

Lorsque chaque élément peut accéder à la mémoire avec la même latence et le même débit, on parle d'accès uniforme à la mémoire (UMA pour *Uniform Memory Access*). C'est le cas des architectures à mémoire partagée. Si les éléments de calcul sont identiques on parle de multiprocesseur symétrique (SMP pour *Symmetric Multi-Processor*). À l'inverse, une architecture à mémoire distribuée aura un accès non-uniforme à la mémoire (NUMA pour *Non-Uniform Memory Access*).

Une architecture à mémoire partagée est plus simple à mettre en place, en particulier un système d'exploitation traditionnel peut être adapté. Elle est également plus facile à programmer, les communications étant facilitées par la présence de la mémoire partagée. Par contre, ces architectures ne sont pas scalables. Il est difficile de rajouter une grande quantité de cœurs de calcul à cause du goulot d'étranglement induit par la mémoire centrale.

L'implantation d'un système d'exploitation est plus compliquée dans une architecture à mémoire distribuée, puisqu'il doit également être distribué. La programmation est également plus compliquée, puisque l'intégralité de la mémoire n'est plus visible par chaque élément. Cependant, l'architecture devient plus facilement scalable, car aucun élément de calcul n'est connecté à un organe unique.

4.1.3 Classification des architectures parallèles

Les architectures parallèles peuvent être classées en différentes catégories selon différents critères.

L'une des premières classifications a été menée par Flynn [Flynn, 1972] et dépend de la présence ou non de concurrence dans les instructions et du parallélisme des données. Les quatre classes suivantes ont été identifées dans la Taxinomie de Flynn :

— Single Instruction Single Data (SISD)
— Single Instruction Multiple Data (SIMD)
— Multiple Instruction Single Data (MISD)
— Multiple Instruction Multiple Data (MIMD)

Un **système SISD** est un processeur mono-cœur capable d'exécuter une seule instruction à la fois, sur un flux de donnée unique. Les instructions et les données sont stockées dans une mémoire principale. La vitesse d'un SISD est limitée par le rythme auquel les informations sont transférées en interne. Un **système SIMD** est une machine multi-cœur capable d'exécuter une même instruction sur tous les cœurs de calcul à la fois, chaque cœur traitant un flux de données différent. Un **système MISD** est une machine multi-cœur capable d'exécuter différentes instructions sur différents cœurs de calcul, sur un même flux de donnée. Ces machines n'ont pas de cas d'utilisation bien défini et demeurent plutôt un exercice intellectuel.

Un **système MIMD** est une machine multi-cœur capable d'exécuter différentes instructions sur différents cœurs de calcul, sur différents flux de donnée. Les cœurs de calcul travaillent de manière indépendante et asynchrone sur des données et des instructions différentes.

On peut également classer les architectures selon le niveau de parallélisme sur lequel elles travaillent.

On retrouve ainsi les **processeurs multi-cœurs**, qui sont constitués de plusieurs unités de calcul sur une même puce. Ils diffèrent des processeurs superscalaires qui exécutent plusieurs instructions en parallèle provenant d'un même flux d'instructions. Les processeurs multi-cœurs quant à eux exécutent plusieurs flux d'instructions en parallèle. La quasi-totalité des processeurs grand-public actuels sont des processeurs multi-cœurs.

Les **processeurs vectoriels** s'opposent aux processeurs scalaires en exécutant leurs instructions sur une grande quantité de données. Ces architectures, héritées des super-calculateurs des années 1970-1980 n'existent plus sous forme de processeur indépendant, mais sont présentes dans les processeurs modernes, avec les jeux d'instructions AVX et SSE par exemple.

Une autre catégorie proche des processeurs vectoriels est formée par les **GPU**. Depuis quelques années en effet, les GPU sont utilisés pour le traitement de données fortement parallèle. On peut ainsi retrouver des cartes graphiques prévues pour le jeu vidéo, ou pour le monde professionnel, comme la famille Quadro de NVidia. Plus récemment les GPGPU Tesla de NVidia sont directement destinés à être utilisés pour traiter des données parallèles.

La catégorie des **architectures reconfigurables**, représentée par les FPGA, est intrinsèquement parallèle. Elles diffèrent des architectures citées précédemment dans le sens où le parallélisme opère à un grain plus fin. On trouve notamment chez Xilinx et plus récemment chez Altera des FPGA capable d'être reconfigurés dynamiquement et partiellement, ce qui permet plus de dynamicité. Ces deux constructeurs proposent également des solutions pour programmer les FPGA depuis un langage de plus niveau, les rendant accessible à un plus large public.

Enfin, la catégorie la plus récente est celle des **Many-cœrs**. Il s'agit ici d'intégrer plusieurs centaines de cœurs de calcul sur une même puce. Parmi les différents projets de manycores industriel, on citera le MPPA-256 de Kalray [de Dinechin et al, 2013] qui implémente 256 cœurs VLIW. Du côté académique, les travaux menés sur l'architecture TSAR [Almaless, 2014] ont cherché à y adapter des systèmes d'exploitation standards, tels Linux on NetBSD.

4.2 La reconfiguration dynamique parallèle

4.2.1 La reconfiguration dynamique partielle

La reconfiguration dynamique partielle, proposée dans certains FPGA haut de gamme modernes, consiste à modifier une partie d'un circuit pendant que le reste du circuit continue à fonctionner. L'utilisateur peut définir une zone reconfigurable, ce qui contraint l'interface entre la partie statique du FPGA et la partie dynamique. La zone reconfigurable peut alors être considérée comme un FPGA plus petit disposant de ses propres entrée-sorties *ad-hoc*.

Elle s'oppose à une reconfiguration classique, où l'intégralité du FPGA doit être en état de *reset* pour charger une nouvelle configuration. Dans une architecture de traitement matérielle conçue de manière modulaire, un composant peut être défini comme étant reconfigurable. Il peut alors être remplacé par un autre composant ayant une autre fonction, pour peu que son interface reste compatible. L'interêt de la reconfiguration dynamique partielle est de pouvoir modifier une partie du circuit sans affecter le fonctionnement du reste du circuit.

Un composant matériel peut donc être considéré de la même manière qu'une tâche logicielle. On peut alors appliquer des mécanismes bien connus dans le domaine des systèmes d'exploitations logiciel à ces tâches matérielles. Les recherches récentes et actuelles visent à appliquer efficacement de tels mécanismes.

L'accélérateur le plus adéquat est ainsi placé au moment opportun et peu être retiré dès qu'il a terminé son exécution. La reconfiguration dynamique partielle ajoute virtuellement de la surface de calcul car les composants matériels peuvent être multiplexés temporellement.

La reconfiguration dynamique partielle proposée dans les FPGA du commerce, propose des mécaniques intéressantes. Cependant, elle reste difficile à mettre en place, pour plusieurs raisons, à la fois techniques et intellectuelles.

Tout d'abord, les outils sont propriétaires, et peu documentés. La plupart des mécanismes utilisés dans les projets de l'état de l'art ont été trouvés par rétro-ingénierie. De plus, ces projets sont verrouillés sur une technologie spécifique, rapidement obsolète, et ne peuvent pas être généralisés sur d'autres technologies.

Ensuite, les outils actuels sont très peu automatisés, et demandent une grande quantité de travail fastidieux pour mettre en place les zones reconfigurables et leurs interfaces.

Compte tenu de la difficulté à mettre en place les mécanismes de systèmes d'exploitation nécessaires à l'encadrement de la reconfiguration sur les FPGA du commerce, il semble nécessaire de développer une plateforme intégrant ces mécanismes.

Outre ces difficultés, la reconfiguration dynamique partielle proposée actuellement n'est pas très optimisée en termes de performances.

Alors que le changement de contexte d'une tâche logicielle est facile à identifier et à mettre en place, l'opération est plus compliquée pour une tâche matérielle. Un accélérateur matériel peut être composé d'une grande quantité de registres et de plusieurs mémoires embarquées, spécifiques à cet accélérateur, et donc difficile à prévoir. Le contenu de ces registres et de ces mémoires doit être sauvegardé avant d'opérer la reconfiguration, pour pouvoir être restauré par la suite. Le surcoût de cette sauvegarde de contexte, ajouté au temps de reconfiguration d'une zone dynamique est plus long que le simple changement d'une tâche logicielle.

Par ailleurs, dans une configuration donnée, à cause de la délimitation de la zone dynamique, le circuit ne pourra pas être optimisé de la même manière que si le FPGA avait été entièrement statique. Ainsi, la surface occupée sera plus importante et la fréquence de fonctionnement du circuit sera réduite.

Enfin, les FPGA reconfigurables dynamiquement ne disposent que d'un port de reconfiguration interne. Cela impose que le composant gérant la reconfiguration soit centralisé. La reconfiguration ne peut alors être effectuée qu'à une granularité élevée à cause du temps de reconfiguration.

La reconfiguration dynamique étudiée actuellement laisse entrevoir un large panel de possibilité, mais ne permet pas le passage à l'échelle. Nous étudions dans la suite de ce chapitre comment mettre en place un mécanisme de reconfiguration dynamique, partielle, parallèle et distribuée, en nous basant sur les travaux menés sur la plateforme Confetti.

4.2.2 La plateforme Confetti

La plateforme Confetti est un assemblage d'éléments, identiques et interchangeables, appelés EStacks [Mudry et al, 2007; Vannel, 2007]. Les EStacks sont eux-mêmes constitués des

différents éléments suivants :

— ECell : l'unité de calcul de base de l'architecture,
— ERouting : la carte responsable de la communication,
— EDisplay : un écran tactile permettant d'interagir avec un utilisateur,
— EPower : qui fournit l'énergie aux autres cartes.

L'ECell

La carte ECell, dont l'ensemble forme la couche de calcul, constitue la brique de base de la plateforme Confetti. Il s'agit de la couche programmable par l'utilisateur et forme ainsi la pièce centrale de la plateforme.

Cette carte est construite autour d'un FPGA Xilinx Spartan 3 couplé à une mémoire Static Random Memory Access (SRAM) de $8\ \mathrm{Mbit}$ et d'une sonde de température. Les FPGA Spartan 3 sont configurables dynamiquement, mais non-partiellement. Les dimensions de cette carte sont très réduites – $26 \times 26\ \mathrm{mm}$.

Chaque ECell échange des signaux avec les autres composants du système au travers d'un connecteur spécifique. Ces signaux sont décrits dans la TABLE 4.1.

Type	Nombre	Description
Application	6 LVDS	Paires differentielles haute vitesse connectées au FPGA associé de la carte ERouting. 3 paires dans chaque direction.
Configuration	5	Signaux pour la configuration du FPGA. Les signaux de contrôle et le bitstream proviennent de la carte ERouting.
Horloge	1	Horloge à $50\ \mathrm{MHz}$ générée par la carte ERouting.
Reset	1	Signal de reset de l'ECell.
Communication	6	Bus de donnée destiné aux cartes ERouting et EPower pour l'affichage et les signaux de contrôle.
Température	3	Signaux de mesure de la température.
Alimentation	12	Tensions d'alimentation de l'ECell.

TABLE 4.1 – Signaux d'entrée/sortie de l'ECell

L'ERouting

La carte ERouting permet d'implanter différents type d'algorithmes de routage grâce à des lignes séries haute vitesse. Pour réduire le plus possible la charge des cartes ECell, la communication est implémentée sur la carte ERouting, de même que la gestion de la configuration et de l'écran.

Cette carte complexe mesure $192 \times 96\ \mathrm{mm}$. Elle est constituée d'une grille de trois par six nœuds de routage et peut accueillir dix-huit cartes ECell.

Chaque nœud de routage est constitué d'un FPGA, d'une mémoire Flash, de possibilités de communications et d'une sonde de température. Le FPGA est du même type que celui utilisé sur les cartes ECell, à la différence qu'il est constitué d'un boitier disposant de plus de broches. Il n'est reconfigurable qu'à travers une interface Joint Test Action Group (JTAG), ce qui le rends moins dynamique que l'ECell. Son objectif est de gérer la configuration du FPGA de l'ECell et de prendre en charge les communications. Il continue de fonctionner pendant que l'ECell est reconfiguré.

