Architectures flot de données dédiées au traitement d'images par morphologie mathématique

Christophe Clienti

ED n°431 : Information, Communication, Modélisation et Simulation

T H È S E

pour obtenir le grade de

DOCTEUR DE L'ÉCOLE NATIONALE SUPÉRIEURE DES MINES DE PARIS

Spécialité "Morphologie Mathématique"

présentée et soutenue publiquement par
Christophe Clienti

le 30 septembre 2009

Architectures flot de données dédiées au traitement d'images par Morphologie Mathématique

Directeurs de thèse : Serge Beucher et Michel Bilodeau

Jury

M.	Dominique Jeulin	Président
M.	Alain Mérigot	Rapporteur
M.	Michel Paindavoine	Rapporteur
M.	Mohamed Akil	Examinateur
M.	Serge Beucher	Examinateur
M.	Michel Bilodeau	Examinateur
M.	Didier Demigny	Examinateur
M.	Fabrice Lemonnier	Examinateur

à Mireille

Remerciements

Les remerciements, cette chose que l'on repousse toujours à la fin de la rédaction mais pourtant si difficile à écrire, tant la peur est grande d'oublier un proche, un ami, un collaborateur.

Je commencerai donc par remercier l'École des Mines de Paris, le CMM et Fernand Meyer de m'avoir accueilli au sein du Centre de Morphologie Mathématique.

Merci également à Dominique Jeulin d'avoir été le président de mon jury, à Alain Mérigot et Michel Paindavoine d'avoir été les rapporteurs de mon travail, à Fabrice Lemonnier, Didier Demigny et Mohamed Akil d'avoir accepté d'en être les examinateurs.

Je remercie chaleureusement Michel Bilodeau et Serge Beucher d'avoir été plus que de simples directeurs de thèse et de m'avoir laisser une très grande liberté dans ma recherche. Ils m'ont permis de trouver un véritable équilibre dans la *Force* entre les mathématiques, l'algorithmique et l'architecture. J'ai pu ainsi *croiser ces différents flux* de connaissance sans danger... Serge, je me permets de reprendre une de tes phrases : "... ce fut une collaboration plus qu'une simple relation directeur de thèse vers thésard ..."

Un grand merci à tous les membres du CMM et en particulier à Catherine et Laura pour leur disponibilité, leur gentillesse et leur bonne humeur.

Je tiens également à adresser mes remerciements à Mireille pour sa patience et sa compréhension durant ces longues années de labeurs. Je remercie ma famille pour m'avoir donné le coup de pousse dont j'avais besoin il y a quelques années et sans lequel, aujourd'hui, je n'en serais pas là.

Abstract

Data flow architectures dedicated to image processing using mathematical morphology

Christophe Clienti

This thesis report is focused on studying data flow accelerators dedicated to image processing based on mathematical morphology. The main objective is to provide a programmable and efficient implementation of basic morphological operators, and to assemble them in such a way as to provide complex operators with fast operation. In recent years, morphological algorithm research has been oriented towards finding elegant algorithms to compute these complex operators, such as geodesic reconstruction and watershed using priority queues. These complex algorithms often use specific data structures that are hard to deploy on platforms other than single-core, general-purpose processors. Moreover, these processors continue their development, not in the field of frequency, but in the field of parallelism by heightening the number of cores. And because the frequency wall seems to have been reached, the best way to optimise performance is to use parallelising techniques.

Consequently, we decided on fast implementations of complex mathematical morphology operators, based on highly parallel simpler operations. In the first part, we study existing computational kernels for neighbourhood processors and suggest new ones based on recent advances in mathematical morphology. We also study existing methods of neighbourhood extraction in a data flow context and describe a new approach to parallelise the extraction of contiguous neighbourhoods.

In the second part, we use the neighbourhood processors as building blocks, assembling them in a pipeline. This pipeline is instantiated in a system-on-a-chip as an accelerator controlled by a simple general-purpose processor operating at low frequency for embedded image processing applications. The pipeline composition and description is hand crafted, and so we offer high-level tools to define and generate it easily. Tools also control the merging of multiple high-level descriptions to build one dynamically reconfigurable at a coarse grain level by looking for aggregatable parts.

In the third part, we present a description of a basic general purpose processor using vector instructions deployed in a dataflow context. We combined a VLIW structure with a cluster in charge of wide vector unit to conciliate instruction level parallelism and data level parallelism. Numerous VLIW vector processors can be assembled in a pipeline to exploit temporal parallelism, in the same way of neighbourhood processors, while providing a high flexibility in terms of programming opportunities. Therefore, such a system can be used to describe applications at all levels of granularity : fine, medium and coarse.

Finally, we analyse the performance of our system against a multi-core workstation processor, and against a graphics processor unit composed of hundreds of processing elements and executing thousands of micro-threads. In this section we show the relevance of our approach and in particular the performance of temporal and spatial parallelism exploitations. This joint exploitation allows to reduce drastically the number of cycles per pixel produced by image processing accelerators and allow us to target efficiently implementation of mathematical morphology complex operators using basic ones such as erosions or dilations.

Key words : mathematical morphology, system on chip, FPGA, data flow.

Résumé

Architectures flots de données dédiées au traitement d'images par morphologie mathématique

Christophe Clienti

Nous abordons ici la thématique des opérateurs et processeurs flot de données dédiés au traitement d'images et orientés vers la morphologie mathématique. L'objectif principal est de proposer des architectures performantes capables de réaliser les opérations simples de ce corpus mathématique afin de proposer des opérateurs morphologiques avancés. Ces dernières années, des algorithmes astucieux ont été proposés avec comme objectif de réduire la quantité des calculs nécessaires à la réalisation de transformations telles que les reconstructions géodésiques ou bien encore la ligne de partage des eaux. Toutefois, les mises en œuvre proposées font souvent appel à des structures de données complexes, telles que des files d'attente hiérarchiques, et qui sont difficiles à employer sur des machines différentes des processeurs généralistes monocœurs. Les processeurs standard poursuivant leur évolution, non plus vers un accroissement de la fréquence, mais vers une augmentation du parallélisme, ces implémentations ne nous permettent pas d'obtenir les gains de performance escomptés à chaque nouvelle génération de machine.

Nous proposons alors des mises en œuvre rapides des opérations complexes de la morphologie mathématique par des machines exploitant fortement le parallélisme intrinsèque des opérations basiques. Nous étudions dans une première partie les processeurs de voisinage travaillant directement sur un flot de pixels et nous proposons de nouvelles structures de calculs, ainsi que de nouveaux principes d'extraction du voisinage capables de rechercher plusieurs voisinages contigus à un vecteur de pixels.

Dans la seconde partie, nous proposons d'exploiter un pipeline de processeurs de voisinage dans un système sur puce avec comme but de disposer d'une architecture embarquant un accélérateur programmable piloté par un processeur généraliste. Nous étudions ensuite des techniques de haut niveau simplifiant la description du pipeline utilisé dans le circuit. Ces techniques présentées adressent également la génération à gros grains de pipelines reconfigurables en cherchant parmi plusieurs descriptions les composants pouvant être mutualisés.

La troisième partie est consacrée à la description d'une architecture reposant sur un flot de processeurs généralistes vectoriels. La structure VLIW ainsi que les instructions travaillant avec de larges vecteurs permettent d'obtenir, sur un unique cœur, des performances élevées. Le raccordement flot de données de ces processeurs nous donne ensuite les mêmes possibilités qu'un flot de processeurs de voisinage en terme de parallélisme temporel, mais nous laisse bien plus de souplesse dans la description du parallélisme spatial, ainsi que dans la description des opérations réalisables à chaque étage. Un système tel que décrit dans cette partie peut donc exploiter un parallélisme temporel et spatial avec une granularité oscillant entre fine, moyenne et forte.

Nous terminons par une comparaison des architectures présentées dans ce mémoire avec une machine généraliste multicœurs et également avec un circuit dédié à l'accélération graphique disposant aujourd'hui d'un grand nombre de processeurs élémentaires et exécutant une grande quantité de tâche de calcul en parallèle. Nous montrons dans cette partie la pertinence de notre approche et surtout l'intérêt conjoint de l'exploitation du parallélisme temporel et spatial afin de réduire au maximum le nombre de cycles par pixel produit en sortie des opérateurs de traitement d'images. Les performances atteintes dans le calcul et le chaînage de ces opérateurs prouvent donc l'intérêt de notre approche de décomposition des opérations complexes de morphologie mathématique par une suite d'opérations simples hautement parallélisables.

Mots clefs : morphologie mathématique, système sur puce, FPGA, flot de données, parallélisme.

Table des matières

Liste des figures

Liste des tableaux

Liste des algorithmes

Glossaire

ALU	Arithmetic and Logical Unit
ASIC	Application Specific Integrated Circuit
ASIP	Application Specific Instruction-set Processor
AVX	Advanced Vector Extensions
CISC	Complex Instruction Set Computer
FAH	File d'Attente Hiérarchique
FPGA	Field-Programmable Gate Array
GFlOPS	Giga Floating Operations Per Seconds
GOPS	Giga Operations Per Seconds
GPGPU	General-Purpose Processing on Graphics Processing Units
GPU	Graphics Processing Unit
LPE	Ligne de Partage des Eaux
MIMD	Multiple Instruction stream Multiple Data stream
MISD	Multiple Instruction stream Single Data stream
MMX	MultiMedia eXtensions
NoC	Network on Chip
NOP	No Operation
PE	Processing Element
PoC	PIMM1[40] on Chip
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Level
SIMD	Single Instruction stream Multiple Data stream
SISD	Single Instruction stream Signle Data stream
SMP	Symmetric Multiprocessing
SoC	System on Chip

SPoC	Several PIMM1[40] on Chip
SSE	Streaming SIMD Extensions
TOPS	Tera Operation Per Seconds
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLIW	Very Large Instruction Word

Chapitre 1

Introduction

1.1 Contexte et motivations

Nous entendons très souvent que la complexité des algorithmes ne cesse de croître rendant le travail des architectes toujours plus ardu. Mais derrière cette sempiternelle affirmation se cache malgré tout une réalité liée à la demande de fonctionnalité ou de robustesse accrue des applications. Le traitement des images en temps réel est alors mis à mal, car ces nouveaux algorithmes requièrent une forte puissance de calcul difficile, voire impossible, à atteindre dans différents domaines et principalement celui de l'embarqué.

Trois défis majeurs sont à relever autour du déploiement d'applications temps réel embarquées de traitement d'images. Le premier consiste à disposer d'algorithmes en adéquation avec les architectures et vice versa. En effet, les opérateurs de morphologie mathématique disposent, pour la plupart, de plusieurs mises en œuvre algorithmiques différentes. Ces mises en œuvre vont de la version la plus proche de la définition mathématique jusqu'à la version la plus optimisée algorithmiquement. Les optimisations proposées d'ordinaire supposent l'utilisation de processeurs généralistes et reposent sur un objectif de réduction du nombre d'opérations, ceci même au prix de l'utilisation des structures de données complexes et coûteuses. On peut citer, par exemple, les files d'attente hiérarchiques dans le cadre du calcul de la ligne de partage des eaux. Ces files, lorsqu'elles sont déployées sur des circuits dédiés, donnent naissance à des circuits trop rigides ne pouvant pas réaliser les nombreuses autres étapes d'une application.

Le second défi correspond à notre capacité à déployer de nouvelles applications toujours plus robustes et donc généralement plus coûteuses en calcul. En effet, des opérateurs morphologiques considérés comme complexes il y a quelques années sont considérés comme basiques de nos jours. On peut citer comme exemple les segmentations hiérarchiques d'images qui utilisent maintenant un grand nombre d'opérateurs de ligne de partage des eaux, bien que le calcul d'une seule ligne de partage des eaux fonctionne déjà difficilement en temps réel sur une machine embarquée voire même sur une station de travail.

Le troisième défi est beaucoup plus technologique, car les systèmes embarqués doivent paradoxalement être de plus en plus polyvalents pour supporter des puissances de calcul grandissantes. Ce paradoxe trouve ses fondements dans le fait qu'une augmentation des puissances de calcul est liée à la diminution de la finesse de gravure en usine. Cette finesse rend donc de plus en plus coûteuse la fabrication d'un circuit. Parallèlement, il est de

moins en moins rentable de réaliser un circuit purement dédié à un unique algorithme, car le volume des pièces à atteindre ne cesse d'augmenter. Un circuit plus souple et regroupant plusieurs domaines aura, par construction, un marché plus important et pourra donc supporter plus facilement des coûts élevés de fabrication.

Nous pouvons alors nous demander comment aborder le domaine de l'embarqué qui ne peut bien évidemment pas se satisfaire uniquement des processeurs généralistes. Plusieurs possibilités s'offrent à nous et l'on peut, dans un premier temps, détourner des circuits performants au prix de quelques compromis afin de les utiliser dans d'autres domaines. On peut citer l'exemple des circuits dédiés à l'accélération graphique où leur utilisation dans le domaine scientifique n'est pas un hasard. En effet, il est possible, avec ces circuits, de disposer de systèmes puissants assez bon marché, car ils sont déjà rentabilisés pour leur domaine premier, à savoir le monde du jeu et du graphisme. Toutefois, la consommation de ces systèmes est assez rédhibitoire et les versions mobiles de ces derniers sont de nos jours encore loin de disposer des mêmes puissances de calcul.

Une autre solution serait de proposer un circuit, non plus orienté vers un algorithme, mais plutôt vers un domaine tel que le traitement d'images. Un circuit développé dans ce cadre a beaucoup plus de possibilités de se vendre, pour les raisons que nous avons évoqué précédemment. Une dernière solution très prometteuse, qui n'exclut pas la précédente, est de considérer les circuits reconfigurables. Nous pouvons citer les circuits logiques programmables, qui par leur polyvalence fine, peuvent cibler aussi bien une architecture spécifique d'un algorithme, une architecture spécifique d'un domaine ou bien même une architecture de type processeur du signal ou généraliste.

Notre motivation réside alors dans la nécessité de proposer des architectures suffisamment flexibles pour appréhender simultanément différentes formes de parallélisme, avec pour objectif d'obtenir la plus grande puissance de calcul quelles que soient les étapes d'une application à réaliser.

1.2 Les architectures flots de données

La morphologie mathématique est une discipline formalisée en 1968 par Georges Matheron et Jean Serra et qui, dès le début, s'est toujours attachée aux problèmes de mise en œuvre efficace d'applications. La morphologie mathématique s'est très largement enrichie au fil des années et de nombreuses applications robustes ont été développées aussi bien dans les domaines du médical, de la défense, de l'automobile et bien d'autres encore.

Des opérateurs matériels dédiés, fonctionnant en flots de données dans un premier temps analogiques puis numériques, ont été développés afin de proposer de puissantes machines de traitement d'images. Devant l'émergence des stations de travail, ces machines sont tombées peu à peu en désuétude, car les processeurs généralistes proposaient jusqu'à aujourd'hui une fréquence de fonctionnement croissante. Cette évolution verticale des processeurs garantissait au développeur de plus grandes puissances de calcul, sans modifier son application, à chaque nouvelle génération de processeur.

Le silicium montre aujourd'hui ses limites vis-à-vis de la fréquence de fonctionnement des circuits et la croissance des processeurs se fait maintenant de manière horizontale en augmentant le nombre de cœurs, par exemple, via un modèle à mémoire commune

symétrique (SMP). Or ce type de croissance ne permet pas un gain en performance directe comme dans le cas d'une croissance verticale. La gestion des systèmes multicœurs SMP est matériellement et logiciellement complexe et ces machines ne sont souvent pas les plus efficaces concernant le traitement d'images en raison de la grande quantité d'accès mémoires nécessaire. Des problèmes de quantité de cache, de cohérence et de partage des données surviennent rapidement et diminuent fortement l'accélération escomptée des machines multicœurs SMP.

Il est alors impératif de revoir la façon dont la morphologie mathématique est mise en œuvre dans les systèmes embarqués. Nous proposons de nous reposer sur sa capacité à pouvoir décrire la quasi-totalité des opérateurs complexes via les opérateurs de base que sont l'érosion, la dilatation ou encore les transformations tout ou rien. Ces opérations simples sont hautement parallélisables et même si elles doivent être itérées plusieurs centaines de fois pour composer des opérations complexes, elles présentent tout de même un grand intérêt. Ces opérations de base rendent possible d'une part, l'exploitation spatiale du parallélisme, car le traitement d'un pixel ne dépend que d'un nombre limité de voisins connu à priori et d'autre part, l'exploitation temporelle du parallélisme, car ces opérations peuvent être chaînées au sein d'un pipeline.

L'utilisation d'architectures flot de données, dédiées au calcul des opérations de base de la morphologie mathématique, reprend alors tout son sens puisque ces machines sont efficaces et peuvent être aujourd'hui intégrées en grand nombre dans le même circuit.

1.3 Organisation du mémoire

Nous nous sommes attachés dans ce mémoire à présenter nos travaux autour des architectures flots de données qui, à travers un objectif commun de performance, montrent des degrés croissants de programmabilité, le but final étant de s'approcher au plus près d'une machine idéale alliant performance et généricité.

La première partie se concentre sur les processeurs de voisinage flot de données et bien que cette architecture ne soit pas des plus récentes, cela ne nous empêche pas de faire apparaître de nouvelles structures performantes ainsi que les problèmes afférents de gestion des voisinages, des bords et de l'accès aux données ; problèmes auxquels nous apportons des améliorations et des solutions. La seconde partie revisite l'assemblage de processeurs flot de données en pipeline et les problèmes de rendement et de programmation, dans un contexte reconfigurable, sont analysés. La troisième partie montre, via la mise en place de processeurs VLIW vectoriels, comment peut être améliorée la souplesse et la performance des machines flots de données sur puce. Cette performance est d'ailleurs analysée dans la dernière partie de ce manuscrit.

1.3.1 Première partie : processeurs de voisinage flots de données

Ce premier chapitre présente le fonctionnement des différentes structures de processeurs de voisinage existantes. Nous rappelons notamment la structure la plus standard de processeur de voisinage en exposant clairement les différents noyaux de calcul disponibles et en détaillant, pour chacun d'eux, le mode de gestion des bords à mettre en place. En

effet le bon traitement des bords est crucial, car des erreurs à ce niveau sont désastreuses lorsque des pipelines de processeurs seront mis en œuvre dans la partie suivante.

Après cette analyse, nous présentons nos différentes évolutions de la structure du processeur, de la structure d'extraction du voisinage et de la structure des noyaux de calculs. Nous montrons en particulier une nouvelle méthode d'extraction du voisinage qui permet d'extraire plusieurs voisinages contigus simultanément, sans rompre avec le mode flot de données des processeurs de voisinage et sans consommation mémoire supplémentaire.

1.3.2 Deuxième partie : chaînage de processeurs de voisinage

La force des processeurs de voisinage décrits dans la partie précédente réside dans leur capacité à exploiter le parallélisme temporel. Leur chaînage, au sein d'un pipeline, est donc naturel. Nous détaillons ici, sans nous fixer nécessairement sur une structure interne de processeur, une composition générale d'un système sur puce capable d'exploiter un pipeline de processeurs de voisinage à plusieurs entrées.

Nous proposons une méthodologie de génération et de gestion des pipelines dans le but de répondre à la problématique de sous-utilisation de ces derniers, souvent liée à une profondeur mal adaptée au domaine d'applications considéré. Nous montrons également comment, à partir de plusieurs descriptions fonctionnelles, nous sommes capables de générer un unique pipeline, mutualisant un maximum de ressources de toutes les descriptions initiales. Cette dernière technique nous permet alors de générer automatiquement à la fois des systèmes dynamiques avec des chemins de données paramétrables, souples dans la diversité des opérations réalisables et performants en terme de rapidité de traitement des applications. Cette partie représente une première avancée dans notre objectif de performance et de généricité ou de possibilité de réutilisation de l'architecture.

1.3.3 Troisième partie : processeurs VLIW vectoriels

Ce chapitre marque un changement fort par rapport aux systèmes décrits précédemment et introduit la notion de processeur programmable flot de données.

Nous commençons par présenter la structure d'un cœur microcodé vectoriel pour ensuite décrire comment nous pouvons en structurer plusieurs dans un système multicœurs flots de données sur puce. La structure même du pipeline dépend du microcode déployé dans les processeurs et nous permet de travailler à plusieurs niveaux de granularité. Une telle structure nous permet, soit d'utiliser tout le pipeline pour réaliser une unique opération distribuée sur tous les cœurs, soit de morceler des traitements à un grain moyen entre différents sous-pipelines parallèles, soit de déployer une application complète sur un unique pipeline monolithique. En outre, des exemples variés d'utilisation de l'architecture sont proposés et démontre la nouvelle progression réalisée en terme de performance et de généricité de l'architecture.

1.3.4 Quatrième partie : comparaisons des différentes architectures

Les comparaisons de performance tiennent une place importante dans nos travaux, car elles permettent de juger des gains apportés par nos systèmes vis-à-vis d'architectures cibles répandues et performantes. Une méthodologie simple de test est proposée afin de mettre en exergue les points forts et faibles de chaque système. Des résultats numériques sont présentés et nous nous sommes attachés à proposer un indice de performance en terme de nombre de cycles par pixels produits. Cet indice étant indépendant de la fréquence, il permet de gommer les aspérités liées aux différentes technologies de fabrication ou de réalisation des accélérateurs et de proposer une comparaison juste et cohérente.

Chapitre 2

Processeurs de voisinage flots de données

Un très grand nombre d'opérateurs avancés de morphologie mathématique découlent principalement de briques élémentaires basées sur des opérations de voisinage comme des érosions, dilatations ou bien encore des opérations *tout ou rien*. Des algorithmes permettent de réaliser ces opérateurs sur des processeurs généralistes de manière efficace en exploitant des structures de données plus ou moins complexes telles que des histogrammes ou bien encore des files d'attente hiérarchiques. La réalisation d'architectures dédiées, exploitant de telles structures de données, donne souvent naissance à des systèmes très rigides, disposant parfois de plusieurs bancs mémoires ou encore ne fonctionnant pas dans un temps déterministe. Nous avons donc orienté notre travail vers la réalisation de briques élémentaires via des processeurs de voisinage flots de données. Ces derniers étant efficaces, simples à mettre en œuvre et ayant de fortes possibilités de parallélisation et de chaînage pour créer facilement les opérations avancées de morphologie mathématique.

Ce chapitre aborde les différentes structures des processeurs de voisinage ainsi que les différents calculs réalisables. Nous considérons ici un processeur flot de données comme étant un opérateur câblé dont la transformation est sélectionnée avant l'acheminement, au fil de l'eau, des données. L'objectif est de disposer d'un système capable de traiter au plus vite les pixels d'une image avec un minimum de contrôle. Habituellement, ce type d'opérateur est relégué au niveau de la chaîne de prétraitement d'une application, car jugé trop rigide de par sa structure. Toutefois, nous montrerons leur intérêt dans le cadre de la morphologie mathématique au fil des chapitres.

Nous allons détailler les différentes structures des processeurs de voisinage et nous commencerons par en détailler l'architecture la plus commune. Nous analyserons et proposerons ensuite des structures proches permettant la mise en place d'une récursion pendant le traitement des données ou bien encore une nouvelle structure d'extraction parallèle des voisinages. Cette dernière structure permet un accès simultané à des groupes de voisins contigus, améliorant ainsi les performances et diminuant la latence du traitement. Nous aborderons également d'autres architectures dédiées aux éléments structurants de type segment et nous présenterons une nouvelle optimisation dans ce domaine afin d'économiser une grande partie de la mémoire employée. Enfin, une conclusion est proposée sous forme d'un récapitulatif et d'une analyse des performances des systèmes mis en jeu ici.

2.1 Avant propos

Dans le contexte des architectures dédiées au traitement d'images, il est possible de trouver un grand nombre de structures. Deux catégories existent, les architectures asynchrones [27] et les architectures synchrones. Étant focalisés sur un type de traitement bas niveau, nous ne nous intéresserons ici qu'aux architectures synchrones et parallèles.

Une classification des architectures des ordinateurs a été proposée par Flynn[29] et se décompose en quatre catégories :

- SISD, *Single Instruction stream Single Data stream* : il s'agit de la machine séquentielle standard, ne contenant aucun parallélisme.
- MISD, *Multiple Instruction stream Single Data stream* : très peu employée seule, cette architecture permet à plusieurs processeurs de travailler sur la même donnée
- SIMD, *Single Instruction stream Multiple Data stream* : plusieurs processeurs exécutent la même instruction sur plusieurs données (processeurs vectoriels...)
- MIMD, *Multiple Instruction stream Multiple Data stream* : plusieurs processeurs exécutent plusieurs instructions sur plusieurs données (multiprocesseur...)

Toutefois, cette classification des architectures met principalement en exergue le mode de fonctionnement du parallélisme via la configuration du réseau entre les différentes unités de calculs. Aucune hypothèse n'est formulée concernant la nature spatiale ou temporelle du parallélisme. C'est la raison pour laquelle nous différencions deux grandes classes de machine : les machines à répartition spatiale du parallélisme et les machines à répartition temporelle du parallélisme. Il existe bien entendu des passerelles entre les deux, une machine peut travailler localement avec un parallélisme spatial, mais globalement avec un parallélisme temporel.

2.1.1 Parallélisme spatial

Nous présentons ici les deux grandes classes de parallélisme spatial utilisées en pratique. Toutefois, nous dissocions les machines vectorielles des machines SIMD, car même si dans les deux cas une seule instruction est exécutée par cycle pour plusieurs données traitées, la gestion de la mémoire est souvent très différente. En effet, les machines vectorielles ont très souvent accès à la mémoire dans sa globalité alors que les machines SIMD disposent très souvent d'une mémoire distribuée dans tous les éléments de calcul.

Machines vectorielles Les processeurs vectoriels sont généralement composés de plusieurs unités de calculs connectées à plusieurs bus de communication. On retrouve généralement à la fois des unités vectorielles et scalaires, chacune dédiée à une classe de calcul (entier, flottant...). Un exemple d'une telle machine est donné à la figure 2.1 représentant le premier *Cray*[64].

La particularité des machines vectorielles réside dans leur capacité à abstraire le nombre des éléments d'un registre vectoriel vis-à-vis du nombre physique d'unités de calcul élémentaires. Cette particularité engendre, selon le nombre des unités de calcul, un nombre de cycles variable pour exécuter une instruction.

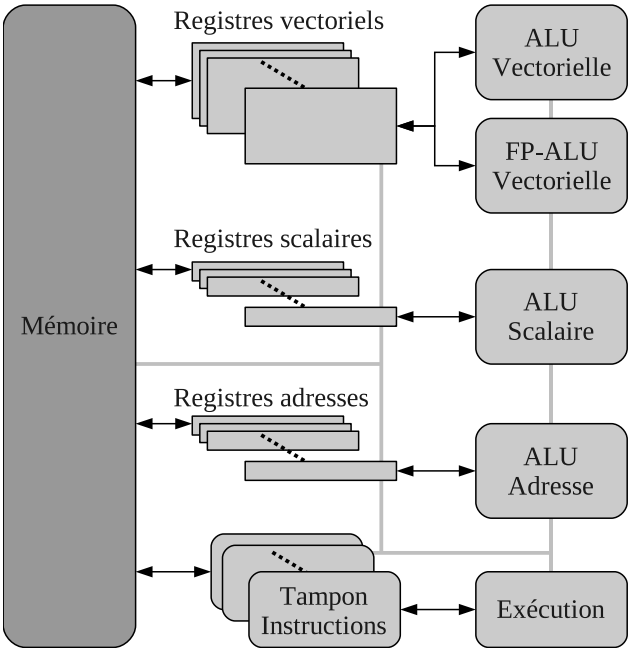


FIG. 2.1: Vue générale de la machine vectorielle *Cray* de première génération

Machines SIMD Les machines SIMD sont composées d'éléments de calcul (PE) raccordés entre eux via un système d'interconnexion mono ou multidimensionnel. Chaque PE exécute la même instruction à un cycle donné et la mémoire du système est généralement distribuée dans chacun d'eux.

La machine *AIS-5000* [66] présente un raccordement des PE bit-série sous la forme d'une chaîne monodimensionnelle. Les différents PE se programment via une table de vérité programmable par l'utilisateur et disposent d'un maximum de quatre opérandes en entrées pouvant être lues en mémoire. La chaîne reliant les différents PE permet également d'alimenter les opérandes via la mémoire des PE voisins. Une vue synthétique du tableau de PE est proposée en figure 2.2.

D'autres topologies entre les PE sont possibles et notamment une structure bidimensionnelle telle que celle de la machine *PAPRICA* [18][19] dédiée au traitement d'images par morphologie mathématique. Cette machine, présentée en figure 2.3, embarque un tableau de 16 PE groupés dans une grille 4×4 afin de gérer simplement les opérations de voisinage ainsi que le traitement des images par fenêtrage. La matrice des PE est optimisée pour réaliser des calculs parallèles sur quatre voisinages 3×3 dont les pixels centraux sont tous juxtaposés.

Machines MIMD Les machines MIMD sont construites à partir de plusieurs processeurs indépendants, exécutant chacun leur programme et reliés par un système d'interconnexion synchrone ou asynchrone. Un bon exemple est le *Transputer*[6],[52] proposé par Barron en 1983, qui pouvait être composé d'un grand nombre de microprocesseurs chacun interconnecté à quatre voisins par un réseau asynchrone. La figure 2.4 présente une vue simplifiée du *Transputer* et la figure 2.5 propose quelques exemples de topologies de réseau réalisables.

D'autres machines MIMD plus orientées vers le traitement d'images existent, notam-

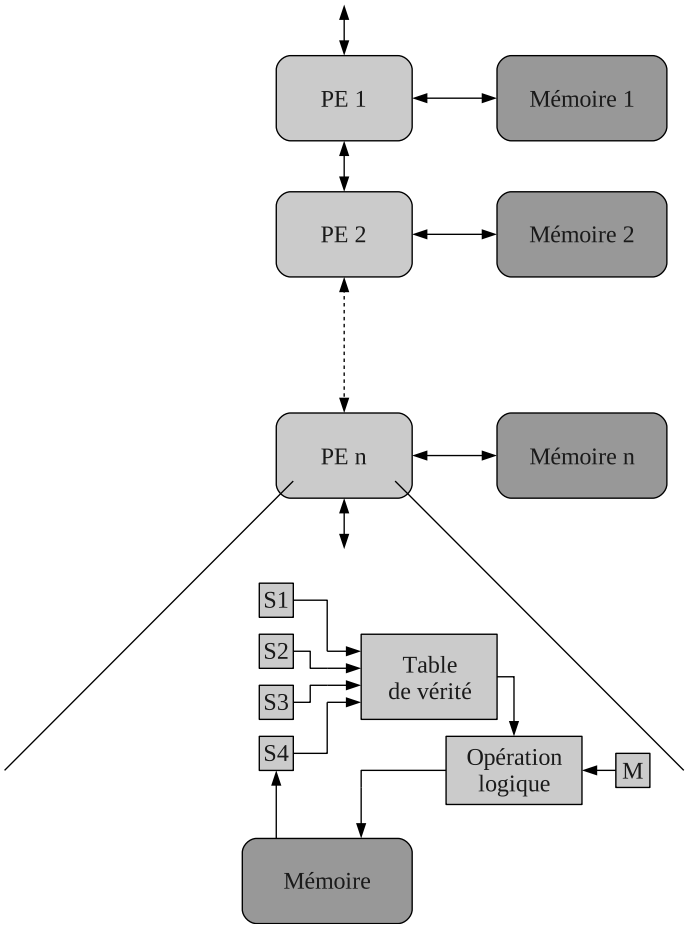


FIG. 2.2: Vue générale de la machine AIS-5000

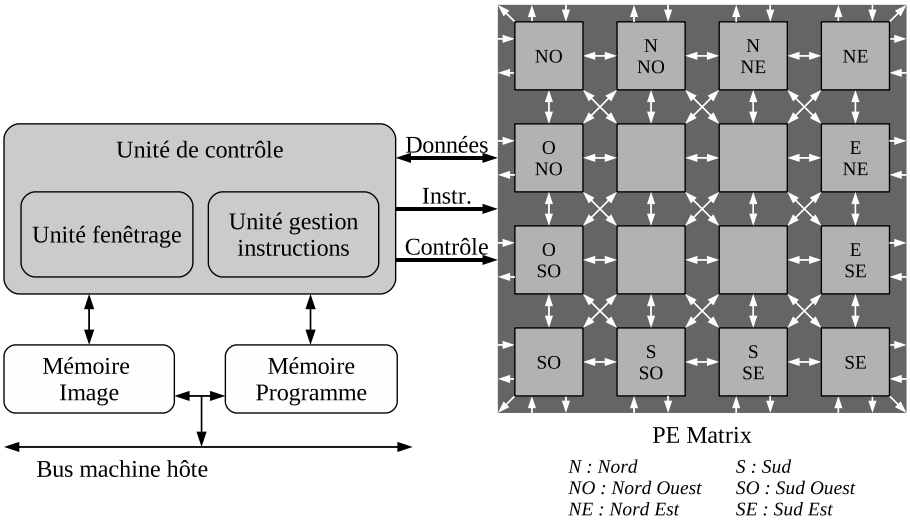


FIG. 2.3: Vue générale de la machine *PAPRICA*

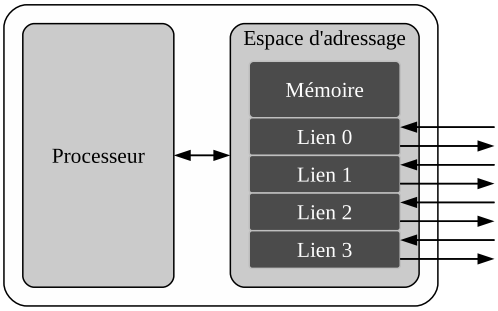
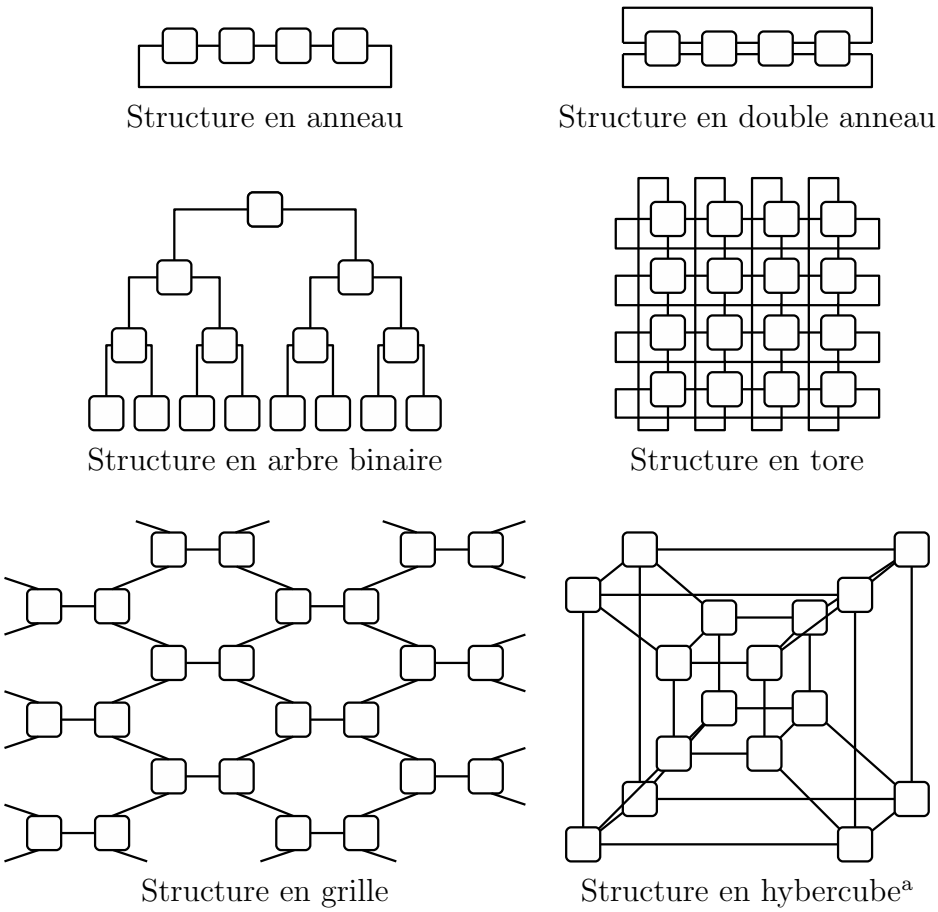


FIG. 2.4: Structure d'un noeud de calcul du Transputer



[a]La dimension maximale d'un hypercube avec le Transputer est quatre en raison du nombre de liens de communication limités à quatre par élément de calcul de la machine

FIG. 2.5: Structure d'un noeud de calcul du *Transputer*

ment la machine *GPIP* d'Hitachi [76]. Cette machine, dont la vue générale est proposée en figure 2.6, se compose de 64 processeurs. Chaque processeur consiste en un DSP accompagné d'une mémoire externe de 512K octets. Une unité de contrôle de chemin, située entre le DSP et sa mémoire locale permet la liaison entre processeurs voisins, la liaison ainsi formée constitue un pipeline en anneau. Cette liaison contrôlée par une unité de gestion de l'anneau fournit un mécanisme pour l'échange de données image entre processeurs, pour la diffusion d'une même image à l'ensemble des processeurs et également pour la circulation de résultats intermédiaires entre les processeurs.

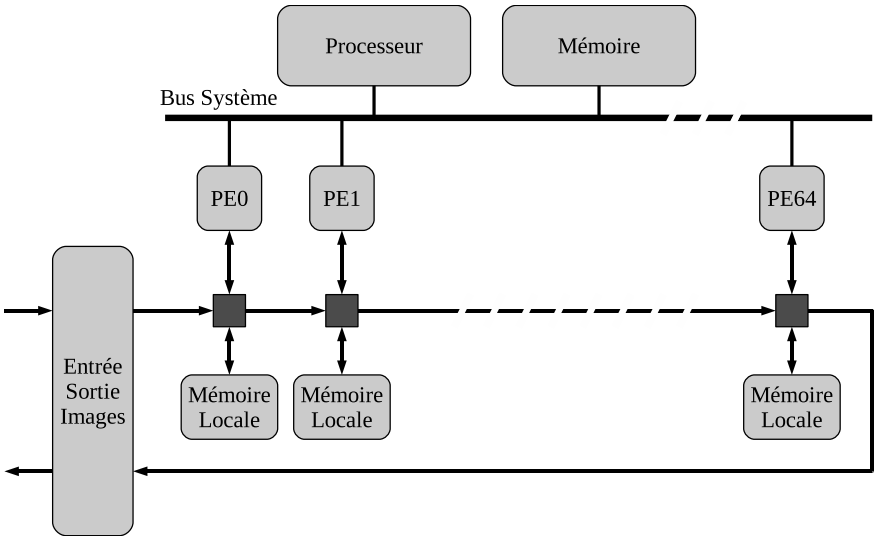


FIG. 2.6: Vue globale de la machine *gpip* d'Hitachi

2.1.2 Parallélisme temporel

L'exploitation de ce parallélisme est généralement réalisée en raccordant, dans une structure systolique ou pipeline, un grand nombre de processeurs exécutant ou non la même opération. Les processeurs sont généralement linéairement connectés et un flot d'images est guidé à travers. L'avantage d'une telle structure est de pouvoir chaîner les différentes opérations de traitement d'images à réaliser en une seule passe dans la machine.