Le FPGA de routage dispose d'une mémoire Flash de $16\ \mathrm{Mibit}$ pour stocker jusqu'à 16 fichiers de configuration du FPGA de l'ECell. Cette mémoire peut également être utilisée par l'ECell pour stocker des données.

Deux bus sont réservés pour le contrôle, et pour la gestion de l'écran et des températures. Un dernier bus est constitué des paires différentielles séries à haute vitesse.

Ce bus est un des points les plus importants de la plateforme Confetti puisque c'est par ce biais que les données de l'application sont transmises. Il relie les FPGA de la couche de routage entre eux selon une grille 4-connexe. Il assure aussi la connexion entre chaque FPGA ECell et son FPGA de routage associé. Cette topologie a été choisie pour limiter des lignes de communication globales trop longues qui auraient limité la bande passante du réseau. La modularité et la scalabilité s'en trouvent alors accrues.

Les connexions entre les FPGA sont réalisées par le biais de drivers Low-Voltage Differential Signaling (LVDS) permettant d'atteindre des débits théoriques de l'ordre de $500\ \mathrm{Mbit/s}$. Une connexion entre deux FPGA est constituée de deux bus – un dans chaque direction – chacun constitué de trois bits.

La carte ERouting dispose également de connecteurs, reliés au FPGA situés au bord de la carte, et fournissent les mêmes connexions que les liens entre FPGA. Plusieurs EStacks peuvent être reliés par ce biais pour augmenter la surface de la plateforme.

L'EDisplay

Dans l'optique de servir de démonstrateur, un système d'affichage a été ajouté à la surface de l'EStack. Il consiste en un affichage couleur 30 bits à LED, capable d'afficher 48×24 pixels à un rythme de 100 rafraîchissements par seconde. Un FPGA Spartan est dédié à cet affichage.

Le but de cet affichage est de donner un aperçu de l'état du système, en montrant par exemple le fonctionnement à vitesse réduite, ou encore en affichant des informations de congestion du réseau, de température ou encore sur la configuration actuelle. Chaque ECell a accès à une sous-partie de l'écran, un carré de 8×8 pixels. Un capteur tactile a été placé à la surface de l'afficheur pour permettre l'interaction avec un utilisateur.

L'EPower

La génération des différentes tensions nécessaires aux trente-six FPGA présents sur la carte ERouting et sur les ECells nécessite une carte dédiée, la carte EPower. Cette carte, de la même taille que l'ERouting, fournit les différentes tensions nécessaires au fonctionnement de tous les composants à partir d'une alimentation 5V externe.

Elle embarque également un microcontrôleur superviseur dont le but est de vérifier plusieurs éléments, comme la stabilité des alimentations, la température au sein de l'EStack, et de manière générale de prévenir les pannes. Ce microcontrôleur supervise également la séquence de démarrage de l'EStack, qui consiste notamment à démarrer les alimentations, vérifier leur stabilité, configurer les FPGA de la carte ERouting et les initialiser. Si l'une de ces étapes ne se déroule pas correctement, en fonction de la gravité de l'erreur, le système complet peut être arrêté pour éviter d'éventuels dommages.

Cette carte dispose aussi d'un FPGA qui s'occupe de la gestion de l'écran de l'EDisplay. Il vient régulièrement lire le contenu de la mémoire vidéo ($8 \times 8 \times 24\ \mathrm{bit}$) de chaque FPGA et les fusionne pour reconstruire une image globale de 48×24 pixels.

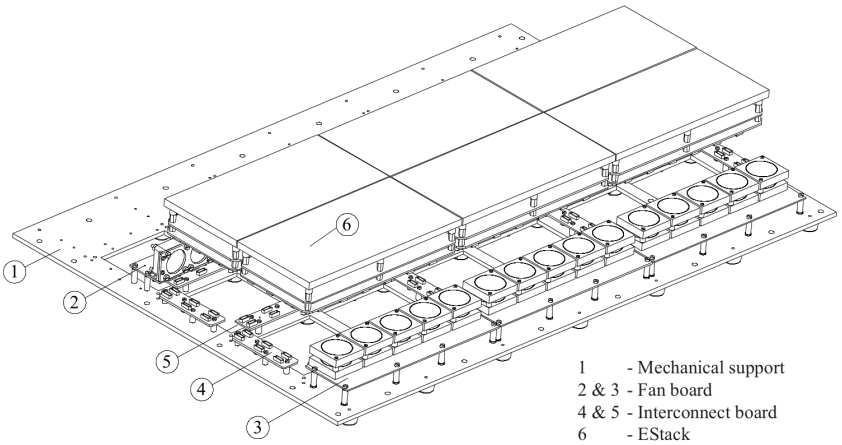


FIGURE 4.1 – **Exemple d'assemblage de la plateforme.** Elle est ici organisée selon une topologie 3×2.

Ce FPGA s'occupe également de lire l'état de la surface tactile pour envoyer le status du capteur à chaque ECell concerné.

La plateforme Confetti

La plateforme Confetti est un assemblage de plusieurs EStack. Grâce à la connectivité des EStacks, il est possible de créer une surface de logique programmable de taille indéfinie. La connexion des EStacks entre eux est rendue possible grâce à une structure mécanique constituée de cartes simples disposant uniquement de connecteurs. La plateforme peut être organisée dans une topologie 3×2 comme dans la FIGURE 4.1.

Compte tenu de la complexité de la plateforme et de la puissance consommée, il a fallu prendre en compte le problème de l'alimentation et de la chaleur dissipée.

Avec une consommation maximale estimée à plus d'1,5W par FPGA et à 30W pour l'écran, un EStack peut consommer jusqu'à 100W. Pour des raisons de simplicité et de coût, chaque EStack est alimenté par une alimentation d'ordinateur de bureau. Ce choix est discutable à posteriori, puisque les EStacks sont alimentés en 5V sous 20A ce qui a tendance à user prématurément les alimentations.

Avec une consommation pouvant aller jusqu'à 100W par EStack, la plateforme doit être convenablement refroidie. Pour cela, un jeu de ventilateur disposés en aspiration et en extraction provoquent un flux d'air dans l'ensemble de la plateforme. Ces ventilateurs peuvent être contrôlés par une carte externe qui mesure la température au sein des EStacks afin de limiter les nuisances sonores.

4.2.3 La programmabilité de la plateforme

Cette section a pour objectif de brosser un tableau général des problématiques bas niveau de programmation logicielle et matérielle de la plateforme.

Le routage

Pour permettre une communication globale dans Confetti, les FPGA de la carte ERouting hébergent un routeur matériel. Ce routeur est basé sur le *framework* Hermes développé dans [Moraes et al, 2004]. Les FPGA de routage sont organisés selon une grille à deux dimensions, ce qui facilite la scalabilité de la plateforme.

Le réseau ainsi formé est basé sur la commutation de paquets, avec un algorithme de routage de type *wormhole*. Pour plus de détails, le lecteur peut se référer à la section 4.3

Les communications locales

Le routeur Hermes a été modifié par Pierre André Mudry au cours de ses travaux de thèse [Mudry, 2009], en suivant deux principaux objectifs : la sérialisation/dé-sérialisation des données et la resynchronisation des différents FPGA.

Un système de sérialisation/dé-sérialisation des données a ainsi été conçu pour pallier à la connectivité restreinte entre les FPGA. Ce module a été nommé Mercury. Les trois paires différentielles disponibles entre chaque FPGA ont été employées selon le schéma suivant :

— `clk` : l'horloge de l'émetteur, transmise localement ;
— `dat` : les données sérialisées ;
— `rdy` : un bit de retour du récepteur indiquant s'il est prêt ou non à recevoir des données.

Une mémoire FIFO interne au récepteur permet la resynchronisation malgré la présence de différents domaines d'horloge entre le récepteur et l'émetteur.

La configuration

La configuration du FPGA d'un ECell est gérée par le FPGA correspondant dans la carte ERouting. Au démarrage de la plateforme, chaque FPGA de routage lit sa mémoire Flash et configure l'ECell avec le premier bitstream.

La programmation des mémoires Flash s'effectue à l'aide d'un bus dédié, disponible sur l'un des connecteurs latéraux des EStacks. Quand le FPGA relié à ce connecteur en reçoit l'ordre, il remplit sa mémoire Flash avec le fichier de configuration entrant. Une fois la mémoire flash programmée, le FPGA de routage peut lancer la phase de configuration de l'ECell. En parallèle, il retransmet l'ordre d'écriture ainsi que le bitstream à son voisinage. Le fichier de configuration traverse ainsi la plateforme suivant un schéma de *broadcast* en arbre. Il est intéressant de noter que les signaux impliqués dans la transmission de ce bitstream sont capables de traverser les EStacks.

Ce mode de configuration est assez limité et pourrait être étendu pour supporter la reconfiguration dynamique parallèle. Dans un premier temps, une extension du routeur Hermes permettrait d'écrire dans la mémoire Flash depuis les ports de communication haute vitesse Mercury. Le FPGA de routage pourrait alors lancer la reconfiguration de l'ECell sans interrompre le fonctionnement global de la plateforme. Pour rendre cette opération possible, il faut également que le FPGA de routage soit capable de lire le contenu de la mémoire Flash et de le diffuser sur le réseau Hermes.

Dans un second temps, il est envisageable de se passer de la programmation de la mémoire Flash en mettant en place un pont entre le port de communication Mercury et le port de configuration du FPGA de l'ECell.

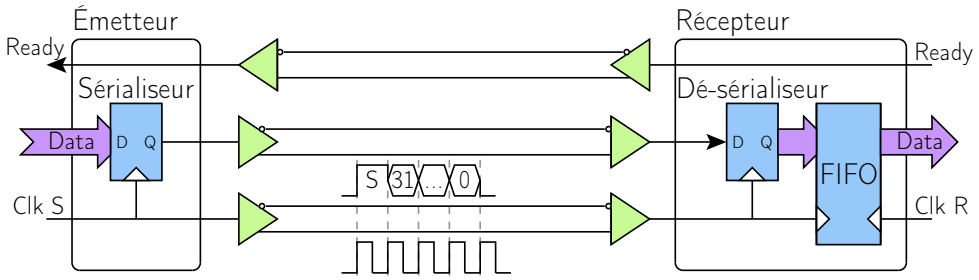


FIGURE 4.2 – **Architecture d'une paire émetteur/récepteur Mercury.** Les signaux *clock* et *data* sont utilisés pour transmettre des données de l'émetteur au récepteur. Le signal *ready* indique à l'émetteur si récepteur est prêt ou non à accepter une donnée.

4.3 Le réseau d'interconnexions

Le réseau d'interconnexions joue un rôle important dans tout système disposant d'une grande quantité d'éléments communicants. Notre architecture, déployé sur la plateforme Confetti ne déroge pas à cette règle. Les éléments de notre architecture sont constitués d'une tuile de calcul et d'un nœud de routage. Sur la plateforme Confetti, chaque élément est réparti sur deux FPGA distinct. Le premier FPGA, l'ECell, héberge la tuile de calcul, et le deuxième FPGA, qui fait parti de la carte ERouting, héberge le nœud de routage.

La tuile est ainsi directement en contact avec son routeur, qui lui-même est connecté aux quatre routeurs voisins. Le bus qui relie la tuile au routeur est le même que celui qui relie les routeurs entre eux, et est constitué d'un nombre limité de signaux. Il faut donc sérialiser les données à transmettre sur ces bus. Nous avons utilisé le *transceiver* développé par Mudry dans [Mudry, 2009] lors de la conception de la plateforme Confetti. L'émetteur est constitué d'un registre à décalage 32-bit pour sérialiser les données avant de les envoyer sur le réseau. Le récepteur est également constitué d'un registre à décalage 32-bit pour dé-sérialiser les données. Il est couplé à une mémoire FIFO *dual-clock* pour resynchroniser les deux domaines d'horloge. Le bus reliant un émetteur à un récepteur est constitué de trois signaux : *clock*, *data* et *ready*. Une paire émetteur/récepteur est représentée en FIGURE 4.2.

Le *transceiver* Mercury permet une communication entre les FPGA avec un nombre réduit de câbles, mais seulement sur un voisinage local. Les nœuds de routage instanciés dans les FPGA de la carte ERouting prennent en charge le routage des données au sein du réseau, permettant une communication entre deux tuiles quelconques. Sur cette plateforme de prototypage, nous voyons l'ensemble des cartes ERouting comme un vaste réseau sur puce.

Nous utilisons le *Framework* Hermes [Moraes et al, 2004], qui a aussi été utilisé par Mudry. Le NoC Hermes est basé sur une technique de commutation de paquets, où les paquets sont constitués d'une en-tête suivi des données. L'en-tête contient les coordonnées (x, y) du destinataire ainsi que la taille du paquet.

Le réseau d'interconnexion est composé de routeurs disposant de cinq ports de communication. L'un d'entre eux est le port local, qui est connecté à la tuile. Les quatre autres sont les ports externes et sont connectés aux routeurs voisins dans les quatre directions cardinales.

Les routeurs permettent une connexion point à point entre chaque paire de tuile dans l'architecture en utilisant l'adresse incluse dans l'en-tête pour rediriger les paquets. Le réseau Hermes utilise un algorithme de routage de type wormhole [Ni and McKinley, 1993] où les données sont divisées en mots de 32 bits appelés *flits*. Chaque routeur Hermes ne stocke qu'un flit, le flit d'en-tête, montré en FIGURE 4.3, qui contient les informations de routages, comme

l'adresse de destination et la longueur du paquet.

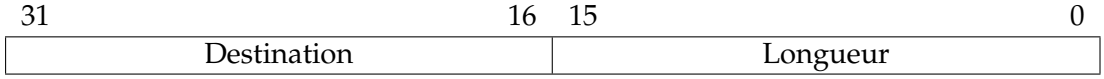


FIGURE 4.3 – **Flit d'en-tête**

Il transmet alors ce flit au prochain routeur en fonction de ses propres coordonnées et de l'adresse de destination. Le décodage de ce flit par les routeurs concernés induit la création d'un chemin dynamique par lequel les données transitent de manière pipelinée. Le paquet progresse alors à la manière d'un vers dans le réseau. Une fois que le paquet a fini de transiter, les routeurs redeviennent disponibles pour accepter un nouveau paquet. Le routeur de destination transmet simplement l'ensemble des flits à sa tuile qui peut ainsi reconstruire le paquet.

L'implémentation de Mudry a apporté plusieurs évolutions par rapport au *framework* Hermes d'origine. D'abord, les flits sont sérialisés par les *transceivers* Mercury pour réduire la quantité de signaux entre les FPGA. Deuxièmement la plateforme Confetti est construite selon un paradigme Globally-Asynchronous Locally-Synchronous (GALS). Ce paradigme permet d'augmenter la scalabilité de l'architecture sans en sacrifier la fréquence de fonctionnement. Enfin, un deuxième flit d'en-tête a été rajouté pour enrichir les fonctionnalités du réseau. Il n'est pas pris en compte par les routeurs, puisqu'il est considéré comme un flit quelconque du message. Il est constitué de trois champs comme montré en FIGURE 4.4. Le premier contient les coordonnées dans le réseau de la tuile émettrice. Le deuxième champ indique le type du message, s'il est point à point ou broadcast, avec ou sans accusé de réception. Le dernier champ indique un numéro de séquence dans le cas où un message devait être séparé en plusieurs paquets.

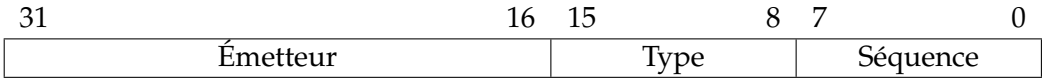


FIGURE 4.4 – **Flit d'extension**

Le broadcast n'étant pas nativement géré par les routeurs, la solution utilisée par Mudry a été de décoder l'extension de l'en-tête au sein des éléments de calculs pour décider de retransmettre ou non le paquet.

Nous avons rajouté à ces évolutions un bloc logique de configuration pour que les FPGA de la carte ERouting puissent configurer les FPGA des ECells. Ce bloc permet de propager un bitstream vers les FPGA voisins et vers sa propre mémoire Flash. Les FPGA des ECells peuvent alors être configurés par ce même bloc avec l'un des bitstreams contenus dans la mémoire Flash.

Enfin, un bloc de logique General-Purpose Input/Output (GPIO) communique avec l'ECell pour lui transmettre son adresse dans le réseau et l'état de la surface tactile, et pour lire les pixels à afficher sur l'écran.

L'architecture de routage est montré en FIGURE 4.5

4.4 Une unité de calcul élémentaire

À l'instar du reste du projet, nous avons conçu la tuile de calcul pour être la plus légère possible. Dans cette phase du projet, l'objectif de cette tuile n'est pas de se comparer en terme

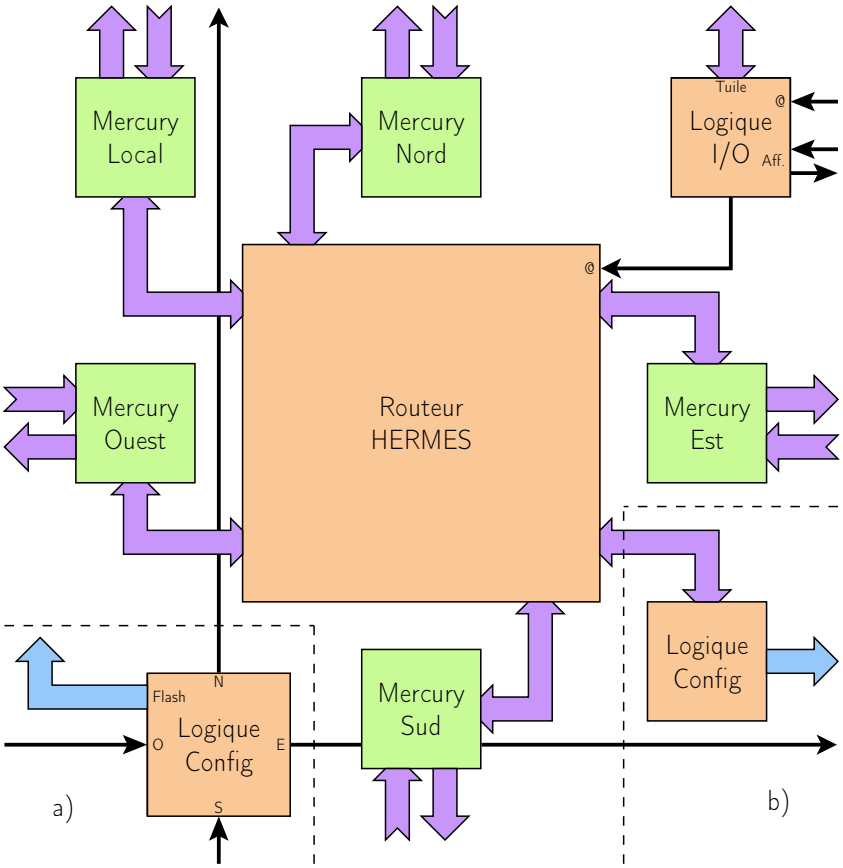


FIGURE 4.5 – **Architecture interne des FPGA de routage.** Le routeur Hermes et les sérialiseurs/dé-sérialiseurs Mercury assurent les communications entre les ECells. Un bloc d'entrée/sorties communique à l'ECell, à l'aide d'un autre port, les informations relatives à l'écran et à l'adresse dans le réseau. Un module de configuration peut être indépendant (voir l'encart a)), ou lié au routeur (encart b)).

de performances à d'autres solutions plus matures. Notre objectif au contraire est de concevoir une tuile légère qui puisse nous permettre d'évaluer les capacités d'auto-organisation de notre architecture.

La tuile de calcul, dont l'architecture interne est représentée en FIGURE 4.6, est constituée du soft-processeur, de sa mémoire de code, de sa mémoire de données et de plusieurs périphériques. Parmi ces périphériques, on trouve un timer et un accès aux différents éléments de Confetti, tels l'écran et la surface tactile, ou encore l'adresse de la tuile dans le réseau. On retrouve aussi le module Mercury, directement connecté au nœud de routage Hermes, permettant de communiquer avec les autres tuiles. Le NPU de la carte auto-organisatrice est également accessible par le processeur grâce au bus périphérique.

Enfin, un DMA spécifique a été développé pour décharger le processeur des différentes communications pouvant survenir.

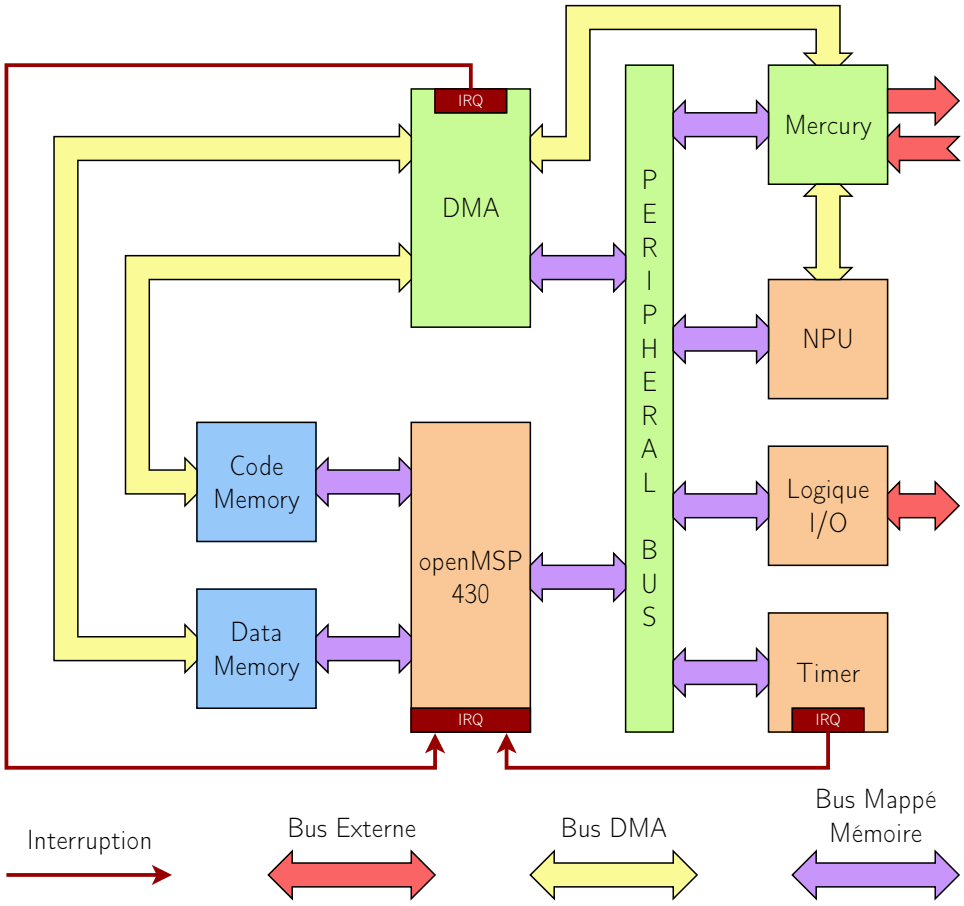


FIGURE 4.6 – **Architecture interne de la tuile de calcul.**

4.4.1 Le processeur

La tuile de calcul est ainsi construite autour du microcontrôleur 16-bit openMSP430, mis à disposition par la communauté OpenCores [OpenCores, 1999]. Nous avons choisi d'utiliser ce soft-processeur selon deux critères : l'indépendance technologique et l'accès libre au code source. Ainsi, nous pouvons envisager d'expérimenter les mécanismes d'auto-reconfiguration sur des plateformes provenant de différents constructeurs. De plus l'accès au code nous paraît indispensable pour garder la possibilité de l'adapter à notre cas d'utilisation particulier. Pour ces raisons, nous nous sommes tournés vers une solution *Open Source*.