La machine *Cytocomputer* [50] est un exemple de pipeline linéaire composé de processeurs de voisinage 3×3. Les images entrent dans le pipeline comme un flot de pixels au format balayage ligne et progressent dans le pipeline à un rythme constant. À l'issue de la phase d'initialisation du pipeline, les pixels calculés sortent au rythme des pixels d'entrée. Chaque étage comprend un ensemble de registres à décalage capable de mémoriser deux lignes contiguës d'une image tandis qu'une fenêtre de registres mémorise les 9 pixels d'un voisinage qui constituent la fenêtre d'entrée 3×3 du module logique de voisinage. Ce module réalise la transformation programmée du pixel central et de ses huit voisins.

On peut également citer les machines PIPE[37],[72] et PIMM1[40] qui sont très similaires au Cytocomputer, mais qui ont l'avantage d'être pilotées par un système hôte afin d'itérer plusieurs passes dans le pipeline.

2.1.3 Synthèse

Les différents types de parallélisme présentés couvrent une grande partie du spectre des machines parallèles. Les structures citées comme exemple présentent souvent l'avantage d'être purement dédiées à une catégorie de parallélisme. Nous verrons dans le chapitre 4 que la tendance de nos jours est de mélanger toutes ces formes de parallélismes pour un besoin sans cesse grandissant en puissance de calcul.

Nous allons, dans les prochaines sections, nous attarder sur le fonctionnement même d'un processeur de voisinage inséré au sein d'une machine exploitant le parallélisme temporel. Nous présenterons d'une part, les nouveaux noyaux de calculs employables dans le cadre de récentes avancées en morphologie mathématique et d'autre part, une nouvelle méthode d'extraction parallélisée du voisinage ainsi que différentes avancées dans la gestion des noyaux de type segment.

2.2 Structure standard

2.2.1 Généralités

Les processeurs de voisinage ont été parmi les premières architectures numériques ou même analogiques à réaliser des opérations de traitement d'images basées sur des calculs linéaires ou non en considérant les voisinages d'une image. On peut citer l'ASIC PIMM1[40] qui permettait de traiter des images 8 bits avec des voisinages de taille maximum 3x3 en trame carrée ou avec des hexagones unitaires en trame hexagonale. Ce circuit était dévoué à des calculs de morphologie mathématique variés comme des érosions, des dilatations, des transformations tout ou rien ou encore des opérations géodésiques par propagation. PIMM1 pouvait soit fonctionner avec ses 8 unités de calculs binaires en parallèle pour traiter des images en teinte de gris, soit fonctionner avec ses 8 unités de calculs en série pour chaîner huit opérations de voisinage sur des images binaires. Les technologies de l'époque ne permettaient raisonnablement pas d'embarquer la mémoire nécessaire au stockage de deux lignes d'une image au sein du circuit. Ces deux lignes étant nécessaires à l'extraction d'un voisinage 3×3 comme nous le verrons plus tard dans ce mémoire. De la même manière, il était difficile de pouvoir disposer de plus d'un processeur de voisinage 8 bits par circuit ou même d'envisager d'autres unités de calcul pour travailler, par exemple, sur des filtres de rang.

Nous proposons ici de reprendre l'étude d'un tel processeur en considérant les moyens actuels, à la fois en terme technologique, mais aussi en terme d'algorithme puisque de nouvelles avancées ont été réalisées dans ces deux domaines. Les processus de fabrication suivent la loi de Moore[56] et permettent aujourd'hui d'envisager la mise en place d'un grand nombre de processeurs de voisinage dans le même circuit. On observe également la maturité de circuits logiques programmables qui permettent la reconfiguration dynamique et partielle[25] du circuit. Nous avons ainsi la possibilité d'embarquer plusieurs structures matérielles optimales à chaque étape d'un traitement. Il n'est donc plus nécessaire de disposer d'un processeur réalisant toutes les opérations possibles.

Avant d'aborder les diverses possibilités de calculs réalisables avec un processeur de voisinage, il est nécessaire de procéder à une petite introduction concernant l'extraction du

voisinage et à une explication du problème des bords. Nous reviendrons plus en détail sur ces points dans les prochaines sections.

Nous définissons un sens de parcours direct comme étant un balayage des lignes d'une image de gauche à droite puis de haut en bas. Le sens de parcours indirect est donc un balayage des lignes d'une image de droite à gauche puis de bas en haut. Un exemple est présenté en figure 2.7 et l'on remarque qu'il est possible de réaliser un parcours indirect d'une image en parcourant dans le sens direct la même image ayant subi une symétrie centrale.

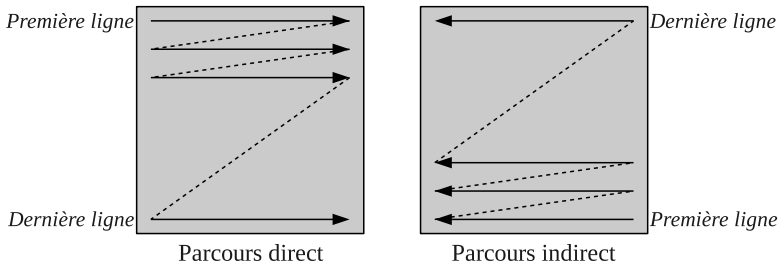


FIG. 2.7: Définition du sens de parcours d'une image

Nous supposons maintenant avoir, à chaque instant t, tous les voisins d'un pixel donné d'une image et dont les bords ont été gérés en remplaçant les pixels par des valeurs n'influençant pas le calcul. Un exemple est présenté en figure 2.8 où l'on extrait un voisinage carré 3×3 au bord de l'image. Le flot de pixels étant stocké en mémoire de façon contiguë et sans *padding*, le système d'extraction seul n'est pas capable de remplacer les valeurs *hors de l'image*. Il faut lui adjoindre un système de gestion des bords comptant les lignes et les colonnes de l'image pour que les bords soient correctement pris en compte à chaque instant.

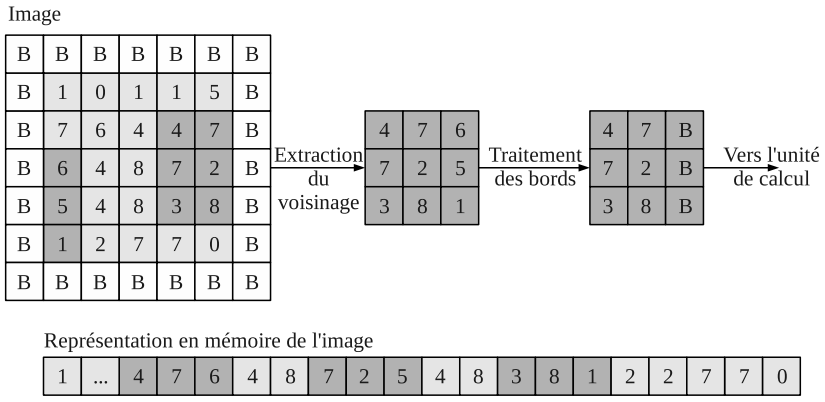


FIG. 2.8: Extraction d'un carré 3×3

Le cas de la maille 6-connexe ou hexagonale est traitée de la même manière que les bords de l'image. En effet pour réaliser un calcul avec un hexagone il est nécessaire de désactiver certains voisins en fonction de la parité des lignes ou des colonnes. La figure 2.9 présente, pour un hexagone de rayon 1, les voisins à prendre en compte en fonction de la parité de la ligne relative au centre de l'élément structurant.

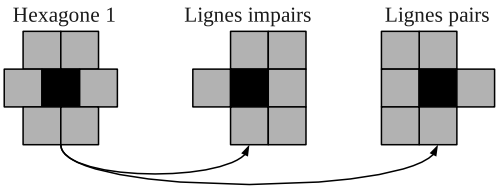


FIG. 2.9: Exemple de simulation d'un hexagone en maille 8-connexe

L'architecture d'un processeur de voisinage est réalisée pour un noyau maximal donné, par exemple un carré 3×3. Il peut être intéressant de pouvoir disposer d'un noyau comportant moins de voisins sans générer une nouvelle architecture. C'est la raison pour laquelle l'état de l'unité de gestion des bords peut être forcé pour toujours invalider certains voisins.

L'unité de gestion des bords est donc un élément central du système, car en plus de sa fonction première, elle est en mesure d'assurer une émulation d'une maille 6-connexe et de paramétrer les formes des noyaux. Nous reviendrons un peu plus en détail sur ce point dans la prochaine section de ce chapitre.

2.2.2 Vue globale

Le but d'une telle structure est de produire à chaque cycle le résultat d'une opération de voisinage. Le système se compose donc d'une unité de calcul, d'une unité d'extraction du voisinage et d'une unité de gestion des bords. Une vue générale de la structure est proposée en figure 2.10. Nous allons nous attarder, dans les prochaines sections, sur la manière de composer un processeur de voisinage et en particulier sur la méthode d'extraction du voisinage, sur la gestion correcte des bords et sur les différents noyaux de calculs pouvant être embarqués.

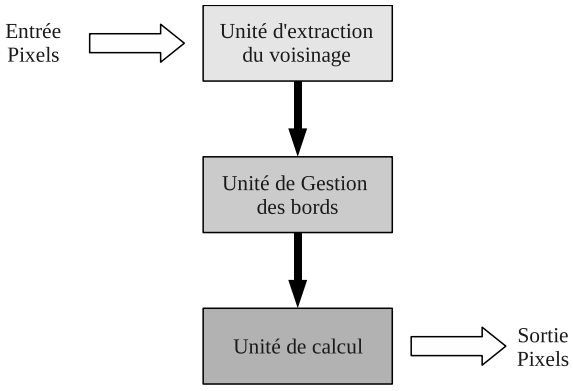


FIG. 2.10: Structure générale d'un processeur de voisinage simple

2.2.3 Extraction du voisinage

L'objectif d'un processeur de voisinage est d'extraire les voisinages d'une image tout en étant alimenté par un flux vidéo standard de pixels. Il n'est donc pas possible d'accéder,

aléatoirement, à tous les voisins désirés d'une image. Il faut ainsi mettre en place des mémoires stockant des lignes de l'image afin de pouvoir faire un calcul avec un noyau composé de M lignes et N colonnes. L'approche standard consiste à utiliser $M-1$ lignes à retard et $M \times N$ registres, deux structures sont présentées en figure 2.11 en considérant par exemple un voisinage maximum 3×3.

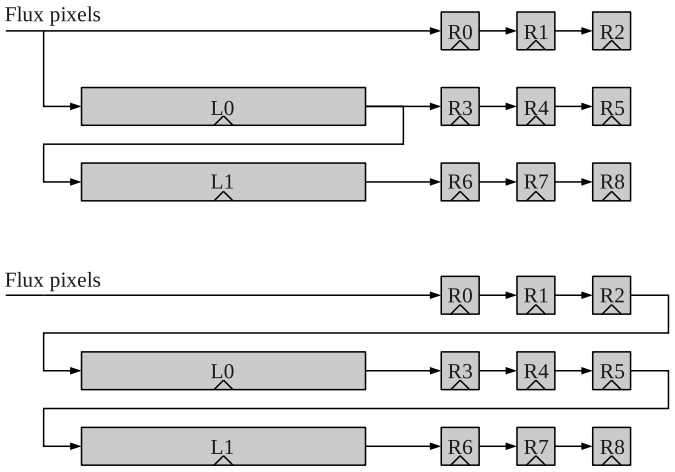


FIG. 2.11: Unités d'extraction de voisinages

Nous privilégierons la seconde structure dans la suite de cette section. Bien entendu, les lignes à retard ne doivent pas avoir une taille fixe, mais doivent pouvoir être paramétrées avec une taille k après synthèse du circuit. On peut pour cela utiliser une mémoire double port avec l'adresse d'écriture initialisée à $k-n$ et l'adresse de lecture initialisée à zéro. Le paramètre n dépend de la latence des mémoires et du nombre de colonnes N du voisinage dans le cas de la seconde structure décrite en 2.11. Les adresses sont incrémentées à chaque cycle grâce à des compteurs ce qui a pour effet de créer une ligne à retard paramétrable.

La structure peut être rendue plus souple par l'ajout de multiplexeurs afin de modifier la structure du voisinage. On peut par exemple court-circuiter les lignes à retard pour disposer d'un voisinage de type segment de taille $M \cdot N$ comme le montre la figure 2.12.

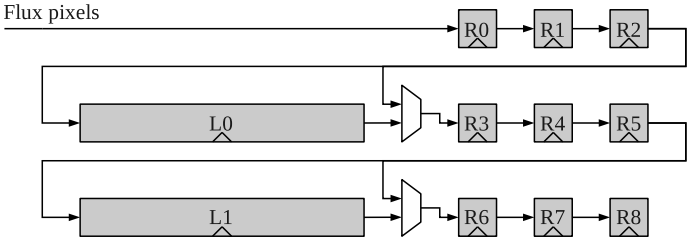


FIG. 2.12: Variante de l'unité d'extraction de voisinages

2.2.4 Gestion des bords

La gestion des bords est un problème complexe, car fonction du type d'algorithme à exécuter. En outre, nous avons même vu qu'il était parfois nécessaire d'en déporter une

partie ou la totalité dans les unités de calcul. C'est pour cette raison que cette unité est composée de deux blocs, un premier chargé de détecter lorsque l'on se trouve sur un bord de l'image et un second chargé d'appliquer une politique de bord en remplaçant les valeurs du voisinage, hors de l'image, par une valeur prédéterminée. Ce dernier bloc peut être court-circuité s'il s'avère nécessaire de gérer les bords au niveau de l'unité de calcul.

Le changement de la topologie du voisinage présentée dans la section précédente influe également sur le bloc chargé de la détection des bords, car même si le nombre de voisins ne change pas, leur politique de désélection varie en fonction de la topologie. Par exemple, un élément structurant 1×9 aura des problèmes de bords uniquement sur les bords Est et Ouest alors qu'un élément structurant 3×3 aura des problèmes de bords dans toutes les directions. La figure 2.13 propose une carte du nombre de cycles lignes/colonnes de désélection pour chacun des voisins d'un élément structurant 7×7. Par exemple, pour le voisin Nord-Ouest (3,3), il est nécessaire de le désactiver durant les trois premières lignes et les trois premières colonnes de chaque ligne.

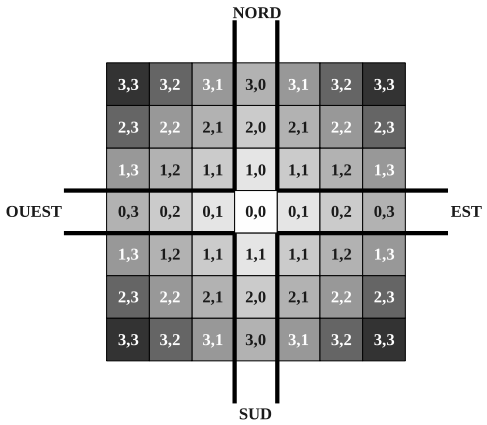


FIG. 2.13: Critère de désélection des voisins d'un élément structurant 7×7

Le premier bloc chargé de la détection des bords est présenté en figure 2.14. Nous nous plaçons dans le cas d'un élément structurant 7×7 avec les colonnes numérotées de gauche à droite de C0 à C6 et les lignes de haut en bas de L0 à L6.

Les sorties de cette unité permettent de choisir quelles couples lignes/colonnes doivent être désélectionnés. L'unité de déselection doit donc juste être capable de réagir à ces signaux pour remplacer certaines valeurs du voisinage par la politique de gestion des bords choisie. Cette unité, présentée en figure 2.15, se compose ainsi d'un multiplexeur à deux entrées par voisin, une entrée pour le pixel provenant de l'unité de voisinage et une entrée provenant d'un registre contenant la valeur du bord désirée. On peut aussi utiliser deux registres (B1, B2) contenant chacun une valeur de bord pour simuler un damier autour de l'image, utile pour les filtres de rang. La commande de ces multiplexeurs est réalisée par l'unité de détection des bords, mais peut être également forcée (signal "s") pour avoir la possibilité de choisir quels voisins doivent être actifs pour le calcul. On utilise alors, dans ce cas de figure, une porte OU avant la commande du multiplexeur. Il est encore possible à ce niveau de mettre en place la gestion de la maille hexagonale en désactivant une ligne sur deux des couples de voisins.

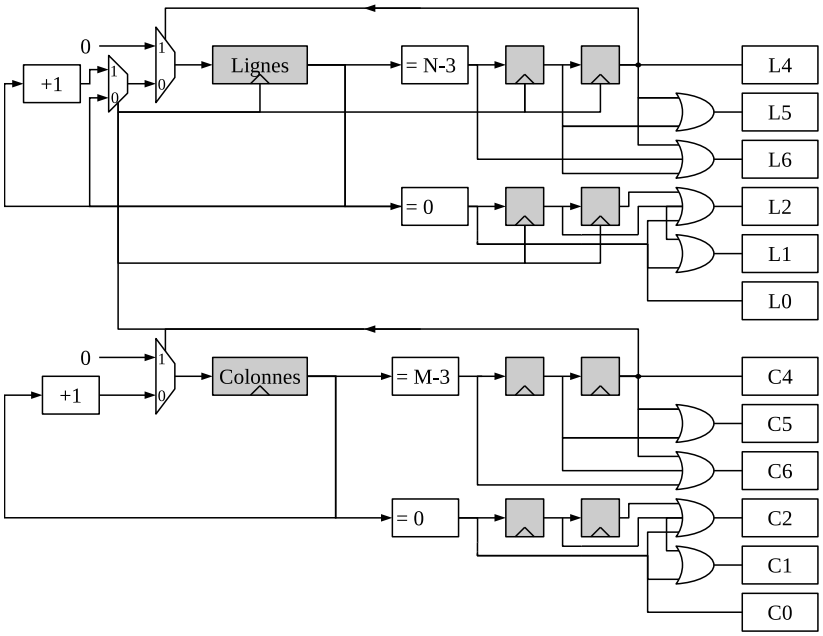


FIG. 2.14: Bloc de détection des bords pour un élément structurant 7×7

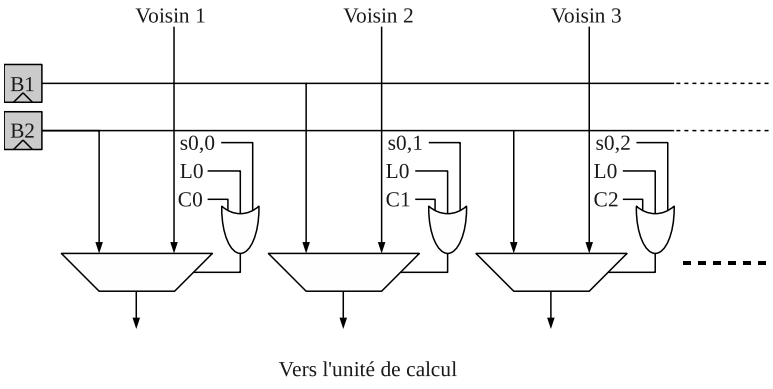


FIG. 2.15: Unité de déselection du voisinage

2.2.5 Noyaux de calcul basiques

2.2.5.1 Érosion et Dilatation

L'érosion ε et la dilatation δ font parties des opérations de base de la morphologie mathématique et sont définies de la manière suivante :

$$[\varepsilon_B(f)](x) = \min_{b \in B}(f(x+b))$$

$$[\delta_B(f)](x) = \max_{b \in B}(f(x+b))$$

L'élément structurant B représente la fenêtre dans laquelle est recherché le minimum ou le maximum autour de chaque pixel de f.

La propriété de dualité entre l'érosion et la dilatation, importante dans le domaine des architectures matérielles, est proposée ci-après :

$$[\varepsilon_B(f)](x) = ([\delta_B(f^c)](x))^c$$

Cette propriété permet d'utiliser un circuit dédié uniquement à l'érosion et de complémenter l'entrée et la sortie du système afin de disposer *gratuitement* d'une dilatation. Cette propriété est valable aussi bien dans le cadre des images binaires que dans le cadre des images à niveaux de gris.

Ces opérations ne sont pas nécessairement utiles seules, mais doivent être combinées pour créer des opérateurs de filtrages, de détection de contours, ou bien des opérateurs de segmentations tels que présentés dans la suite de l'ouvrage.

L'érosion ou la dilatation sont des cas particuliers de filtres de rang où il n'est pas obligatoire de procéder à un tri des pixels. Un simple minimum ou maximum sur le pixel central et l'ensemble de ses voisins suffit comme le montre la figure 2.16 où l'on recherche le minimum parmi les 9 valeurs d'un voisinage carré 3×3.

Les registres entre les différents opérateurs de calcul ne sont là qu'à titre indicatif et leur quantité dépend de la taille maximale des chemins critiques admissibles lors de la synthèse de l'architecture.

Une telle structure voit sa taille croître lorsque l'on augmente le nombre de voisins, car ce dernier correspond au nombre d'opérateurs min/max composant l'arbre de calcul. Dans le cas général, si le noyau dispose de N éléments (en comptant donc le pixel central) il faudra $N-1$ opérateurs min/max. Dans le cas d'un élément structurant rectangulaire de taille $I \times J$ (I correspond au nombre de lignes et J au nombre de colonnes) le nombre d'opérateurs min/max est :

$$N_{min/max} = I \cdot J - 1$$

Les calculs sur tous les voisins étant identiques, il est possible de réduire la taille de l'arbre dès lors qu'un sens de parcours de l'image est connu. Par exemple, un balayage direct tel que présenté dans la figure 2.17 permet d'économiser des opérateurs, car le calcul du minimum ou du maximum sur une colonne du voisinage à l'instant t est stocké dans un registre pour ne pas refaire ce même calcul au temps $t+1$ sur la colonne voisine.

L'exemple d'arbre optimisé permet de travailler avec les éléments structurants centrés décrits en 2.18. Les multiplexeurs ont plusieurs rôles, ils permettent une gestion des bords Est et Ouest de l'image, mais aussi de prendre en compte soit le minimum de la colonne

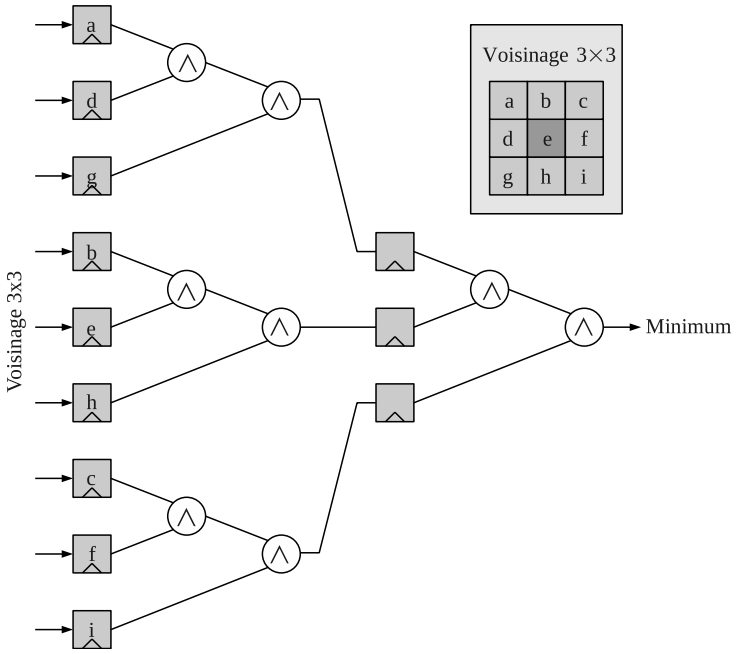


FIG. 2.16: Arbre de recherche du minimum considérant un pixel et ses 8 voisins

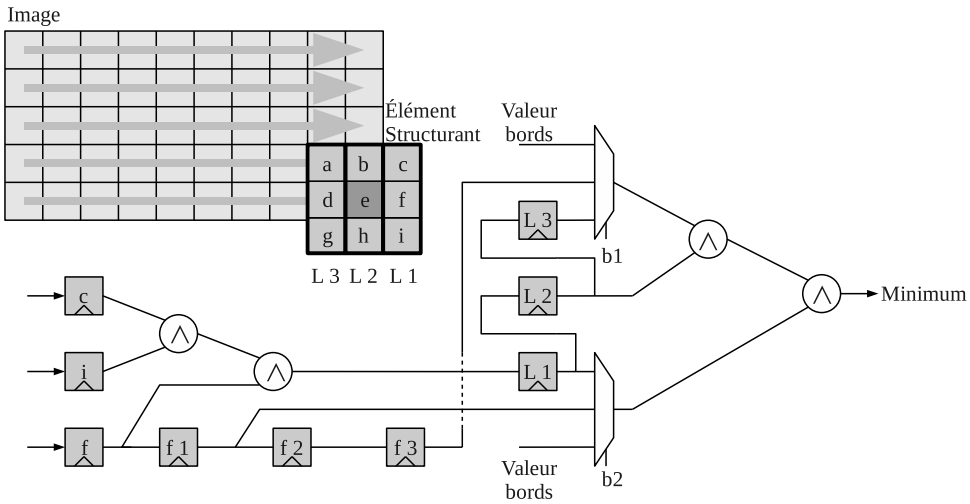


FIG. 2.17: Arbre réduit de recherche du minimum considérant un pixel et ses 8 voisins

soit la valeur centrale de cette dernière grâce aux registres “f1”, “f2” ou “f3”. Les commandes “b1” et “b2” doivent donc être pilotées via l'unité d'extraction des pixels de façon à avoir un contrôle cohérent puisque c'est à ce niveau que les bords Nord et Sud de l'image sont pris en compte.

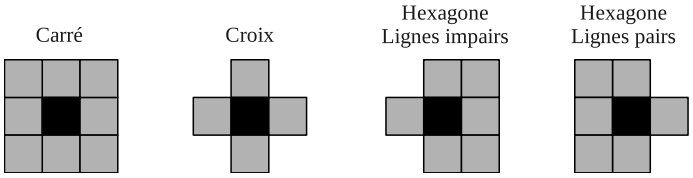


FIG. 2.18: Éléments structurants utilisables avec l'arbre de calcul optimisé du minimum de la figure 2.17

La quantité d'opérateurs min/max de cette structure optimisée, avec l'utilisation d'un élément rectangulaire de taille $I \times J$, est définie de la manière suivante :

$$N_{min/max} = I + J - 2$$

Il faut tout de même prendre garde aux bords de l'image, car des colonnes et des lignes du voisinage doivent être désactivées lors du calcul du minimum/maximum. Nous avions supposé précédemment que l'unité en charge de l'extraction des voisinages changeait les valeurs *hors de l'image* de manière à ne pas perturber le calcul (dans le cas d'une érosion, on change ces valeurs par la plus grande valeur possible donnée par la taille du codage des pixels), mais dans le cas de l'arbre de calcul optimisé, le système de gestion des bords doit être déporté en partie dans ce dernier. De plus, le cas de la maille hexagonale simulée en forçant certaines valeurs du voisinage doit être complètement pris en charge, non plus au niveau de l'extraction du voisinage, mais dans l'unité de calcul. Nous avons également perdu la possibilité d'invalider n'importe quel voisin de l'élément structurant sans compliquer davantage l'arbre de calcul.

2.2.5.2 Les filtres de rang

Les filtres de rang sont des filtres non linéaires permettant la sélection d'un pixel parmi un ensemble de pixels connexes. Le calcul sur un noyau se déroule en deux étapes, tout d'abord un tri des pixels est réalisé afin d'ordonner l'ensemble pour ensuite choisir un pixel en fonction de sa position (de son rang) dans cet ensemble. Trois rangs se distinguent plus particulièrement et sont la base des opérateurs suivants lorsqu'ils sont itérés sur tous les voisinages de l'image :

- le rang minimum qui produit l'érosion
- le rang médian qui produit le filtre médian
- le rang maximum qui produit la dilatation

Si l'on considère que le tri doit fonctionner avec une cadence correspondant à la période du système, et que le nombre des valeurs est assez limité, il est judicieux de mettre à plat un algorithme de tri simple tel qu'un tri à bulle. Cet algorithme souvent peu utilisé pour trier une quantité de données supérieure à une vingtaine d'éléments possède dans notre cas de bonnes propriétés concernant les chemins de données. En effet les comparaisons des

valeurs se faisant deux à deux, la complexité des interconnexions dans l'arbre de calcul est relativement simple à prédéterminer.

On définit un atome de tri comme étant un opérateur renvoyant la valeur minimale et maximale parmi deux valeurs, cet opérateur est donc composé d'un opérateur max et d'un opérateur min. On définit également un étage de tri comme étant une structure dans laquelle un certain nombre d'atomes de tri sont disposés en parallèle. Un exemple de tri avec un élément structurant 8-connexe est présenté en figure 2.19.

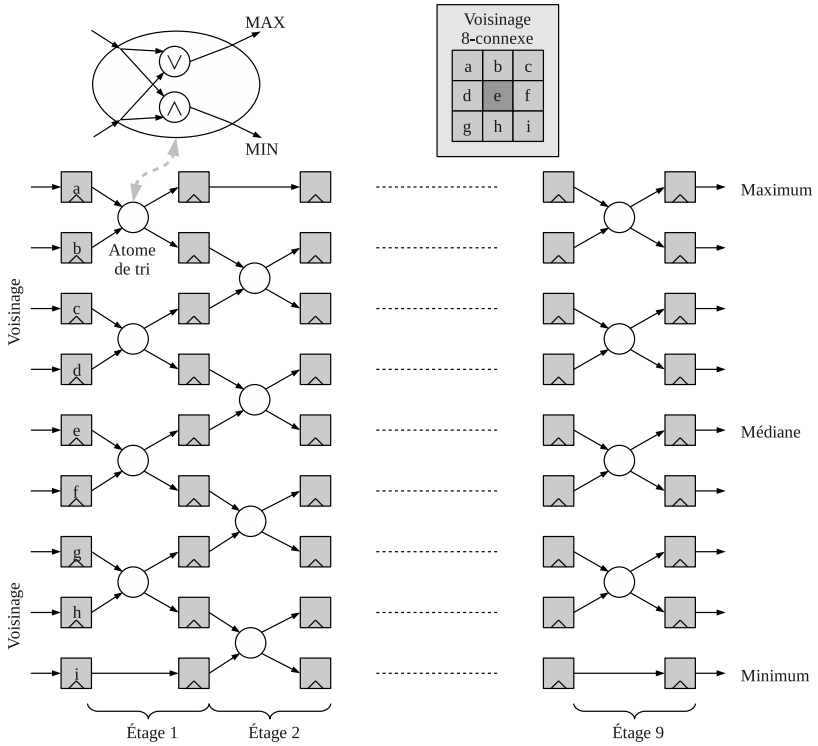


FIG. 2.19: Unité de tri à bulle de 9 valeurs

L'équation ci-dessous permet de connaître, pour un nombre N de valeurs à trier, le nombre nécessaire d'atomes de tri N_{atomes} pour la mise en œuvre du tri à bulle :

$$N_{atomes} = \frac{N \cdot (N-1)}{2}$$

Le tri à bulle présente l'avantage d'avoir des chemins de données relativement simples, mais consomme un grand nombre d'atomes de tri. Il est possible de mettre en place le principe *diviser pour régner* utilisé dans les algorithmes de tri rapide au prix d'une interconnexion plus complexe entre les étages de l'unité de calcul. Un tel exemple est présenté en figure 2.20 où le tri de neuf valeurs est réalisé avec seulement 25 atomes de tri contre 36 pour un tri à bulle standard.

La gestion des bords est problématique dans le cadre des filtres de rang autres que l'érosion et la dilatation. En effet, le nombre de valeurs à trier peut varier selon la position de l'élément structurant dans l'image. Par exemple, pour un filtre médian avec un élément structurant carré centré de rayon unitaire, le nombre de valeurs à trier passe de quatre sur

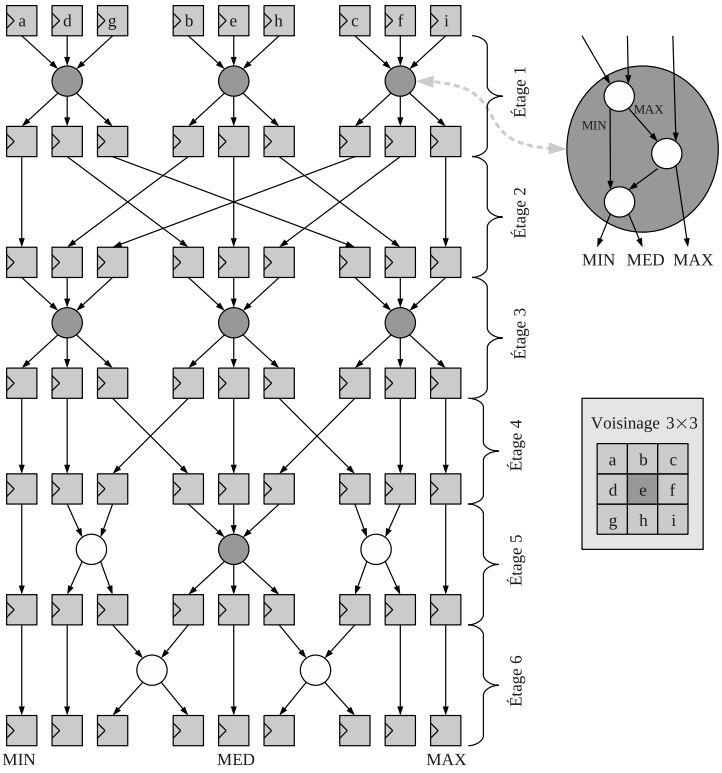


FIG. 2.20: Unité de tri fusion de 9 valeurs

un coin de l'image, à six dans un coté et neuf partout ailleurs. Deux possibilités s'offrent à nous pour gérer ce phénomène. La première consiste à changer la sélection du rang en sortie de l'unité de tri pour toujours "cibler" la bonne valeur lorsque le nombre de données à trier change. Cette possibilité est réalisable, mais nous posera problème lorsque l'on voudra optimiser l'unité de calcul en élaguant les atomes de tri menant aux filtres de rangs non nécessaires. La seconde possibilité est de faire en sorte que le *padding* émulé ne soit plus uniforme, mais un damier. Ceci à pour effet de concentrer les données utiles dans la partie centrale de la zone de tri évitant ainsi de changer la sélection du rang en sortie de l'unité de tri lorsqu'il y a moins de valeurs à trier. Un exemple est présenté en figure 2.21. Cette dernière méthode présente tout de même un petit inconvénient. Reprenons notre exemple de filtre médian avec un élément structurant carré de rayon unitaire et supposons que nous nous trouvons sur un bord dans une ligne paire où il y a donc six valeurs à trier (indexées de un à six). Notre *padding* en damier ajoute deux pixels blancs et un noir pour que l'unité de tri fonctionne bien avec neuf pixels. En choisissant la médiane, nous choisissons en réalité la valeur numéro quatre parmi les valeurs utiles triées (ce cas est présenté dans le cas B de la figure 2.21), alors que si nous avions été sur le même bord, mais dans une ligne impaire, le pixel blanc et les deux pixels noirs du *padding* nous auraient conduits à choisir la valeur numéro 3 parmi les valeurs utiles triées. Ceci n'est pas réellement problématique puisqu'il n'y a pas de bonne façon de choisir une valeur médiane parmi un nombre pair de valeurs. Certains algorithmes calculent la moyenne entre les deux médianes ce qui introduit une nouvelle valeur non présente initialement dans l'image, d'autres se fixent sur l'une des deux. Avec notre système nous changeons à chaque ligne le choix du rang de la médiane parmi les

deux possibles lorsqu'il y a un nombre pair de valeurs à trier. Un autre problème survient avec ce type de *padding*, si nous nous intéressons à d'autres rangs que la médiane, il se peut que sur les bords le calcul soit inexact à cause des valeurs du damier introduites dans le calcul. On observe ce phénomène sur la figure 2.21 en considérant cette fois, non plus la médiane, mais par exemple la valeur minimale. Il faut donc être en mesure, pour chaque rang voulu, de pouvoir spécifier avant le calcul le type de bord émulé, principalement un bord noir (dilatation), blanc (érosion) ou un damier (médian).

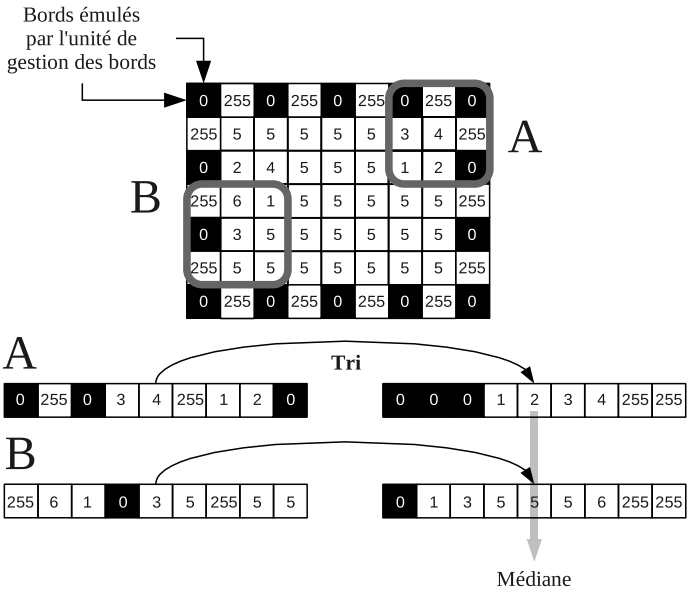


FIG. 2.21: Politique de gestion des bords des filtres de rang

L'émulation de la maille 6-connexe avec une unité de tri travaillant pour la maille 8-connexe peut se faire aussi en remplaçant alternativement, selon la parité de la ligne, le couple des voisins (Nord-Est, Sud-Est) ou (Nord-Ouest, Sud-Ouest) par un pixel blanc et un pixel noir. Le tri ne sera alors pas perturbé et donnera le même résultat que s'il avait été fait directement sur 7 valeurs. Bien sûr, les remarques précédentes doivent être prises en compte.

Nous allons voir maintenant qu'il est possible d'optimiser la structure de l'unité de calcul afin de grandement réduire la quantité d'atomes de tri grâce aux deux hypothèses développées dans les prochains paragraphes. La plupart de ces techniques de tri sont inspirées de l'ouvrage *Graphics GEMS*[32] et en particulier sur les travaux de Paeth [62] qui propose des tris optimaux dans le cadre d'une grille 3×3.

Première optimisation Il n'est pas nécessaire de procéder à un tri complet des données, car le plus souvent, seules quelques valeurs bien particulières comme la médiane, le minimum ou bien le maximum sont nécessaires. Il est ainsi possible d'élaguer tous les atomes de tri ne rentrant pas dans le calcul du ou des rangs considérés. Deux exemples sont présentés ci-après, le premier, figure 2.22, montre un tri fusion élagué sur cinq valeurs ne produisant que la valeur médiane. Le second, figure 2.23 est un tri fusion élagué sur neuf valeurs produisant le minimum, le maximum et la valeur médiane [33].

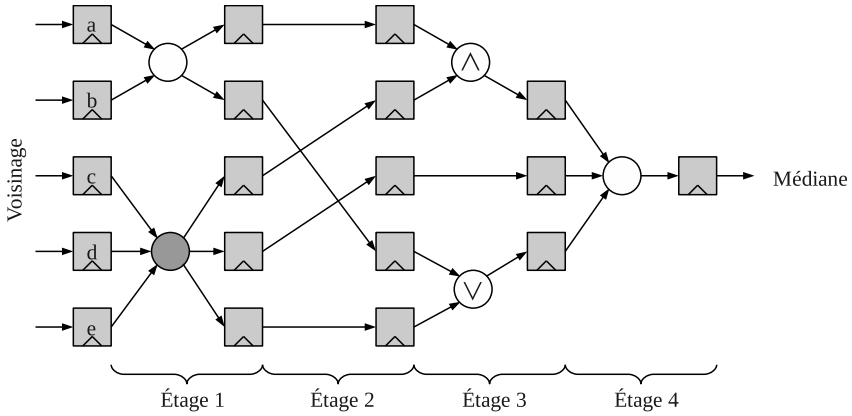


FIG. 2.22: Architecture de recherche de la valeur médiane sur 5 valeurs utilisant un tri fusion élagué

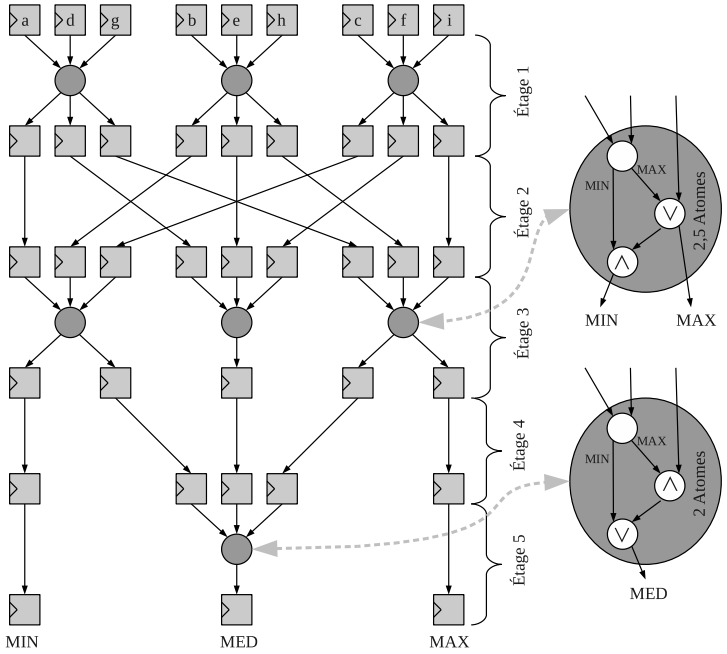


FIG. 2.23: Architecture de recherche du minimum, maximum et de la valeur médiane sur neuf valeurs utilisant un tri fusion élagué

Un récapitulatif des gains est proposé dans le tableau 2.1 pour différents types de tri considérant un jeu de données de taille 5, 7 et 9.

Type / Nb. Elt.	Tri Bulle			Tri Fusion		
	Complet	Min Max Médian	Médian seul	Complet	Min Max Médian	Médian seul
5 éléments	10	8.5[a]	7	9	8	7
	100%	85%	70%	90%	80%	70%
7 éléments	21	18	15	18	15[b]	14[b]
	100%	85%	71%	85%	71%	66%
9 éléments	36	31	26	25	18	15
	100%	86%	72%	70%	50%	42%

[a]Soit 8 atomes plus un min ou un max selon la manière dont a commencé le tri à bulle.
[b]L'architecture de l'unité de tri a été modifiée pour optimiser l'élagage, cf figure 2.24.

TAB. 2.1: Tableau récapitulatif des tailles de différentes unités de tri en nombre d'atomes

On observe avec ce tableau que le principe d'élagage commence à porter ses fruits dès lors qu'il y a un nombre suffisant de données à trier. Dans ce dernier cas, les étages où l'élagage est possible sont peu nombreux puisqu'il faut toujours conserver le premier étage du tri fusion. L'élagage sur l'unité de tri à bulle est au mieux équivalent à l'élagage de l'unité de tri fusion pour de petits jeux de données, mais devient beaucoup moins intéressant lorsque la taille du jeu de données croît.

Le nombre d'atomes N_{atome} de tri pour un tri à bulle élagué comportant N éléments et ne produisant que la valeur médiane est donné par la relation suivante :

$$N_{atome} = \left(\frac{N-1}{2}\right)^2 + \frac{1}{2} \cdot \frac{N^2-1}{4}, \text{ avec N impair}$$

Le nombre d'atomes de tri pour un tri à bulle élagué comportant N éléments produisant uniquement le minimum, la médiane et le maximum est donné par la relation suivante :

$$N_{atome} = \frac{N^2-1}{4} + \frac{3}{4} \cdot \left(\frac{N+1}{2}\right)^2 \text{avec N impair}$$

Les tris fusion sont réalisés de façon empirique en se basant, pour chacune des tailles croissantes de jeu de données, sur les tris de tailles moindres réalisés à l'étape précédente. De plus, certaines structures peuvent être optimales pour un tri complet des données, mais sous-optimale pour un tri fusion élagué. Un exemple est donné en figure 2.24 où l'on considère deux unités de tri de 7 éléments. La première structure consomme seulement 18 atomes de tri alors que la seconde en utilise 19, pourtant cette dernière après élagage utilise moins d'atomes de tri que la première. L'élaboration de tris fusions optimaux est donc un problème complexe et où les structures sont de moins en moins triviales lorsque le nombre de données augmente.

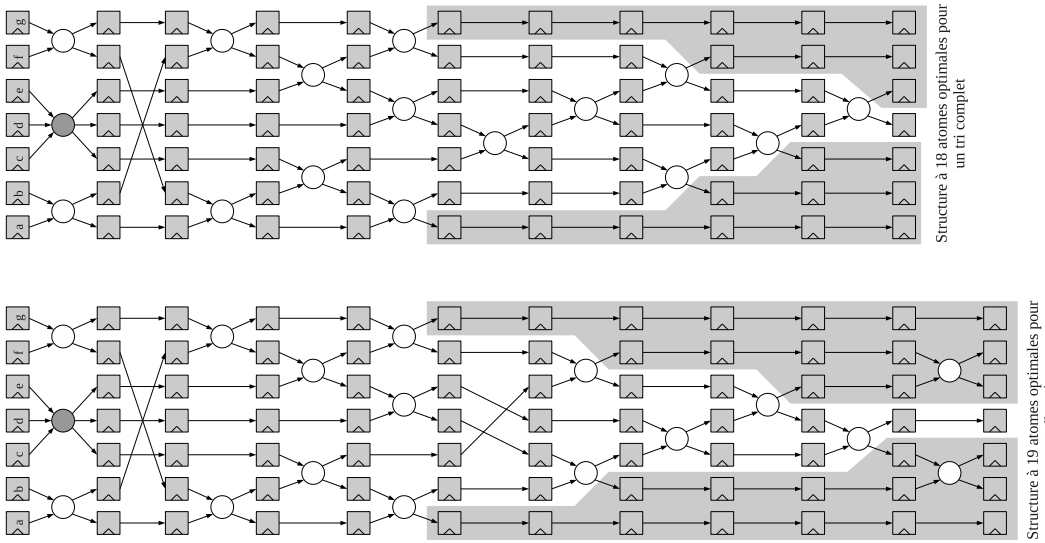


FIG. 2.24: Différentes architectures de tri fusion de 7 valeurs, optimales pour un tri complet ou pour un tri élagué

Seconde optimisation Si un sens de parcours est connu, il est possible de simplifier l'unité de tri comme nous avons pu le faire pour l'arbre de calcul d'une érosion ou d'une dilatation. Le principe reste le même, l'élément structurant se déplace par exemple dans le sens vidéo (sens direct), nous observons ainsi que le tri réalisé sur une colonne du voisinage à l'instant t est identique à celui de la colonne voisine à l'instant $t+1$. Cette optimisation ne fonctionne efficacement que s'il est possible de trier indépendamment les colonnes du voisinage comme dans le cas du tri fusion présenté en figure 2.20. D'une manière générale, pour un élément structurant rectangulaire composé de I lignes et J colonnes, il faut être capable de diviser le premier étage du tri en J tris de I éléments comme le montre la figure 2.25.

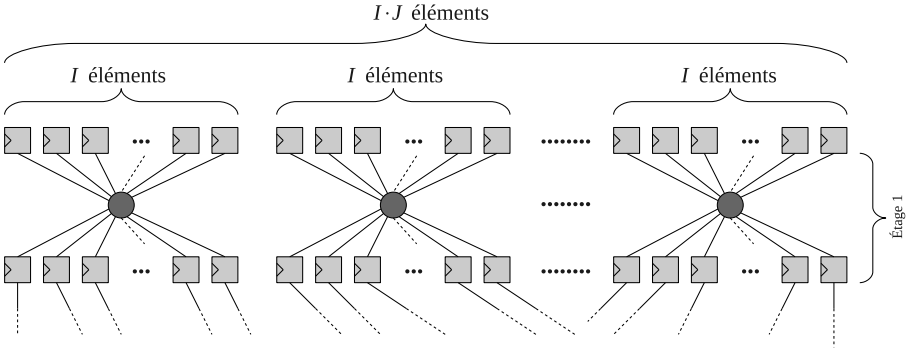


FIG. 2.25: Architecture de tri fusion de $I\dot{J}$ valeurs

Le tri obtenu n'est plus optimal dès que l'on dépasse le voisinage 3×3 puisque les sous-ensembles à trier doivent avoir une taille I, alors qu'il aurait fallu pousser le découpage plus loin. Par exemple, une unité de tri fusion élaguée fournissant uniquement la médiane d'un voisinage carré 5×5, avec un découpage du premier étage du tri en cinq sous-tris, utilise 112

atomes (figure 2.26). La version la plus optimale, en considérant environ 8 sous-ensembles de taille 3 dans le premier étage du tri, utilise quant à elle 99 atomes de tri [70].

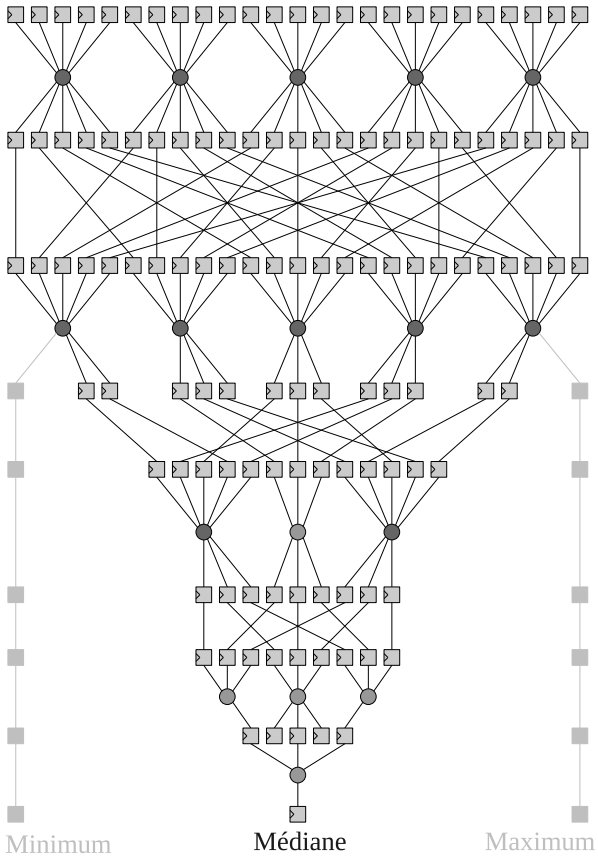


FIG. 2.26: Architecture de tri fusion élagué de 25 valeurs

Bien que non optimal cette structure à l'avantage de travailler sur les colonnes du voisinage permettant de conserver une seule unité de tri dans le premier étage de l'unité de calcul. Les unités de tri économisées sont simplement remplacées par des registres à décalage. Une telle structure renvoyant le minimum, la médiane et le maximum et considérant voisinage 3×3 est présentée en figure 2.27.

Le même problème sur les bords survient, il faut déporter une partie de la gestion de ces derniers dans l'unité de calcul, c'est le rôle du composant "GESTION BORDS" qui n'est rien d'autre que trois multiplexeurs recopiant l'entrée ou la valeur du *padding* sur la sortie grâce aux signaux "b1" ou "b2". Le *padding* étant un damier, il est nécessaire que le composant de gestion des bords ait une information à propos de la parité de la ligne afin de choisir la bonne séquence de pixels noir/blanc. Cette information est transmise via le signal "P" du bloc de gestion des bords.

L'économie d'atomes de tri est assez substantielle et permet par exemple de construire une unité de tri travaillant sur un voisinage 3×3 et produisant le minimum, le maximum et la médiane, avec seulement 29% des ressources d'un tri à bulle produisant les mêmes résultats. Toutes ces informations sont regroupées dans le tableau 2.2.

Les filtres de rang regroupent un grand nombre d'opérations dont les structures de calcul

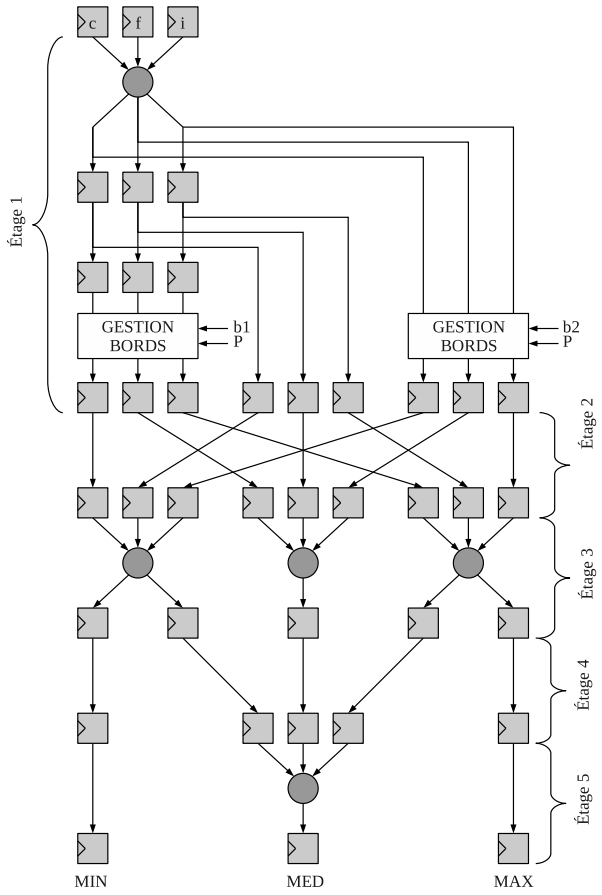


FIG. 2.27: Architecture de tri fusion élagué de 9 valeurs réutilisant les tris des colonnes des voisinages précédents

Nb. Elt. \ Type	Tri Bulle		Tri Fusion			
	Min Max Médian	Médian seul	Min Max Médian	Médian seul	Min Max Médian [a]	Médian seul [a]
9 éléments	31	26	18	15	11	9
	100%	84%	58%	48%	35%	29%
25 éléments	261	222	113	112	77	76
	100%	85%	43%	43%	30%	29%

[a]Structure réutilisant les tris déjà réalisés sur les colonnes voisines.

TAB. 2.2: Tableau récapitulatif des différentes tailles des unités de tri optimisées

peuvent beaucoup varier. Le tri des pixels, même avec plusieurs niveaux d'optimisation, est une opération coûteuse et si l'on considère un pipeline de processeurs de voisinage il faut absolument éviter de l'utiliser comme unité de calcul dans tous les processeurs. En effet, la surface de l'architecture risque d'être très importante et l'utilité n'est pas garantie, car décomposer un filtre de rang (autre que certaines érosions ou dilatations) n'est pas possible en cascadant plusieurs opérations de tailles réduites. De plus, parmi les filtres de rang, le plus utilisé est sans conteste le filtre médian. Ce dernier est souvent mis en œuvre dans une première phase de débruitage ce qui implique de mettre une unité de tri uniquement dans le premier étage d'un flot de processeurs de voisinage, les étages suivants utiliseront par exemple des arbres de calculs du minimum et/ou du maximum.

2.2.5.3 Opérateurs gradients

Les opérations résiduelles sont une famille d'opérateurs important en morphologie mathématique et permettent pour les plus simples de construire différents types de gradients basés sur des opérations d'érosion et/ou de dilatation. Nous définissons trois gradients :

- le gradient intérieur : $g^- = f - \varepsilon(f)$
- le gradient supérieur : $g^+ = \delta(f) - f$
- le gradient épais : $g = g^+ + g^- = \delta(f) - \varepsilon(f)$.

Cette notion d'*épaisseur* du gradient se comprend bien avec des images binaires : le gradient intérieur est inscrit dans les zones blanches de l'image tandis que le gradient supérieur est à la frontière extérieure, le gradient épais est, quant à lui, l'union du gradient intérieur et supérieur.

Avec de telles structures, il est facile de prévoir dans l'unité de calcul d'un filtre de rang un opérateur de soustraction permettant de produire un gradient. En revanche, pour des opérations résiduelles plus complexes, il faudra utiliser plusieurs processeurs de voisinage chaînés en série ou en parallèle via des unités arithmétiques et logiques (ALU). La figure 2.28 propose une architecture basée sur les unités de tri des filtres de rang permettant de produire plusieurs types de gradients morphologiques : le gradient intérieur g^-, le gradient extérieur g^+ et le gradient épais g. Ce dernier présente un bord erroné, car même si le tri des données est en mesure de nous fournir à la fois l'érosion et la dilatation, la politique des bords émulés autour de l'image est mise en place seulement pour un rang précis alors que nous utilisons simultanément deux valeurs en sortie de tri.

Il est alors préférable de calculer le gradient intérieur ou extérieur à partir d'un arbre de recherche du minimum ou du maximum et de calculer un gradient épais en utilisant deux processeurs de voisinage en parallèle (nous approfondirons ce dernier point dans le chapitre 3).

2.2.5.4 Érosion et dilatation géodésiques

Quelques notions sur la géodésie Les transformations géodésiques [7] prennent en entrées deux images, une appelée marqueur qui subira une transformation et une autre appelée masque qui permettra de contraindre le marqueur à un certain niveau. En pratique, il n'a pas de problème de choix d'une taille d'un élément structurant puisque ces transformations sont opérées jusqu'à stabilité, ou idempotence, du marqueur.

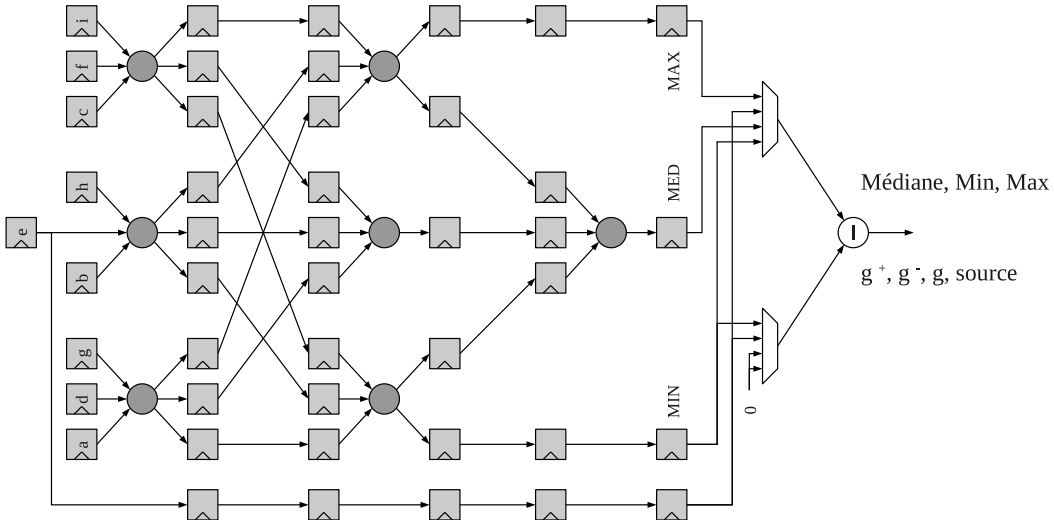


FIG. 2.28: Calcul de gradients basés sur une unité de tri

On définit une érosion géodésique comme étant l'érosion du marqueur f avec un élément structurant de rayon 1, suivie d'un infimum avec le masque g :

$$\varepsilon_g^{(1)}(f) = \varepsilon^{(1)}(f) \vee g \text{ et } f \geq g$$

De manière duale, la dilatation géodésique est définie de la façon suivante :

$$\delta_g^{(1)}(f) = \delta^{(1)}(f) \wedge g \text{ et } f \leq g$$

La dualité entre l'érosion géodésique et la dilatation géodésique se vérifie ainsi :

$$\begin{aligned} \varepsilon_g^{(1)}(f) &= [\delta^{(1)}(f^c) \wedge g^c]^c \\ &= [(\varepsilon^{(1)}(f))^c \wedge g^c]^c \\ &= \varepsilon^{(1)}(f) \vee g \end{aligned}$$

On peut également définir une transformation géodésique autoduale qui considère une dilatation géodésique lorsque $f(p) \leq g(p)$ et une érosion géodésique lorsque $f(p) > g(p)$:

$$[\nu_g^{(1)}(f)](p) = \begin{cases} [\delta_g^{(1)}(f)](p), & \text{si } f(p) \leq g(p) \\ [\varepsilon_g^{(1)}(f)](p), & \text{sinon} \end{cases}$$

La reconstruction géodésique R_g se définit comme étant la répétition d'une dilatation géodésique ou d'une érosion géodésique jusqu'à idempotence. On distingue alors :

– la reconstruction par dilatation :

$$R_g^{\delta}(f) = \delta_g^{(i)}(f) = \delta_g^{(1)}[\delta_g^{(i-1)}(f)]$$

– la reconstruction par érosion :

$$R_g^{\varepsilon}(f) = \varepsilon_g^{(i)}(f) = \varepsilon_g^{(1)}[\varepsilon_g^{(i-1)}(f)]$$

– la reconstruction autoduale ou un cas particulier des nivellements [51], [54] :

$$R_g^{\nu}(f) = \nu_g^{(i)}(f) = \nu_g^{(1)}[\nu_g^{(i-1)}(f)]$$

L'idempotence est vérifiée par l'égalité suivante :

$$\phi_g^{(i)}(f) = \phi_g^{(i+1)}(f), \text{ avec } \phi = \varepsilon,\ \phi = \delta \text{ ou } \phi = \nu$$

Les figures 2.29.a, 2.29.b, 2.29.c représentent respectivement une reconstruction par dilatation, une reconstruction par érosion et un nivellement (reconstruction autoduale). Dans un contexte monodimensionnel, le nombre d'étapes de reconstruction est assez faible, mais dans un cas bidimensionnel, ce nombre peut fortement croître. Il est alors intéressant d'envisager des approches récursives qui permettent de conserver une architecture de processeur de voisinage tout en accélérant les calculs.

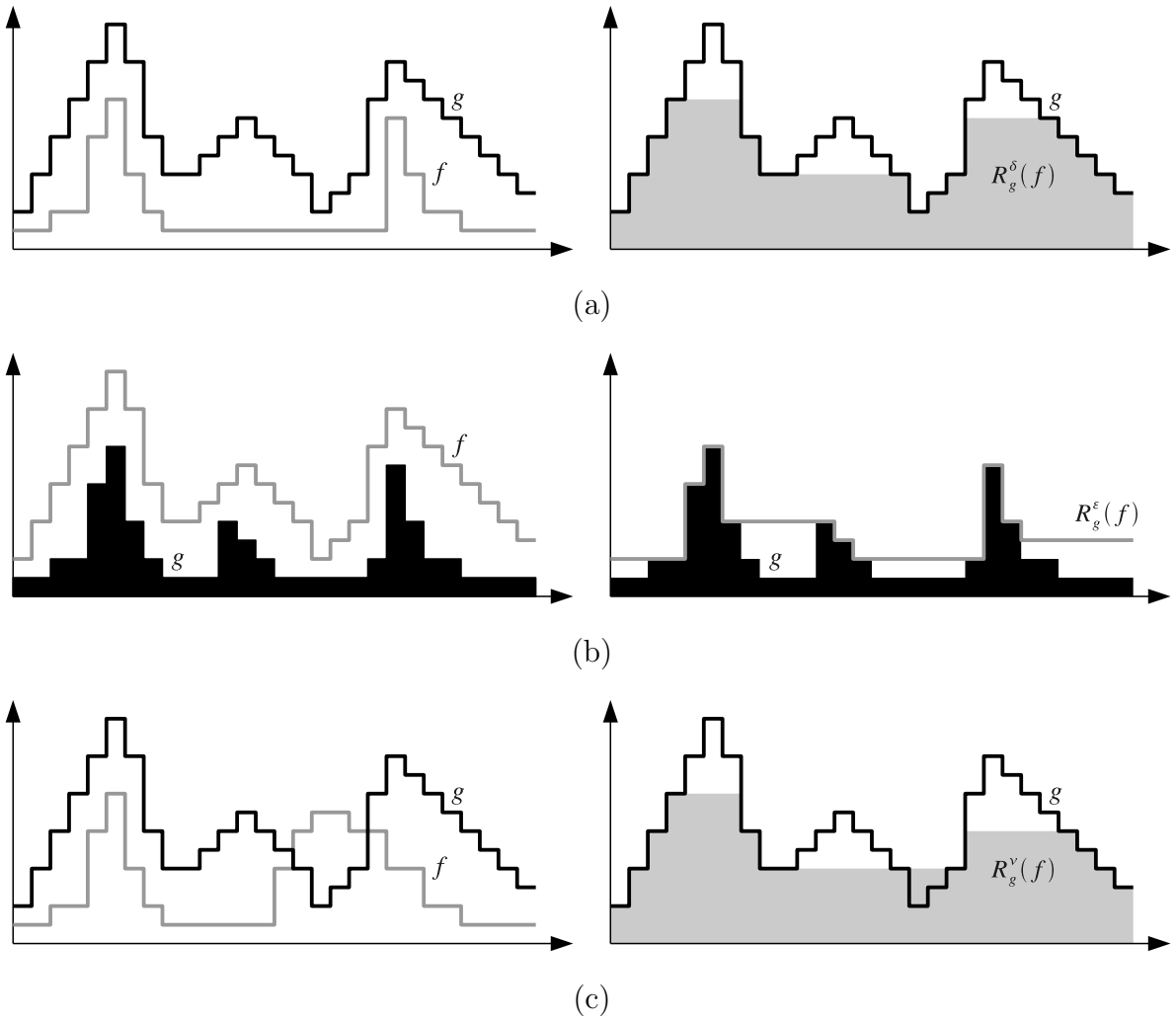
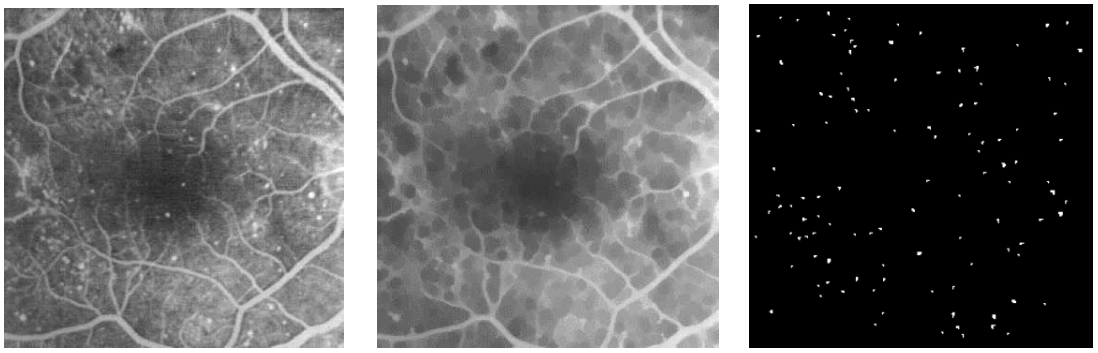


FIG. 2.29: Exemples de reconstructions géodésiques monodimensionnel

Les reconstructions géodésiques ont de nombreuses applications telles que dans la recherche de maxima/minima régionaux, dans l'amélioration du contraste ou bien encore

dans le domaine médical avec la détection de microanévrismes dans des images rétiniennes (F. Zana et J.C. Klein) comme le montre la figure 2.30. Les nivellements sont principalement utilisés pour simplifier une image en créant des zones plates sans supprimer les contours présents dans l'image.



(a) Image originale du fond de l'oeil

(b) Ouverture géodésique $(R_g^{\delta}(\varepsilon_B(f)))$ suivi d'une fermeture géodésique $(R_g^{\varepsilon}(\delta_B(f)))$

(c) Microanévrisme détécté par soustraction de l'image reconstruite avec l'image originale

FIG. 2.30: Utilisation des reconstructions géodésiques dans le cadre de la détection de microanévrismes

Aspects architecturaux La reconstruction géodésique est une opération complexe de propagation jusqu'à idempotence et repose donc sur une itération d'opérations simples avec un élément structurant de rayon unitaire. Cette décomposition est propice à l'utilisation d'un processeur de voisinage embarquant une unité de calcul chargée de calculer une dilatation/érosion de l'image marqueur et de comparer le résultat via une opération de minimum/maximum avec l'image masque.

Une telle unité de calcul n'est pas sans impact sur la structure d'extraction du voisinage des processeurs flots de données. En effet, il est nécessaire d'acheminer à l'unité de calcul, chargée des opérations géodésiques, les pixels de mêmes indices du masque et du marqueur.

La latence importante des pixels, entre le moment où ils rentrent dans le processeur et le moment où ils sont pris en charge dans l'unité de calcul, nous oblige à retarder d'autant les pixels du masque pour qu'ils soient acheminés vers l'unité de calcul en même temps que les pixels du marqueur.

Deux stratégies s'offrent à nous, on peut dans un premier temps ajouter une seconde entrée au processeur de voisinage afin de traiter le problème de l'acheminement synchrone de deux images vers l'unité de calcul (figure 2.31). Cette solution a pour effet de doubler la largeur des mémoires des lignes à retard. On peut, dans un second temps, mettre en place deux processeurs de voisinage en parallèle, disposant chacun d'une seule entrée image et dont les sorties de calculs sont dirigées vers une ALU réalisant un minimum. L'un des processeurs réalise l'érosion et le second ne réalise aucun calcul (par exemple en spécifiant un élément structurant composé uniquement du pixel central). Un exemple de cette dernière solution est présenté en figure 2.32 et la généralisation du concept est présentée au chapitre 3. Nous montrerons dans ce chapitre comment décrire un grand nombre des opérations utiles aux morphologues en utilisant un pipeline de processeurs de voisinage.

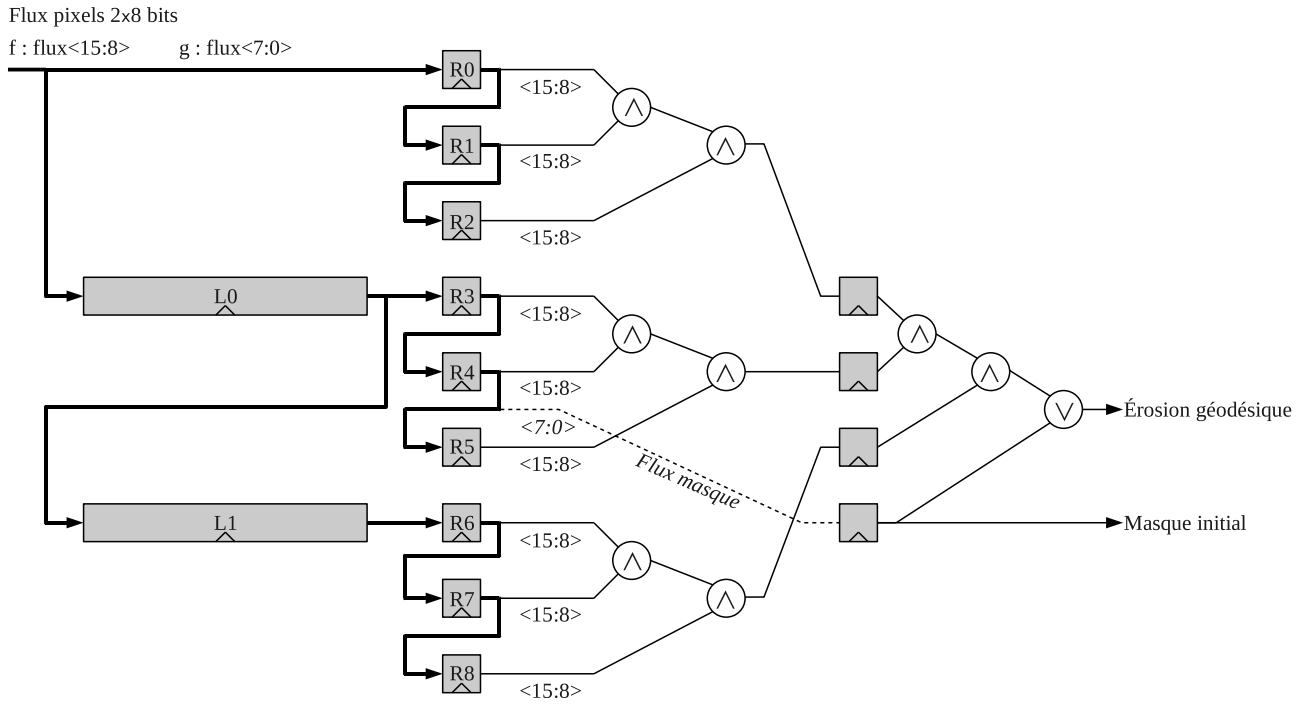


FIG. 2.31: Architecture avancée d'un processeur de voisinage à deux entrées, dédiée au calcul de l'érodée géodésique

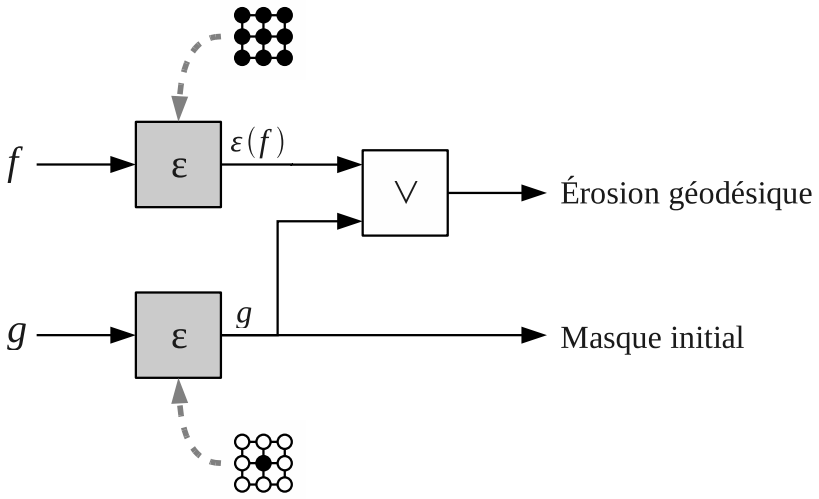


FIG. 2.32: Architecture à deux processeurs de voisinage simple, dédié au calcul de l'érodée géodésique

Le chaînage de ces architectures dans un pipeline permet alors de réaliser plusieurs itérations d'une reconstruction géodésique en une passe. On comprend alors bien l'intérêt de disposer d'un pipeline de processeur de voisinage afin d'exploiter au maximum le parallélisme temporel des opérations géodésiques. Toutefois, le nombre de passes pour atteindre l'idempotence est inconnu à priori et en pratique ce chiffre est extrêmement variable. L'emploi de processeurs de voisinage *récursif*, présentés en section 2.3, permet de minimiser ce nombre, mais supprime l'intérêt de disposer d'un pipeline de processeur. Il existe donc un compromis entre utiliser un pipeline profond dédié à la géodésie ou utiliser un seul processeur de voisinage récursif. Ce type de compromis est abordé dans le chapitre 3.

2.2.5.5 Opérateurs tout ou rien

L'opérateur de transformation tout ou rien est principalement utilisé sur des images binaires et permet de mettre en évidence certaines configurations des pixels. On définit une configuration par l'utilisation de deux éléments structurants, un pour le fond et un pour la forme. Si le voisinage extrait de l'image correspond exactement à la configuration établie, on renvoie alors 1 et dans le cas contraire 0. On peut définir cette transformation, pour une configuration B, comme étant l'intersection de l'érosion ε de l'image en considérant l'élément structurant de la forme avec la dilatation δ de l'image inversée considérant l'élément structurant du fond.

$$\eta_B(X) = \varepsilon_{B_{\text{forme}}}(X) \cap \delta_{B_{\text{fond}}}(X^c)$$

À partir de cette définition, on peut construire deux autres transformations appelées amincissement et épaississement. L'amincissement θ d'un ensemble X consiste à enlever des points dont le voisinage correspond à une configuration donnée. Il s'agit en fait du résidu morphologique entre l'image initiale et la transformation tout ou rien correspondant à la configuration B.

$$\theta(X) = X - \eta_B(X)$$

L'épaississement ξ consiste à ajouter des points dont le voisinage correspond à la configuration B.

$$\xi(X) = X \cup \eta_B(X)$$

L'architecture de l'unité de calcul de la transformation tout ou rien est présentée en figure 2.33. Le premier étage permet de contrôler si le pixel du voisinage est dans le fond, dans la forme ou bien s'il ne doit pas être pris en compte. Le signal Sx0 permet de définir un voisin comme étant dans le fond (0) ou dans la forme (1). Le signal Sx1 permet quant à lui de ne pas prendre en compte, lorsqu'il est actif, un voisin dans le calcul en forçant la sortie du premier étage à 1. En effet, le second étage est composé d'un arbre de ET logique, il faut donc forcer à 1 les voisins ne devant pas rentrer dans le calcul ou étant sur un bord de l'image pour qu'ils ne soient pas pris en compte. La désélection des voisins hors de l'image se fait aussi dans l'unité de calcul en actionnant le signal ACT_x. L'utilisateur a donc le libre choix de définir toutes les configurations de voisinage qu'il désire en trame carrée ou hexagonale.