Parmi le large choix de soft-processeurs *Open Source*, notre choix c'est porté sur l'openMSP430 car le projet est mature et régulièrement maintenu. De plus le code est bien documenté, ce qui nous permet d'ajouter facilement nos propres IP, voire de modifier le code du processeur. Enfin, comme son jeu d'instructions est compatible avec la famille de micro-contrôleurs MSP430 de Texas Instruments, il peut exécuter le code généré par les outils de compilation du MSP430, ce qui rend le développement plus aisé.

Ce soft-processeur souffre néamoins de l'absence de bus multi-maître, ce qui complexifie l'intégration d'un DMA, pourtant nécessaire aux nombreuses communications présentes dans notre projet.

4.4.2 Les communications

Pour pallier à l'absence de bus multi-maître de l'openMSP430 énoncée plus haut, nous tirons parti des mémoires *dual-port* embarquées dans le FPGA. Le DMA peut ainsi accéder à un port de la mémoire pendant que le processeur y accède par le second port. Il se connecte aux mémoires de code et de données d'une part, et aux périphériques d'autre part. Il doit donc être capable d'accéder matériellement aux périphériques. Comme on ne peut pas non plus avoir un autre maître sur le bus périphérique du openMSP430, nous avons ajouté un bus dédié aux périphériques concernés. Le DMA se connecte donc aux périphériques à travers des canaux dédiés, formés par des interfaces de type *Streaming*.

Actuellement, seul le périphérique Mercury est pris en charge par le DMA. Pour économiser quelques ressources FPGA, les transmissions de données du processeur au reste du réseau par le périphérique Mercury ne peuvent être effectuées que depuis le DMA. Ces connexions permettent les échanges suivants :

— De la **mémoire de donnée** au **périphérique Mercury**. Elle sert naturellement à émettre des données sur le réseau.
— Du **périphérique Mercury** à la **mémoire de donnée**. De manière analogue, elle sert à recevoir des données d'une autre tuile.
— Du **périphérique Mercury** à la **mémoire de code**. Elle sert à reprogrammer le processeur.
— De la **mémoire de code** au **périphérique Mercury**. Elle sert à reprogrammer une autre tuile.

Le périphérique Mercury étant prévu pour transmettre des flits de 32 bits a du être adapté pour être utilisable par le processeur 16 bit. Une petite machine à états permet d'attendre d'avoir deux mots de 16 bits avant de lancer l'émission d'un flit de 32 bits.

Le périphérique Mercury dispose de deux ports DMA. Alors que le premier est évidemment connecté au DMA, le second est connecté au NPU. Le NPU peut ainsi, grâce à un système d'arbitrage interne au périphérique Mercury prendre la main sur le réseau pour envoyer et recevoir des données aux NPU voisins.

Certains bits inutilisés du *header* servent à indiquer la nature des trames. On distingue dans un premier temps les données processeurs, les données de reprogrammation et les données neuronales. Pour ne pas encombrer le processeur avec la gestion des données ne le concernant pas, le décodage est effectué au sein du transceiver Mercury. L'interface du DMA aux mémoires de code et de données, respectivement pour la reprogrammation et les données processeurs, se fait à l'aide d'un bus adressable classique.

Pour la réception de données côté processeur, nous avons défini un *buffer* de donnée, dans lequel le DMA écrit les données les unes à la suite des autres. Quand il arrive à la fin du *buffer*, il reboucle. Quand l'utilisateur demande une lecture au travers de la fonction `read` de l'Application Programming Interface (API), le contenu du *buffer* est recopié. Pour savoir quelles données sont *fraiches*, la fonction `read` doit avoir accès au pointeur d'écriture.

De la même manière, pour ne pas écraser des données non lues, le DMA doit avoir accès au pointeur de lecture. En ce qui concerne la mémoire de code, chaque programme sera écrit à la même adresse. Il n'y a donc pas de gestion de *buffer* circulaire.

À l'émission on distingue trois cas. La donnée vient de la mémoire de donnée, de la mémoire de code, ou du NPU. Dans les deux premiers cas, le processeur donne l'adresse de départ, la quantité de donnée à transmettre et initie la transmission. Dans le troisième cas, c'est le NPU qui se charge de ces points. Le périphérique *Mercury* joue donc le rôle d'arbitre, en donnant la priorité au NPU ou au processeur.

Le *Mercury* étant capable d'initier des transferts DMA, il doit également indiquer au DMA à quelle mémoire le transfert s'adresse. Un signal permet d'indiquer au DMA que les données concernent la mémoire de code.

Certains choix fait en amont des travaux sur la plateforme Confetti ont ajouté quelques difficultés au moment de l'intégration. On pensera particulièrement à l'architecture 16 bits du processeur choisi qui entre en contradiction avec les architectures 32 bits déjà en place. La tuile, dans son état actuel, n'exploite pas toute les spécificités de la plateforme Confetti. Il manque en particulier la lecture des sondes de températures, et la gestion des mémoires Flash et SRAM depuis la tuile.

En dépit de ces limites, l'implémentation de la tuile de calcul dans les FPGA ECell, associée à quelques adaptations sur les FPGA de routage ont pu mener à plusieurs expérimentations sur les performances du réseau Hermes, puis au déploiement d'automates cellulaires, mettant en œvre une grande partie des périphériques de la plateforme.

4.5 Résultats

4.5.1 Déploiement de l'architecture

Le déploiement et le prototypage d'une nouvelle architecture sur la plateforme Confetti n'est pas un problème trivial. La difficulté vient entre autres de la quantité de FPGA à programmer, et de leur organisation en deux couches, mais aussi du manque de solution de *debug* efficace, et de l'asynchronie de la plateforme.

L'ERouting

Deux ports JTAG sont prévus pour la configuration des FPGA de la carte ERouting. Le premier, partagé par les 18 FPGA de cette carte, n'est pas utilisable sans apporter de modifications à la sonde JTAG, qui n'est pas prévue pour gérer autant de FPGA. Le second accède à trois mémoires flash, chacune reliée à 6 FPGA. Il nécessite cependant un reset de la carte ERouting pour charger le fichier de configuration dans les FPGA et ne permet aucune forme de debug. De plus, chaque EStack doit être programmé séparément.

Dans un premier temps, deux configurations étaient nécessaires, la première permettant de configurer les FPGA des ECell, et le deuxième contenant le routeur en lui-même. Dans le but de simplifier la phase de configuration des ECell, nous avons fusionné la gestion de la configuration avec le routeur Hermes (voir FIGURE 4.5a)). À cause de la place limitée des FPGA de routage, il a fallu supprimer le superflux et ne garder que le nécessaire au routage, à la configuration et à la gestion de l'écran.

Le taux d'utilisation de cette architecture est donné en TABLE 4.2. Un résultat plus détaillé est montré en FIGURE 4.7, avec le taux d'utilisation des différents sous-blocs de l'architecture des FPGA de routage. Compte tenu du taux d'utilisation élevé, l'algorithme de placement-routage doit utiliser des LUT du FPGA pour le routage. La fréquence globale du système en est impacté, la fréquence maximale atteignable du circuit est de 50 MHz pour le routeur et de 100 MHz pour les sérialiseurs/dé-sérialiseurs Mercury.

Resource	Utilisation	Maximum	%
LUT	3168	3840	82
Registres	1683	3840	43
Slices	1918	1920	99

TABLE 4.2 – **Taux d'utilisation de l'architecture de routage** sur FPGA `xc3s200-ft256`.

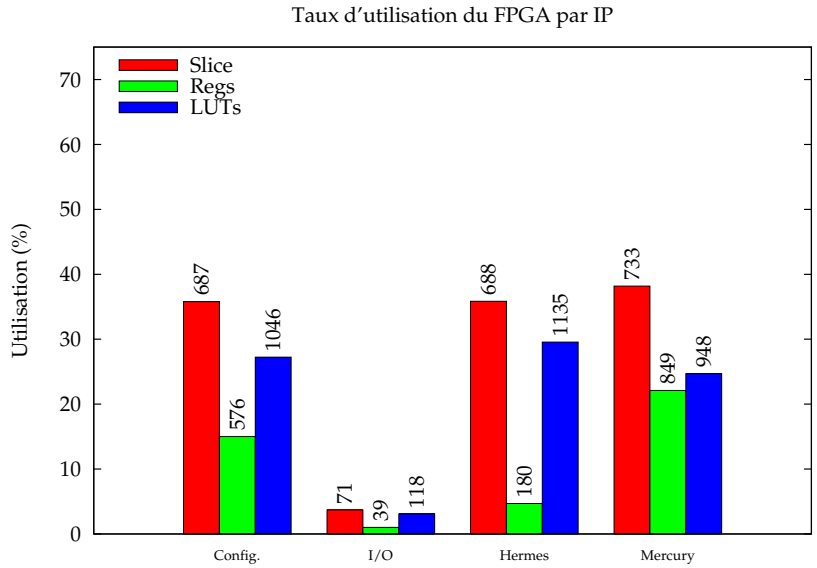


FIGURE 4.7 – **Résultats d'implémentation de l'architecture de routage.** Taux d'utilisation des modules de configuration, de gestion des entrées/sorties, du routeur Hermes et des sérialiseurs/dé-sérialiseurs Mercury. Le nombre de *slices* est donnée à titre indicatif. Si une *slice* contient deux *LUT* appartenant à deux modules différents, elle sera comptabilisé deux fois.

L'ECell

Avec une architecture de routage et de configuration unifiée, le déploiement d'une architecture matérielle de calcul est facilité. Une carte d'interface se connecte sur le FPGA sud-ouest de la plateforme, par le biais de deux signaux série asynchrone Universal Asynchronous Receiver/Transmitter (UART). Le premier sert à transmettre des informations de contrôle, et le deuxième transporte le bistream en lui-même. Ces deux signaux UART sont transmis entre les EStacks, ce qui permet une configuration de la plateforme entière. Le bitstream est ainsi chargé dans les mémoires Flash réservés aux ECell, et peut être chargé sur requête de l'UART de contrôle.

L'architecture présentée en section 4.4 a également nécessité quelques optimisations pour entrer dans les FPGA de la plateforme Confetti. Le NPU a ainsi dû être retiré de la tuile. Le taux d'occupation après ces modifications est donné en TABLE 4.3. Un histogramme du taux d'occupation par sous-bloc est également donné en FIGURE 4.8. La fréquence de fonctionnement de la tuile est également réduite à cause du taux d'occupation élevé de la tuile. Elle atteind 27 MHz pour le cœur de la tuile et 100 MHz pour les sérialiseurs/dé-sérialiseurs Mercury.