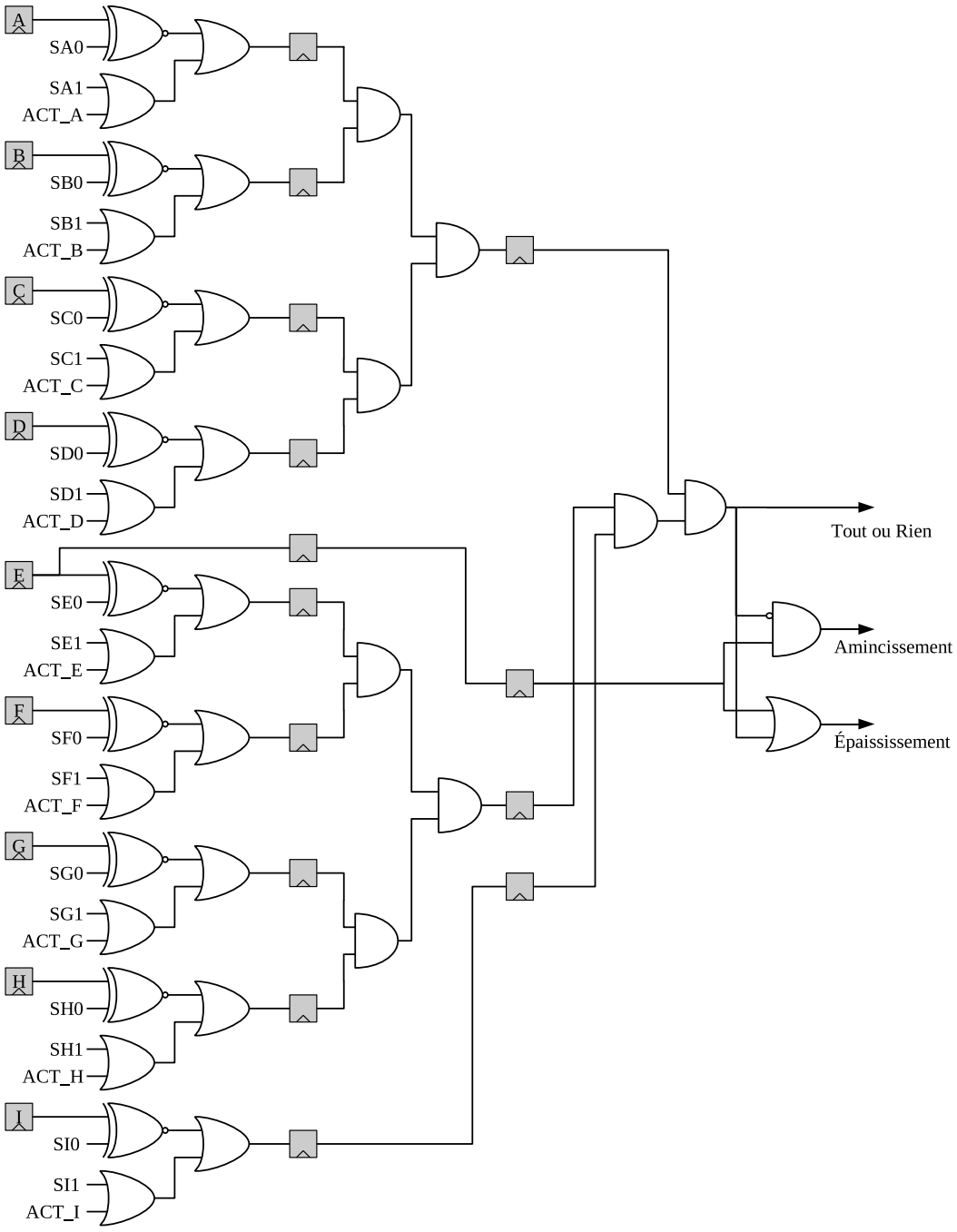


FIG. 2.33: Unité de calcul des opérateurs Tout ou Rien, d'amincissement et d'épaississement

2.2.5.6 Opérateurs linéaires

Cette unité de calcul a pour but de réaliser de simples convolutions avec de petits noyaux afin de débruiter une image ou de calculer un filtre de Sobel. Elle est présentée en figure 2.34 et est composée dans le premier étage de multiplieurs pipelines. Ces derniers multiplient la fenêtre extraite par les coefficients du filtre. Afin d'économiser des ressources sur le circuit, il est envisageable de coder les coefficients avec un nombre de bits plus réduit que celui employé pour le codage des pixels. L'arbre d'opérateurs d'addition dans les étages suivants somme les sorties des multiplieurs et à chaque étage le nombre de bits nécessaire pour le codage s'incrémente. C'est la raison pour laquelle il est judicieux de ne pas utiliser trop de bits pour le codage des coefficients du filtre. Enfin, le dernier étage permet la normalisation de la sortie en utilisant un diviseur ou un multiplieur à virgule fixe.

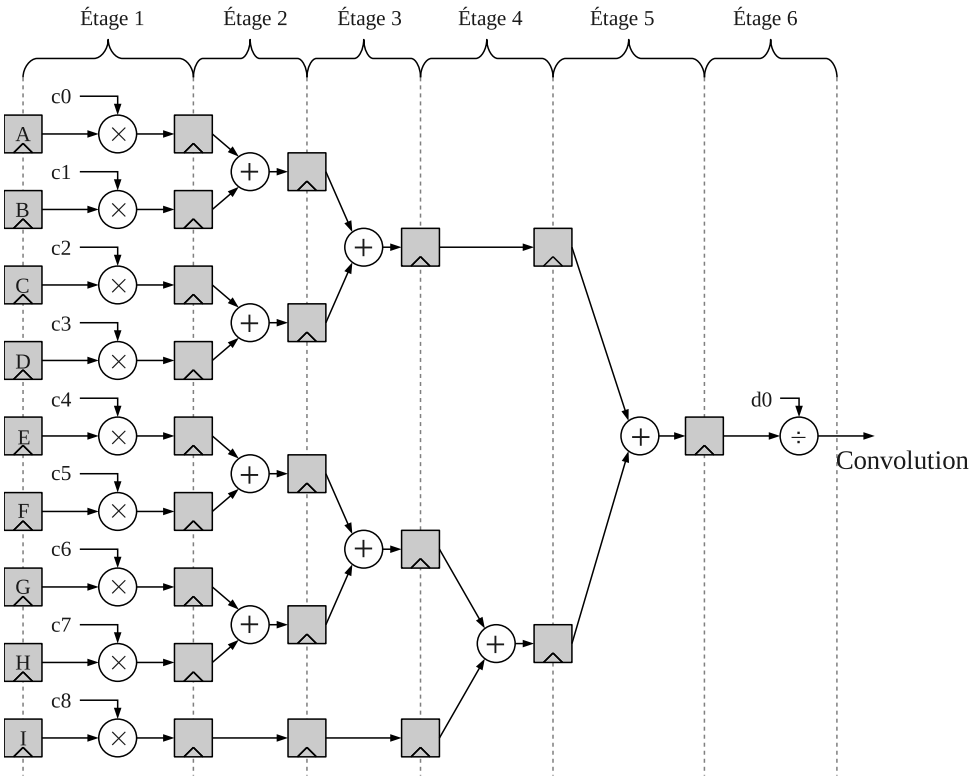


FIG. 2.34: Unité de calcul de convolutions

2.2.6 Nouveaux noyaux de calcul

2.2.6.1 Érosion et dilatation microvisqueuses

Les opérations microvisqueuses ont été introduites récemment en morphologie mathématique par Angulo et Meyer[55] et sont à la base d'une nouvelle catégorie d'opérateur de nivellement. En effet, les nivellements définis à la section 2.2.5.4 ne permettent pas toujours de créer suffisamment de zones plates dans une image, car ils reconstruisent trop de détail provenant de l'image masque. Ce genre de problème engendre souvent des segmentations

d'images comportant un nombre de labels trop important. La solution proposée par Angulo et Meyer consiste à remplacer dans la reconstruction géodésique la définition standard des érosions et dilatations par des définitions équivalentes *moins fortes*. En outre, ces érosions et dilatations microvisqueuses peuvent être employées lorsqu'il est nécessaire de travailler sur des images bruitées sans étapes de restauration préalable.

Nous proposons donc dans cette section une mise en œuvre matérielle de ces érosions/dilatations en couplant à une unité d'extraction du voisinage, une unité de calcul dédiée. Le principe est de considérer le graphe d'un voisinage et de travailler avec les arêtes plutôt qu'avec les noeuds comme on le fait, par exemple, avec une simple érosion. Les arêtes sont calculées grâce à un supremum ou infimum et l'on cherche ensuite l'infimum ou le supremum des arêtes avec le pixel central. Les éléments structurants que nous considérons pour cette unité de calcul sont décrits en figure 2.35.

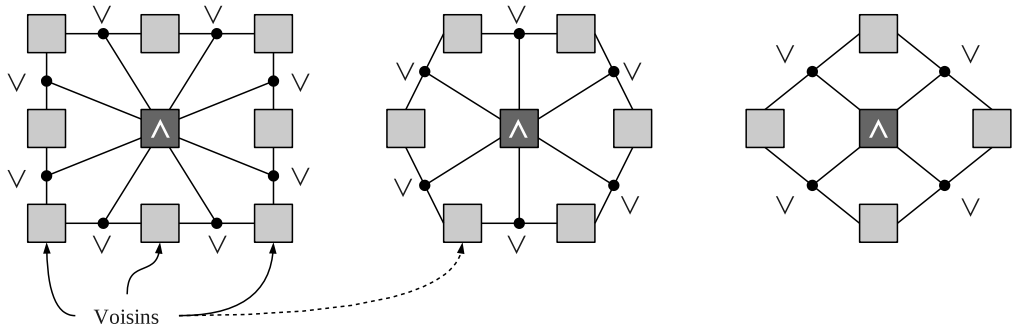


FIG. 2.35: Éléments structurants considérés pour une érosion microvisqueuse

La gestion des bords est complexe et l'on ne peut pas simplement remplacer un voisin hors de l'image par une valeur ne perturbant pas le calcul. En effet, le fait d'être sur un bord modifie le graphe du voisinage et il faut donc reconsidérer tout l'arbre de calcul. Il faut également bien garder à l'esprit que la trame hexagonale doit être simulée dans la maille carrée, et le fait de vouloir la gérer comme nous avons pu le faire précédemment n'est pas possible. C'est-à-dire que le fait de ne pas tenir compte de certains voisins modifie aussi le graphe du voisinage. L'unité de calcul va donc devoir calculer tous les sous graphes utiles à la composition des voisinages décrits précédemment comme le montre la figure 2.36. De plus, cette liste de sous-graphes nous permet également de prendre en compte la modification du graphe du voisinage lors que ce dernier est sur un bord de l'image.

L'unité de calcul se compose donc d'un premier niveau dans lequel tous les sous graphes sont calculés, et d'un second niveau plus conventionnel où l'on calcule l'infimum ou le suprémum des arêtes que l'on aura choisies. Ce choix est matérialisé par l'usage de multiplexeurs juste avant le second niveau comme le montre la figure 2.37.

La commande judicieuse de chacun des multiplexeurs permet à la fois de définir un élément structurant, mais également toutes les déclinaisons associées au traitement des bords Nord (N), Sud (S), Est(E), Ouest(O), NE, NO, SE, SO. Les tableaux 2.3, 2.4 présentent les tables de vérité pour appliquer un opérateur microvisqueux avec un carré ou une croix de rayon 1 en tenant compte des bords. Comme toujours, la maille hexagonale dispose de deux tables de vérité, une pour les commandes des multiplexeurs lorsque l'on travaille sur les lignes paires, tableau 2.5, et une autre pour les lignes impaires, tableau 2.6.

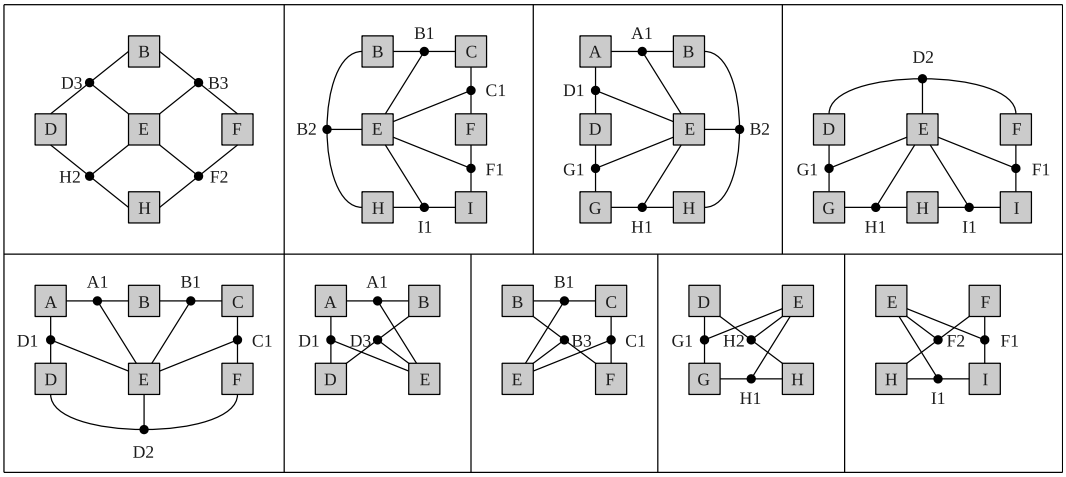


FIG. 2.36: Sous graphes nécessaires à la construction de voisinages microvisqueux complexes

FIG. 2.37: Unité de calcul de l'érosion microvisqueuse pour un voisinage 8, 6 ou 4 connexe

Pos. Vois. / Mux	N	S	E	O	NE	NO	SE	SO	Ailleurs
X1	D2	A1	A1	Pad	Pad	Pad	Pad	A1	A1
X2	Pad	B1	B2	B1	Pad	Pad	B1	Pad	B1
X3	Pad	C1	Pad	C1	Pad	Pad	C1	D3	C1
X4	F1	D2	Pad	F1	Pad	F1	Pad	Pad	F1
X5	I1	Pad	Pad	I1	H2	I1	Pad	Pad	I1
X6	H1	Pad	H1	B2	H1	Pad	Pad	Pad	H1
X7	G1	Pad	G1	Pad	G1	Pad	B3	Pad	G1
X8	Pad	D1	D1	Pad	Pad	F2	Pad	D1	D1

TAB. 2.3: Table de vérité pour le calcul des différents graphes de voisinages en 8-connexité

Pos. Vois. / Mux	N	S	E	O	NE	NO	SE	SO	Ailleurs
X1	D2	D2	D3	Pad	Pad	Pad	Pad	Pad	D3
X2	Pad	Pad	B2	B2	Pad	Pad	Pad	Pad	Pad
X3	Pad	D3	Pad	B3	Pad	Pad	D3	B3	B3
X4	Pad	Pad	Pad	Pad	Pad	Pad	Pad	Pad	Pad
X5	H2	Pad	H2	F2	H2	F2	Pad	Pad	H2
X6	Pad	Pad	Pad	Pad	Pad	Pad	Pad	Pad	Pad
X7	Pad	B3	Pad	Pad	Pad	Pad	Pad	Pad	Pad
X8	F2	Pad	Pad	Pad	Pad	Pad	Pad	Pad	F2

TAB. 2.4: Table de vérité pour le calcul des différents graphes de voisinages en 4-connexité

Pos. Vois. / Mux	N	S	E	O	NE	NO	SE	SO	Ailleurs
X1	D2	D3	D3	Pad	Pad	Pad	Pad	A1	D3
X2	Pad	B1	B2	B1	Pad	Pad	Pad	Pad	B1
X3	Pad	C1	Pad	C1	Pad	Pad	D3	D3	C1
X4	F1	D2	Pad	F1	Pad	F1	Pad	Pad	F1
X5	I1	Pad	Pad	I1	H2	I1	Pad	Pad	I1
X6	H2	Pad	H2	B2	Pad	Pad	Pad	Pad	H2
X7	Pad	Pad	Pad	Pad	Pad	Pad	Pad	Pad	Pad
X8	Pad	Pad	Pad	Pad	Pad	F2	Pad	D1	Pad

TAB. 2.5: Table de vérité pour le calcul des différents graphes de voisinages en 6-connexité pour les lignes paires

Pos. Vois. / Mux	N	S	E	O	NE	NO	SE	SO	Ailleurs
X1	D2	A1	A1	Pad	Pad	Pad	Pad	Pad	A1
X2	Pad	Pad	B2	B2	Pad	Pad	B1	Pad	Pad
X3	Pad	B3	Pad	B3	Pad	Pad	C1	B3	B3
X4	Pad	D2	Pad	Pad	Pad	Pad	Pad	Pad	Pad
X5	F2	Pad	Pad	F2	H2	F2	Pad	Pad	F2
X6	H1	Pad	H1	Pad	H1	Pad	Pad	Pad	H1
X7	G1	Pad	G1	Pad	G1	Pad	B3	Pad	G1
X8	Pad	D1	D1	Pad	Pad	Pad	Pad	Pad	D1

TAB. 2.6: Table de vérité pour le calcul des différents graphes de voisinages en 6-connexité pour les lignes impaires

2.2.6.2 Opérateur dédié au SKIZ isotrope

L'opérateur SKIZ, ou squelette par zone d'influence, est un opérateur permettant de calculer la frontière entre les différentes zones d'influence d'une image binaire.

Ce squelette est traditionnellement obtenu par des amincissements successifs de l'image binaire inversée en considérant une configuration de type L, définie dans l'alphabet de Golay [69], ainsi que toutes les rotations associées (8 en maille carrée, 6 en maille hexagonale, etc.). Le tableau 2.7 présente les principales configurations, aux rotations prêtes, de l'alphabet de Golay pour la maille hexagonale et carrée.

On définit un amincissement séquentiel d'une image binaire X en considérant n amincissements successifs avec les n rotations discrètes d'un élément structurant B :

$$X\ \theta_n\ B = (\cdots((X\ \theta\ B_1)\ \theta\ B_2)\cdots)\ \theta\ B_n$$

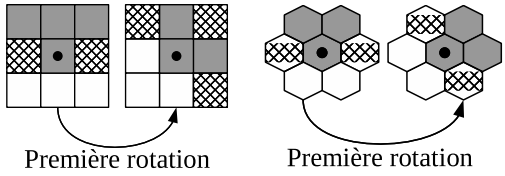
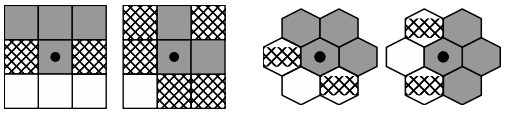
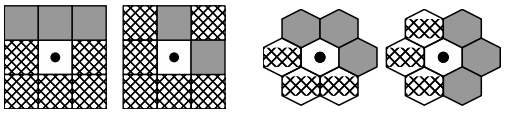
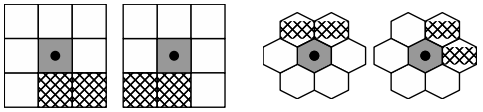
Un squelette est obtenu en répétant jusqu'à idempotence un amincissement séquentiel θ_n sur une image X avec l'élément L de l'alphabet de Golay :

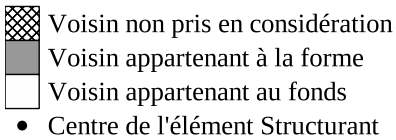
$$SK_L(X) = (X\ \theta_n\ L)^{\infty}$$

Il est également possible de construire le SKIZ en réalisant des dilatations successives de l'image binaire labellisée où l'on stoppe la dilatation dès que deux labels se chevauchent [10]. L'intérêt de cette transformation est de pouvoir réaliser une étape de la transformation dans toutes les directions avec un seul balayage de l'image. C'est la raison pour laquelle cette opération est qualifiée d'isotrope contrairement au SKIZ par amincissements où, pour réaliser une itération équivalente du SKIZ isotrope, des amincissements successifs avec les n rotations d'un motif de Golay sont nécessaires.

L'architecture de cette unité de calcul est assez simple puisqu'il suffit de reprendre l'arbre de calcul d'une dilatation et de conditionner le résultat par un critère nous informant si un autre label, différent du résultat de la dilatation, est présent dans le voisinage. Si tel est le cas on renvoie alors 0 et dans le cas contraire, on renvoie le résultat de la dilatation.

En itérant jusqu'à stabilité ce calcul, on obtient le squelette par zone d'influence d'une image binaire. Un exemple est présenté en figure 2.38. Plus de détails concernant la mise en œuvre de cet opérateur de manière logicielle sont donnés dans le chapitre 4.

	Configuration du voisinage B	Amincissement avec B	Épaississement avec B^c	*Hit-or-Miss*
L		Squelette homotopique de la forme	Squelette homotopique du fond	–
M		Squelette homotopique de la forme	Squelette homotopique du fond	–
C			Enveloppe convexe	–
E		Ébarbulage de forme	Ébarbulage du fond	Points terminaux du squelette
I		Supprime les pixels isolés	Bouche les trous d'un pixel	Points isolés



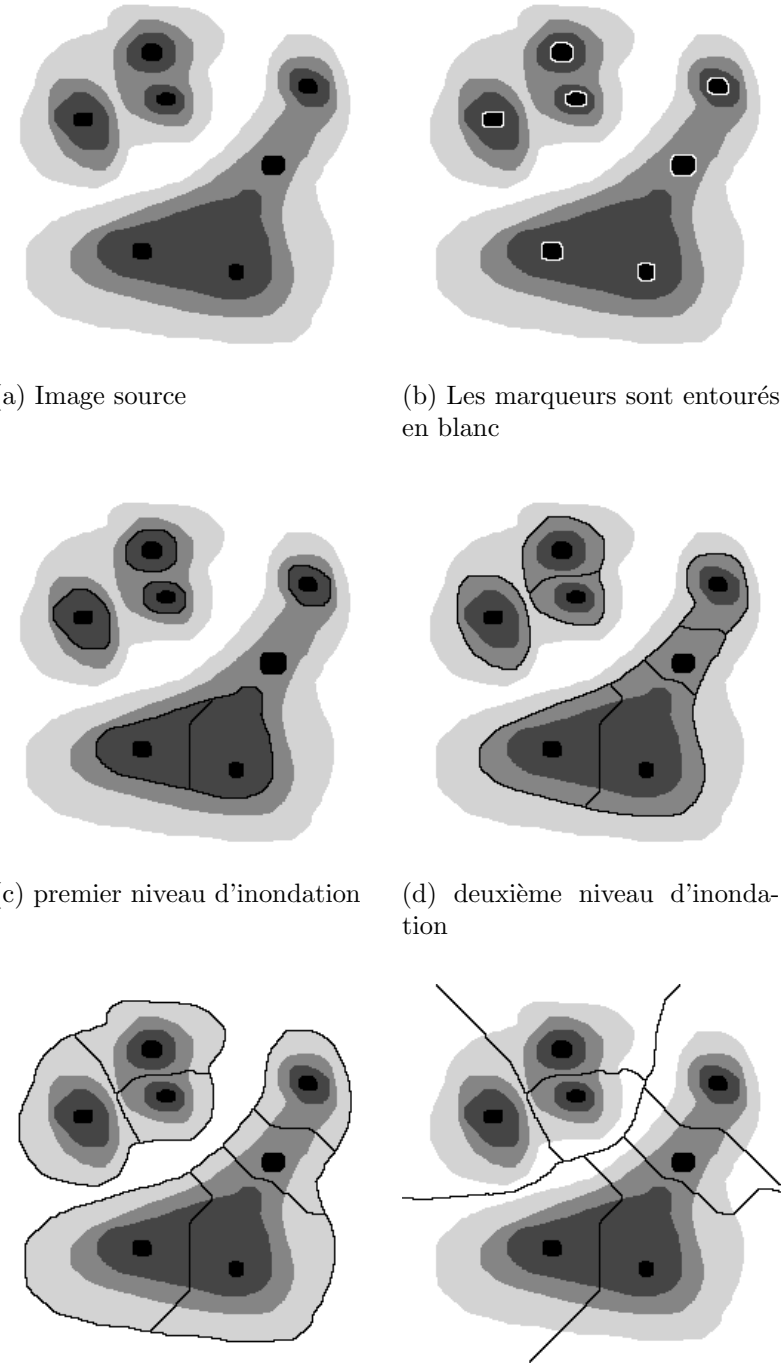
TAB. 2.7: Alphabet de Golay : configuration du voisinage pour la maille carrée et hexagonale

(a) Image binaire labélisée

(b) Résultat du SKIZ

(c) Superposition du résultat sur l'image binaire

FIG. 2.38: SKIZ d'une image labelisée



(a) Image source

(b) Les marqueurs sont entourés en blanc

(c) premier niveau d'inondation

(d) deuxième niveau d'inondation

(e) troisième niveau d'inondation

(f) dernier niveau d'inondation

Les lignes noires représentent la ligne de partage des eaux

FIG. 2.39: Principe d'inondation avec le SKIZ isotrope pour le calcul de la LPE

L'intérêt de cette transformation est de pouvoir réaliser une ligne de partage des eaux (LPE) [9] par inondation en itérant niveau de gris par niveau de gris un SKIZ isotrope. La figure 2.39 montre sur un exemple simple le fonctionnement de la LPE par inondation. On inonde *en faisant jaillir de l'eau* à partir des marqueurs 2.39.b et progressivement, le niveau de l'eau s'élève. On érige au fur et à mesure un *barrage* entre les différentes eaux pour éviter qu'elles ne se mélangent. Une fois l'inondation terminée, figures 2.39.e, les barrages représentent la ligne de partage des eaux.

La LPE peut être calculée efficacement sur un processeur généraliste par l'utilisation d'une file d'attente hiérarchique [53][10]. Une mise en œuvre matérielle [41] d'une telle structure conduit vers une architecture très spécifique et difficilement employable pour le calcul d'autres opérations morphologiques. Notre approche n'est pas de fournir une architecture dédiée à un calcul qui ne représente qu'une étape d'une application, mais plutôt de donner le moyen au morphologue de décrire toute son application via un pipeline de processeurs de voisinage. La structuration d'un tel pipeline est discutée au chapitre 3, mais nous pouvons d'ores et déjà comprendre l'intérêt d'une telle approche. En effet en sélectionnant soigneusement les différents noyaux de calcul à intégrer aux processeurs composant le pipeline, la majeure partie des opérations de morphologie mathématique peuvent être réalisées efficacement.

2.3 Structure récursive

2.3.1 Fonctionnement

Une autre catégorie d'opérateurs récursifs existe, ayant pour objectif non plus de composer de très gros éléments structurants, mais plutôt d'accélérer les calculs en diminuant le nombre de passes nécessaires sur une image pour calculer, par exemple, une reconstruction géodésique (présentée à la section 2.2.5.4) où encore une transformation distance. Le principe est d'appliquer une récursion dans les calculs en réinjectant dans l'image source les résultats provenant de calculs sur les voisinages précédents vis-à-vis d'un parcours de l'image. Les éléments structurants employés sont le plus souvent des carrés, des croix, des hexagones de rayon unitaire, auxquels on supprime les voisins *futurs* par rapport au centre de l'élément structurant pour un certain parcours de l'image.

Ces éléments structurants sont présentés en figure 2.40. Le carré noir représente le centre des éléments structurant et les zones grises représentent le voisinage considéré.

On peut très bien envisager de ne pas utiliser les éléments structurants nécessaires aux passes avec un parcours indirect. Il est en effet possible de parcourir dans un sens indirect une image en parcourant dans le sens direct cette dernière ayant subi une symétrie centrale. Lorsque nous envoyons un flux de pixels aux processeurs de voisinage, ces derniers travaillent toujours dans le sens direct et cette symétrie centrale est implicite, selon la manière dont sont envoyés les pixels. En d'autres termes, il est uniquement nécessaire de symétriser le flux pixel dans le cas d'un parcours indirect équivalent. Quant à la structure "direct" du processeur de voisinage, celle-ci ne change pas, la figure 2.41 illustre ce point.

On remarquera tout de suite qu'une architecture de processeur de voisinage pour fonctionner avec de tels éléments structurants peut être réalisée avec une seule ligne à retard, car nous considérons dans cette approche récursive uniquement des éléments structurants

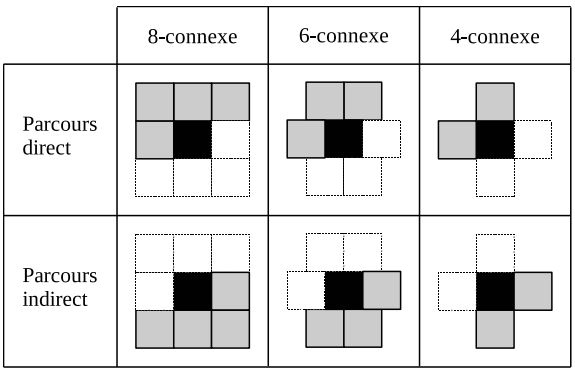


FIG. 2.40: Éléments structurants utilisés pour le calcul d'opérations morphologiques par une approche récursive

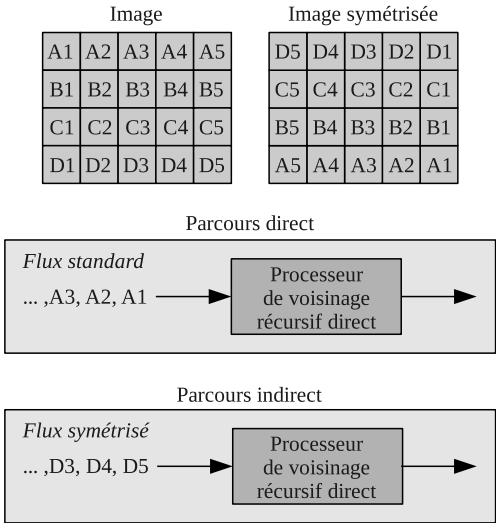


FIG. 2.41: Symétrie des flux pixels pour le parcours direct et indirect d'une image dans un processeur de voisinage

composés de deux lignes. Toutefois, nous souhaitons, avec la même architecture, également faire des opérations de voisinages classiques, c'est la raison pour laquelle nous présentons une structure de processeur de voisinage standard, dans laquelle nous avons ajouté les ressources nécessaires à la construction d'opérations récursives. Cette structure est présentée en figure 2.42 et permet, grâce au multiplexeur, de boucler la sortie du calcul vers les registres du voisinage autorisant la construction de tous types de propagation.

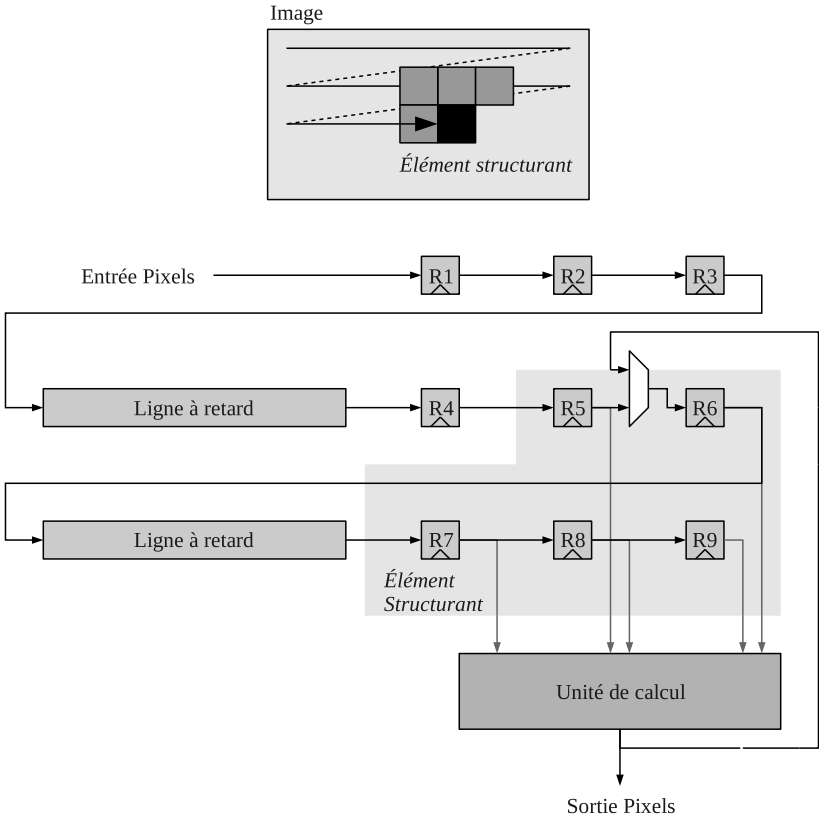


FIG. 2.42: Structure simple d'un processeur de voisinage standard pouvant fonctionner en mode récursif

Avec cette dernière figure, on observe une contrainte forte de latence dans l'unité de calcul. Le bouclage se faisant immédiatement après le registre central, il est nécessaire que la valeur du calcul soit disponible instantanément. C'est-à-dire que, dans notre cas, l'unité de calcul ne doit comporter aucun registre. Cette contrainte est assez peu réaliste en pratique, car nous visons des fréquences de traitements assez élevées. Il n'est donc pas envisageable de laisser de longs chemins critiques dans notre architecture comme on pourrait en trouver dans des unités de tris sans aucun registre. Nous devons donc prendre en compte la latence dans l'unité de calcul à deux niveaux :

- le premier se situe au niveau du registre R6 de la figure 2.42 et provient du fait qu'il faut être capable de prendre en compte le bouclage au niveau de R6. Nous venons de dire qu'il n'était pas possible de réellement boucler le calcul, il faudra donc modifier l'unité de calcul pour faire entrer au temps t, le résultat du temps $t-1$.
- le second se situe dans la propagation des résultats dans la ligne à retard. Dans l'état actuel des choses, on ne peut pas modifier simplement le contenu de la ligne à retard.

En effet, en considérant une latence k de l'unité de calcul et lorsque le résultat est disponible, il faudrait être en mesure d'écraser la valeur entrée dans la ligne $k-1$ cycles en arrière. Une telle opération est envisageable si l'on raccourcit la ligne à retard de $k-1$ éléments.

La figure 2.43 présente un processeur de voisinage standard pouvant à la fois travailler dans un cadre classique et dans un cadre récursif. Pour travailler dans le cadre récursif, il suffit d'affecter une taille plus petite à la ligne à retard 2 de $k-1$ éléments, d'activer le multiplexeur réalisant le bouclage sur la ligne à retard et de prendre en compte dans le dernier étage du calcul la valeur du registre Rb.

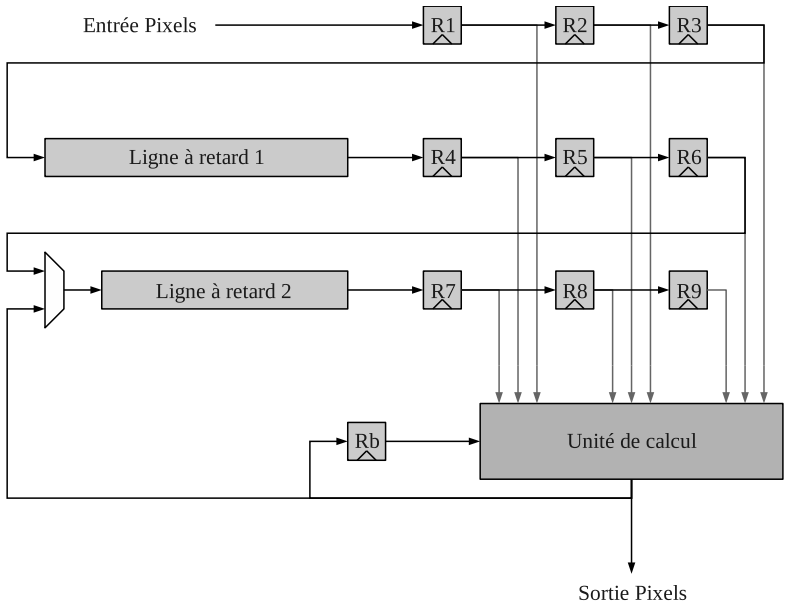


FIG. 2.43: Structure d'un processeur de voisinage standard pouvant fonctionner en mode récursif ou non

Il est théoriquement possible d'appliquer le même principe aux processeurs à extraction des voisinages parallélisés de la section suivante, mais il faut être en mesure de propager les valeurs d'une unité de calcul à l'autre, car ces dernières prennent en entrée des voisinages connexes de l'image. Cette propagation est coûteuse, car elle met en œuvre des chemins critiques où il est difficile d'intercaler des registres pour accélérer la cadence.

2.3.2 Domaines applicatifs

2.3.2.1 Transformation distance

Une première utilisation de ce type d'opérateur repose sur le calcul de la distance discrète en 4,6 ou 8-connexité d'une image binaire. Ce type de distance peut être calculée de façon séquentielle en accumulant dans une image numérique les érosions de taille croissante de l'image binaire.

La transformation distance telle que décrite par Rosenfeld et Pflaz [63], se décompose en deux étapes :

– Parcours direct de tous les pixels p de l'image f en considérant le calcul ci-dessous, où le graphe du voisinage $N_{\overleftarrow{g}}$ correspond à un de ceux définis en figure 2.40 pour le parcours direct.

$$\text{si } f(p) = 1 \text{ alors } f(p) \leftarrow 1 + \min\{f(q) | q \in N_{\overleftarrow{g}}(p)\}$$

– Parcours indirect de tous les pixels p de l'image f en considérant le calcul ci-dessous, avec un voisinage adapté au parcours indirect.

$$f(p) \leftarrow \min[f(p), 1 + \min\{f(q) | q \in N_{\overrightarrow{g}}(p)\}]$$

On peut utiliser uniquement la seconde formule pour les deux passes si l'on considère que l'image binaire est codée non plus entre 0 et 1 mais entre 0 et n, où n représente la valeur maximale du codage d'un pixel.

L'arbre de calcul permettant d'obtenir la transformation distance est présenté en figure 2.44. Cet arbre permet la réalisation des deux passes de calcul et suppose que les valeurs de l'image binaire, pour la première passe, soient 0 ou n. On retrouve le registre Rb qui permet de prendre en compte le premier niveau de la récursion en conservant le résultat précédent du calcul.

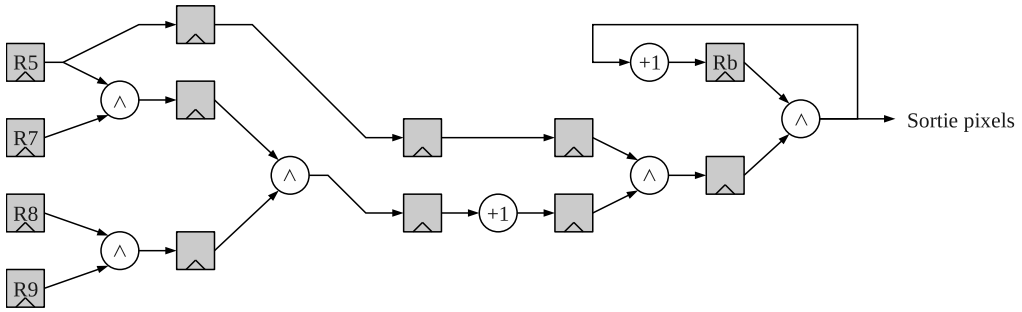


FIG. 2.44: Arbre de calcul de la transformation distance pour les passes avec flux direct et indirect

Un exemple est présenté en figure 2.45, où l'on observe le résultat partiel de la première passe de calcul dans le sens direct et le résultat final après la seconde passe dans le sens indirect. Les histogrammes des images ont été normalisés afin d'améliorer le rendu de l'affichage.

2.3.2.2 Reconstruction géodésique

Comme dans le cadre de la transformation distance, la mise en place d'un traitement récursif impose l'utilisation d'éléments structurants spécifiques que nous avons déjà présentés en figure 2.40. Nous utilisons donc un élément structurant $N_{\overleftarrow{g}}$ dans le cadre d'une passe directe, et un élément structurant $N_{\overrightarrow{g}}$ dans le cadre d'une passe indirecte. La reconstruction géodésique par dilatation consiste à répéter alternativement sur tous les pixels de l'image et jusqu'à stabilité les deux équations suivantes :

$$f(p) \leftarrow \min[g(p), \max\{f(q) | q \in N_{\overleftarrow{g}}(p) \cup p\}]$$
$$f(p) \leftarrow \min[g(p), \max\{f(q) | q \in N_{\overrightarrow{g}}(p) \cup p\}]$$

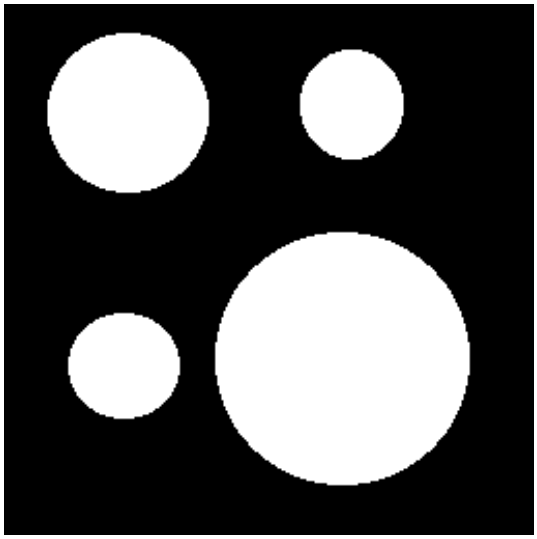
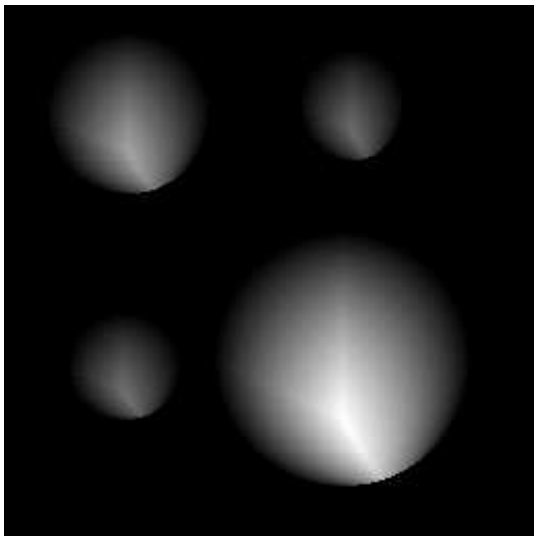
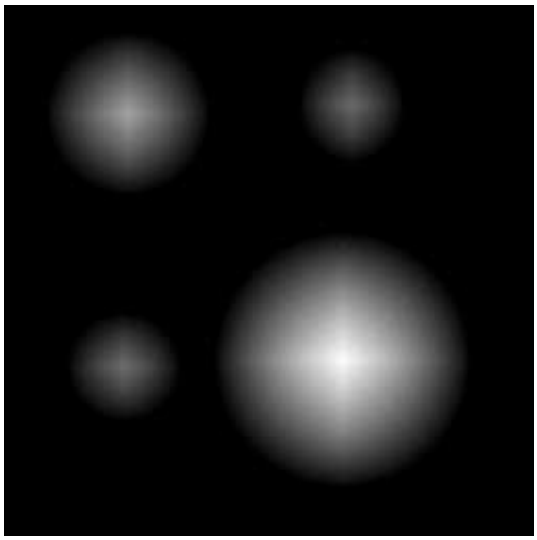


image originale



première passe



seconde passe

FIG. 2.45: Calcul de la transformation distance 8-connexe

On peut facilement étendre ces formules à la reconstruction par érosion en remplaçant les *min* par des *max* et les *max* par des *min*. La reconstruction autoduale nécessite toujours un traitement particulier selon la position du masque (en dessous ou au dessus), il est nécessaire d'utiliser à la place les formules suivantes :

$$f(p) \leftarrow \begin{cases} \min[g(p), \max\{f(q) | q \in N_{\overleftarrow{g}}(p) \cup p\}], & \text{si } f(p) \leq g(p) \\ \max[g(p), \min\{f(q) | q \in N_{\overleftarrow{g}}(p) \cup p\}], & \text{sinon} \end{cases}$$

$$f(p) \leftarrow \begin{cases} \min[g(p), \max\{f(q) | q \in N_{\overrightarrow{g}}(p) \cup p\}], & \text{si } f(p) \leq g(p) \\ \max[g(p), \min\{f(q) | q \in N_{\overrightarrow{g}}(p) \cup p\}], & \text{sinon} \end{cases}$$

L'unité de calcul présentée en figure 2.46, pour le processeur de voisinage récursif défini en figure 2.43, permet de réaliser tous les types de reconstructions géodésiques que nous avons décrits ici. Les registres R5, R7, R8, R9 contiennent les voisinages d'un pixel de l'image f et le registre Rg contient le pixel correspondant au centre de l'élément structurant dans l'image g (le masque). Le registre Rb permet de prendre en compte le calcul précédent, car la latence de l'unité de calcul empêche d'écraser R6 (ce problème a déjà été présenté précédemment). Le comparateur permet de mettre en place le calcul des nivellements et la sortie de ce dernier commande l'opération à effectuer dans les étages suivants, à savoir un minimum ou un maximum. Ainsi, selon la position du masque par rapport au marqueur, on obtient une reconstruction par dilatation, une reconstruction par érosion ou un nivellement.

Par ailleurs, il convient d'utiliser deux extracteurs de voisinages en parallèle, parce que nous avons besoin de deux images. On peut également ajouter au processeur de voisinage une ligne à retard indépendante qui servira uniquement à alimenter l'unité de calcul avec les pixels du masque. Puisqu'il n'est pas nécessaire d'extraire le voisinage du masque, cette dernière solution est satisfaisante.

2.4 Structure parallélisée

2.4.1 Vue globale

Nous détaillons ici une nouvelle voie de traitement du parallélisme avec les processeurs de voisinage. La méthode traditionnellement employée pour traiter une image avec un plus fort parallélisme consiste à la découper en morceaux afin d'alimenter plusieurs processeurs en parallèle. Ce principe est largement abordé dans le chapitre 4 et n'est pas nécessairement optimal notamment au niveau du découpage des imagettes puisqu'il est nécessaire de prévoir des zones de recouvrement à cause du traitement de l'image par un voisinage.

Nous proposons dans cette section, ainsi que dans l'article [23], une autre méthode à un grain beaucoup plus fin. En effet, s'il on considère que le système alimentant le processeur de voisinage envoie les pixels groupés par paquets de n (ce qui est traditionnellement le cas avec les mémoires employées de nos jours), il est envisageable d'extraire n voisinages contigus par cycle et ainsi produire n pixels résultats par cycles. Le principe général présenté en figure 2.47 n'est pas très différent de la structure standard décrite précédemment. Il est juste nécessaire de prévoir n unités de gestion des bords, n unités de calcul et un système d'extraction du voisinage modifié. Nous verrons également qu'il est possible de réduire

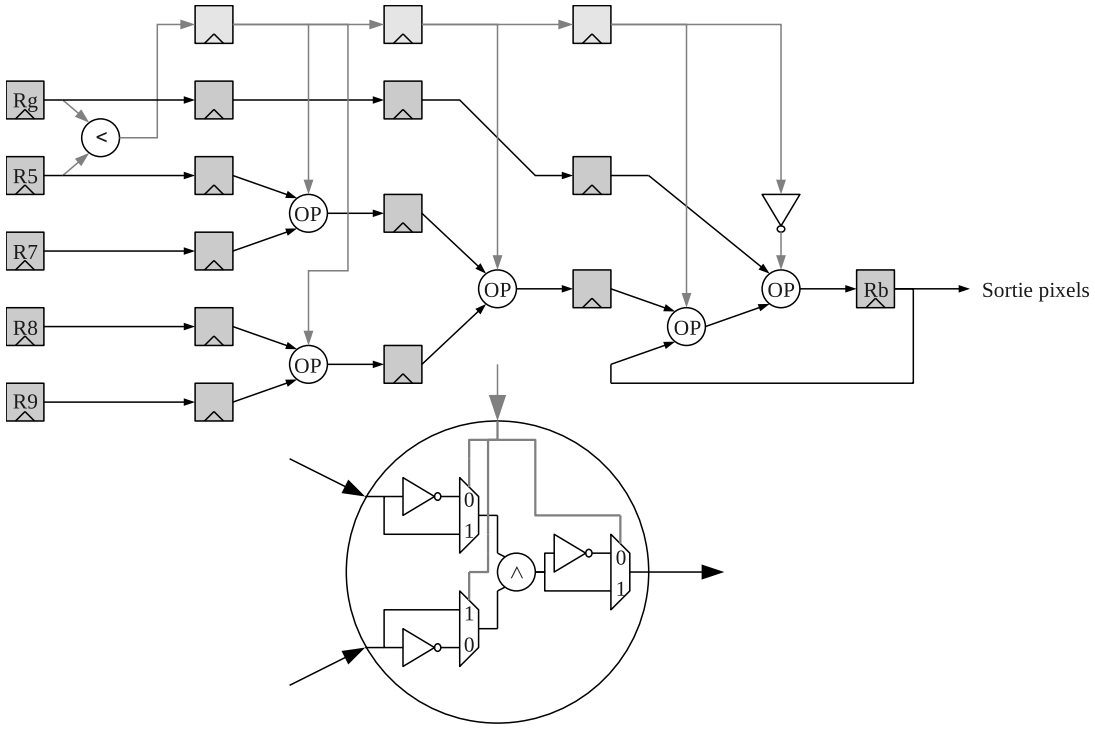


FIG. 2.46: Arbre de calcul de la reconstruction géodésique

la quantité de ressources nécessaires, par exemple dans les unités de calcul, puisque des opérations redondantes entre les voisinages contigus peuvent être regroupées.

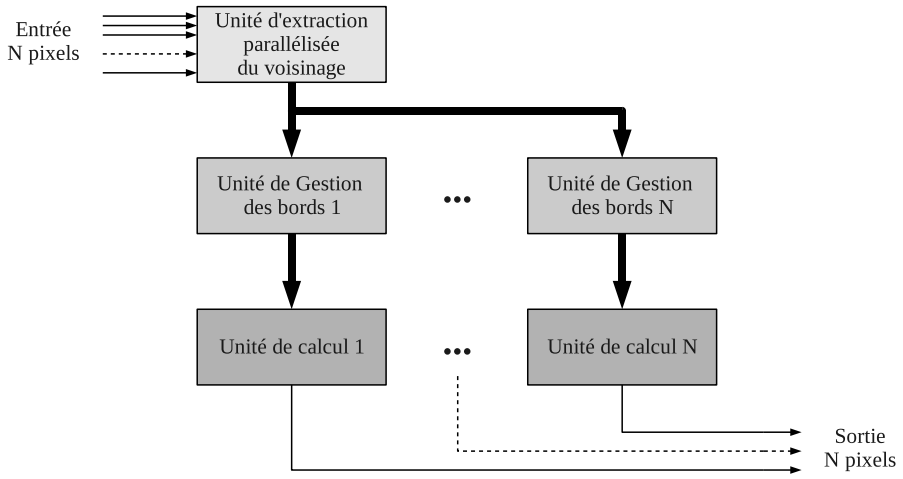


FIG. 2.47: Structure générale d'un processeur de voisinage parallélisé

2.4.2 Extraction du voisinage

L'objectif est d'exploiter au maximum le débit mémoire amont, c'est-à-dire que, si le système alimentant le processeur fournit n pixels contigus par cycle, il faut pouvoir exploiter

au maximum ce parallélisme. S'il n'était pas exploité, il serait nécessaire de découper les paquets de pixels pour les traiter un à un dans un processeur de voisinage standard et donc multiplier le temps de traitement par n.

L'exploitation du parallélisme des données se fait donc en extrayant n voisinages connexes par cycle comme le montre la figure 2.48. On remarque qu'il est nécessaire de prévoir une zone de recollement entre deux cycles ce qui a pour effet d'ajouter un certain nombre de registres à l'unité d'extraction du voisinage. Si l'on considère des voisinages ayant N lignes et M colonnes il faut prévoir $N \cdot (\frac{M}{2} + 1)$ registres de recollement.

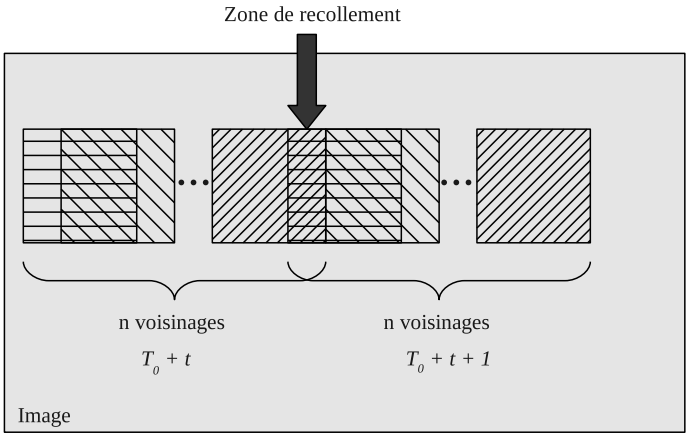


FIG. 2.48: Principe d'extraction parallélisé de voisinages contigus

Les lignes à retard sont conservées et leur capacité ne change pas, seule la taille des mots est modifiée pour correspondre au degré de parallélisme n. Le nombre de registres nécessaires à l'extraction des voisinages s'exprime de la manière suivante :

$$f(n) = N \cdot n + N \cdot \left(\frac{M}{2} + 1\right) = N \cdot n + \frac{N \cdot M}{2} + N$$

Si l'on ne considérait pas une extraction des voisinages contigus, il serait nécessaire d'utiliser le nombre de registres suivant :

$$g(n) = N \cdot M \cdot n$$

Afin de connaître la quantité de registres économisée, on peut écrire la relation suivante :

$$\lim_{n \to +\infty} h(n) = \lim_{n \to +\infty} \frac{f(n)}{g(n)} = \frac{1}{M}$$

Ainsi le nombre de registres nécessaires à l'extraction de n voisinages connexes tend à être M fois plus petit que le nombre de registres utilisés pour l'extraction de n voisinages non connexes.

L'architecture d'une telle unité d'extraction est présentée en figure 2.49, on retrouve les registres de recollement RBx, les registres RAx recevant les nouveaux paquets de pixels et les lignes à retard Lx.

A chaque cycle les registres RAx reçoivent n pixels en provenance de la ligne à retard Lx ou, dans le cas de RA0, directement depuis l'entrée pixels. On conserve alors les $M/2+$

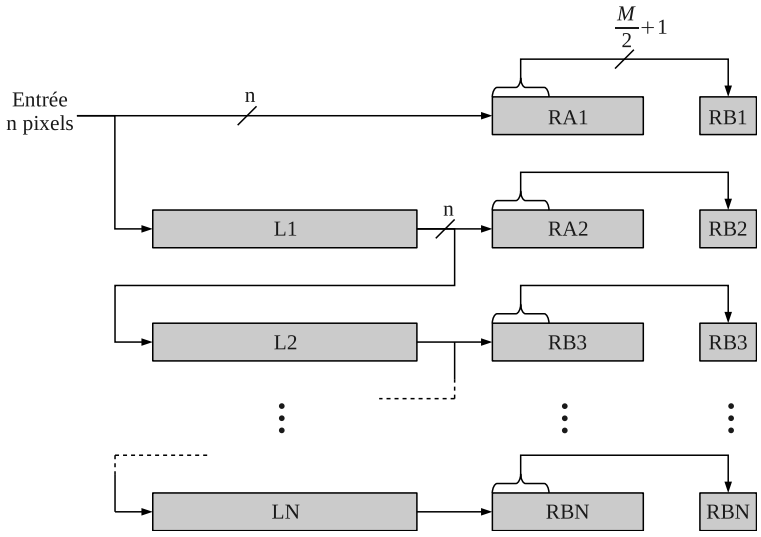


FIG. 2.49: Structure d'un processeur de voisinage $N \times M$ parallélisé par n

1 valeurs du dernier voisinage dans les registres RBx afin de garantir un recouvrement correct lors du cycle suivant. Une cinématique est proposée en figure 2.50 dans le cas d'un extracteur de voisinage 3×3 parallélisé par 4.

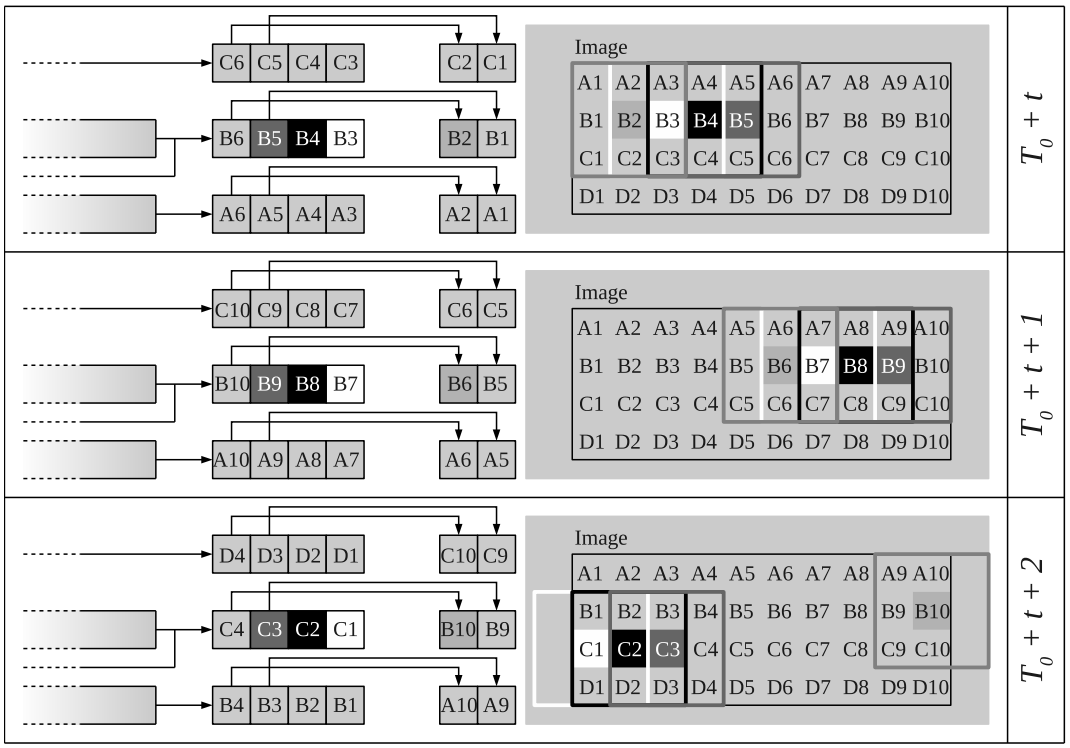


FIG. 2.50: Exemple de processeur de voisinage 3×3 parallélisé par 4

Cette dernière figure montre également, dans le cas $T_0 + t + 2$, les problèmes de gestion des bords. Selon la taille des lignes, les n voisinages n'ont pas les mêmes problèmes de

bords aux mêmes instants, cet aspect est abordé dans la section suivante.

Le fonctionnement des lignes à retard n'est pas aussi simple qu'il y paraît et il n'est pas uniquement nécessaire de changer la taille des mots mémoire afin de les rendre compatibles avec notre extracteur de voisinage parallélisé. En effet, le nombre de mots mémoire (un mot recevant n pixels) multiplié par n doit être égal à la taille de la ligne. En d'autres termes, sans modification de la structure des lignes à retard, la taille d'une ligne doit être multiple de n. Si cette contrainte n'est pas respectée, l'extraction des voisinages ne sera pas correcte, car les lignes à retard n'auront pas une taille suffisante pour mémoriser une ligne. On observera alors un décalage grandissant des voisinages extraits au fur et à mesure que le processeur recevra de nouvelles lignes à traiter.

La manière la plus simple de s'affranchir de cette contrainte est d'ajouter aux lignes à retard le nombre de registres nécessaires pour obtenir une taille correspondante à celle d'une ligne de l'image. Le nombre maximum de registres ainsi ajoutés sur la ligne à retard ne peut pas excéder $n - 1$ et ces derniers sont ajoutés un à un sur les sorties de la ligne à retard. Un exemple est présenté en figure 2.51 où l'on considère une image 9×4, un voisinage 3×3 et une extraction parallélisée d'ordre 4. Il est donc nécessaire d'ajouter un registre en sortie de la ligne à retard au niveau du premier élément pour compléter la taille permettant d'atteindre une capacité de 9 pixels. Si les lignes de l'image avaient une taille de 10 pixels, il aurait fallu ajouter deux registres, un au niveau du premier élément et un autre au niveau du second.

Lorsque des registres sont ajoutés, on remarque qu'il est nécessaire de procéder à un réordonnancement des éléments (a1, b1, c1, d1, dans le schéma de la figure 2.51). Les registres ajoutés ont pour effet d'introduire un retard impliquant de rerouter ces signaux par rapport à ceux ne disposant pas d'un tel registre. Ce reroutage dépend de la taille de la ligne et du degré de parallélisme. On peut tout à fait imaginer de figer le degré de parallélisme pour un processeur de voisinage, mais pour être suffisamment souple, ce dernier doit être capable de traiter plusieurs tailles de lignes. Il faut ainsi prévoir tous les registres en sortie des lignes à retard ainsi que tous les chemins de données nécessaires à leur activation, à leur désactivation et au reroutage des données. Un tel système est présenté en figure 2.52 pour des mots de quatre pixels, mais peut être généralisé à des mots de taille n.

La table de vérité commandant les quatre multiplexeurs est proposée en 2.8. Elle permet de gérer toutes les tailles de lignes pour un processeur de voisinage parallélisé par 4.

Mux / **Nb. Reg.**	**ax**	**bx**	**cx**	**dx**
0	s1	s2	s3	s4
1	s2	s3	s4	s5
2	s3	s4	s5	s6
3	s4	s5	s6	s7

TAB. 2.8: Table de vérité de sélection des registres de complément de la ligne à retard dans le cas d'un extracteur parallélisé de taille 4.

Une telle structure d'extraction de voisinage permet de réduire la latence d'une opération de voisinage. En effet la latence de ces opérateurs de voisinage est traditionnellement

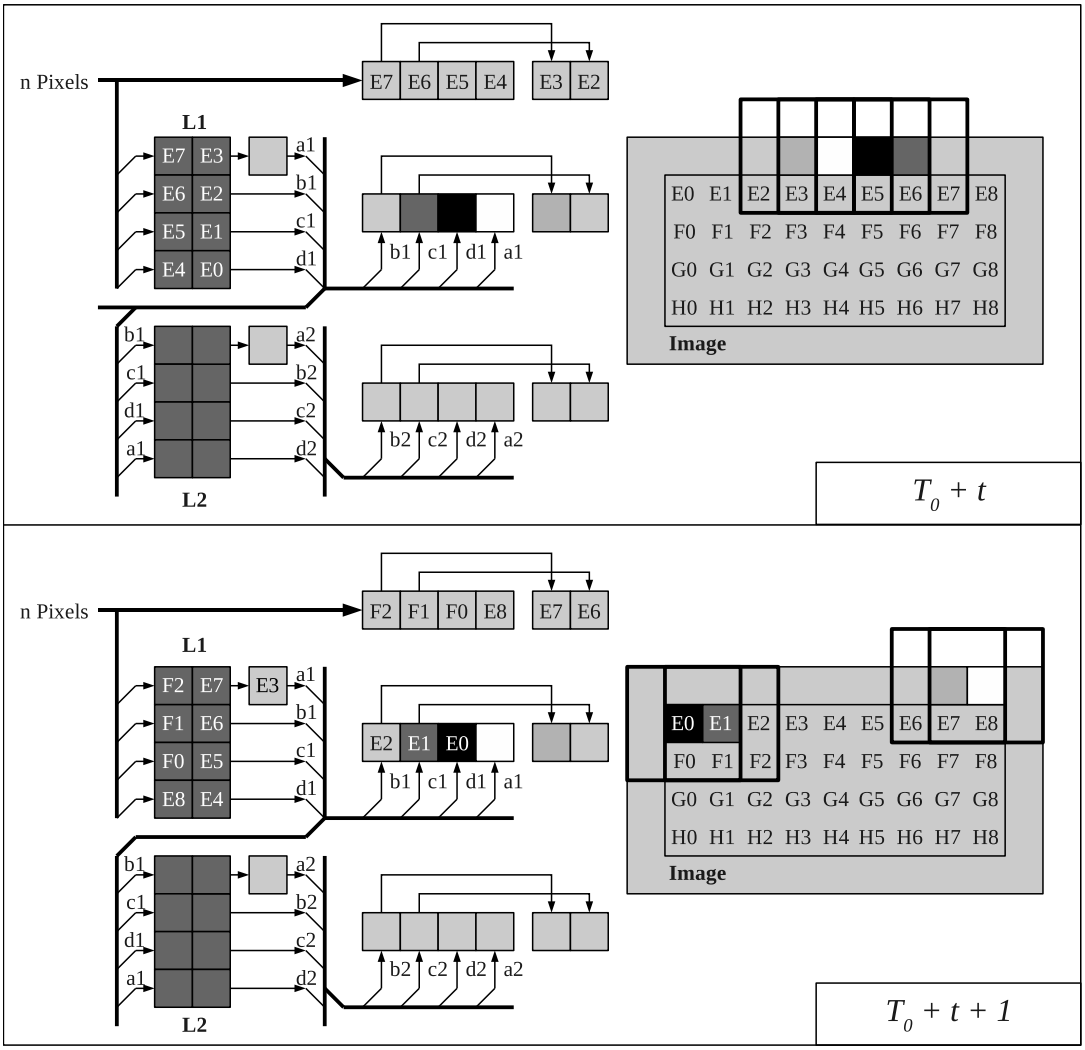


FIG. 2.51: Exemple complet de processeur de voisinage 3×3 parallélisé par 4

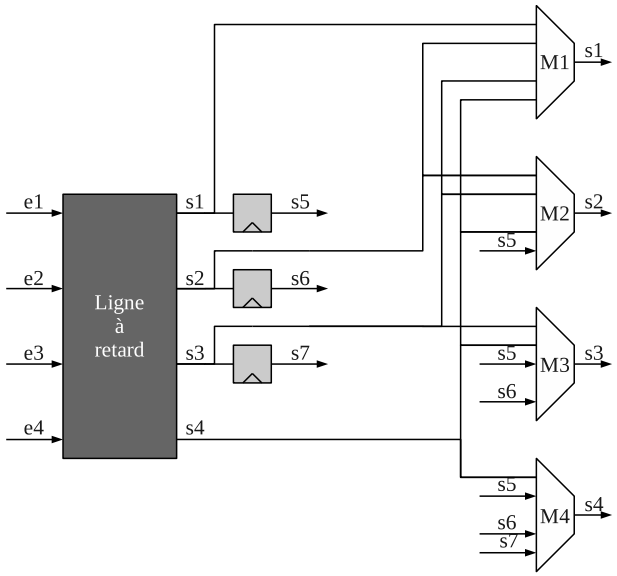


FIG. 2.52: Gestion des différentes tailles de lignes pour un extracteur de voisinage parallélisé

due aux lignes à retard fonctionnant pixel par pixel. Avec cette nouvelle structure la latence est environ divisée par n. Effectivement, même si la taille de la ligne à retard n'a pas changée, la taille des mots mémoire transmis à chaque cycle se compose maintenant de n pixels.

2.4.3 Gestion des bords

La gestion des bords est assez similaire à celle mise en place dans les processeurs de voisinage flot de donnée standard de la figure 2.13. Il est juste nécessaire de dupliquer n fois cette unité et de modifier les compteurs de lignes et de colonnes. Chaque unité de gestion de bords doit embarquer ses propres compteurs, car comme nous avons pu le voir dans les figures précédentes, les voisinages extraits à instant t peuvent être à cheval sur deux lignes. Chacun des compteurs de colonnes doit être initialisé avec l'indice du voisinage extrait. Par exemple, le compteur de colonnes correspondant au voisinage le plus à l'est doit être initialisé avec la valeur zéro et le compteur de colonnes correspondant au voisinage le plus à l'ouest doit être initialisé avec la valeur $n - 1$. L'incrément des compteurs de colonnes doit être aussi remplacé par la valeur n.

Les signaux informant de la position des n voisinages vis-à-vis des bords de l'image sont ensuite générés pour être utilisés par les unités de remplacement de pixels décrites en figure 2.14. Ces dernières sont, elles aussi, répliquées pour chaque voisinage extrait.

Nous disposons alors de n voisinages dont les bords ont été gérés en remplaçant les voisins hors de l'image par des valeurs qui ne perturberont pas ou peu le calcul. Il est aussi envisageable de transmettre directement les signaux informant de la présence d'un problème de bords pour chacun des n voisinages vers les unités de calculs respectives, dans le cas où ces dernières ne supporteraient pas le remplacement de pixels proposé.

2.4.4 Optimisation de l'arbre de calcul

Sachant que les n voisinages extraits de l'image se recouvrent, il est peut-être plus intéressant d'envisager une unité de calcul pour tous les voisinages plutôt que n unités de calcul. En effet, les calculs sur les colonnes d'un voisinage peuvent être réutilisés pour d'autres voisinages. Ce principe à déjà été abordé dans la section traitant des unités de calculs des opérateurs de rang, mais l'économie se faisait entre deux cycles lorsque le parcours de l'image le permettait, alors qu'ici l'économie se fait spatialement.

Un exemple d'une unité de calcul optimisé pour le calcul d'une érosion avec un élément structurant 3×3 et un degré de parallélisation 4 est proposé en figure 2.53.

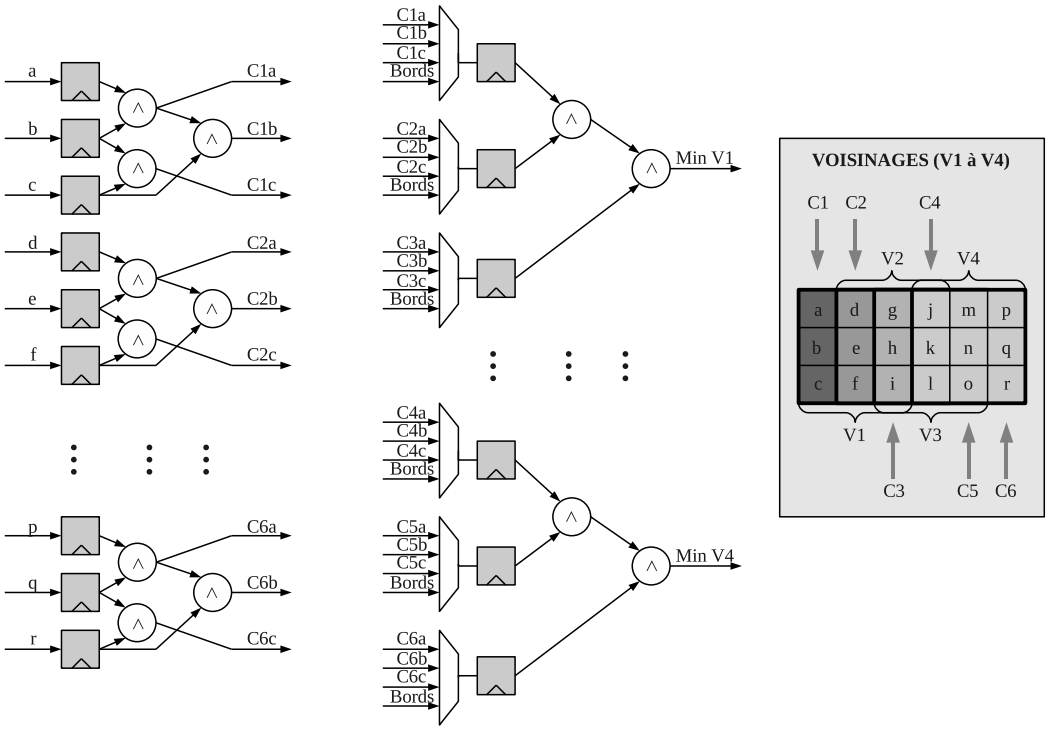


FIG. 2.53: Arbre de recherche du minimum de quatre voisinages connexe

Comme dans le cas des arbres de calculs optimisés des processeurs de voisinage classiques, il est nécessaire de déporter la gestion des bords au sein de l'unité de calcul. En effet une même colonne peut servir au calcul de deux voisinages. Lorsque cette dernière ne doit pas être prise en compte dans le calcul, car se trouvant dans le bord EST d'un voisinage k, elle doit être utilisée dans le calcul du voisinage $k+1$. C'est la raison pour laquelle tous les cas de calculs de minima sur une colonne sont pris en compte comme le montre la figure 2.53.

La figure 2.54 illustre ce phénomène en considérant un extracteur de voisinage 3×3 parallélisé par 2. On remarque que la colonne C3 du voisinage V1 ne doit pas être prise en compte alors qu'elle doit être utilisée dans le calcul de V2. Ceci est dû au fait que la colonne C3 n'est pas réellement hors de l'image puisqu'il n'existe pas de *padding*, mais se trouve déjà sur la ligne suivante.

Cette optimisation est intéressante uniquement dans le cadre de voisinages 3×3 car

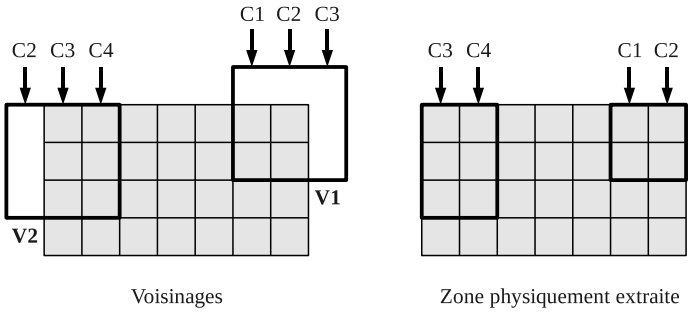


FIG. 2.54: Arbre de recherche du minimum de quatre voisinages connexes

la combinatoire du calcul de toutes les opérations d'une colonne est faible. Dans le cadre de la figure 2.53 on utilise finalement que 26 opérateurs au lieu de 36 dans le cadre non optimisé. Dès lors que l'on considère des voisinages plus importants, la combinatoire sur les colonnes est tellement importante qu'il est préférable de considérer une unité de calcul indépendante par voisinage. Par exemple, pour des extracteurs parallélisés d'ordre 4, une unité de calcul "optimisée" considérant des éléments structurants 5×5 utilise 96 opérateurs (par exemple min/max pour des érosions/dilatations). La version standard utilisera également 96 opérateurs, mais avec des chemins de données beaucoup plus simples et sans multiplexeurs.

2.5 Structure segment

2.5.1 Avant propos

Nous nous sommes concentrés depuis le début de ce chapitre sur le calcul d'opération sur des voisinages 2D. Cependant, des algorithmes très optimisés calculant des érosions/dilatations avec des éléments structurants de type segment (ESS) existent. Ces algorithmes travaillent en temps constant quelle que soit la taille de l'ESS. Les propriétés de décomposition des éléments structurants pour les opérations d'érosion et de dilatation permettent ainsi de considérer toutes les tailles de segments sans complexifier les calculs. La figure 2.55 illustre ce principe, nous utilisons ici des ESS dans différentes directions afin d'obtenir une opération équivalente au traitement pas un élément structurant hexagonal de rayon 3.

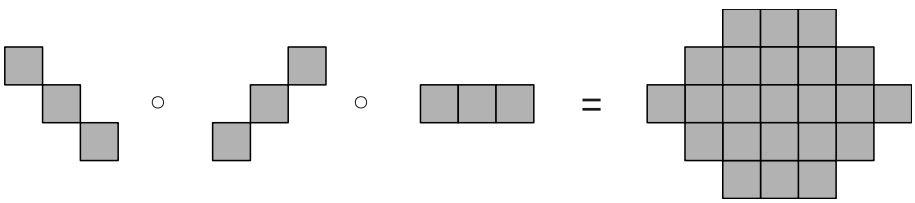


FIG. 2.55: Composition de trois opérations utilisant des segments permettant d'obtenir une opération équivalente à un hexagone

Cette décomposition est bien évidemment possible à condition de disposer d'un adressage des pixels d'une image utilisant l'algorithme de Bresenham [16].

2.5.2 Algorithme et architectures existantes

2.5.2.1 Approche de Lemonnier

L'algorithme de Lemonnier [47] ne traite que le cas de la dilatation, l'érosion peut être calculée par dualité :

$$\varepsilon_B(f^c) = [\delta_B(f)]^c$$

Les algorithmes 1 et 2 présentent l'algorithme de Lemonnier qui permet de réaliser une dilatation sur une ligne f de taille M avec un élément structurant $K = 2 \cdot k + 1$ et se déroule en deux passes avec un temps constant quelque soit la taille de k. Tout d'abord, une propagation des pixels dans le sens direct est réalisée en considérant la taille de l'élément structurant. Cette passe gère les bords grâce au calcul de t qui corrige les erreurs de propagation en bout de ligne. Ensuite une seconde passe est réalisée dans le sens indirect, durant laquelle les bords sont gérés dès le début afin d'être en mesure de correctement propager ces derniers obtenus lors de la première passe du calcul. La mémoire tampon g est un tableau temporaire initialisé à zéro où l'on stocke les propagations de pixels avant de produire le résultat final avec $f(x-k)$. Le vecteur g peut être remplacé par deux variables, car seules la valeur courante et la valeur précédente sont utilisées dans les calculs.