Resource	Utilisation	Maximum	%
LUT	2885	3840	75
Registres	1406	3840	36
Slices	1889	1920	98

TABLE 4.3 – Taux d'utilisation de l'architecture de la tuile de calcul sur FPGA xc3s200-vq100

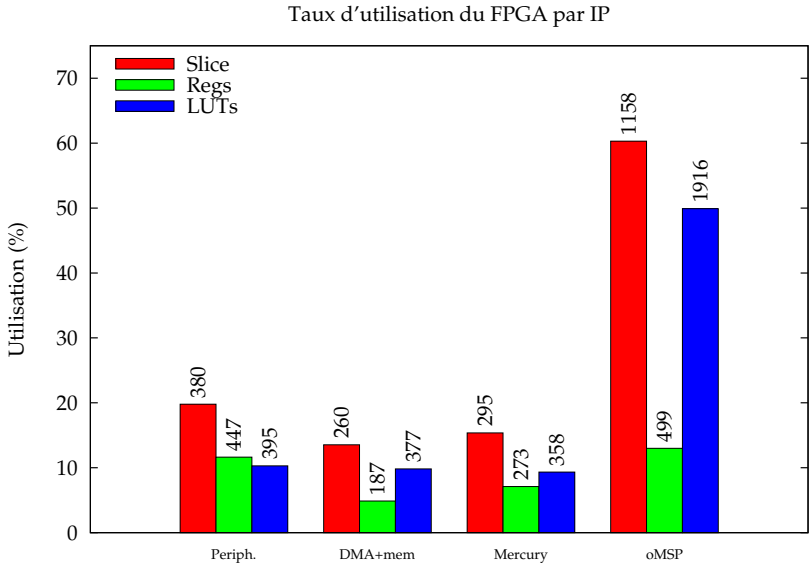


FIGURE 4.8 – **Taux d'occupation des différents modules d'une Tuile.** Le nombre indiqué représente le nombre de *slices* utilisé par ce module. Il est à noter que si une *slice* contient deux LUT appartenant à deux modules différents, elle sera comptabilisé deux fois.

4.5.2 Déploiement d'une application

Le déploiement d'une application logicielle simple a pour but de mettre à l'épreuve la programmabilité de la nouvelle tuile intégrée dans l'environement distribué et asynchrone qu'est la plateforme Confetti.

La tuile, construite autour d'un soft-processeur openMSP430 dispose de peu de mémoire, inférée sous forme de blocs de mémoire BRAM. Le contenu des mémoires est alors intégré au fichier de configuration du FPGA. Il est donc possible de programmer l'espace logiciel en même temps que l'espace matériel par la méthode décrite plus haut. Comme cette solution nécessite de re-synthétiser et de reconfigurer toute l'architecture, il a fallu trouver une alternative.

Le DMA de la tuile présentée en section 4.4 permet d'accéder à la mémoire de code du soft-processeur. Un petit programme préchargé à la configuration du FPGA est chargé d'écouter les données entrantes et de programmer la mémoire de code le cas échéant. Ce *bootloader* peut rester actif pendant l'exécution du logiciel pour une reprogrammation, pilotée par la tuile elle-même, une autre tuile, ou encore l'utilisateur, à travers la carte d'interface. Ce mode de programmation est par nature scalable puisqu'il s'appuie sur le réseau de routage de la plateforme.

La première application déployée sur la plateforme est un automate cellulaire de type Jeu de la Vie de Conway [Conway, 1970]. Les cellules du Jeu de la Vie sont organisées selon une

grille à deux dimensions et peuvent prendre deux états distincts : vivantes ou mortes. À chaque cycle, le nouvel état de chaque cellule dépend de l'état actuel et de l'état des cellules voisines. Il est calculé de la façon suivante :

— Une cellule morte possédant exactement trois voisines vivantes devient vivante,

— Une cellule vivante possédant deux ou trois voisines vivantes le reste, sinon elle meurt.

L'implémentation de cet automate met en œvre différentes fonctionnalités de la plateforme. Elle a en effet permis de tester le réseau de routeurs Hermes, pour des communications locales et distantes. L'écran et la surface tactile sont aussi sollicités pour intéragir avec un utilisateur.

Dans notre cas, chaque cellule est représentée par une LED de l'écran. Une cellule vivante est représentée par une LED allumée et une cellule morte par une LED éteinte. Ainsi, chaque tuile de calcul travaille sur une grille de taille 8×8 et partage la bordure de sa grille avec ses voisins directs. Dans le but de tester les communications sur de plus longues distances, nous avons choisi d'implanter une version torique de l'automate.

La re-synchronisation en terme de cycles de l'automate est assurée par un mécanisme de lecture blocants. Lors d'une transmission de données, le DMA du récepteur copie automatiquement le paquet dans un *buffer* interne. L'appel à la fonction `read` copie le buffer interne vers le buffer de l'utilisateur, et le débloque. L'automate reste alors localement synchronisé car une tuile ne peut pas commencer une itération avant que son voisinage n'ait terminé l'itération précédente.

4.5.3 Le réseau Hermes

Deux expériences ont été menées pour caractériser le débit et la latence du réseau Hermes. L'objectif de ces manipulations est double : d'une part il permet d'estimer les temps de configuration des FPGA ECell, d'autre part, il permet de connaître le débit disponible pour une application.

Débit et latences optimales

La première expérience consiste à transmettre des données entre deux ECell adjacents. Cela permet de mesurer le débit maximal et la latence induite par la paire émetteur/récepteur et par une paire de routeurs Hermes. L'un des deux ECell, l'émetteur, envoie une série de paquets à l'autre ECell, le récepteur. Le récepteur, renvoie chaque paquet à l'émetteur, qui peut mesurer le temps de transmission global.

Pour mesurer indépendamment le débit et la latence, l'émetteur envoie des paquets de taille différente. Sur la courbe du temps de transmission en fonction de la taille du paquet, montrée en FIGURE 4.9, l'ordonnée à l'origine nous indique la latence et la pente de la courbe nous renseigne sur le débit.

L'expérience a été menée avec trois stratégies différentes au niveau du récepteur pour mesurer son impact sur la transmission :

1. Recopie complète du buffer par un appel à la fonction `read`,
2. Suppression des données mais incrémentation du pointeur du buffer,
3. Suppression des données et non prise en compte du pointeur.

Ces trois expériences ont été faites à une fréquence de fonctionnement de 20MHz et ont donné les résultats montrés en FIGURE 4.9. Un récapitulatif des résultats extraits de ces courbes est donné dans la TABLE 4.4.

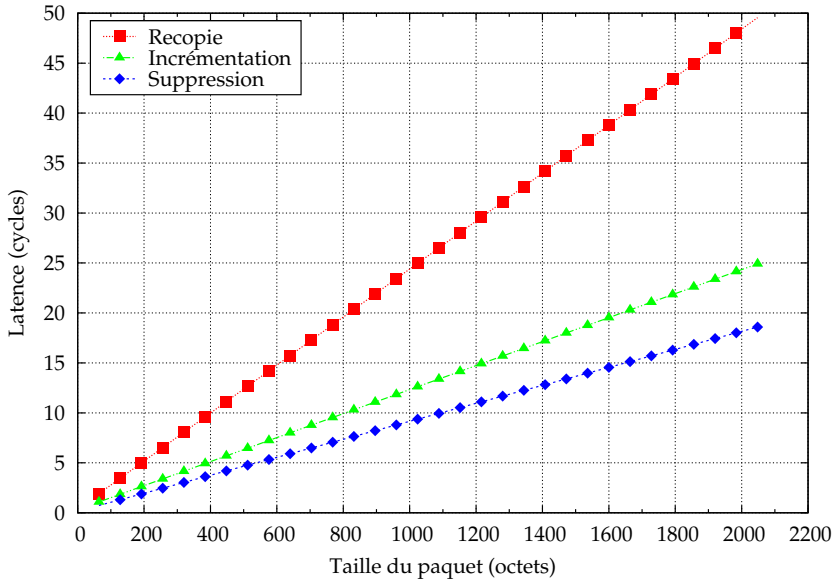


FIGURE 4.9 – **Nombre de cycles requis en fonction de la taille du paquet à envoyer.** Les trois courbes représentent trois stratégies différentes au niveau du récepteur.

Expérience	Débit (B/Cycle)	Débit (MB/s)	Latence (Cycles)	Latence (μs)
1	0.04	0.8	378	19
2	0.08	1.6	316	16
3	0.11	2.2	154	7.7

TABLE 4.4 – **Résultat de la mesure du débit et de la latence.**

Ces résultats nous permettent de prévoir le temps de reconfiguration d'un FPGA ECell depuis une cellule voisine. La configuration d'un ECell se fait par le FPGA correspondant de la carte ERouting à travers un module matériel dédié. Comme l'impact de ce matériel sur la latence est négligeable, on pourra utiliser les temps obtenus avec la troisième expérience.

La taille du fichier de configuration d'un FPGA Xilinx Spartan 3 `xc3s200` est de 128 kB. Le temps de reconfiguration peut être calculé selon l'équation 4.1.

$$\frac{128\ \text{kB}}{2.2\ \text{MB/s}} = 60\ \text{ms} \tag{4.1}$$

Il est à noter que cette durée est légèrement plus longue que les 20 ms nécessaires à la reconfiguration d'un Spartan 3 `xc3s200`. Néamoins, ce chiffre reste du même ordre de grandeur.

Le temps de reprogrammation logiciel peut être calculé de la même manière. La taille du code compilé peut varier, mais n'excèdera pas 64 kB, à cause de la largeur du bus de l'openMSP de 16 bits. Le temps de communication maximal pour la reprogrammation d'une tuile peut être calculé selon la l'équation 4.2.

$$\frac{64\ \text{kB}}{2.2\ \text{MB/s}} = 30\ \text{ms} \tag{4.2}$$

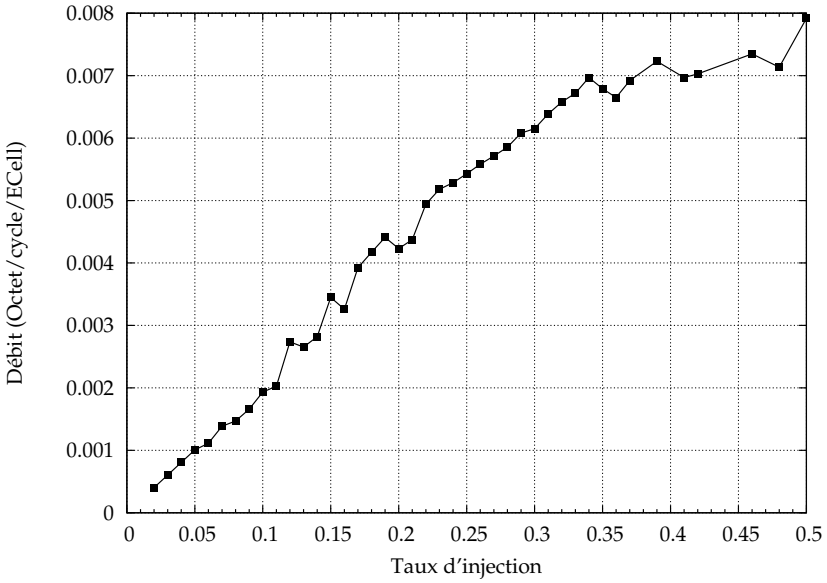


FIGURE 4.10 – **Évolution du débit du réseau en fonction du taux d'injection.**

Débit et latences moyens

L'objectif de la seconde expérience est de trouver la capacité du réseau. La capacité d'un réseau représente le débit réellement atteignable, dans notre cas si toutes les tuiles cherchent à communiquer.

Les auteurs de [Pande et al, 2005] proposent une méthode intéressante pour mesurer la capacité d'un réseau qui s'adapte bien à notre cas. L'expérience est divisée en *slots* de temps au cours des quels chaque ECell se voit attribuer une probabilité d'envoyer un paquet. Cette probabilité est appellée *taux d'injection*. On calcule le débit selon l'équation 4.3.

$$D = \frac{(\text{Nombre de paquets complétés}) \times (\text{Longueur d'un paquet})}{(\text{Nombre d'}\texttt{ECells}) \times (\text{Temps total})} \tag{4.3}$$

où *Nombre de paquets complétés* indique le nombre total de paquets reçus en intégralité, *Longueur d'un paquet* peut être mesuré en octets. *Nombre d'`ECells`* est le nombre d'éléments pouvant communiquer et *Temps total* est le temps – pouvant être mesuré en cycles d'horloge – entre l'émission du premier paquet et la réception du dernier. Cette capacité est ainsi mesurée en octet/cycle/`ECell`, et une capacité de $D = 1$ signifie que chaque nœud reçoit un octet à chaque cycle.

Pour mesurer la capacité du réseau, on fera varier le taux d'injection à chaque nœud, jusqu'à observer un phénomène de saturation.