Algorithme 1 Première passe de l'algorithme de dilatation 1D de Lemonnier [47]

```
ENTRÉES: f, k, M
SORTIES: h

  n := 0
  Pour x=0 to M-1 Faire
    Si x>M-k-1 Alors
      Si t[x-1]>f[x] Alors
        t[x] := f[x]
      Sinon
        t[x] := t[x-1]
      Fin si
    Sinon
      t[x] := 0
    Fin si
    Si g[x-1]>f[x] ∧ n<k Alors
      g[x] := g[x-1]
      n := n+1
    Sinon
      Si x>M-k-1 ∧ n ≥ k Alors
        g[x] := t[x]
      Sinon
        g[x] := f[x]
      Fin si
      t[x] := 0
      n := 0
    Fin si
    h[x] := max(g[x],f[x-k])
  Fin pour
```

Une architecture flot de donnée, proposée en figure 2.56, mettant en oeuvre cet algorithme utilise uniquement deux opérateurs max à deux entrées. Le traitement des lignes dans des sens différents implique de mettre en place deux mémoires tampons entre les unités réalisant chacune les passes de l'algorithme, nous définissons ce mécanisme comme un système de double tampon. En effet, pendant que la première unité de propagation produit, dans le sens direct, le résultat intermédiaire de la ligne n, la seconde unité de propagation produit le résultat de la ligne $n-1$ en utilisant le résultat intermédiaire de la

Algorithme 2 Seconde passe de l'algorithme de dilatation 1D de Lemonnier [47]

```
ENTRÉES: h, k, M
SORTIES: δ_f

  n := 0
  Pour x=M-1 downto 0 Faire
    Si (h[x]>h[x+1]) ∧ (t[x]<h[x])
    Alors
      t[x] := h[x]
    Sinon
      t[x] := t[x+1]
    Fin si

    Si (g[x+1]>h[x]) ∧ (n<k) Alors
      g[x] := g[x+1]
      n := n+1
    Sinon
      Si n ≥ k Alors
        g[x] := t[x]
      Sinon
        g[x] := h[x]
      Fin si
      t[x] := 0
      n := 0
    Fin si
    Si x<k Alors
      δ_f[x] := g[x]
    Sinon
      δ_f[x] := max(g[x],h[x+k])
    Fin si
  Fin pour
```

seconde mémoire tampon dans le sens indirect. À chaque nouvelle ligne de l'image à traiter, les chemins de données conduisant à ces mémoires sont échangés. De plus, les unités de propagations doivent être en mesure d'accéder à $f(x-k)$ ou à $h(x+k)$, ce qui implique que ces dernières disposent chacune d'une mémoire de taille k. Cette architecture présente l'inconvénient de retourner les lignes de l'image en sortie à cause de la seconde passe dans le sens indirect.

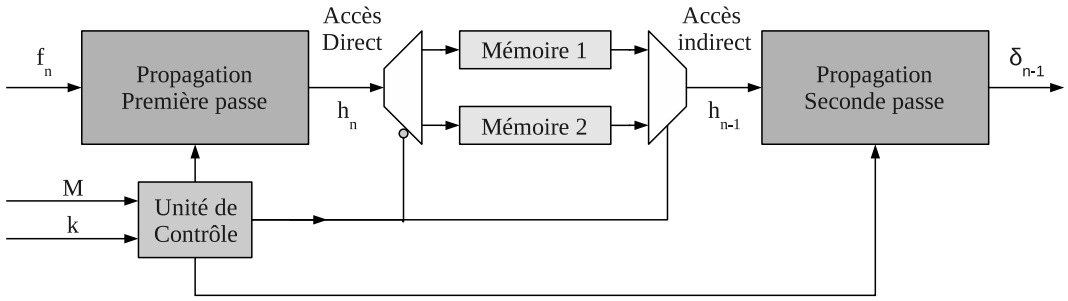


FIG. 2.56: Mise en œuvre matérielle flot de données de l'algorithme de dilatation de Lemonnier

2.5.2.2 Approche HGW

Cet algorithme travaille également avec des ESS, et a été décrit à la fois par van Herk[79] et par Gil-Werman[31]. Il permet de calculer toutes tailles de dilatations ou d'érosions sur les lignes d'une image en utilisant trois opérateurs min ou max selon que l'on désire une érosion ou une dilatation. Nous considérons une ligne f de taille M et un élément structurant segment de taille k (impair). Il se décompose en trois étapes, tout d'abord une propagation dans le sens direct permettant l'obtention de g à partir de f. Ensuite une passe dans le sens indirect permettant l'obtention de h toujours à partir de f. Finalement, l'érosion ou la dilatation sont calculées dans une troisième passe calculant la fusion de g et

h. Les équations ci-dessous regroupent le calcul de ces trois étapes dans le cadre du calcul d'une dilatation. Pour une érosion, il est juste nécessaire de remplacer les opérateurs min par des max.

$$g(x) = \begin{cases} f(x) & \text{si } x \bmod k = 0 \\ max(g(x-1), f(x)) & \text{sinon} \end{cases}, x = 0, 1, \cdots, M-1$$

$$h(x) = \begin{cases} f(x) & \text{si } x \bmod (k-1) = 0 \\ max(h(x+1), f(x)) & \text{sinon} \end{cases}, x = M-1, \cdots, 1, 0$$

$$\delta_f(x)) = max\left(g\left(x + \frac{k}{2}\right), h\left(x - \frac{k}{2}\right)\right), x = 0, 1, \cdots, M-1$$

Les expressions ci-dessus ne tiennent pas compte du problème qui apparait sur les bords lorsque la taille de l'image n'est pas un multiple de la taille k de l'élément structurant. Une solution pour traiter correctement les bords est d'ajouter les pixels manquants en bout de ligne pour arriver à une taille multiple de k. Cette méthode n'est pas satisfaisante, car elle entraîne des calculs inutiles. Une méthode plus efficace est de protéger l'accès hors de l'intervalle de définition en renvoyant directement la valeur du *padding*. La figure 2.57 montre un exemple de *padding* pour le traitement d'une ligne de 19 pixels par un élément structurant segment de taille 7.

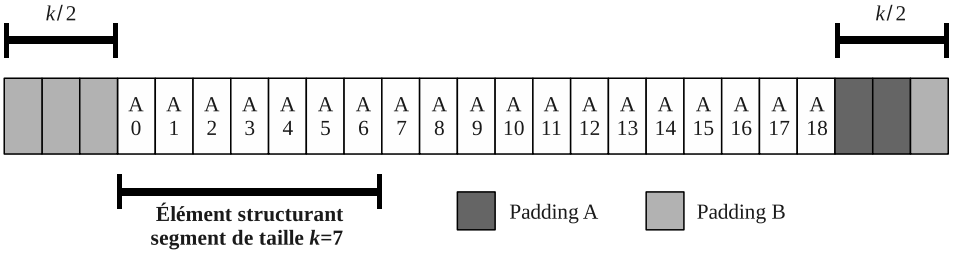


FIG. 2.57: Exemple de *padding* nécessaire à l'algorithme HGW

Le *padding* A est ajouté pour obtenir une taille de ligne multiple de k et le *padding* B est ajouté pour protéger les accès hors de la ligne. La taille du *padding* A (PSA) est défini de la manière suivante :

$$\text{PSA} = (k - (M-1) \bmod k) - 1$$

Il est donc possible de réécrire les équations de l'algorithme HGW pour émuler le *padding* et ainsi ne pas avoir de problèmes sur les bords :

$$g(x) = \begin{cases} f(x) & \text{si } x \bmod k = 0 \\ max(g(x-1), f(x)) & \text{sinon} \end{cases}$$

$$h(x) = \begin{cases} f(x) & \text{si } x \bmod (k-1) = 0 \\ f(x) & \text{si } x = M-1 \\ max(h(x+1), f(x)) & \text{sinon} \end{cases}$$

$$\delta_f(x) = \begin{cases} g(x+\frac{k}{2}) & \text{si } x-\frac{k}{2} < 0 \\ max(g(M-1), h(x-\frac{k}{2})) & \text{si } M \leq x+\frac{k}{2} < M+\text{PSA} \\ h(x-\frac{k}{2}) & \text{si } x+\frac{k}{2} \geq M+\text{PSA} \\ max\left(g\left(x+\frac{k}{2}\right), h\left(x-\frac{k}{2}\right)\right) & \text{sinon} \end{cases}$$

L'algorithme 3 permet de calculer l'algorithme HGW en utilisant les équations ci dessous tenant compte du problème des bords.

Algorithme 3 Dilatation 1D avec l'algorithme HGW [79] [31]

```
ENTRÉES: f, k, M
SORTIES: δf

  PSA :=(k-(M-1) mod k)-1

  Pour x from 0 to M-1 Faire
    Si x mod k=0 Alors
      g[x] :=f[x]
    Sinon
      g[x] :=max(g[x-1],f[x])
    Fin si
  Fin pour

  Pour x from M-1 downto 0 Faire
    Si x=M-1 Alors
      h[x] :=f[x]
    Sinon Si (x+1) mod k =0 Alors
      h[x] :=f[x]
    Sinon
      h[x] :=max(h[x+1],f[x])
    Fin si
  Fin pour

  Pour x from 0 to M-1 Faire
    Si x-k/2<0 Alors
      δf[x] :=g[x+k/2]
    Sinon Si x+k/2 ≥ M Alors
      Si x+k/2 < M+PSA Alors
        δf[x] :=max(g[M-1],h[x-k/2])
      Sinon
        δf[x] :=h[x-k/2]
      Fin si
    Sinon
      δf[x] :=max(g[x+k/2],h[x-k/2])
    Fin si
  Fin pour
```

Cet algorithme nécessite trois opérateurs min ou max par pixels calculés pour produire une érosion ou une dilatation, c'est un opérateur de plus que l'algorithme de Lemonnier. Le temps de calcul est donc constant quelque soit la taille de l'ESS. L'architecture flot de données présentée en figure 2.58 correspondant à cet algorithme, requiert des mémoires de la taille d'une ligne pour calculer h à partir de f (lu dans le sens indirect). On met alors en place deux mémoires de lignes, pendant que l'on stocke les pixels f_n dans le sens direct dans l'une des deux mémoires, on lit dans le sens indirect f_{n-1} dans l'autre mémoire. On échange alors ces mémoires dès qu'une nouvelle ligne doit être traitée. Ce double tampon introduit un retard d'une ligne dans le calcul de h et a pour effet d'inverser le flux de h par rapport au flux de g. Il faut donc ajouter aussi un double tampon après le calcul de g pour que l'unité de fusion opère les flux dans le même sens. Cette dernière doit avoir accès à $g(x+k/2)$ et à $h[x-k/2]$ ce qui impose de retarder le flux de h avec une ligne à retard de taille k. Les lignes de l'image résultat sont également retournées en sortie de cette architecture.

Cette mise en œuvre matérielle, pour fonctionner sans aucun temps mort, doit donc disposer de pas moins de quatre mémoires de lignes et d'une mémoire de la taille d'un élément structurant, la rendant ainsi beaucoup moins intéressante que l'approche de Lemonnier, cela sans compter le fait qu'elle utilise un opérateur min/max supplémentaire.

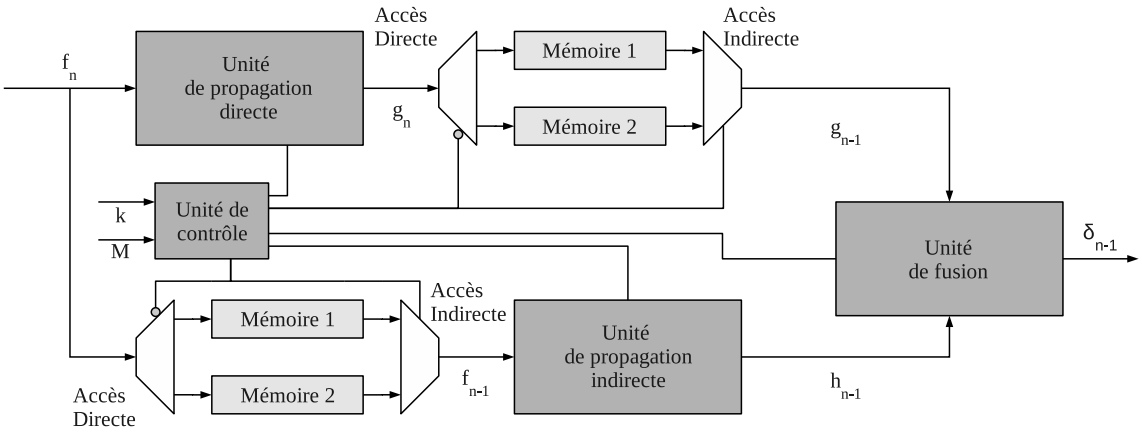


FIG. 2.58: Architecture fonctionnelle de l'algorithme HGW

Toutefois, nous allons voir dans la section suivante qu'une petite modification de l'algorithme peut rendre l'architecture HGW extrêmement intéressante en terme de quantité de mémoire mise en jeu et donc en terme de surface du circuit.

2.5.3 Amélioration de l'architecture HGW

La modification de l'algorithme HGW que nous proposons [24] est motivée par le fait de pouvoir traiter des images de grandes tailles sans avoir à stocker en entier la ligne à éroder/dilater. Ceci permet pour des architectures flots de données de retirer les mémoires de lignes. En effet, le principal défaut de l'approche standard est lié au fait que les propagations réalisant le calcul de g et h doivent être faite dans un sens opposé ce qui implique d'avoir connaissance de la ligne en entier ou de la mémoriser complètement.

2.5.3.1 Principe de fonctionnement

La figure 2.59 montre à quels instants, pour un élément structurant $k = 7$, sont réinitialisés les propagations. Les pixels représentés en noir symbolisent les pixels réintroduits (provenant de la ligne originale) et les gris clair les valeurs propagées.

Un bloc est défini comme étant un groupe de pixels entre deux insertions de pixels originaux, la propagation au sein d'un bloc est indépendante de celle des autres. Par exemple, dans la figure 2.59, les pixels de g $[A0, A6]$ forment un bloc. On remarque dans le calcul de h que si une rotation centrée des blocs est réalisée avant propagation, les pixels originaux réintroduits dans g et h le sont aux mêmes indices. Cette rotation avant propagation, produisant le tableau f', supprime la nécessité d'effectuer un passage dans le sens vidéo inverse. Un exemple de rotation des blocs de f produisant f' avant la propagation de h' est présenté en figure 2.60. Il est bien sûr nécessaire après propagation de reconstruire h à partir de h' en retournant les blocs de ce dernier.

L'architecture fonctionnelle de la mise en œuvre de l'algorithme HGW modifié est proposée en figure 2.61. Elle permet donc de supprimer la nécessité d'une propagation dans le sens indirect et permettra d'économiser une importante quantité de mémoire.

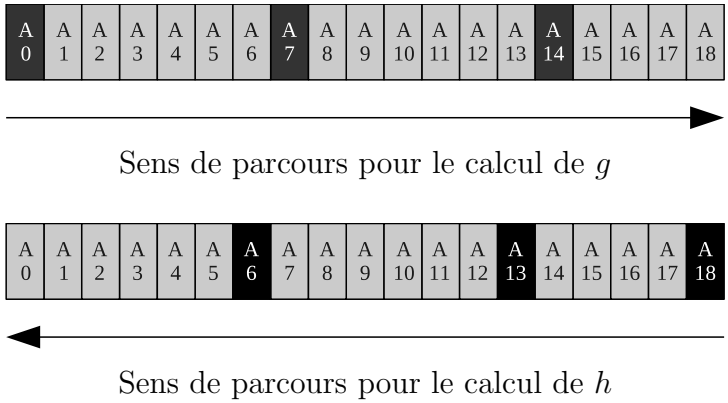


FIG. 2.59: Instants des initialisations de g et h avec f dans le cadre de l'algorithme HGW

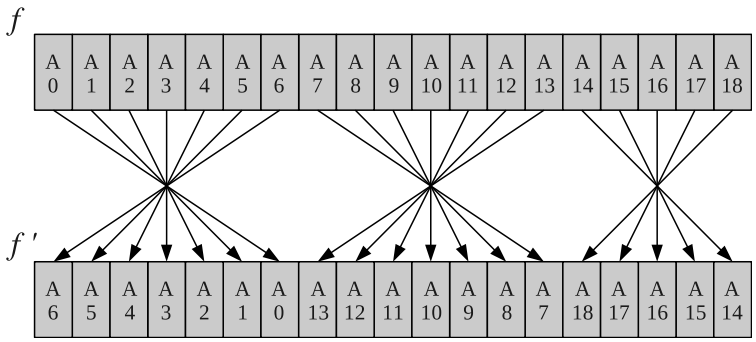


FIG. 2.60: Rotation des blocs de taille 7 dans le cadre de l'algorithme HGW modifié

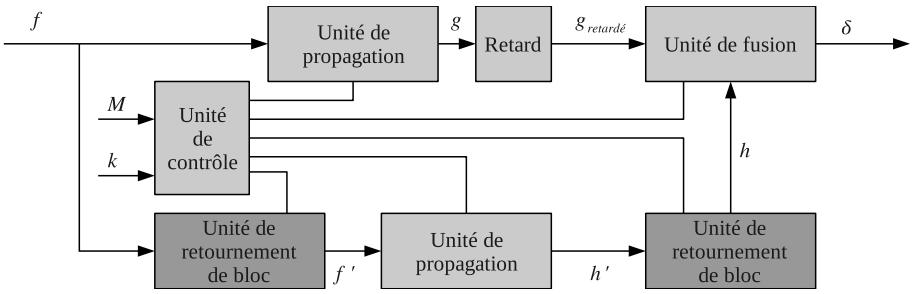


FIG. 2.61: L'architecture fonctionnelle flot de données pour l'algorithme HGW modifié

2.5.3.2 Architecture flot de données

Plusieurs architectures flot de données existent pour réaliser des érosions/dilatations, mais les mémoires utilisées n'ont pas une taille indépendante de la taille de l'image considérée. L'architecture présentée ici reprend l'approche HGW modifiée afin de produire une architecture où la seule dépendance, en terme de taille des mémoires, est relative à la taille maximale de l'élément structurant segment considéré.

L'architecture réalisée est complètement pipeline et produit un pixel résultat par cycle, sans interruption entre le traitement de deux lignes (de même taille) d'une image. Toutes les remarques citées précédemment relatives à l'approche HGW ont été prises en compte afin d'obtenir une unité matérielle réalisant des érosions/dilatations sans erreurs sur les bords.

L'unité de propagation doit propager les pixels dans le sens vidéo en considérant la taille k de l'élément structurant, mais aussi la taille M de la ligne. La figure 2.62 présente une vue simplifiée de l'unité. Elle est composée d'un compteur modulo k et d'un compteur de pixels modulo M. Ceux-ci commandent, via un comparateur, le multiplexeur permettant la réinitialisation de la propagation lorsqu'un nouveau bloc ou une nouvelle ligne se présentent.

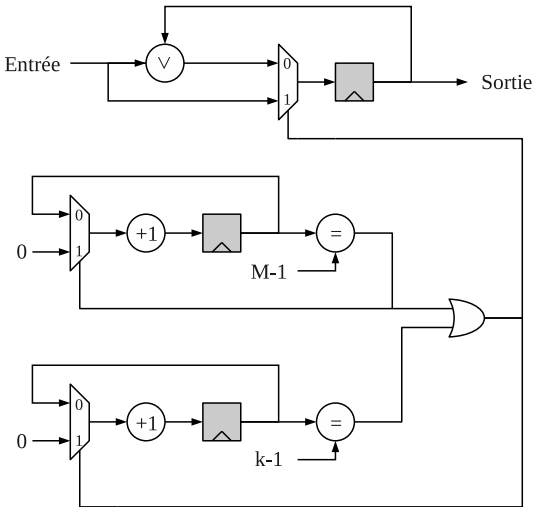


FIG. 2.62: Schéma simplifié de l'unité de propagation de l'architecture HGW modifiée produisant des dilatations de taille k sur des lignes de taille M

C'est sur l'unité de retournement des blocs qu'est basée la modification de l'algorithme HGW. Elle est capable de retourner des blocs de données de façon pipeline avec une cadence de un pixel par cycle, c'est-à-dire sans temps mort. Le retournement des blocs fonctionne de la manière suivante : pendant qu'un bloc n est écrit dans la mémoire, le bloc $n-1$ est lu dans le sens inverse. Ce mode de fonctionnement implique l'utilisation d'une mémoire double port avec d'un côté une écriture avec, par exemple, un décompteur et de l'autre une lecture avec un compteur.

Un problème subsiste lorsque M n'est pas un multiple de k (la taille de la ligne n'est pas un multiple de la taille de l'élément structurant), le dernier bloc étant plus court, la lecture de l'avant-dernier bloc n'est pas complète. Il faut gérer ce cas en écrivant dans un

autre ordre le dernier bloc pour qu'il puisse sortir lorsque la prochaine ligne commence. La figure 2.63 présente un exemple de retournement de blocs sur deux lignes.

La gestion de la mémoire hors du dernier bloc est réalisée de sorte que lorsqu'on écrit à une adresse $2^n + x$ (avec $x, k < 2^n$) on lit les données à l'adresse $k - x$. À l'arrivée d'un nouveau bloc, il suffit de faire le contraire, c'est-à-dire écrire en x et lire en $2^n + k - x$.

Lorsque l'on arrive en bout de ligne, il faut écrire les données avec des adresses décroissantes. Au début de la nouvelle ligne, on reprend l'écriture de façon standard, mais les dernières données de la ligne précédente doivent être lues juste avant l'écriture puisqu'écrites dans le même espace d'adresse. La mémoire double port à lecture prioritaire garantit que lorsqu'une adresse est présentée sur le port, il est possible de récupérer la valeur indexée avant de l'écraser.

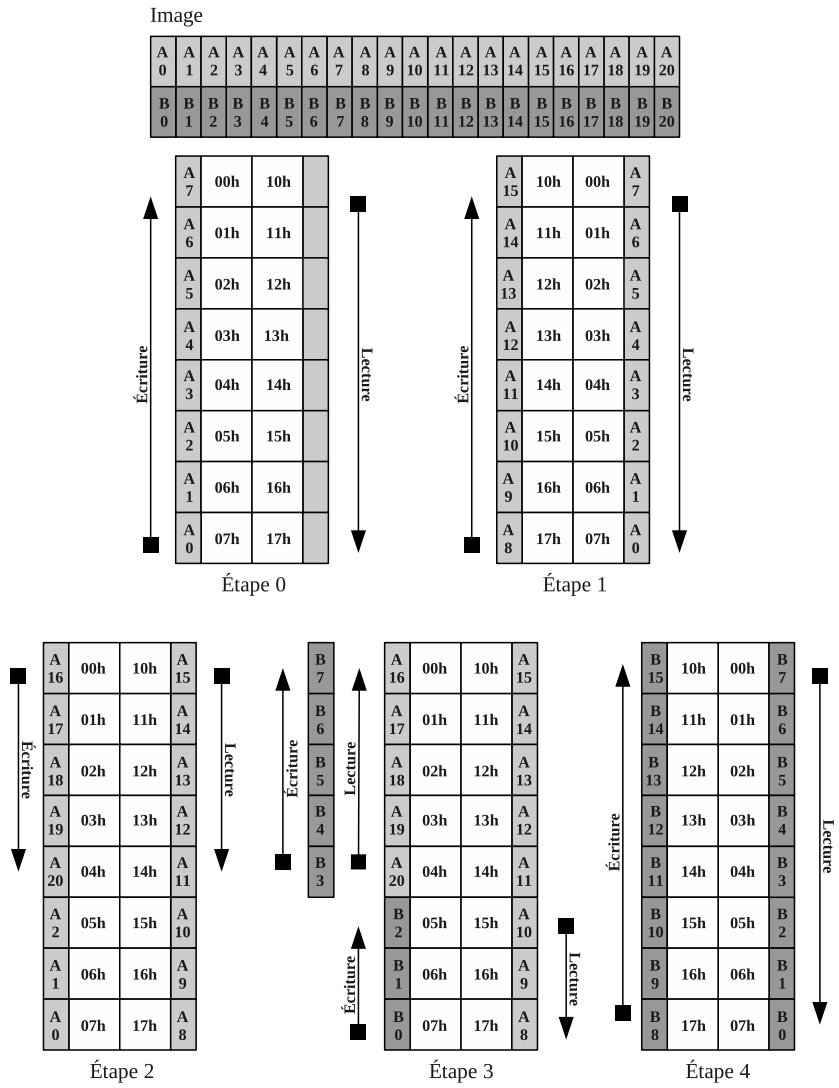


FIG. 2.63: État des mémoires de retournement de bloc de l'architecture HGW

Sur la figure 2.64 est présentée une vue simplifiée du système de retournement de blocs. On retrouve les compteurs de pixels modulo k ainsi que l'électronique de gestion des bancs

mémoires. Les multiplexeurs permettent, lorsque le dernier bloc se présente, de changer la logique d'écriture des pixels telle que présentée aux étapes 2 et 3 de la figure 2.64.

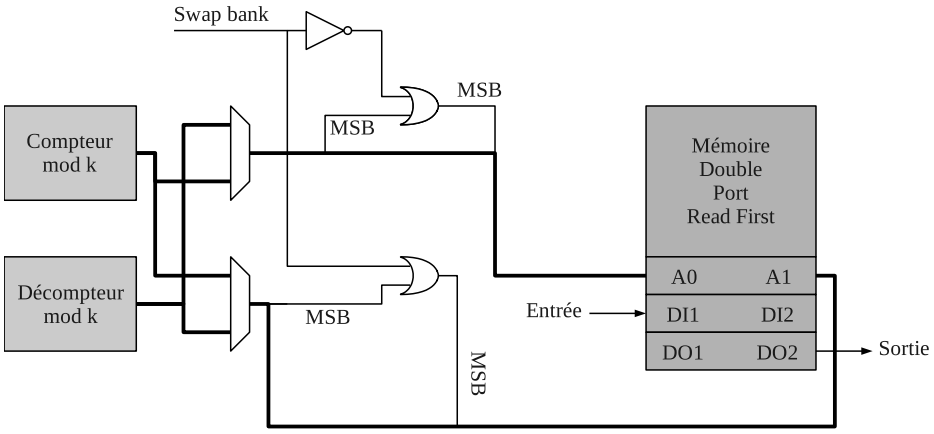


FIG. 2.64: Schéma fonctionnel de l'unité de retournement de blocs de l'architecture HGW

La latence, engendrée par la succession de deux unités de retournement, correspond à deux blocs de taille k. Afin de présenter les pixels de g et h de façon synchronisée, il est nécessaire de mettre en place une ligne à retard en sortie de la propagation produisant g. Sachant que dans le calcul de r, on accède à $g(x + \frac{k}{2})$ et à $h(x - \frac{k}{2})$, et que les pixels provenant de h arrivent deux blocs en avance par rapport à ceux de g, la taille de la ligne à retard doit avoir une taille équivalente à un bloc, soit k éléments.

La figure 2.65 montre comment est retardé g pour que les sorties des unités produisant g et h soient correctement synchronisées.

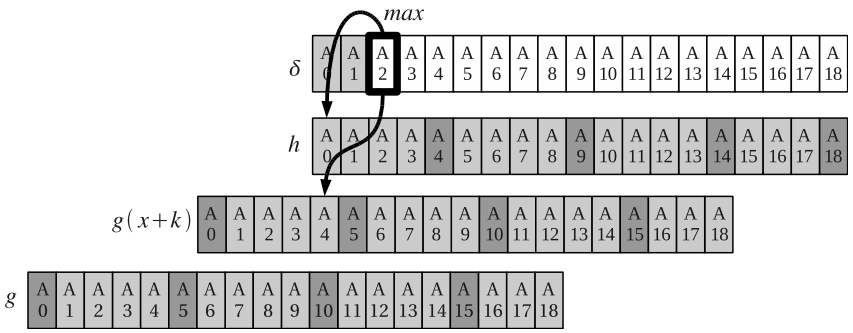


FIG. 2.65: Synchronisation des données de l'architectrure HGW modifiée

Plusieurs possibilités existent quant à la réalisation de cette unité. On peut utiliser une mémoire double port avec un port en écriture à une adresse x et un port en lecture à une adresse $x + k$. On peut également utiliser une mémoire simple port à lecture prioritaire avec une gestion d'adresse modulo k. Il est également possible, pour une taille maximale d'élément structurant pas trop importante, d'utiliser une mémoire distribuée sur le circuit. Ces possibilités dépendent évidemment du circuit visé (FPGA, ASIC, ...).

Les deux propagations réalisées, il est maintenant nécessaire de fusionner les pixels afin de produire le résultat de la dilatation. Cette opération suppose que les sorties pixels g

et h soient synchronisées. Cette unité dispose d'un compteur modulo M pour mettre en place la bonne politique de gestion des bords à chaque instant. Elle prend en entrée la taille de la ligne, le nombre de pixels à propager dans le *padding* et la taille de l'élément structurant. La figure 2.66 présente le schéma simplifié de cette unité. Le comparateur "Detection fin de ligne" indique lorsqu'il ne faut plus prendre en compte la sortie de g pour la ligne en cours ($x + \frac{k}{2} \geq M - 1$ avec x l'indice du pixel). Le comparateur "Détection *padding* propagé" indique jusqu'à quel indice doit être propagé le dernier maximum de g (lorsque $M \leq x + \frac{k}{2} < M + \text{PSA}$). Le comparateur "Détection début de ligne" sert à ne pas tenir compte de h au début d'une ligne (c'est à dire lorsque $x - \frac{k}{2} < 0$).

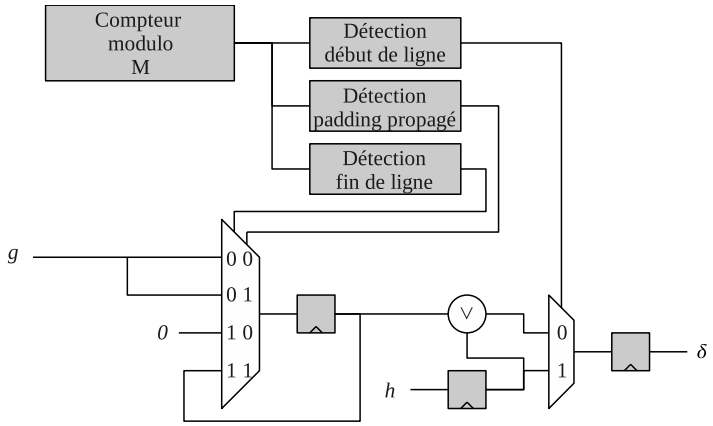


FIG. 2.66: Schéma simplifié de l'unité de fusion de l'architecture HGW réalisant une dilatation

La latence de cette architecture est proportionnelle à la taille de l'élément structurant, le temps de traitement d'une image peut donc varier quelque peu. Cette latence s'exprime de la façon suivante : $L = 3 \cdot \frac{k}{2} \cdot T_{clk}$ avec T_{clk} correspondant à la période en seconde du système et k la taille de l'élément structurant segment.

L'architecture proposée permet de disposer des premiers pixels résultats avant que la première ligne n'ait été totalement envoyée, contrairement à la mise en œuvre matérielle de l'algorithme de Lemonnier ou même lors de l'utilisation de processeurs de voisinage standard. Cette réduction de la latence est importante lors de la cascade d'un grand nombre d'opérateurs mais également lorsque plusieurs passes dans le même système sont nécessaires.

Si des a priori sur la taille des éléments structurants sont connus, il serait très intéressant de réduire la taille des mémoires afin de diminuer fortement la taille du circuit. À titre d'exemple le circuit à une surface de 260000 portes avec des ESS de taille 1024, mais avec des ESS de taille 128, la surface du circuit tombe à 38000 portes. La figure 2.67 montre l'évolution du nombre de portes en fonction de la taille maximale k en considérant une taille de ligne de 2048 pixels. On remarque que la solution proposée est pertinente, car la surface du circuit est moindre pour des éléments structurants strictement inférieurs à la taille d'une demi-ligne.

La surface occupée par la mémoire dans ces systèmes est plus importante que la logique. C'est la raison pour laquelle une architecture capable de se dispenser de coûteuses mémoires de lignes est pertinente. Il est possible avec ce système d'augmenter à moindres coûts la taille des images pouvant être traitées tout en ayant un nombre de portes largement

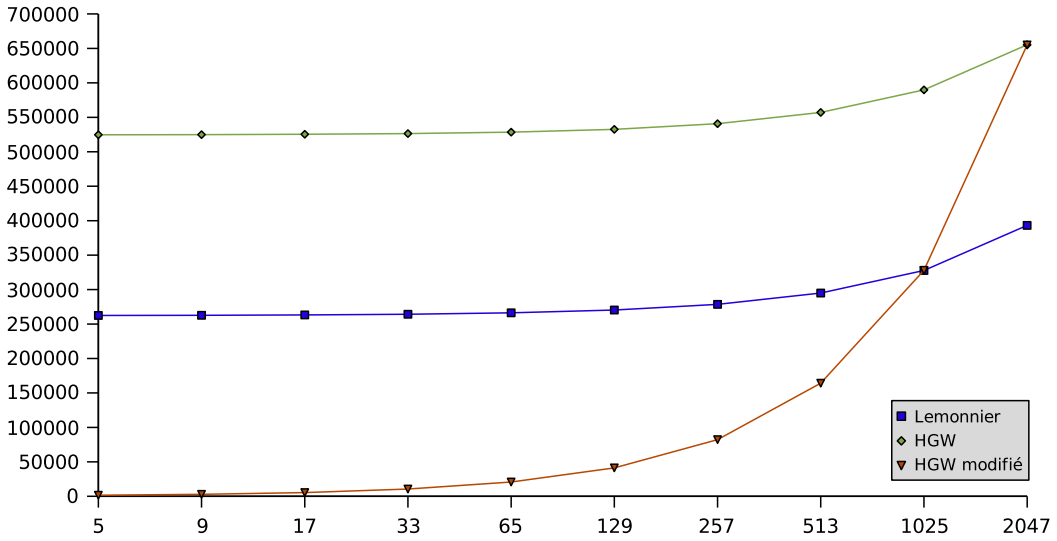


FIG. 2.67: Nombre de portes en fonction du k maximal

inférieur aux architectures actuelles, pour peu qu'on limite la taille maximale de l'élément structurant au quart de la taille de la ligne.

2.6 Conclusion

Nous avons montré ici une grande partie des calculs atomiques nécessaires en traitement d'image par morphologie mathématique. Ces opérateurs ne constituent pas une fin en soi et sont bien souvent une brique élémentaire d'opérateurs plus complexes.

De nouvelles structures de processeurs ont été proposées, comme des extracteurs de voisinages parallélisés où l'on dispose de N voisinages par cycles. Ce principe permet de voir les processeurs de voisinage sous un autre jour, car il est maintenant possible d'augmenter très fortement la puissance de calcul en considérant des bus mémoires plus larges. La nature très répétitive des calculs de base en morphologie mathématique est donc une véritable aubaine pour les systèmes considérant non plus une fréquence élevée de fonctionnement, mais plutôt un parallélisme massif. En effet, nous voyons apparaître aujourd'hui des mémoires dédiées aux cartes graphiques avec des débits de l'ordre de 150Go/s avec des mots de 512 bits. Si un processeur de voisinage parallélisé était capable d'exploiter ce débit mémoire, il serait possible de calculer 77672 érosions sur des images en résolution 1920×1080 en une seconde, soit un temps de calcul de $13\mu s$ par érosion. Bien sûr ce calcul semble décorrélé de la réalité, mais montre qu'une évolution possible du traitement d'image par morphologie mathématique est de considérer uniquement les opérations de base s'appuyant sur des mises en œuvre extrêmement véloces. Ceci dans le but de réaliser toutes les autres opérations comme des SKIZ où même des lignes de partage des eaux niveau par niveau.

La problématique majeure se dégageant de ce chapitre est l'inexistence d'une unité de calcul pouvant réaliser toutes les opérations décrites ici. Ces unités peuvent être très simples comme assez complexes, mais toujours limitées à un certain nombre d'opérations.

Il peut exister aussi plusieurs façons plus ou moins standard de réaliser un calcul, nous avons d'ailleurs déjà abordé ce sujet lors de la description des opérateurs géodésiques où plusieurs stratégies s'offraient à nous pour réaliser des reconstructions géodésiques. Nous avons montré qu'il était possible de réaliser une itération d'une reconstruction géodésique avec un seul processeur. Il est nécessaire dans ce cas d'employer une structure de processeur de voisinage très spécifique comportant à la fois des lignes à retard pour l'extraction des voisinages du marqueur et des lignes à retard pour acheminer le masque jusqu'à l'unité de calcul. Toutefois cette structure sous-exploite largement la logique présente dès lors que l'on réalise d'autres opérations comme de simples érosions. Il est alors préférable de privilégier une structure plus souple et d'utiliser plusieurs processeurs de voisinage raccordés en série et en parallèle via des ALU afin de réaliser la majeure partie des opérations décrites dans ce chapitre. Nous disposerions alors d'un pipeline plus ou moins profond où l'on pourrait chaîner des opérations basiques pour en construire de plus complexes.

Un problème subsiste concernant le choix des unités de calcul que l'on doit embarquer dans les processeurs de voisinage afin de réaliser un maximum de calculs. Les unités de tri seraient de bons candidats à la standardisation de notre structure de pipeline de processeurs de voisinage puisqu'il est possible de réaliser tous les filtres de rang. Cependant, la taille de ces unités, même réduites au maximum, est un frein à leur adoption. De plus, un des objectifs du pipeline peut être la construction de filtres avec des noyaux de grandes tailles décomposés sur un grand nombre de processeurs, or ce type de décomposition ne fonctionne d'une part que pour certains éléments structurants et d'autre part, qu'avec les érosions et les dilatations. Dans ce cas il est inutile de disposer de filtres de rang dans tous les étages du pipeline.

Nous avons analysé dans le détail les différentes structures de processeurs de voisinage et nous venons de formuler quelques propositions concernant leur chaînage en vue d'obtenir une amélioration significative des performances. Ce chapitre se situe à une échelle plutôt microscopique et nous allons passer, dans le chapitre suivant, à une échelle macroscopique. C'est-à-dire que nous allons cette fois analyser, sans se préoccuper de la structure interne d'un processeur, le chaînage d'opérateurs câblés dans des pipelines statiques et dynamiques, ceci dans le but de toujours répondre au mieux aux besoins applicatifs tout en garantissant une meilleure généricité des architectures proposées.

Chapitre 3

Chaînage de processeurs de voisinage

Les flots de processeurs de voisinage permettent d'exécuter les opérations élémentaires de morphologie mathématique de manière très efficace. Le nombre volontairement réduit d'opérations réalisables par processeur permet d'en limiter la surface et autorise leurs chaînages dans des flots profonds de processeurs. On peut alors construire des opérateurs complexes puisque le corpus théorique de la morphologie mathématique repose principalement sur la composition d'opérations basiques qui sont prises en charge par nos processeurs de voisinage.

Se pose alors le problème de la généricité du flot de processeurs, car le traitement opéré par les briques de calcul reste à un grain architectural assez élevé. Il peut être alors difficile de trouver une structure unique autorisant la réalisation de toutes les opérations de morphologie mathématique. Un autre paramètre du flot de processeurs est donc à prendre en compte, comme la régularité. Nous devons nous demander s'il est nécessaire de découper le flot de processeurs en plusieurs étages ayant une structure identique. En effet, un flot régulier est principalement avantageux dans le cas où l'architecture est fondue dans un circuit non reconfigurable, car il est possible de disposer d'un système plus souple, mais pas toujours optimal vis-à-vis de l'application. De la même manière, un flot profond de processeurs peut être difficile à employer, car l'application n'est pas toujours simplement distribuable sur ce dernier dans son ensemble, principalement à cause de dépendances entre les données. Il convient donc dans le cadre d'une structure régulière de bien analyser les contraintes des applications ou du domaine visé afin de correctement dimensionner la profondeur et la topologie du flot de processeurs.

Une alternative à une structure fixe des différents étages d'un flot de processeurs de voisinage consiste, dans le cadre de l'utilisation de circuits reconfigurables, à décrire la structure du flot de façon spécifique pour chacune des étapes de l'application afin de toujours disposer du matériel le plus efficace. On peut alors cibler un circuit à grain fin comme un FPGA afin d'y mettre en place la reconfiguration dynamique des accélérateurs. Toutefois, nous pouvons également mettre en place une reconfiguration avec un grain beaucoup plus important en essayant de fusionner toutes les descriptions des accélérateurs d'une ou plusieurs applications pour en générer une unique dont les ressources les plus consommatrices sont mutualisées et où les chemins de données nécessaires sont ajoutés.

Nous allons tout d'abord dans ce chapitre décrire comment mettre en œuvre un flot de processeurs et donc décrire la structure d'un système hôte capable de piloter un accélérateur

type flot de données à plusieurs entrées sorties. Une fois les bases du système hôte détaillées, nous pourrons ainsi mieux comprendre, dans une seconde partie, les limitations que l'on peut rencontrer en procédant à une description manuelle du flot de processeurs et qui orientent quelque peu la composition de l'accélérateur vers une structure visant la généricité pour des raisons de temps de développement. Nous terminerons ce chapitre, dans une troisième partie, par une présentation d'un système de description de haut niveau des flots de processeurs facilitant la mise en place de structures très particulières et optimales pour chaque opération avec des possibilités de fusion de différents accélérateurs.

3.1 Avant propos

Nous allons aborder trois aspects différents dans ce chapitre. Nous allons commencer par décrire comment s'insère un accélérateur flot de données au sein d'un circuit généraliste et en particulier comment réitérer avec les mêmes données plusieurs passes de calcul dans un accélérateur flot de données. En effet, ce dernier point est crucial dans le cadre du traitement d'image et en particulier dans le cadre de la morphologie mathématique car, c'est en agrégeant une grande quantité de traitements simples que l'on peut réaliser des applications complexes. L'interfaçage générique d'IP flots de données, tel que celui proposé par Fraboulet & Risset [30], permet de disposer d'une interface simple et évolutive en termes de nombre de flots de données vers l'accélérateur.

Une fois la problématique d'utilisation d'un flot de processeurs au sein d'un SoC traitée, nous pourrons d'abord proposer quelles options nous avons à notre disposition pour rendre plus souple d'utilisation et plus versatile un SoC composé d'un pipeline d'opérateurs flots de données. Nous détaillerons ici à la fois les aspects liés à la description manuelle d'un pipeline, aux différents niveaux de granularité de reconfiguration envisageable ainsi qu'à la description de haut niveau.

La description de haut niveau est là pour fournir un moyen simple d'agréger les composants de bases d'un étage de flots de processeurs que nous appellerons *pattern*. En effet, un *pattern* est une agrégation de composants tels que des processeurs de voisinage, des unités arithmétiques, des unités de seuillage ou bien encore des multiplexeurs. L'agencement de ces composants via un outil de haut niveau permet d'accélérer le développement de pipelines spécifiques à des applications. Nous ciblons majoritairement les circuits FPGA disposant d'une capacité de reconfiguration dynamique partielle [74], mais le travail proposé ici pourrait être utilisé sur des architectures types ambric [20] composées de centaines de coeurs de processeurs simples reliés entre eux via des interconnexions programmables et utilisant un modèle de programmation orienté composant.

Des outils tels que Gaut [21], dont l'objectif principal est de synthétiser des IP matérielles à partir d'un langage tel que le C, peuvent être utilisés afin de générer simplement l'agencement des composants dans un *pattern*. Nous avons toutefois décidé de spécifier plus simplement un *pattern* afin de valider les concepts de fusion. Cette technique est inspirée de ce que l'on peut trouver aujourd'hui dans le monde des ASIP lorsque plusieurs descriptions matérielles d'instructions spécialisées sont fusionnées pour mutualiser les ressources disponibles sur le silicium [17], [58], [85], [84]. Ce principe, appelé *Compound Circuit*, a été mis en place, à une granularité plus élevée, au niveau des *patterns* servant à la description

d'un pipeline d'opérateurs flot de données et permet de mutualiser les ressources. Cette technique permet de disposer de toutes les opérations proposées par les différents pipelines au sein d'un unique accélérateur de taille réduite.

3.2 Gestion d'un accélérateur flot de données

3.2.1 Problèmatique

Généralement, les processeurs de voisinage sont utilisés de manière assez standard juste après un imageur pour réaliser quelques corrections sur l'image, car cette dernière est souvent bruitée en sortie du capteur. Le processeur est donc utilisé dans une mode flot de données *pure* où il n'est pas possible de réinjecter en entrée les calculs sortant du flot. Ce mode de fonctionnement est adapté à la mise en place d'opérations simples d'amélioration des images, mais dans notre cas, nous ciblons plutôt une utilisation de l'accélérateur pour des traitements plus complexes.

Une application de traitement d'images peut être difficilement déployable dans sa totalité sur un flot de processeurs pour plusieurs raisons. Premièrement, l'application peut faire appel à une très grande quantité d'opérateurs et nécessiterait ainsi une surface de silicium beaucoup trop importante si elle était déployée totalement dans un flot profond d'opérateurs. Deuxièmement, la dépendance entre les données et en particulier l'utilisation d'opérateurs de réduction rend impossible le déploiement matériel complet d'une application. Un exemple est donné en figure 3.1 où la nécessité de calculer le maximum et le minimum global sur l'image nous contraint de casser le flot de calculs. Il n'est pas possible de mettre en œuvre cette application sans stocker une image intermédiaire, car le calcul du maximum global est prêt uniquement lorsque tous les pixels ont été envoyés. L'opérateur utilisant ce résultat global ne peut donc pas fonctionner en pipeline rendant impossible la mise en place complète de l'application dans un seul flot d'opérateurs.

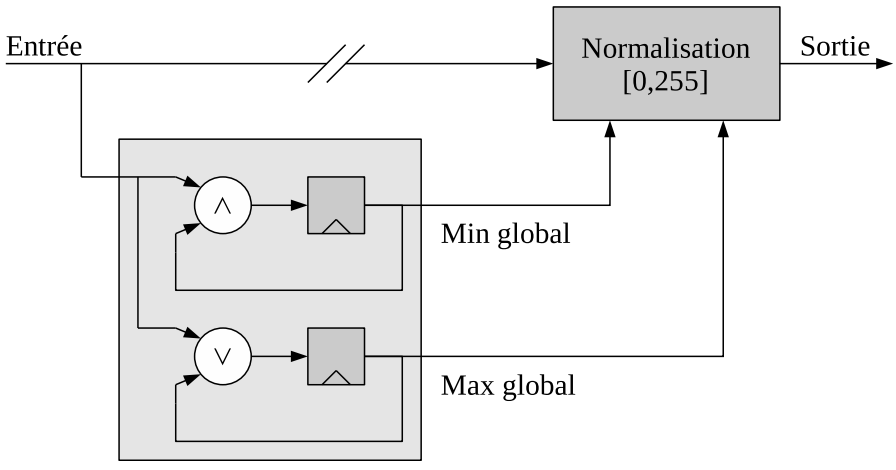


FIG. 3.1: Normalisation d'images : le flot de calcul est interrompu par la présence d'opérations de réduction

La mise en place d'un accélérateur type flot de données avec la possibilité de réaliser des stockages intermédiaires est donc impérative. Il faut ainsi prévoir l'intégration de l'accélé-

rateur dans un système plus général que l'on appellera hôte et qui est capable d'amorcer des transferts vers et depuis une mémoire de stockage des images intermédiaires.

3.2.2 Intégration dans un système sur puce

La gestion de l'accélérateur dans un système sur puce (SoC) est un moyen simple et efficace d'intégrer un flot de processeurs de voisinage. On dispose ainsi, dans le même circuit, d'un processeur généraliste, de contrôleurs mémoires et pourquoi pas de périphériques de captures d'images autorisant une utilisation optimale de l'accélérateur. En effet, un tel système élimine les problèmes de gestion du flot de données cités dans le section précédente et permet le rebouclage des traitements.

Nous avons envisagé deux architectures fonctionnelles décrites ci-après. La première permet une intégration simple, mais conduit à une surcharge du bus principal du circuit et la seconde est plus complexe, mais permet des performances plus élevées tout en ne monopolisant pas le bus principal du processeur. En effet, la congestion du bus principal par les DMA peut être problématique surtout si le processeur accède à ses instructions et ses données en mémoire principale sans cache ni tampon.

3.2.2.1 Intégration simple

L'objectif ici est d'intégrer l'accélérateur de la manière la plus standard possible en l'interfaçant directement sur le bus principal du SoC et où l'on retrouve les composants de ce dernier : processeurs, DMA, contrôleur mémoire...

La figure 3.2 présente la vue fonctionnelle du circuit. On retrouve le processeur généraliste avec ses mémoires statiques d'instructions et de données. Cette dernière est de faible taille et doit être uniquement considérée comme une zone de travail temporaire où l'on peut stocker uniquement que quelques lignes d'une image. Le stockage des images complètes est assuré par la mémoire dynamique externe au circuit et deux composants sur le bus autre que le processeur généraliste peuvent y écrire des données : le *DMA* et la logique d'acquisition de signaux vidéo.

Une application déployée sur cette architecture se déroule de la manière suivante : une image est automatiquement acquise sur ordre du processeur généraliste et est stockée en mémoire externe. Une interruption est déclenchée lorsque cette opération est terminée. Le processeur généraliste peut alors programmer les opérations à réaliser dans l'accélérateur et orchestrer la programmation des DMA pour transmettre avec un flux le plus tendu possible l'envoi et la réception des images dans l'accélérateur. Les FIFO permettent une bonne synchronisation des données à traiter par le flot de processeurs lorsque plusieurs images sont nécessaires à une opération. Dès lors que tous les pixels sont ressortis de l'accélérateur et sont stockés en mémoire externe, le processeur peut soit passer à l'étape suivante de l'application soit demander l'acquisition d'une nouvelle image si toutes les opérations sur une trame ont été réalisées.

On remarque que ce mode de fonctionnement n'est pas optimal, car il n'est pas possible de recevoir une image venant d'une caméra et dans le même temps utiliser l'accélérateur, et ce, pour deux raisons : la mémoire externe est partagée entre l'accélérateur et le système d'acquisition vidéo et le bus du SoC est également partagé entre tous les périphériques. Un

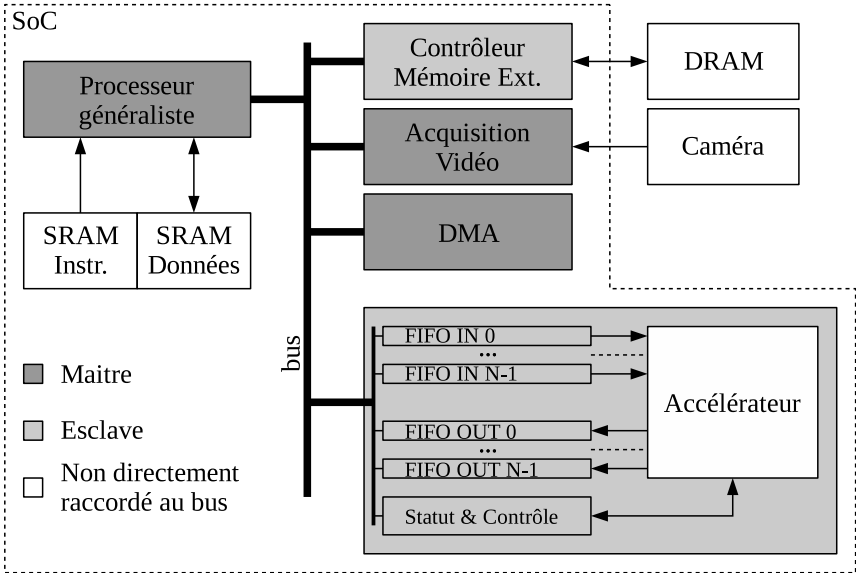


FIG. 3.2: Intégration simple d'un accélérateur type flot de processeurs dans un SoC

tel système trouve son intérêt si, d'une part, les contraintes en termes de temps de calcul des applications ciblées ne sont pas fortes et, d'autre part, si le coût du système est un facteur important, comme c'est dans le cas des applications automobiles.

3.2.2.2 Intégration avancée

Une intégration plus complexe peut être réalisée en reprenant la structure simple et en y ajoutant une mémoire pouvant contenir plusieurs images locales à l'accélérateur. Cette structure est présentée en figure 3.3. Ainsi lorsque toutes les étapes d'une opération sur une même image n'ont pu être réalisées complètement en une passe dans l'accélérateur, il est possible de stocker la ou les images en mémoire locale pour réitérer une nouvelle passe dans le flot de processeurs. Pendant ce temps le processeur généraliste est libre de l'utilisation du bus et peut très bien demander l'acquisition d'une nouvelle image sans perturber la deuxième passe de calcul dans l'accélérateur.

Cette structure est plus coûteuse à la fois matériellement, car il est nécessaire d'ajouter une deuxième mémoire externe ainsi que toute la logique de contrôle supplémentaire, mais également logiciellement, car l'ordonnancement des calculs sur l'accélérateur devient beaucoup plus complexe afin de paralléliser les acquisitions d'images et les traitements. On expose alors le parallélisme de gestion des images à l'utilisateur et on pourrait chercher à optimiser automatiquement cette fonction un peu comme est géré le pipeline logiciel dans les processeurs VLIW.

3.2.3 Synchronisme et gestion logicielle

La gestion logicielle d'un accélérateur type flot de données est une tâche assez fine dès que plusieurs flots de données en parallèle doivent être pris en compte. En effet, les composants de l'accélérateur situés aux mêmes étages du pipeline fonctionnent de manière

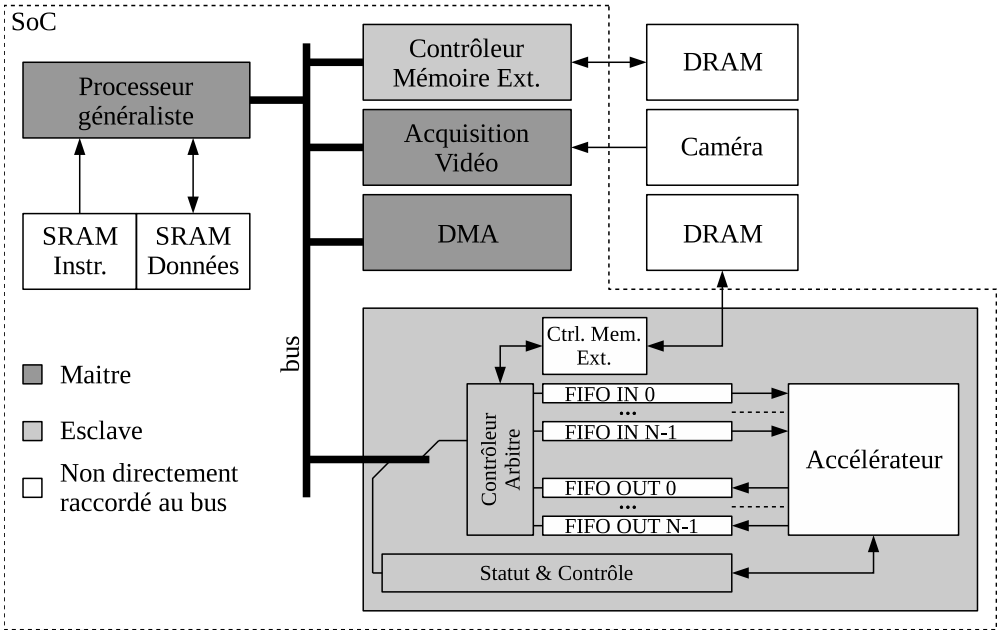


FIG. 3.3: Intégration avancée d'un accélérateur type flot de processeurs dans un SoC

synchrone et doivent traiter les pixels de mêmes indices aux mêmes instants. Ceci n'est vrai que si des échanges de flux doivent être effectués entre les composants disposés en parallèle, dans le cas contraire la gestion est assez simple et peut être considérée de la même façon que lors de l'acheminement d'un flot unique de données où finalement aucune FIFO en entrée n'est nécessaire.

La figure 3.4 illustre, pour une architecture capable de traiter deux flots de données en parallèle, le cas où il est nécessaire de synchroniser les flux avant l'acheminement vers l'accélérateur et le cas où cette synchronisation n'est pas nécessaire. En effet, dans l'accélérateur 1, la présence d'un opérateur de soustraction relié aux deux branches parallèles réalisant une érosion et une dilatation impose que les flux présentés aux entrées E0 et E1 soient parfaitement synchrones. De manière duale, il n'est pas nécessaire pour l'accélérateur 2 de présenter des flux synchronisés aux entrées E1 et E2 puisqu'il n'existe pas d'opérateurs mettant en jeu des calculs à partir des flux des deux branches d'opérateurs.

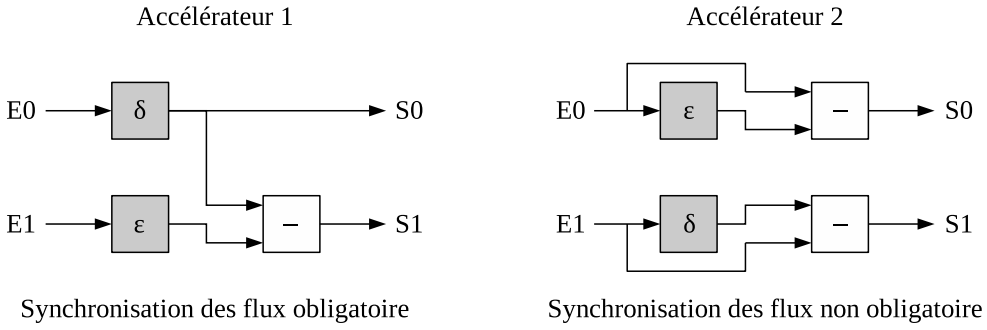


FIG. 3.4: Gestion du synchronisme dans des accélérateurs multi flot de données

Nous allons maintenant nous placer dans le cas où plusieurs flots de données doivent être envoyés et reçus en parallèle puisque nous ne faisons pas d'hypothèse spécifique quant à la structure interne des accélérateurs et nous considérons alors le pire cas. Une barrière de synchronisation est nécessaire avant les entrées de l'accélérateur pour garantir le synchronisme des flots de données avant l'accélérateur. Nous supposons en outre que l'accélérateur ne désynchronise pas en interne les flots de données. Chaque entrée de l'accélérateur doit être précédée d'une FIFO stockant quelques centaines de pixels et informant le contrôleur de l'accélérateur de leurs niveaux de remplissage. En effet, dès qu'une des FIFO en entrée est vide il est nécessaire de *geler* l'accélérateur. Il est également impératif de placer des FIFO en sorties de l'accélérateur pour stocker momentanément les résultats des calculs. Ces dernières doivent permettre également au contrôleur de geler l'accélérateur lorsqu'elles sont pleines. Nous mettons donc en place un verrou global qui gèle l'accélérateur dès qu'une FIFO d'entrée ou de sortie est pleine. La figure 3.5 présente l'architecture de contrôle du synchronisme autour de l'accélérateur.

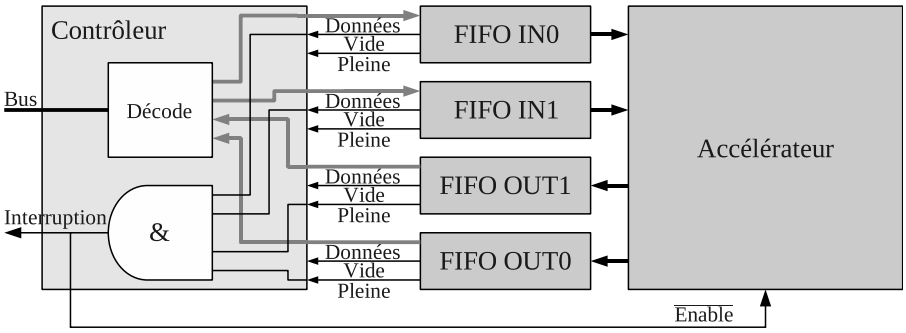


FIG. 3.5: Controleur d'un accélérateur multiflot de données

Ce verrou matériel simplifie grandement la gestion logicielle dédiée à l'alimentation des accélérateurs en données puisqu'il est possible d'accéder aux FIFO directement depuis un DMA programmé pour envoyer ou recevoir quelques lignes d'une image. On peut donc écrire et lire alternativement par paquet de lignes dans les FIFO sans risquer de désynchroniser les flux de données vers l'accélérateur. Le processeur généraliste se charge du contrôle et ordonne les transferts DMA pour minimiser les périodes d'inactivité de l'accélérateur. Le contrôleur tient informé le processeur soit par un mécanisme d'interruption matérielle soit en changeant l'état de registres de statut que le processeur vient consulter à intervalle régulier.

3.3 Description manuelle

3.3.1 Intérêts et inconvénients

Nous sommes capables de mettre en place un accélérateur flot de données avec plusieurs entrées et plusieurs sorties au sein d'un système complexe mettant en jeu un processeur généraliste ainsi que tous les composants habituellement associés : DMA, contrôleur mémoire... Un tel système nous permet de réaliser plusieurs traitements séquentiels d'une

même donnée et nous donne la possibilité d'homogénéiser et de réduire la taille de l'accélérateur. En effet, il n'est plus nécessaire de déployer matériellement toutes les étapes d'une application, mais il est plutôt nécessaire de trouver une structure matérielle commune à plusieurs sous-parties de l'application. Il convient alors de procéder à une description des opérateurs à employer ainsi que des chemins de données à mettre en place. On considère d'ailleurs que le découpage d'une application en sous-parties peut produire des blocs non homogènes et que l'on tombe alors dans un cas équivalent multiapplications.

Une description manuelle d'un accélérateur utilisant les briques décrites au chapitre 2 permet de disposer, pour une application spécifique, d'une structure optimale à la fois en termes d'utilisation de ressources et en termes de contraintes temporelles. Si l'optimalité est un paramètre prépondérant dans le système, il est donc nécessaire de procéder à une nouvelle description d'un accélérateur pour toutes nouvelles applications à réaliser. Au contraire, si l'on peut se satisfaire d'un accélérateur quasi optimal, nous pouvons procéder à une nouvelle description plus générique capable d'exécuter plusieurs applications différentes.

On observe alors deux niveaux de granularité dès que l'on souhaite traiter plusieurs applications avec le même matériel. L'utilisation de structures optimales impose de mettre en place un mécanisme de reconfiguration à grain fin type FPGA où l'on peut venir reconfigurer l'accélérateur en fonction de l'application ciblée. Cette granularité ne nous affranchit pas de la nécessité de décrire matériellement un accélérateur pour chaque grande étape d'une application. La mise en place d'une description plus générique d'un accélérateur prenant en charge un ensemble d'applications correspond à une architecture reconfigurable gros grains où l'on vient modifier principalement des chemins de donnés pour *sélectionner* une application parmi celles réalisables.

Les figures 3.6 et 3.7 présentent les deux niveaux de granularité pouvant être mis en place dans le cadre d'une opération de gradient morphologique et d'une opération d'ouverture morphologique. Dans le premier cas, l'utilisateur procède à une description manuelle structurelle de chaque opération en utilisant des briques matérielles définies au chapitre 2. Pour chacune de ces descriptions, un fichier de configuration est généré et est stocké sur la plateforme. Le contrôleur du système hôte peut donc charger, en fonction des besoins, une des deux configurations à disposition dans la zone reconfigurable. Dans le second cas, les deux flots d'opérateurs sont fusionnés manuellement en économisant les ressources disponibles et en ajoutant les différents chemins de données possibles. Cette étape réalisée, l'utilisateur procède ensuite à une description manuelle structurelle du résultat de la fusion pour disposer d'un unique accélérateur ayant la possibilité de changer de configuration en manipulant uniquement les multiplexeurs et les différents éléments de configurations inhérents aux composants mis en place.

3.3.2 Exemple d'application : chaînage dédié à la quasi-distance

La quasi-distance [12] est une transformation de type distance adaptée aux images en niveaux de gris. Elle est inscrite dans le cadre plus générique des transformations résiduelles. Une transformation résiduelle θ sur un ensemble $\mathbf{X}$ est définie à l'aide de deux familles de transformations ψ_i et ζ_i, dépendant d'un paramètre $i \in \mathbf{I}$.

Les éléments des familles ψ_i et ζ_i sont appelés primitives et on définit le résidu de taille

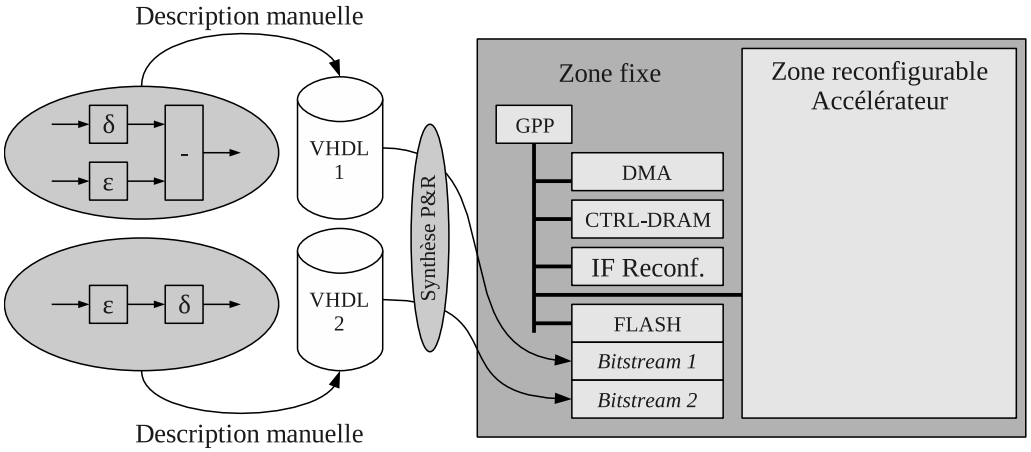


FIG. 3.6: Mise en œuvre d'une reconfiguration à grain fin dans un contexte multiapplications

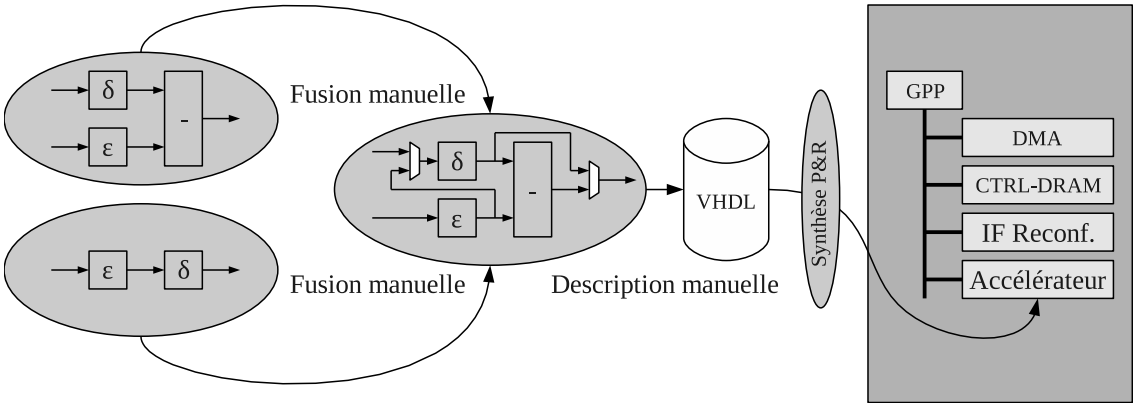


FIG. 3.7: Mise en œuvre d'une reconfiguration à gros grains dans un contexte multiapplications

i comme :

$$r_i = \psi_i / \zeta_i$$

La transformation résiduelle est alors définie comme étant l'union des résidus :

$$\theta_i = \bigcup_{i \in \mathbf{I}} \psi_i / \zeta_i$$

Ces formules sont très généralistes et voici quelques exemples d'utilisation :

– Érodée ultime [46] obtenue en utilisant pour ψ_i une érosion ε_i de f et pour ζ_i une ouverture élémentaire par reconstruction de l'érodée ε_i :

$$\psi_i = \varepsilon_i$$
$$\zeta_i = R_f^{\delta}(\varepsilon_i)$$

– Le squelette par boule maximale obtenu en utilisant pour ψ_i une érosion ε_i de f et pour ζ_i une élémentaire de l'érodée ε_i :

$$\psi_i = \varepsilon_i$$
$$\zeta_i = \varphi(\varepsilon_i)$$

Ces transformations résiduelles sont accompagnées d'une fonction associée q qui nous informe en chaque point x de θ l'indice du résidu r_i.

$$q(x) = i + 1, \text{ avec } x \in r_i$$

Dans le cadre de l'érosion ultime, la fonction associée nous informe de la taille des érodés et dans le cadre du squelette par boule maximale, cette fonction nous informe de la taille des boules en chaque point.

La transformation distance est définie par la fonction associée obtenue en opérant une transformation résiduelle dont r_i est obtenue par soustraction ensembliste et dont les primitives sont des érosions de tailles croissantes :

$$\psi_i = \varepsilon_i$$
$$\zeta_i = \varepsilon_{i+1}$$

Dans le cas binaire, les primitives sont choisies de manière à ce que la fonction associée soit unique pour chaque point de θ. Dans le cadre numérique ce choix est beaucoup plus complexe, car il faut d'une part trouver un équivalent numérique à la soustraction ensembliste et d'autre part prendre en compte le fait que les résidus ne sont plus disjoints entre les itérations i et $i + 1$. Une possibilité pour le calcul de la fonction associée est de s'intéresser à la dernière occurrence du résidu maximal [12] :

$$q(x) = \bigvee_{i \in \mathbf{I}} i + 1, \text{ avec } x \in \bigvee_{i \in \mathbf{i}} r_i$$

Les primitives employées dans le cadre de la quasi-distance sont les mêmes que celles employées pour obtenir la fonction associée produisant la transformation distance. On définit alors la transformation résiduelle numérique comme étant le suprémum des différences des primitives :

$$r_i = |\psi_i - \zeta_i|$$
$$\theta_i = \bigvee_{i \in \mathbf{I}} \psi_i - \zeta_i$$

Un algorithme de calcul de la transformation résiduelle est donné en 4.

Dans le cadre de la quasi-distance, le chevauchement des résidus aux itérations i et $i+1$ conduit à une fonction associée q présentant des irrégularités et nous faisant perdre la propriété 1-lipschitzienne de la transformation distance. C'est-à-dire que l'on ne peut plus garantir :

$$|q(x) - q(y)| \leq \|x - y\|$$

Il convient alors de procéder à une régularisation de la fonction associée détaillée à l'algorithme 5.

Algorithme 4 Quasi-distance : calcul de la fonction associée, Beucher [11]

ENTRÉES: $\mathtt{I}$
SORTIES: $\mathtt{I}_{\mathtt{tresiduelle}}$, $\mathtt{I}_{\mathtt{associée}}$

$\mathtt{I}_{\mathtt{tresiduelle}} \leftarrow 0$
$\mathtt{I}_{\mathtt{associée}} \leftarrow 0$
$\mathtt{I}_{\mathtt{tmp1}} \leftarrow \mathtt{I}$

$\mathtt{v1} \leftarrow \mathtt{Volume}(\mathtt{I}_{\mathtt{tmp1}})$
$\mathtt{v2} \leftarrow \mathtt{v1} + 1$
$\mathtt{i} \leftarrow 0$

Tantque v2>v1 **Faire**
  $\mathtt{i} \leftarrow \mathtt{i} + 1$
  $\mathtt{v2} \leftarrow \mathtt{v1}$

  $\mathtt{I}_{\mathtt{tmp2}} \leftarrow \varepsilon(\mathtt{I}_{\mathtt{tmp1}})$
  $\mathtt{I}_{\mathtt{tmp1}} \leftarrow \mathtt{I}_{\mathtt{tmp1}} - \mathtt{I}_{\mathtt{tmp2}}$

  $\mathtt{I}_{\mathtt{tmp3}} \leftarrow (\mathtt{I}_{\mathtt{tmp1}} \geq \mathtt{I}_{\mathtt{tresiduelle}} \,?\, \mathtt{i} : 0)$
  $\mathtt{I}_{\mathtt{tmp3}} \leftarrow (\mathtt{I}_{\mathtt{tmp1}} \neq 0 \,?\, \mathtt{I}_{\mathtt{tmp3}} : 0)$

  $\mathtt{I}_{\mathtt{associée}} \leftarrow (\mathtt{I}_{\mathtt{tmp3}} \vee \mathtt{I}_{\mathtt{associée}})$
  $\mathtt{I}_{\mathtt{tresiduelle}} \leftarrow (\mathtt{I}_{\mathtt{tmp1}} \vee \mathtt{I}_{\mathtt{tresiduelle}})$

  $\mathtt{I}_{\mathtt{tmp1}} \leftarrow \mathtt{I}_{\mathtt{tmp2}}$

  $\mathtt{v1} \leftarrow \mathtt{Volume}(\mathtt{I}_{\mathtt{tmp1}})$
Fin tantque

La figure 3.8 présente un exemple d'image ayant subi une transformation résiduelle de type quasi-distance. On remarque effectivement ici que le suprémum des résidus présente assez peu d'intérêt comparé à l'image de la fonction associée. On remarque que les zones claires de l'image faiblement contrastées présentent dans l'image de la fonction associée des distances assez élevées.

Image originale

Suprémum des résidus θ

Fonction associée q non régularisée

Fonction associée q régularisée

FIG. 3.8: Exemple de quasi-distance en maille carrée

Algorithme 5 Quasi-distance : régularisation de la fonction associée, Beucher [11]

ENTRÉES: $I_{associée}$
SORTIES: I_{qdist}

$I_{qdist} \leftarrow I_{associée}$

$v1 \leftarrow Volume(I_{qdist})$
$v2 \leftarrow v1 + 1$

Tantque $v2 > v1$ **Faire**
  $v2 \leftarrow v1$

  $I_{tmp1} \leftarrow \varepsilon(I_{qdist})$
  $I_{tmp2} \leftarrow I_{tmp1} + 1$

  $I_{qdist} \leftarrow (I_{tmp2} \wedge I_{qdist})$

  $v1 \leftarrow Volume(I_{qdist})$
Fin tantque

Les algorithmes présentés ici pour le calcul des transformations résiduelles et en particulier pour le calcul de la quasi-distance sont très itératifs, mais ont l'avantage d'utiliser des opérations arithmétiques, logiques et de voisinage simplement parallélisables. Il existe une mise en œuvre de ces algorithmes par une approche hybride avec une structure de données type files d'attente[28]. Cette approche réalise les premières étapes du calcul telles que décrites dans les algorithmes 4 et 5 pour ensuite basculer sur une approche à base de files. Cette méthode reprend le principe utilisé dans les reconstructions géodésiques hybrides [80]. Toutefois, il est difficile et même parfois impossible d'avoir un critère fiable permettant de savoir quand basculer d'une mise en œuvre classique vers une mise en œuvre par file d'attente. En outre, des structures de données type file sont facilement utilisables dans un processeur généraliste, mais requièrent une mise en œuvre très, voire trop, spécialisée dans un circuit dédié.

Nous avons donc choisi de décrire cet algorithme en réalisant un *pattern* d'architecture dédié à une itération du calcul de la fonction associée (algorithme 4) et à une itération de la régularisation de la fonction associée (algorithme 5). En fusionnant ces deux descriptions et en chaînant k fois cette dernière il est possible de réaliser k itérations en n'envoyant qu'une seule fois le flot de données. Lorsque nous parlons de fusion, nous sous-entendons la fusion des descriptions matérielles et non la fusion des algorithmes, c'est-à-dire que l'architecture résultante recevra d'abord n fois le flot de données pour réaliser le calcul de la fonction associée pour seulement ensuite recevoir m fois un autre flot de données afin de réaliser la phase de régularisation.

Les figures 3.9 et 3.10 présentent la description matérielle d'un étage du calcul de la quasi-distance et d'un étage de la régularisation. On remarque la présence de *NOP* qui se traduisent de manière matérielle, soit en termes de registres si la synchronisation implique des briques de calcul à faible latence, soit en termes de lignes à retard si la synchronisation implique des briques de calcul à forte latence, comme des processeurs de voisinage. Le processeur de voisinage utilisé dans ces deux descriptions permet d'économiser un *NOP*

car un processeur de voisinage peut aisément ressortir à la fois le résultat du calcul et la valeur du pixel source. Les unités arithmétiques et logiques (ALU) prennent en entrées deux flux de données et si nécessaire une constante. Elles réalisent des calculs simples tels que des additions, des soustractions ou bien encore des recopies conditionnelles (*cmove*) de valeurs.