Les paquets transmis contiennent un horodatage pour permettre de calculer la latence.

Le débit dans le réseau, montré en FIGURE 4.10, croît linéairement en fonction du taux d'injection jusqu'à un plafond de 140 kB/s/ECell pour un taux d'injection de 34 %. La capacité du réseau vaut ainsi 0.007 Byte/Cycle/ECell.

De manière analogue, la latence, dont l'évolution est représentée en FIGURE 4.11, augmente de manière exponentielle jusqu'à un seuil pour un taux d'injection de 40 %. Ce seuil est dû à

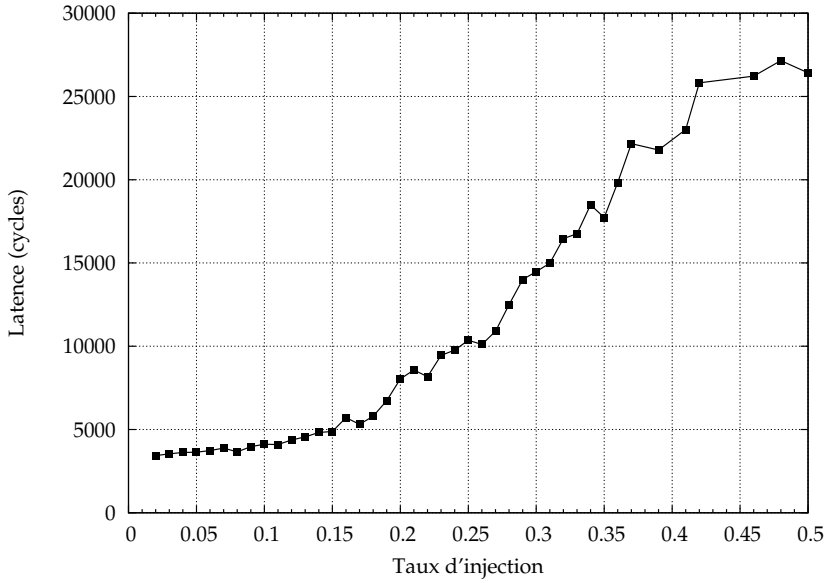


FIGURE 4.11 – **Évolution de la latence des paquets dans le réseau en fonction du taux d'injection.**

la suppression des messages s'ils ne peuvent être envoyés pendant un *slot* de temps. Sinon, la latence aurait continué à croître.

4.5.4 Discussions

En conclusion, la plateforme Confetti est une plateforme unique pour le prototypage d'architectures cellulaires, neuronales, et distribuées de manière générale. Une série d'expérimentations a montré la faisabilité de la mise en place de la reconfiguration dynamique parallèle et partielle à l'échelle de la plateforme.

Cependant, la version actuelle montre quelques limites sur différents aspects. La première limite est liée à la complexité de la plateforme, en particulier en ce qui concerne la programmabilité et le prototypage de la plateforme. Cette complexité rallonge le temps de développement et de validations d'architectures et impose une simulation en profondeur avant tout déploiement.

La deuxième limite est liée à l'âge de la plateforme, dans un monde où la technologie avance à grand pas. Les FPGA choisis, bien qu'à la pointe de la technologie à l'époque de la conception de la plateforme, ne sont pas assez puissants pour supporter l'ensemble du projet que nous souhaitons prototyper.

Une autre limite concerne le choix du réseau de routage, imposé une fois de plus par la puissance limitée des FPGA de routage. Le manque d'une solution de *broadcast* général ou de *multicast* localisé est un frein au déploiement du réseau de neurones matériel d'auto-adaptation, qui nécessite une propagation globale des données d'entrée. De plus, les congestions du réseau pourraient être repoussées avec l'utilisation par exemple de canaux virtuels.

Enfin, la consommation électrique des EStacks a également été source de problèmes. Tout d'abord, la température à l'intérieur de la plateforme a été à l'origine de plusieurs erreurs sur certains tests. Ensuite, les alimentations d'ordinateurs de bureau utilisées pour alimenter la

plateforme ne sont pas adaptées, et ont tendance à tomber en panne.

Pour ces raisons, une deuxième version de la plateforme est en cours de développement.

Cette version sera évidemment doté d'un FPGA plus puissant. La reconfiguration dynamique partielle proposée par les FPGA modernes permettrait de fusionner les architectures de routage et de calcul au sein de la même puce, tout en laissant la possiblité au routage de fonctionner pendant la reconfiguration de la couche de calcul. Cela permettrait de réduire la complexité globale de la plateforme. La programabilité en particulier serait améliorée, avec un port de programmation unifié. Enfin, un FPGA plus gros permettrait d'instancier un routeur plus complet.

Les FPGA modernes sont également dotés de laisons série haute vitesse, ce qui augmenterait le débit maximal atteignable. La granularité de la nouvelle plateforme pourrait être réduite, avec un unique FPGA par carte.

Le problème de l'alimentation de cette plateforme reste une question ouverte.

CHAPITRE 5

Conclusion

Sommaire

5.1 Synthèse

Dans cette thèse, nous avons d'abord observé l'évolution passée des architectures des processeurs qui a conduit à l'apparition des multi-cœurs, jusqu'à l'émergence, récente et à venir, des many-cœurs. Nous avons pu constater de nombreuses difficultés concernant le passage de quelques processeurs communiquant sur un bus et partageant une mémoire, à l'intégration de plusieurs milliers de cœurs de calcul. Dans le même temps, nous avons vu apparaître les architectures supportant la reconfiguration dynamique et partielle. Ces architectures pourtant prometteuses souffrent également de nombreuses limites.

Les concepteurs de ces architectures mettent en avant des applications certes gourmandes en puissance de calcul et qui mettent en échec les micro-processeurs classiques, mais ces applications représentent souvent des problèmes statiques requérant peu de contrôle, et dont la complexité vient de la quantité de données à traiter. Pour adresser des applications plus complexes et dynamiques, nous avons analysé plusieurs architectures auto-organisées, suivant une approche bio-inspirée. Nous n'avons pas trouvé d'architecture répondant à la fois à la contrainte de la scalabilité et de la dynamicité. Les projets offrant le plus de dynamicité se limitent à quelques éléments de calcul, et ceux proposant des architectures massivement parallèles sont relativement figées.

Pour répondre à ces problématiques, nous avons conçu un calculateur capable d'auto-organiser son architecture interne en fonction de la nature et de la richesse des informations contenues dans l'environnement dans lequel il est placé. Ce contrôleur s'inscrit dans la boucle sensori-motrice d'un robot mobile, illustrant ainsi un large choix d'applications complexes, évoluant dans un environnement dynamique. L'auto-organisation de l'architecture se manifeste sous la forme d'émergence d'aires de traitement sur la surface de la puce.

Le développement et l'évolution de ces aires sont pilotés par un réseau de neurones matériel intégré à la couche de calcul programmable. L'originalité de ce réseau de neurones de type carte auto-organisatrice est d'être complètement distribué, et de disposer d'une connectivité limitée. Nous pensons en effet que ces conditions soient nécessaires pour qu'une architecture puisse passer à l'échelle.

Cette couche neuronale tire ses données d'entrée dans une couche de pré-traitement qui a pour but d'extraire l'information pertinente de l'environnement. Dans le cadre de ces travaux,

elle est implémentée sous la forme d'un système de vision bio-inspiré, par ailleurs validé dans un contexte robotique.

5.2 Réflexions

J'aimerais profiter de cette section pour identifier quelques problèmes à venir et exprimer quelques idées plus personnelles.

5.2.1 L'auto-organisation

Dans la nature, on peut observer le développement et l'émergence de différentes structures et architectures ou de comportements, issus des effets et de la dynamique de leur environnement. Le développement du cerveau d'un nouveau-né dépend fortement des interactions avec son entourage, de la manipulation d'objets, et de manière générale de la perception par ces différents sens. La plasticité cérébrale permet de modifier, au cours de la vie d'un individu l'organisation du cerveau, suite à la perte d'un organe par exemple. À un niveau plus élevé, l'évolution des espèces depuis l'apparition de la vie sur Terre est également liée aux effets de l'environnement.

Ces structures et comportements agissent eux-mêmes sur leur environnement, créant ainsi des boucles, plus ou moins complexes. Ces boucles d'interactions peuvent converger, diverger ou osciller. Par exemple, les équations de Lotka-Volterra, décrivent les oscillations formées au cours du temps par les populations de proies et de prédateurs dans un environnement donné.

L'auto-organisation, l'adaptation, et les interactions avec l'environnement sont difficiles à modéliser et à plus forte raison à reproduire. L'information contenue dans l'environnement est riche et dense, dynamique et non prédictible, et bruitée. Il existe pourtant de nombreux systèmes artificiels capable d'adapter leur comportement à leur environnement. Ainsi un simple logiciel de traitement de texte par exemple modifie son comportement en fonction de ces interactions avec un utilisateur humain à travers une interface homme-machine. Ces adaptations s'effectuent le plus souvent à l'aide de machines à états finis pré-programmées, et ne peuvent pas prévoir tous les cas. Elles peuvent même être bloquées si un comportement non prévu survient.

Les domaines de l'intelligence artificielle et des neuronsciences computationnelles ont pour vocation d'adapter les comportements de systèmes artificiels de manière plus naturelle, à l'aide d'algorithmes inspirés des observations faites dans la nature. Les réseaux de neurones artificiels permettent de faire émerger un comportement, qui sera spécifique à un environnement, et à une structure neuronale donnée. La robotique en particulier offre un large éventail de défis à relever pour ces disciplines. L'environnement peut être similaire à celui des êtres vivants à partir desquels les modèles neuronaux ont été inspirés.

Les architectures reconfigurables, comme les FPGA sont un substrat intéressant pour expérimenter le développement d'architectures matérielles en fonction des effets de l'environnement. Au delà de l'adaptation d'un comportement, ici c'est la structure même de l'organe responsable de la prise de décision qui s'adapte. Dans un contexte de robotique mobile, cette structure s'adapte à la morphologie du robot, aux tâches demandées et aux informations perçues à travers différents capteurs. Ces vecteurs d'adaptation forment l'environnement du contrôleur.

Nous avons, tout au long de cette thèse, proposé les briques de base pour permettre à l'architecture d'un tel contrôleur de s'adapter aux effets, statiques et dynamiques, de l'environnement.

5.2.2 Les modèles de calcul

La démocratisation à venir des architectures massivement parallèles sur puce soulève des questions quant aux modèles de calcul à utiliser. La représentation classique d'un problème informatique se traduit par un algorithme qui souvent résulte en une séquence d'instructions, pouvant contenir des conditions et des dépendances entre instructions, ce qui complique leur parallélisation.

Avant d'aborder les perspectives architecturales, nous pouvons nous poser la question suivante :
"Quels modèles de calcul pouvons-nous utiliser pour la couche de calcul programmable ?"

Considérons pour cela une aire de traitement, consistant en un circuit massivement parallèle, reconfigurable de manière dynamique et dont les dimensions sont variables. Nous pouvons supposer, grâce à la couche d'auto-organisation, qu'une diminution de la taille de l'aire soit provoquée par une diminution de la quantité de données à traiter.

Nous avons vu qu'une représentation en flot de données se prétait bien à une implémentation sur une architecture parallèle. Lorsque l'aire de traitement diminue, plusieurs agents peuvent être multiplexés temporellement sur un nombre réduit d'éléments de calcul, qu'ils soient logiciels ou matériels.

Un autre modèle de calcul à considérer est celui des mailles associatives [Mérigot, 1997]. Le modèle des mailles associatives considère les communications dans un ensemble arbitrairement grand de processeurs. Ces ensembles appartiennent à une composante connexe d'un graphe de communication (appelé mgraph). Un composant connecté correspond au sous-ensemble d'un graphe dont les nœuds sont reliés entre eux, directement ou indirectement, par des arcs. Les graphes considérés sont toujours des sous-graphes du graphe de communication physique, mais ils sont définis dynamiquement, en stockant, dans chaque processeur, une description de ses arêtes entrantes pouvant être calculée ou téléchargée avec un motif prédéterminé localement. Les primitives de communication concernent toujours les processeurs appartenant au même ensemble du graphe connexe de communication. Les données de ces processeurs sont combinées avec un opérateur arithmétique ou logique. Il est possible par exemple de recueillir des données à partir de ces processeurs interconnectés, et de calculer leur somme globale, le maximum, etc... Les processeurs considérés par le modèle des mailles associatives étant à faible granularité, il est possible de les virtualiser à l'intérieur d'un élément de calcul. Cette voie pourrait être étudiée dans la suite de ces travaux pour doter l'architecture d'un modèle de calcul complet apte à rendre son usage plus général.

5.3 Perspectives

Dans cette thèse, les quatre couches composant l'architecture SATURN ont été étudiées. Les couches d'adaptation et de calcul programmable s'interfacent naturellement. Nous avons proposé un système de vision en guise d'exemple pour la couche de pré-traitement. À l'avenir, il faudra enrichir cette couche avec d'autres modalités, comme l'information auditive, d'autres informations visuelles – comme le mouvement ou les textures – ou encore la proprioception.

Il y a également un travail d'équilibrage entre ces différentes modalités, ainsi que d'interfaçage avec les couches les plus hautes.

Le modèle neuronal choisi fonctionne bien pour quelques centaines de neurones, mais il n'est plus capable de converger quand ce chiffre dépasse les milliers. Un nouveau modèle a été proposé plus récemment par Rodriguez, qui mime une connectivité complète de manière

itérative. Ce modèle peut être exécuté par les NPU si l'on modifie le module de communication. Il faudra alors voir en quoi les performances temporelles seraient affectées par cet ajout d'itérations dans de grands réseaux.

Il faudra ensuite déployer des applications sur cette architecture. On pourrait, dans un premier temps, implémenter les réseaux de contrôle présentés au chapitre 2. L'étape suivante serait d'étendre ces modèles à d'autres modalités pour permettre au robot des interactions plus complexes. La concurrence d'aires de traitement nous permet également d'envisager des scénarii multi-applications voire multi-utilisateurs.

Publications personnelles

Fiack L, Lefebvre T, Miramond B (2012a) Integration of a bio-inspired robotic vision system on fpga. In : GDR SoC-SiP, vol 2012

Fiack L, Miramond B, Cuperlier N (2012b) Fpga-based perception architecture for robotic missions. In : SCaBot (Smart Camera for roBotic applications)

Fiack L, Cuperlier N, Miramond B (2013) Hardware vision architecture for autonomous navigation. In : GDR SoC-SiP

Fiack L, Cuperlier N, Miramond B (2014a) Embedded and real-time architecture for bio-inspired vision-based robot navigation. Journal of Real-Time Image Processing

Fiack L, Miramond B, Upegui A, Vannel F (2014b) Dynamic parallel reconfiguration for self-adaptive hardware architectures. In : Adaptive Hardware and Systems (AHS), IEEE, pp 218–224

Fiack L, Rodriguez L, Miramond B (2015) Hardware design of a neural processing unit for bio-inspired computing. In : New Circuits and Systems Conference (NEWCAS)

Rodriguez L, Fiack L, Miramond B (2013a) Hardware architecture of self-organizing maps. In : NeuComp

Rodriguez L, Fiack L, Miramond B (2013b) A neural model for hardware plasticity in artificial vision systems. In : Conference on Design and Architectures for Signal and Image Processing (Dasip 2013)

Rodriguez L, Fiack L, Miramond B, Hochapfel E (2013c) Validation of neural networks onto FPGA. In : NeuComp

Bibliographie

Almaless G (2014) Operating System Design and Implementation for Single-Chip cc-NUMA Many-Core. PhD thesis, Paris 6

Altera (2013) User-Customizable ARM-Based SoCs for Next-Generation Embedded Systems. URL https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01167-custom-arm-soc.pdf

Amari Si (1977) Dynamics of pattern formation in lateral-inhibition type neural fields. Biological cybernetics 27(2) :77–87

Appiah K, Hunter A, Meng H, Yue S, Hobden M, Priestley N, Hobden P, Pettit C (2009) A binary self-organizing map and its FPGA implementation. In : Neural Networks, 2009. IJCNN 2009. International Joint Conference on, IEEE, pp 164–171

Battezzati N, Colazzo S, Maffione M, Senepa L (2012) SURF algorithm in FPGA : A novel architecture for high demanding industrial applications. In : Rosenstiel W, Thiele L (eds) DATE, IEEE, pp 161–162

Berton F, Sandini G, Metta G (2006) Anthropomorphic visual sensors. Encyclopedia of Sensors X :1–16

Bilsen G, Engels M, Lauwereins R, Peperstraete J (1996) Cycle-static dataflow. IEEE Transactions on Signal Processing 44(2) :397–408

Birem M, Berry F (2012) FPGA-based Real time Extraction of visual features. In : Proceedings of Circuits and Systems (ISCAS), DOI 10.1109/ISCAS.2012.6271964

Bonato V, Holanda J, Marques E (2006) An embedded multi-camera system for simultaneous localization and mapping. In : Proceedings of Applied Reconfigurable Computing, Lecture Notes on Computer Science

Bonato V, Marques E, Constantinides GA (2008) A parallel hardware architecture for scale and rotation invariant feature detection. IEEE Trans Cir and Sys for Video Technol 18(12) :1703–1712, DOI 10.1109/TCSVT.2008.2004936

Borkar S (2007) Thousand core chips : a technology perspective. In : Proceedings of the 44th annual Design Automation Conference, ACM, pp 746–749

Bourgeault M (2011) Alteras partial reconfiguration flow

Bouris D, Nikitakis A, Walters J (2010) Fast and Efficient FPGA-Based Feature Detection Employing the SURF Algorithm. In : Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on, pp 3–10, DOI 10.1109/FCCM.2010.11

Cardeira C, Mammeri Z (1995) Preemptive and non-preemptive real-time scheduling based on neural networks. Proceedings DCCS 95 :67–72

Carrillo S, Harkin J, McDaid L, Morgan F, Pande S, Cawley S, McGinley B (2013) Scalable hierarchical network-on-chip architecture for spiking neural network hardware implementations. IEEE Transactions on Parallel and Distributed Systems 24(12) :2451–2461, DOI 10.1109/TPDS.2012.289

Chillet D, Eiche A, Pillement S, Sentieys O (2011) Real-time scheduling on heterogeneous system-on-chip architectures using an optimised artificial neural network. Journal of Systems Architecture 57(4) :340–353

Clark A (1999) An embodied cognitive science ? Trends in Cognitive Sciences

Colonnier F, Manecy A, Juston R, Mallot H, Leitel R, Floreano D, Viollet S (2015) A small-scale hyperacute compound eye featuring active eye tremor : application to visual stabilization, target tracking, and short-range odometry. Bioinspiration & biomimetics 10(2) :026,002

Conway J (1970) The game of life. Scientific American 223(4) :4

Crowley JL, Riff O (2003) Fast computation of scale normalised gaussian receptive fields. Springer Lecture Notes in Computer Science 2695

Cuperlier N, Quoy M, Gaussier P (2007) Neurobiologically inspired mobile robot navigation and planning. Frontiers in NeuroRobotics 1(1)

Diaz J, Munoz-Caro C, Nino A (2012) A survey of parallel programming models and tools in the multi and many-core era. IEEE Transactions on Parallel and Distributed Systems 23(8) :1369–1386

de Dinechin BD, Ayrignac R, Beaucamps PE, Couvert P, Ganne B, de Massas PG, Jacquet F, Jones S, Chaisemartin NM, Riss F, et al (2013) A clustered manycore processor architecture for embedded and accelerated applications. In : High Performance Extreme Computing Conference (HPEC), 2013 IEEE, IEEE, pp 1–6

Dye D (2010) Partial reconfiguration of xilinx fpgas using ise design suite

Economakos C, Sidiropoulos H, Economakos G (2013) Rapid prototyping of digital controllers using FPGAs and ESL/HLS design methodologies. In : Automation and Computing (ICAC), 2013 19th International Conference on, IEEE, pp 1–6

Emery R, Yakovlev A, Chester G (2009) Connection-centric network for spiking neural networks. In : Proceedings of the 2009 3rd ACM/IEEE International Symposium on Networks-on-Chip, IEEE Computer Society, pp 144–152

Ewo R, Pinna A, Granado B, Mbouenda M, Fotsin HB (2014) A hardware mpi spawn for distributed multiprocessing reconfigurable system on chip (mp-rsoc). In : Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on, IEEE, pp 238–238

Flynn MJ (1972) Some computer organizations and their effectiveness. IEEE Transactions on Computers 100(9) :948–960

Furber S, Temple S (2007) Neural systems engineering. Journal of the Royal Society interface 4(13) :193–206

Gallego A, Mora J, Otero A, Salvador R, de la Torre E, Riesgo T (2013) A Novel FPGA-based Evolvable Hardware System Based on Multiple Processing Arrays. In : Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International, IEEE, pp 182–191

GAMOM NGOUNOU EWO RC (2015) Déploiement d'applications parallèles sur une architecture distribuée matériellement reconfigurable. PhD thesis, UCP

Gantel L (2014) Hardware and software architecture facilitating the operation by the industry of dynamically adaptable heterogeneous embedded systems. PhD thesis, Cergy Pontoise

Gaussier P, Joulain C, Banquet JP, Leprêtre S, Revel A (2000) The visual homing problem : an examle of robotic/biology cross fertilization. Robotics and Autonomous Systems 30 :115–180

Gaussier P, Revel A, Banquet JP, Babeau V (2002) From view cells and place cells to cognitive map learning : processing stages of the hippocampal system. Biological Cybernetics 86 :15–28

Giovannangeli C, Gaussier P, Banquet J (2006a) Robustness of visual place cells in dynamic indoor and outdoor environment. International Journal of Advanced Robotic Systems 3(2) :115–124

Giovannangeli C, Gaussier P, Banquet JP (2006b) Robustness of visual place cells in dynamic indoor and outdoor environment. International Journal of Advanced Robotic Systems 3(2) :115–124

Goodale MA, Milner AD (1992) Separate visual pathways for perception and action. Trends in Neurosciences 15(1) :20–25

Greiner A (2009) Tsar : a scalable, shared memory, many-cores architecture with global cache coherence. In : 9th International Forum on Embedded MPSoC and Multicore (MPSoC'09), vol 15

Guerrier P, Greiner A (2000) A generic architecture for on-chip packet-switched interconnections. In : Proceedings of the conference on Design, automation and test in Europe, ACM, pp 250–256

Harris C, Stephens M (1988) A combined corner and edge detector. In : Alvey vision conference, Citeseer, vol 15, p 50

Hirel J, Gaussier P, Quoy M (2011) Biologically inspired neural networks for spatio-temporal planning in robotic navigation tasks. In : Robotics and Biomimetics (ROBIO), 2011 IEEE International Conference on, pp 1627–1632, DOI 10.1109/ROBIO.2011.6181522

Hopfield JJ (1982) Neural networks and physical systems with emergent collective computational abilities. Proceedings of the national academy of sciences 79(8) :2554–2558

ITRS (2013) The international technology roadmap for semiconductors

Kohonen T (1990) The self-organizing map. Proceedings of the IEEE 78 :17

Kohonen T (2012) Self-organization and associative memory, vol 8. Springer

Kolb B, Tees R (1990) The Cerebral Cortex of the Rat. MIT Press

Lagarde M, Andry P, Gaussier P (2008) Distributed real time neural networks in interactive complex systems. In : Proceedings of the 5th international conference on Soft computing as transdisciplinary science and technology, ACM, New York, NY, USA, CSTST '08, pp 95–100, DOI 10.1145/1456223.1456247

Lindeberg T (1998) Feature detection with automatic scale selection. International Journal of Computer Vision vol. 30, no. 2 :79,116

Lowe DG (2004) Distinctive image features from scale-invariant key-points. International Journal of Computer Vision vol. 60, no. 2 :91,110

Maillard M (2007) Formalisation de la perception comme dynamique sensori-motrice : application dans un cadre de reconnaissance d'objets par un robot autonome. PhD thesis, Cergy

Maillard M, Gapenne O, Hafemeister L, Gaussier P (2005) Perception as a dynamical sensorimotor attraction basin. In : Proceedings of the 8th European Conference on Advances in Artificial Life (ECAL '05), vol 3630, p 37–46

Marchesan Almeida G, Sassatelli G, Benoit P, Saint-Jean N, Varyani S, Torres L, Robert M (2009) An adaptive message passing mpsoc framework. International Journal of Reconfigurable Computing 2009

Markram H (2006) The blue brain project. Nature Reviews Neuroscience 7(2) :153–160

Mérigot A (1997) Associative nets : A graph-based parallel computing model. IEEE Transactions on Computers 46(5) :558–571

Merolla P, Arthur J, Akopyan F, Imam N, Manohar R, Modha DS (2011) A digital neurosynaptic core using embedded crossbar memory with 45pJ per spike in 45nm. In : Custom Integrated Circuits Conference (CICC), 2011 IEEE, IEEE, pp 1–4

Mikolajczyk K, Schmid C (2004) Scale & affine invariant interest point detectors. International Journal of Computer Vision 60(1)

Miramond B (2014) Contributions à la conception de systèmes sur puce reconfigurables. Des systèmes embarqués multiprocesseurs aux architectures bio-inspirées. PhD thesis

Modha DS, Ananthanarayanan R, Esser SK, Ndirango A, Sherbondy AJ, Singh R (2011) Cognitive computing. Communications of the ACM 54(8) :62–71

Mora J, Gallego A, Otero A, Lopez B, de la Torre E, Riesgo T (2013) A noise-agnostic self-adaptive image processing application based on evolvable hardware. In : Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on, IEEE, pp 351–352

Moraes F, Calazans N, Mello A, Möller L, Ost L (2004) Hermes : an infrastructure for low area overhead packet-switching networks on chip. INTEGRATION, the VLSI journal 38(1) :69–93

Mudry PA (2009) A hardware-software codesign framework for cellular computing. PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Mudry PA, Vannel F, Tempesti G, Mange D (2007) Confetti : A reconfigurable hardware platform for prototyping cellular architectures. In : IPDPS, IEEE, pp 1–8

Musa P, Sudiro SA, Wibowo EP, Harmanto S, Paindavoine M (2012) Design and implementation of non-linear image processing functions for cmos image sensor. In : Photonics Asia, International Society for Optics and Photonics, pp 85,580O–85,580O

Ni LM, McKinley PK (1993) A survey of wormhole routing techniques in direct networks. Computer 26(2) :62–76, DOI 10.1109/2.191995, URL http://dx.doi.org/10.1109/2.191995

O'Keefe J, Nadel N (1978) The Hippocampus As a Cognitive Map. Clarenton Press, Oxford

Olofsson A, Trogan R, Raikhman O, Adapteva L (2011) A 1024-core 70 gflop/w floating point manycore microprocessor. In : Poster on 15th Workshop on High Performance Embedded Computing HPEC2011

OpenCores (1999) OpenCores. URL http://opencores.com/

P Gaussier SZ (1995) Perac : a neural architecture to control artificial animals. Robotics and Autonomous Systems 16(2–4) :291–320

Paindavoine M, Dubois J, Musa P (2015) Neuro inspired smart image sensor : analog hmax implementation. In : IS&T/SPIE Electronic Imaging, International Society for Optics and Photonics, pp 94,030J–94,030J

Panades IM, Greiner A, Sheibanyrad A, STMicroelectronics G (2006) A low cost network-on-chip with guaranteed service well suited to the gals approach. Proc NANONET

Pande PP, Grecu C, Jones M, Ivanov A, Saleh R (2005) Performance evaluation and design trade-offs for network-on-chip interconnect architectures. IEEE Transactions on Computers 54(8) :1025–1040

Patterson D, Anderson T, Cardwell N, Fromm R, Keeton K, Kozyrakis C, Thomas R, Yelick K (1997) A case for intelligent RAM. Micro, IEEE 17(2) :34–44

Rodriguez L, Miramond B, Granado B (2014) Toward a sparse self-organizing map for neuromorphic architectures. ACM Journal on Emerging Technologies in Computing systems

Rodriguez LY (2015) Définition d'un substrat computationnel bio-inspiré : déclinaison de propriétés de plasticité cérébrale dans les architectures de traitement auto-adaptatif. Thèses, Université de Cergy-Pontoise

Salvador R, Otero A, Mora J, de la Torre E, Sekanina L, Riesgo T (2011) Fault tolerance analysis and self-healing strategy of autonomous, evolvable hardware systems. In : Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on, IEEE, pp 164–169

Sanchez E, Pérez-Uribe A, Upegui A, Thoma Y, Moreno JM, Villa AEP, Volken H, Napieralski A, Sassatelli G, Lavarec E (2007) Perplexus : Pervasive computing framework for modeling complex virtually-unbounded systems. In : Arslan T, Stoica A, Suess M, Keymeulen D, Higuchi T, Zebulum RS, Erdogan AT (eds) AHS, IEEE Computer Society, pp 587–591, URL http://dblp.uni-trier.de/db/conf/ahs/ahs2007.html#SanchezPUTMVVNSL07

Santarini M (2012) Xilinx unveils vivado design suite for the next decade of'all programmable'devices. Xcell Journal-Solutions for a Programmable World (79)

SATURN (2010) Projet SATURN. URL http://projet-saturn.ensea.fr/

Schaeferling M (2010) Flex-SURF : A Flexible Architecture for FPGA Based Robust Feature Extraction for Optical Tracking Systems. In : Conference on Reconfigurable Computing and FPGAs

Schemmel J, Bruderle D, Grubl A, Hock M, Meier K, Millner S (2010) A wafer-scale neuromorphic hardware system for large-scale neural modeling. In : Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on, IEEE, pp 1947–1950

So HKH, Brodersen R (2008) A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH. ACM Transactions on Embedded Computing Systems (TECS) 7(2) :14

Sousa ER, Tanase A, Hannig F, Teich J (2013) Accuracy and performance analysis of Harris corner computation on tightly-coupled processor arrays. In : Design and Architectures for Signal and Image Processing (DASIP), 2013 Conference on, IEEE, pp 88–95

Stone JE, Gohara D, Shi G (2010) Opencl : A parallel programming standard for heterogeneous computing systems. Computing in science & engineering 12(1-3) :66–73

Teich J (2008) Invasive algorithms and architectures. it-Information Technology Methoden und innovative Anwendungen der Informatik und Informationstechnik 50(5) :300–310

Teich J, Henkel J, Herkersdorf A, Schmitt-Landsiedel D, Schröder-Preikschat W, Snelting G (2011) Invasive computing : An overview. In : Multiprocessor System-on-Chip, Springer, pp 241–268

Treisman AM, Gelade G (1980) A feature-integration theory of attention. Cognitive psychology 12(1) :97–136

Tyrrell AM, Sanchez E, Floreano D, Tempesti G, Mange D, Moreno JM, Rosenberg J, Villa AE (2003) Poetic tissue : An integrated architecture for bio-inspired hardware. In : Evolvable Systems : From Biology to Hardware, Springer, pp 129–140

Vannel F (2007) Bio-inspired cellular machines : Towards a new electronic paper architecture. PhD thesis, EPFL

Wilson RA, Foglia L (2011) Embodied cognition

Xilinx (2011) UG761 Design Reference Guide. URL `http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf`

Xilinx (2015) All programmable 7 series product selection guide. URL `http://www.xilinx.com/support/documentation/selector-guides/7-series-product-selection-guide.pdf`

Zamarreño-Ramos C, Linares-Barranco A, Serrano-Gotarredona T, Linares-Barranco B (2013) Multicasting mesh aer : a scalable assembly approach for reconfigurable neuromorphic structured aer systems. application to convnets. IEEE Transactions on Biomedical Circuits and Systems 7(1) :82–102

Zhong S, Wang J, Yan L, Kang L, Cao Z (2013) A real-time embedded architecture for SIFT. J Syst Archit 59(1) :16–29, DOI 10.1016/j.sysarc.2012.09.002

Les effets de l'environnement sur le développement et l'organisation d'architectures de traitement matériel auto-organisées

Résumé : Les avancées technologiques récentes ont permis d'intégrer plusieurs milliards de transistors au sein d'une même puce, et ce chiffre ne cesse d'augmenter. Il n'est plus possible depuis quelques années, à cause de limitations physiques, d'augmenter la fréquence de fonctionnement des micro-processeurs. Pour adresser des applications toujours plus complexes, la tendance actuelle consiste à multiplier le nombre de cœurs de calcul. Au-delà d'une dizaine de processeurs, de nombreuses problématiques apparaissent, comme la gestion de la mémoire, les communications, la manière de représenter le calcul ou encore l'ordonnancement de tâches.

Pour répondre à ces problématiques, nous avons conçu un calculateur capable d'auto-organiser son architecture interne en fonction de la nature et de la richesse des informations contenues dans l'environnement dans lequel il est placé. Ce contrôleur s'inscrit dans la boucle sensori-motrice d'un robot mobile, illustrant ainsi un large choix d'applications complexes, évoluant dans un environnement dynamique. Il est constitué d'une grille 2D d'éléments de calcul prenant la forme d'une surface reconfigurable, pouvant héberger un processeur ou un accélérateur matériel. L'auto-organisation de l'architecture se manifeste sous la forme d'émergence d'aires de traitement sur la surface de la puce, parmi les éléments de calcul.

Le développement et l'évolution de ces aires sont pilotés par un réseau de neurones matériel intégré à la couche de calcul. L'originalité de ce réseau de neurones de type carte auto-organisatrice est d'être complètement distribué, et de disposer d'une connectivité limitée. Nous pensons en effet que ces conditions soient nécessaires pour qu'une architecture puisse passer à l'échelle.

Cette couche neuronale tire ses données d'entrée dans une couche de pré-traitement qui a pour but d'extraire l'information pertinente de l'environnement. Dans le cadre de ces travaux, elle est implémentée sous la forme d'un système de vision bio-inspiré, par ailleurs validé dans un contexte robotique.

Mots clés : Architectures auto-organisées, Many-cœurs, Réseau de neurones matériels, Systèmes de vision

The effects of the environment on the development and the organization of self-organizing hardware processing architectures

Abstract : Recent technological progress have allowed the integration of billions of transistors inside a single chip, and this figure keeps growing. Since a few years, due to physical limitations, it has not been possible to increase the micro-processors frequency anymore. To tackle more and more complex applications, the current trend is to increase the number of computation cores. However, beyond tens of cores, some issues emerge. Among them, we can cite memory management, communications, programming or task scheduling.

To address these challenges, we have designed a computer which is able to self-organize its own internal architecture depending on the nature and the richness of the informations sensed in the environment in which it is placed. This computer is part of a sensori-motor loop of a mobile robot, thus allowing a large panel of complex applications, running in a dynamic environment. It is made of a 2D mesh network of processing elements which are reconfigurable surfaces, that can host a processor or a hardware accelerator. The self-organization of the architecture takes the form of the emergence of computation areas on the chip surface, among the processing elements.

The developpment and the evolution of these areas are driven by a hardware neural network, integrated in the programmable computation layer. The novelty of this self-organizing map neural network is that it is completely distributed, and that the neurons are sparsely connected. We think indeed that these conditions are necessary for the scalability of an architecture.

This neuronal layer takes its inputs from a pre-processing sublayer which extracts the relevant information in the environment. In the context of this work, this layer is implemented as a bio-inspired vision system, also validated in a robotics context.

Keywords : Auto-organized architectures, Many-cores, Hardware neural networks, Vision systems