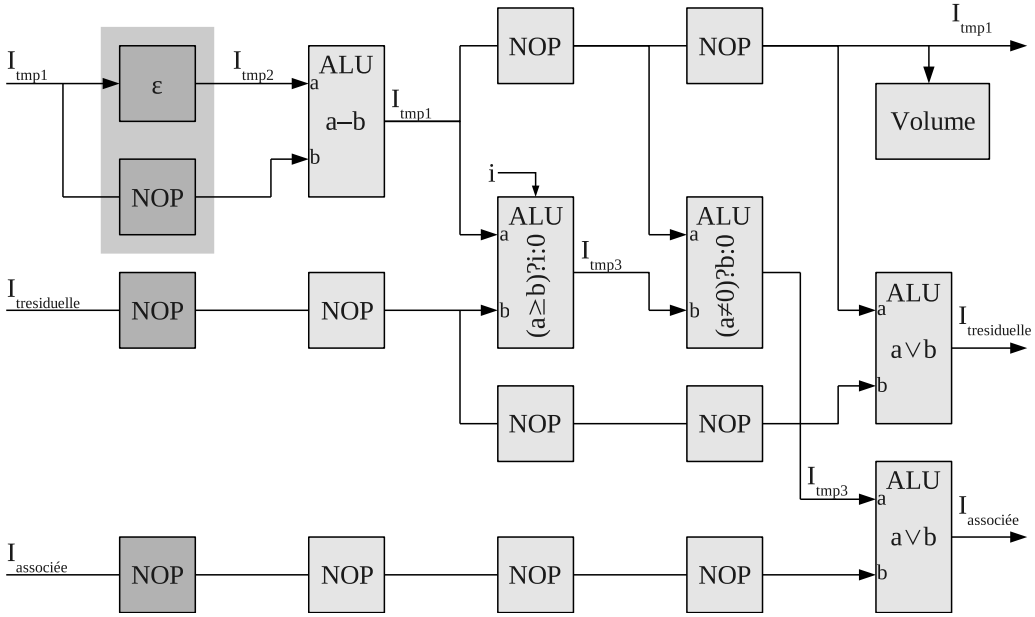


FIG. 3.9: Structure d'un étage matériel du flot de calcul dédié à la quasi-distance

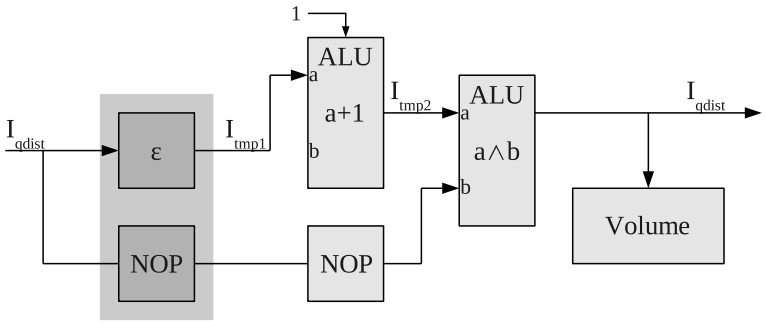


FIG. 3.10: Structure d'un étage matériel du flot de calcul dédié à la régularisation de la quasi-distance

L'opération de fusion manuelle du *pattern* d'architecture consiste à rechercher manuellement, parmi les deux descriptions des figures 3.9 et 3.10, les ressources communes et ajouter les chemins de données nécessaires à la reconfiguration d'un *pattern* à l'autre. Plusieurs solutions optimales existent et une version est proposée en figure 3.11. Nous supposons dans cette version pouvoir également fusionner, dans une même ALU, les opérations nécessaires aux deux utilisations du flot d'opérateurs. Si tel n'était pas le cas, il serait obligatoire d'ajouter les ALUs nécessaires en plus des chemins de données et du multiplexeur déjà présent. De la même manière, si les opérateurs à fusionner n'impliquaient pas uniquement des

opérations de voisinages identiques, on peut supposer aisément qu'un processeur réalisant une érosion, dont l'élément structurant ne dépasse pas une taille critique, peut également être en mesure de réaliser une dilatation.

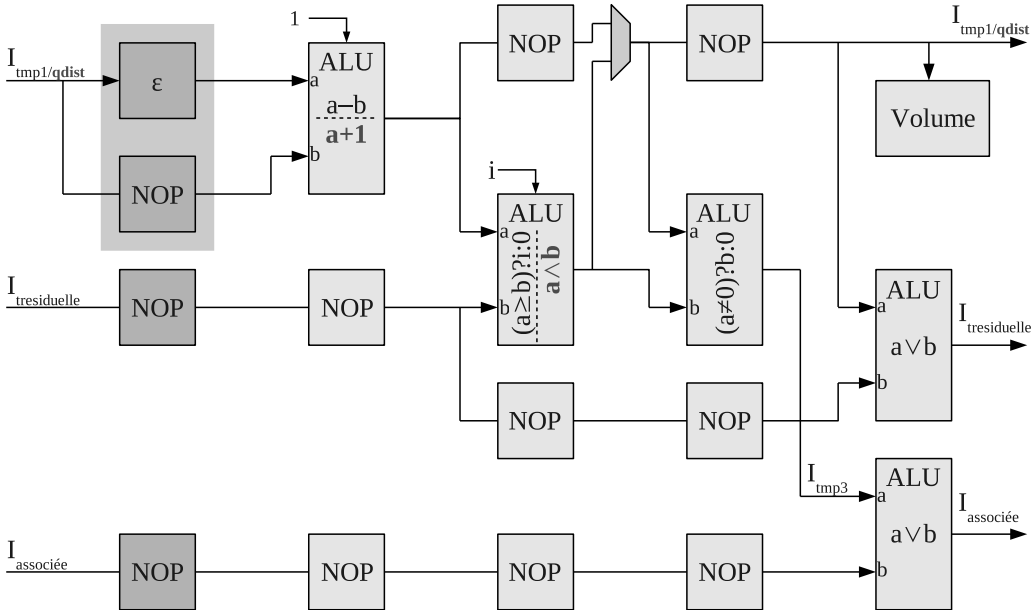


FIG. 3.11: Fusion du *pattern* dédié au calcul de la quasi-distance et du *pattern* dédié à sa régularisation

L'approche de description manuelle de *patterns* d'architecture composés d'opérateurs flots de données dédiés au traitement d'images montre ses limites et principalement dans le cadre de la reconfiguration à grain fin du fait de la grande quantité de flots à décrire manuellement. On peut dans cette optique utiliser des générateurs de *patterns* d'architecture, tel que CatapultC de Mentor ou bien encore Gaut, qui permettent la description de *pattern* à un plus haut niveau d'abstraction.

Une autre approche est de considérer la reconfiguration à gros grains et essayer de décrire manuellement un *pattern* d'architecture satisfaisant pour une majorité de traitements. Une telle structure est présentée dans l'annexe A. Cette approche est limitée et souvent non optimale. On peut donc envisager un outil permettant à la fois une description de haut niveau, mais également une génération automatique d'un *pattern* d'architecture résultat de la fusion de plusieurs *patterns* sources. Tout en restant dans une optique de reconfiguration à gros grains, on peut ainsi prendre en entrée un ensemble d'opérations à réaliser, avec pour chacune d'entre elles le *pattern* d'architecture correspondant, et construire automatiquement un *pattern* commun réalisant toutes les opérations considérées.

3.4 Description de haut niveau

3.4.1 Problématique

L'enjeu ici est de disposer d'un outil dédié à la description rapide de *patterns* d'architecture, que *patterns* l'on pourra répliquer un certain nombre de fois afin de disposer d'une

plus grande puissance de calcul. Il ne s'agit pas de décrire un atelier logiciel prenant en entrée le cœur d'un algorithme décrit au niveau de la manipulation des pixels les uns par rapport aux autres, mais plutôt de disposer d'un outil décrivant l'agencement d'opérations de traitement d'images simples à travers des composants tels que ceux décrits dans le chapitre 2 et que nous avons commencé à utiliser dans des descriptions manuelles à la section précédente.

Nous avons sélectionné deux aspects importants concernant la génération automatique d'un *pattern* d'architecture. Le premier concerne la synchronisation des flots de données, car nous avons choisi de masquer cette contrainte à l'utilisateur et le second aborde la fusion automatique de plusieurs descriptions dans le but d'obtenir automatiquement une architecture reconfigurable, ou reparamétrable dynamiquement, à gros grains.

3.4.2 Description d'un *pattern* d'architecture

Nous avons vu dans la section traitant de la description manuelle que la description d'un *pattern* d'architecture pouvait être une tâche longue puisqu'il est nécessaire d'intervenir manuellement pour spécifier, à un assez bas niveau, les instances de composants nécessaires au *pattern* ainsi que leurs connexions les unes par rapport aux autres.

En supposant disposer d'un ensemble de briques de traitement d'images assimilées à une *boîte noire* et prenant un certain nombre de paramètres en entrées et en sorties, nous avons défini une méthode graphique de description de *patterns*. Ce langage ou cette méthode n'est pas un objectif en soi, mais plutôt un moyen de construire un *pattern* rapidement et simplement par un utilisateur. Rapidement, car ce dernier ne se préoccupe pas d'implémenter le *pattern* dans un langage de description matérielle (VHDL par exemple) et simplement, car l'utilisateur n'a pas à gérer le synchronisme hétérogène des composants de façon très poussée. En outre, la description graphique du *pattern* produit un modèle permettant la fusion avec d'autres descriptions avec des algorithmes beaucoup plus simples que s'ils étaient mis en œuvre directement dans la description matérielle au niveau RTL.

La figure 3.12 montre une description réalisée à l'aide du langage graphique afin d'obtenir un *pattern* d'architecture dont le but est de réaliser les opérations de base de morphologie mathématiques. Lors de la synthèse du circuit, l'utilisateur doit uniquement se soucier du nombre d'étages nécessaires, c'est-à-dire spécifier le nombre de fois qu'il souhaite voir répliquer le *pattern* pour former un pipeline plus ou moins profond.

Les briques d'opérateurs utilisées ici sont les suivantes :

- *Input* : point d'entrée d'un flux de données émanant soit d'une files d'attente temporisant les données provenant de l'hôte soit d'une sortie du *pattern* située dans l'étage précédent du pipeline.
- *Output* : point de sortie d'un flux de données allant soit vers les file d'attente temporisant l'évacuation des données vers l'hôte soit vers une entrée du *pattern* située dans l'étage suivant du pipeline.
- *PoC* : processeur de voisinage dédié aux calculs d'une érosion ou d'une dilatation avec tous types d'éléments structurants composés au maximum de 9 voisins. L'opération ainsi que l'élément structurant sont paramétrables avant l'envoi d'un flot de données.
- *Alu* : unité de calcul prenant en entrée deux flots de données et une constante et réalisant divers calculs paramétrables avant l'envoi du flot de données.

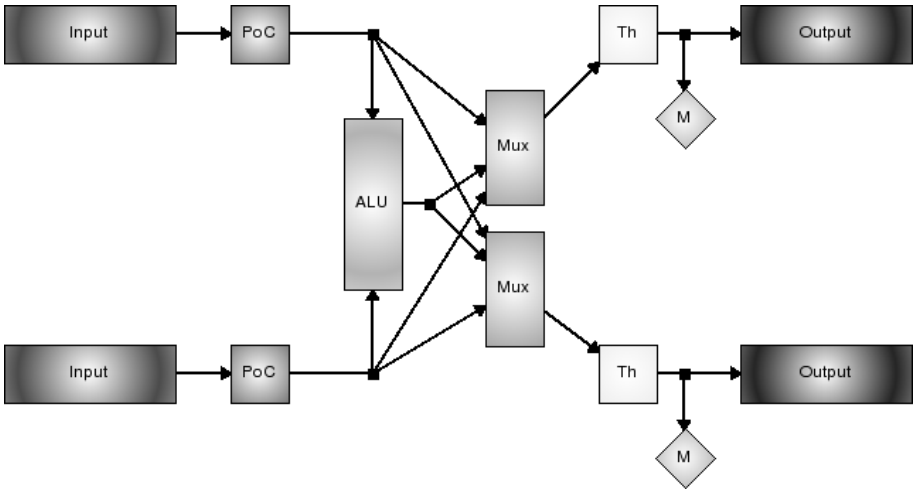


FIG. 3.12: Exemple de description conduisant à une génération matérielle automatique d'un *pattern* d'architecture

- *Mux* : multiplexeur permettant de rediriger des flots de données. L'état du multiplexeur se paramètre avant l'envoi d'un flot de données.
- *Th* : Unité de seuillage prenant en entrée une constante pour un seuil haut et une constante pour un seuil bas, ainsi qu'un bit indiquant si une binarisation est nécessaire. Cette brique est donc tout aussi bien en mesure de ne rien seuiller et de seuiller avec ou sans binarisation du flot de données. Toutes ces possibilités sont paramétrables avant le lancement d'un flot de données.
- *M* : Unité de mesure calculant le minimum, le maximum et le volume global d'un flot de données. Les valeurs sont à jour lorsque le flot de données a entièrement circulé sur la branche concernée.

On peut, avec le *pattern* décrit à la figure 3.12, réaliser des opérations géodésiques, des calculs de fonctions distances, calculer des gradients et ou bien encore réaliser des filtres alternés séquentiels. Les performances de ces opérations complexes sont évidemment fonction du nombre d'étages de *patterns* au sein du pipeline. Toutefois, la génération du matériel correspondant n'est pas directe, car il faut dans un premier temps résoudre les problèmes de synchronisation engendrés par les latences hétérogènes des composants ou opérateurs du *pattern*.

3.4.3 Synchronisation du flot de données

La synchronisation d'un *pattern* d'architecture décrit avec notre système consiste à ajouter des *NOP*, qui sont matérialisés par un certain nombre de registres en séries où par une ligne à retard, afin de présenter aux entrées des opérateurs suivants les pixels de même indices pouvant provenir de flux différents.

Le problème de la synchronisation peut être vu en partant des entrées du *pattern* et en découpant la description en plusieurs étages. Un exemple reprenant le cas de la figure 3.12 est présenté en figure 3.13. Dès lors que l'on dispose d'une représentation en étage, il est facile de rechercher et ajouter récursivement des *NOP* partout où un signal est connecté

directement à la fois dans l'étage courant et dans l'étage suivant.

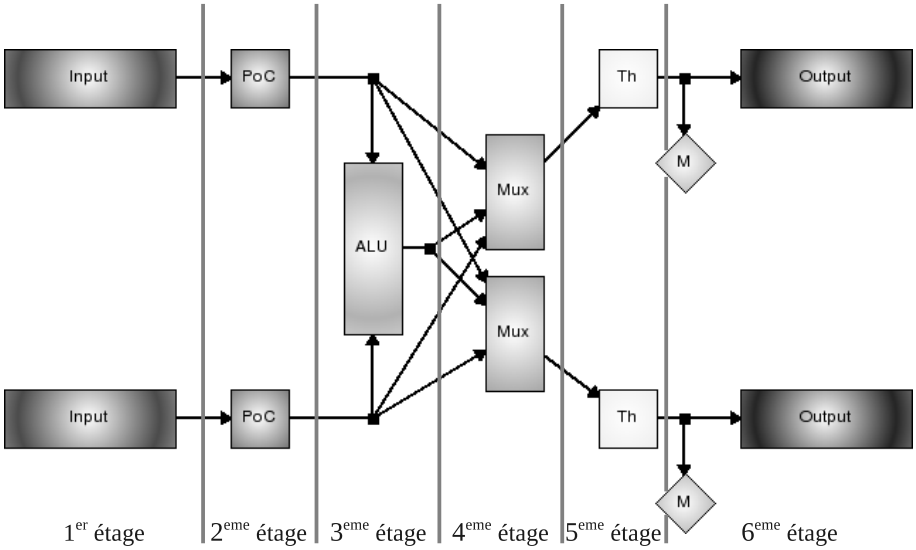


FIG. 3.13: *Pattern* d'architecture synchronisé automatiquement

Un exemple de synchronisation est présenté en figure 3.14, il s'agit toujours du *pattern* décrit en 3.12 où les *NOP* ont été spécifiés automatiquement. Dans le premier cas, des *NOP* ont été ajoutés sur tous les signaux et même s'ils proviennent du même composant. Le second cas montre comment des *NOP* peuvent être regroupés afin d'économiser quelques ressources logiques.

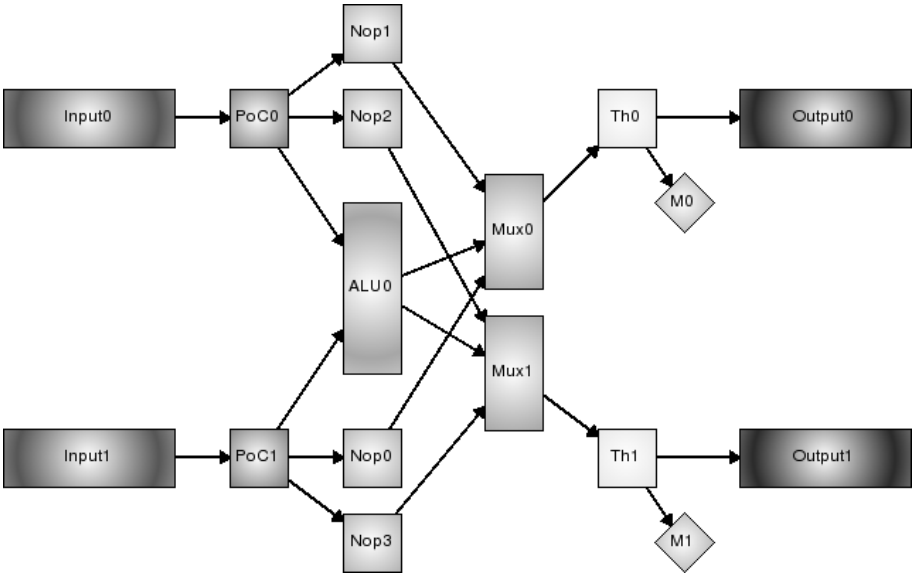
Un autre problème de synchronisation existe, mais ce dernier est implicite et peut se régler lors de la génération du matériel ou du logiciel. En effet, Il est possible que, dans le même étage, les opérateurs mis en jeu aient des latences différentes. Il convient alors de prendre en compte l'opérateur ayant la plus grande latence pour ajouter implicitement un certain retard derrière chaque opérateur pour ne pas désynchroniser les flux. On peut très bien imaginer que l'on retrouve dans un même étage un PoC ayant une forte latence et une ALU ayant une faible latence. Ainsi, lors de la génération du matériel, il faudra veiller à ajouter en sortie de l'ALU une ligne à retard permettant d'avoir une latence au niveau de l'ALU égale à celle engendrée par le PoC dans son ensemble.

3.4.4 Génération de code

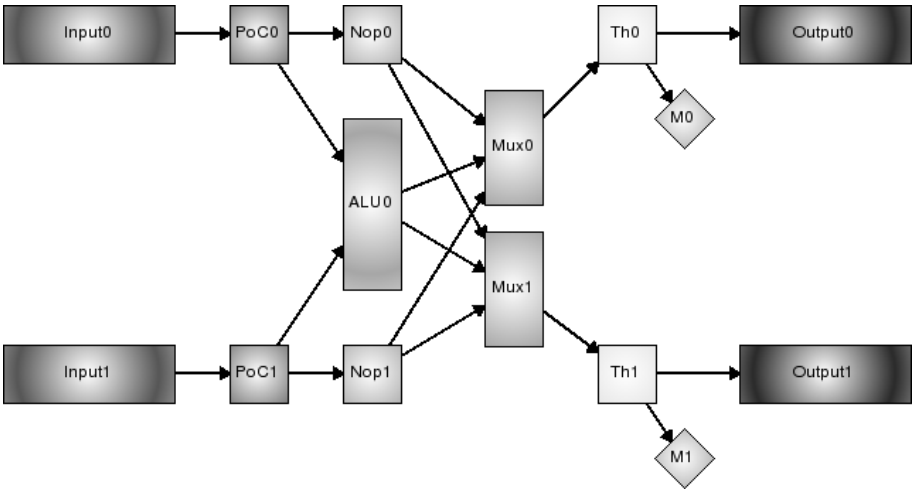
3.4.4.1 Génération du simulateur

La description d'un *pattern* a pour finalité de générer automatiquement un pipeline ayant une profondeur fixée avant synthèse afin de le mettre en place dans un SoC tel que proposé en figure 3.3. Mais avant de générer un circuit, il peut être intéressant de valider de manière fonctionnelle et rapide le bon fonctionnement de la structure du pipeline.

Nous avons donc choisi d'avoir la possibilité de générer une fonction de simulation reposant sur une bibliothèque de traitement d'image. Cette dernière est capable de réaliser tous les traitements élémentaires du langage de description et la fonction de simulation se



(a) Sans fusion de *NOP*



(b) Avec fusion de *NOP*

FIG. 3.14: *Pattern* d'architecture synchronisé automatiquement

charge d'ordonnancer les calculs décrits dans le pipeline de *pattern*. Cette fonction prend en paramètres autant d'images que d'entrées/sorties au niveau du pipeline ainsi que la configuration des différents étages. La configuration représente l'état du pipeline pour le traitement d'un flot de données et permet de spécifier, par exemple, les opérations réalisées par les *PoC* ou bien encore les états des multiplexeurs.

La figure 3.15 présente les couches et les interfaces logicielles qui sont mises en œuvre pour valider de manière fonctionnelle une application utilisant un pipeline spécifique de *patterns*. La génération de la fonction de simulation est relativement simple, mais doit se faire après synchronisation du flux de données et en parcourant le *pattern* des entrées vers les sorties en considérant un découpage temporel tel que présenté en figure 3.14.

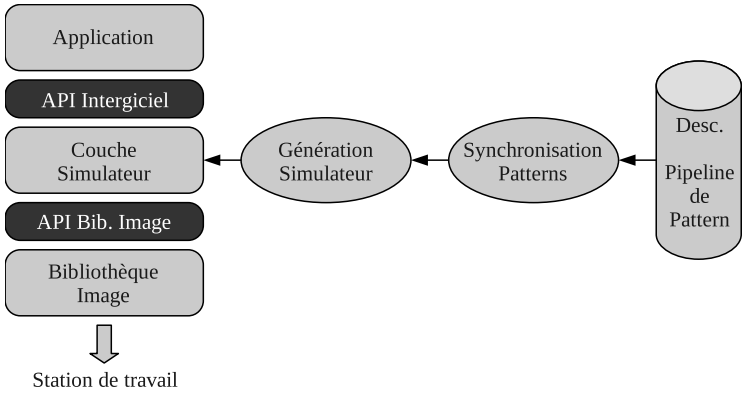


FIG. 3.15: Flot et architecture logicielle de validation fonctionnelle d'un pipeline de processeurs de voisinage décrit via un langage de haut niveau

3.4.4.2 Génération du matériel

La génération du matériel fonctionne de manière analogue à la génération du logiciel. Cette étape génère uniquement les interconnexions entre les différents composants. En effet, pour chaque composant disponible dans le langage de description de haut niveau, il existe un représentant au niveau RTL dans un langage tel que VHDL. Le générateur de code se charge donc d'instancier et de raccorder les différents composants conformément à ce qui est décrit dans la description de haut niveau.

L'étape de génération du matériel doit être précédée d'une étape de synchronisation du *pattern*. Toutefois, cette étape doit encore être raffinée, car la synchronisation réalisée, qui est finalement fonctionnelle, suppose que tous les composants ont la même latence. Cette hypothèse, suffisante dans le cadre de la génération d'un simulateur sur station de travail, est abusive dans le cadre de la génération du matériel. Une étape supplémentaire doit donc être mise en œuvre afin de synchroniser plus finement le pipeline en recherchant pour chaque sous-étage du *pattern*, le composant ayant la latence maximale afin d'ajuster via des registres ou des files d'attente les latences des composants du même sous-étage. Ainsi, en plus du fichier VHDL décrivant matériellement un composant, il est nécessaire de renseigner dans un fichier tiers la latence de chaque composant.

La figure 3.16 présente l'architecture logicielle ainsi que le flot automatique de génération d'un pipeline matériel de processeurs de voisinage. Une application appelle des

fonctions de traitements d'images proposées au niveau de l'API Intergiciel, API commune avec le simulateur, afin de cacher et d'abstraire la mécanique de gestion des flots mémoire et de configuration bas niveau du pipeline.

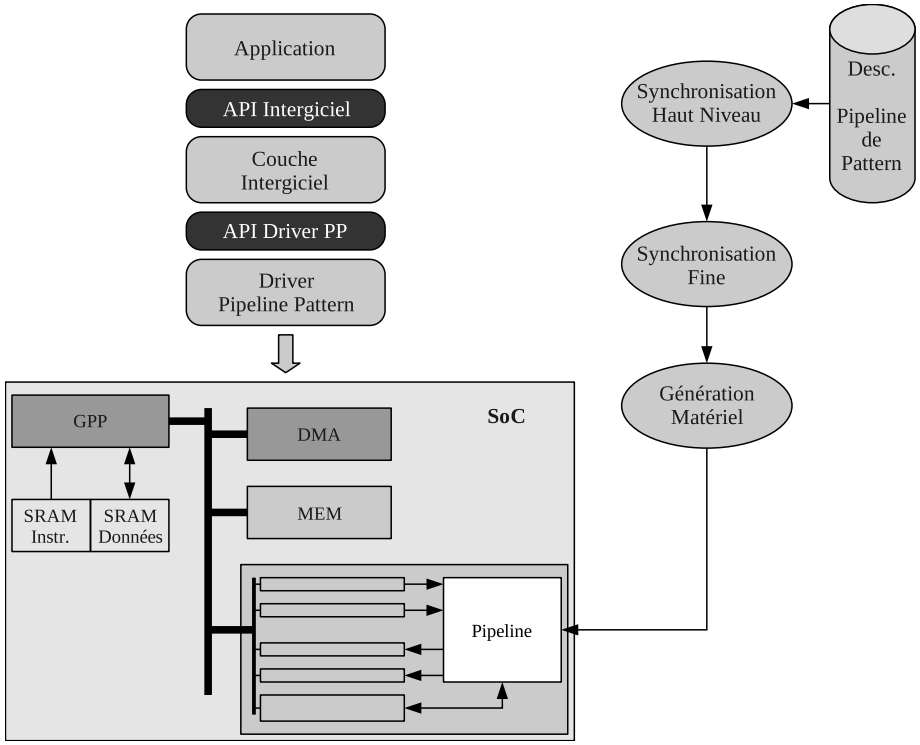


FIG. 3.16: Flot de génération du matériel et architecture logicielle de gestion d'un pipeline de processeurs de voisinage décrit via un langage de haut niveau

3.4.5 Fusion de descriptions

La fusion de différentes descriptions est un point intéressant puisque l'on peut essayer de construire automatiquement un *pattern* reconfigurable à gros grains à partir de plusieurs *patterns* décrits avec notre formalisme.

Un exemple serait de pouvoir construire le *pattern* réalisant à la fois la quasi-distance et la régularisation à partir des descriptions réalisant uniquement la quasi-distance et uniquement la régularisation. On peut alors se demander quelles ressources il est impératif d'économiser et quelles sont celles que l'on peut dupliquer. Dans notre cas, les processeurs PoC sont des ressources à utiliser avec précaution, car coûteuses à la fois en termes de mémoires embarquées et en termes de ressources logiques. On peut également, dans une moindre mesure, chercher à économiser les ALU lorsque ces dernières embarquent une grande quantité d'opérations réalisables.

Afin de simplifier la présentation de la méthode de fusion, nous considérons la fusion d'un *pattern* A avec un *pattern* B visant à produire un *pattern* C. On admet que chacun des composants d'un même *pattern* dispose un identifiant unique.

Le principe est assez simple et consiste à lister tous les composants utilisés dans les différentes descriptions afin de rechercher ceux que l'on doit regrouper et ceux que l'on

peut répliquer. Pour ce faire, un dictionnaire propre à chacun des *patterns* est construit afin de lister, par type de composant, toutes les ressources devant être mutualisées.

La méthode de fusion se décompose en deux phases. La première, présentée à l'algorithme 6, a pour objectif de créer le *pattern* C comportant uniquement les composants issus de la fusion de A et B. Cet algorithme repose sur l'utilisation de dictionnaires, propre aux *patterns* A et B, permettant de lister les identifiants par type de composant à mutualiser. Les dictionnaires sont créés en parcourant les designs des entrées vers les sorties pour fusionner des composants les plus proches afin d'éviter d'ajouter une grande quantité de composants de synchronisation.

Lors de la création du dictionnaire A, tous les composants du *pattern* A sont copiés dans le *pattern* C et une table de correspondance des identifiants est créée garantissant de retrouver un composant de A dans C. Durant la création du dictionnaire B, les composants de B ne faisant pas l'objet d'une mutualisation sont ajoutés dans C et de la même manière, une table de correspondance des identifiants est créée. Enfin, cet algorithme finit par mutualiser les ressources en réaffectant dans les tables de correspondances les identifiants mutualisés et en ajoutant éventuellement les composants de B restant à fusionner et n'ayant pas trouvé de correspondance dans le *pattern* A.

La seconde phase de fusion, décrite à l'algorithme 7, est chargée de reconnecter les signaux entre les composants et, le cas échéant, ajouter des multiplexeurs aux entrées des composants mutualisés.

L'algorithme de fusion étant maintenant décrit, nous pouvons observer sur la figure 3.17 les différents résultats d'une fusion de deux *patterns* simples. Le premier *pattern* source, figure 3.17.a permet le chaînage en série de deux opérations de voisinage type érosion/dilatation. Le second *pattern* source, figure 3.17.b permet quant à lui la réalisation de deux opérations de voisinage en parallèle où les deux flots de données sont totalement indépendants. L'algorithme à été paramétré de manière à ce qu'il mutualise les ressources de type *Input*, *Output*, *PoC* et le résultat de la fusion, figure 3.17.c, montre que les calculs réalisés dans les *patterns* sources peuvent maintenant être réalisés dans un unique *pattern* via le paramétrage de deux multiplexeurs. On observe toutefois que le multiplexeur *Mux0* est un composant superflu, mais l'algorithme proposé se force de conserver, pour un *pattern* donné, la position des entrées sorties. Une extension de l'algorithme serait donc de supprimer ce type de multiplexeur lors de l'affectation des signaux dans le *pattern* fusionné (algorithme 7 étape 2).

Nous allons maintenant revenir sur notre exemple de quasi-distance et utiliser le système de description proposé pour détailler les deux étapes de l'opération dans deux *patterns* différents.

3.4.6 Exemple d'application : chaînage dédié à la quasi-distance

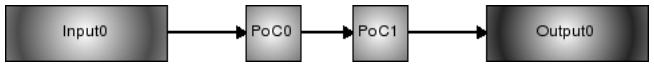
La figure 3.18 présente la fusion automatique des deux *patterns* impliqués dans le calcul de la quasi-distance. On remarque que les composants à mutualiser choisis (PoC et ALU) ont bien été fusionnés et qu'il est maintenant possible de réaliser, à moindre coût, les deux passes de la quasi-distance avec un matériel optimisé. Toutefois, on remarque une fois de plus la présence d'un multiplexeur supplémentaire par rapport à la fusion manuelle présentée à la figure 3.11. Ceci provient du fait que l'algorithme cherche à fusionner les

Algorithme 6 Fusion de *pattern* : Mutualisation des composants

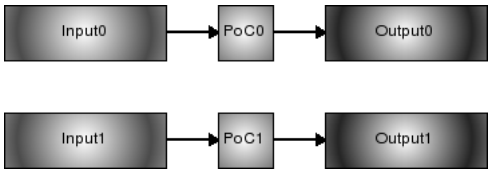
```
ENTRÉES: A,B,liste
SORTIES: C,hashA,hashB

  /* Étape 1 : Création du dictionnaire des composants mutualisés du pattern
  A et copie des composants de A dans C */
  Pour Chaque composant de A Faire
    Si  liste.contient(composant.type)  Alors
      dicoA[composant.type].ajouteComposant(composant.id)
    Fin si
    ncomposant = C.ajouteComposant(composant.type)
    hashA[composant.id] = ncomposant.id
  Fin pour

  /* Étape 2 : Création du dictionnaire des composants mutualisés du pattern
  B et copie des composants de B non-mutualisables dans C */
  Pour Chaque composant de B Faire
    Si  liste.contient(composant.type)  Alors
      dicoB[composant.type].ajouteComposant(composant.id)
    Sinon
      ncomposant = C.ajouteComposant(composant.type)
      hashB[composant.id] = ncomposant.id
    Fin si
  Fin pour

  /* Étape 3 : Mutualisation des ressources et ajoute si nécessaire les
  composants du pattern B à mutualiser s'il ne sont pas déjà présent dans A
  */
  Pour Chaque type de liste Faire
    Pour Chaque idB de dicoB[type] Faire
      Si !dicoA[type].estVide  Alors
        /*  Mutualisation  */
        idA = dicoA[type].pop
        hashB[idB] = hashA[idA]
      Sinon
        /*  Ajout composant à mutualiser non présent dans A ou déjà
        mutulalisé */
        ncomposant = C.ajouteComposant(type)
        hashB[idB] = ncomposant.id
      Fin si
    Fin pour
  Fin pour
```

Algorithme 7 Fusion de *pattern* : Génération des connexions

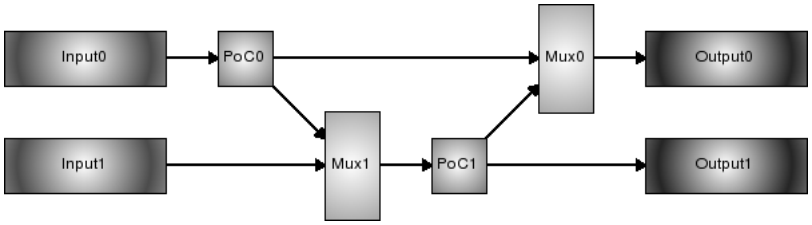
```
ENTRÉES: A, B, C, hashA, hashB
SORTIES: C

  /* Étape 1 : Affectation des signaux du design A dans le design C */
  Pour Chaque signal de A Faire
    C.ajouteSignal(hashA[signal.start],hashA[signal.stop])
  Fin pour

  /* Étape 2 : Affectation des signaux du design B dans le design C
  et ajoute les multiplexeurs. La méthode ajouteSignal vérifie si un
  multiplexeur est nécessaire lorsqu'un un signal existant pointe déjà sur
  un composant */
  Pour Chaque signal de B Faire
    Si !C.signalExiste(hashB[signal.start],hashB[signal.stop]) Alors
      C.ajouteSignal(hashB[signal.start],hashB[signal.stop])
    Fin si
  Fin pour
```



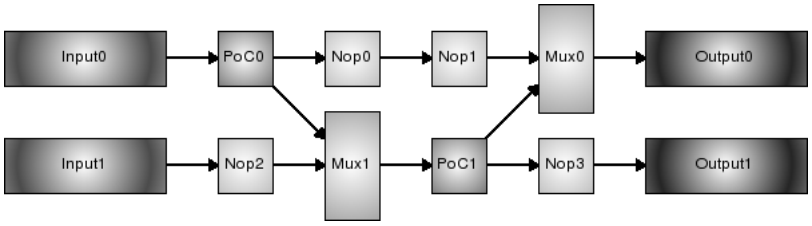
(a) *Pattern* source 1



(b) *Pattern* source 2



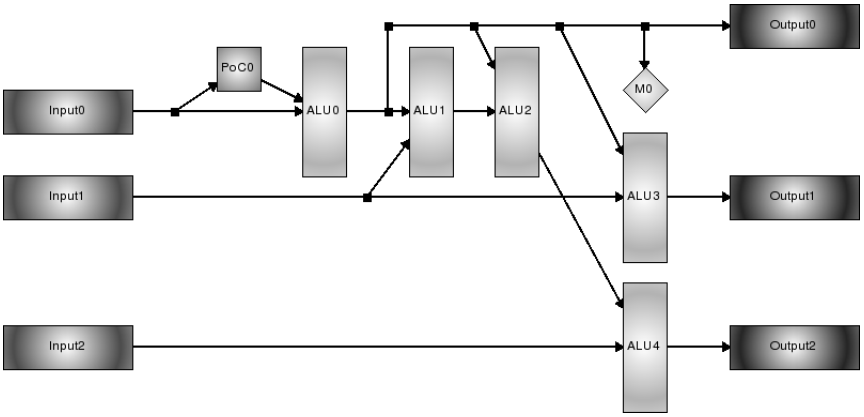
(c) Résultat de la fusion



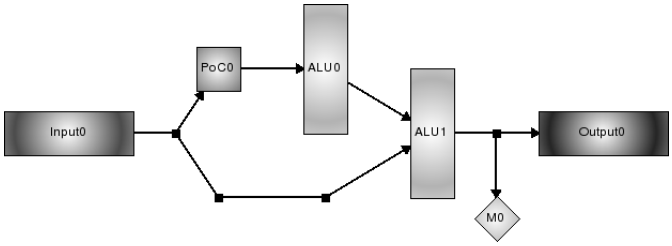
(d) Résultat synchronisé de la fusion

FIG. 3.17: Exemple de fusion automatique de deux *patterns* simples

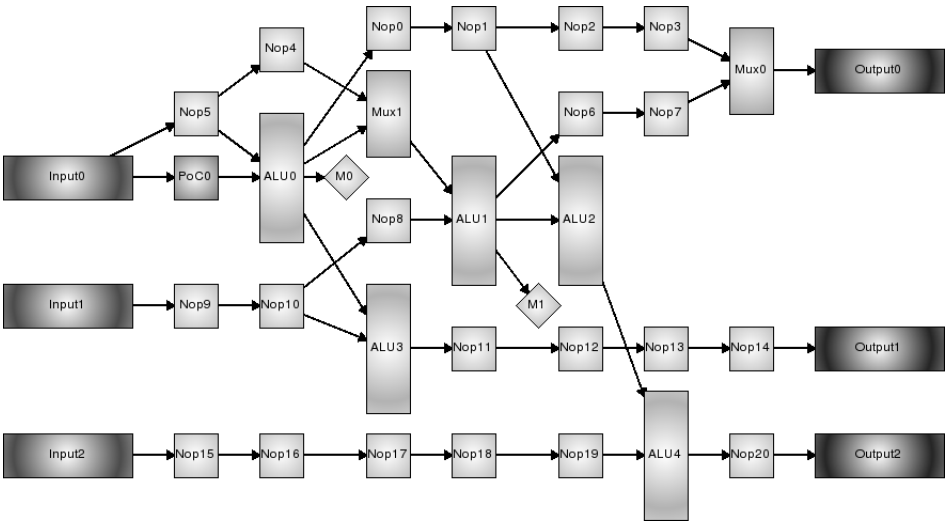
composants les plus proches, c'est-à-dire en minimisant la distance entre les sous-étages du *pattern*, mais aussi parce que l'algorithme cherche à diriger le flux vers les sorties dont l'indice correspond à celui du *pattern* source. Ce dernier point a déjà été énoncé dans la sous-section précédente.



(a) *Pattern* source 1



(b) *Pattern* source 2



(c) Résultat de la fusion

FIG. 3.18: Exemple de fusion automatique des deux *patterns* de la quasi-distance

3.5 Conclusion

Nous avons présenté dans ce chapitre une méthode de gestion d'un accélérateur flot de données composé de ressources assez hétérogènes et qui sont utilisées normalement en sortie de capteur sans possibilité de réitérer des calculs sur un même flot de données. Ce mode de fonctionnement est aujourd'hui trop rigide et compte tenu des technologies à notre disposition et des complexités croissantes des applications, il est aujourd'hui possible de proposer des circuits capables de réaliser plusieurs applications grâce à une souplesse accrue des possibilités de programmation. En particulier, l'utilisation d'un processeur généraliste couplé à un DMA permet de gérer un accélérateur flot de données et laisse l'utilisateur libre d'ordonnancer les calculs. Cette souplesse offerte aux développeurs d'applications montre toutefois ses limites, car si l'accélérateur est trop générique, des problèmes de performances peuvent survenir, et si l'accélérateur est trop typé, certaines applications ne pourront pas être déployées.

L'équilibre entre performance et généricité est variable d'une application ou famille d'applications à une autre et nous avons, dans un premier temps, proposé un outil de description de haut niveau simplifiant la création d'une nouvelle structure d'un accélérateur flot de données. Cet outil seul, même s'il simplifie grandement la construction d'un accélérateur, ne règle pas réellement le compromis qu'un développeur doit trouver entre spécificité de l'accélérateur et généricité dû au domaine d'application visé. L'utilisateur doit donc décrire manuellement un accélérateur pouvant être le résultat d'une fusion de plusieurs accélérateurs dédiés à une application ou famille d'applications.

La fusion de *patterns* d'accélérateurs peut être automatisée et nous avons montré que des résultats similaires à ceux obtenus manuellement étaient observés. L'algorithme proposé permet de générer automatiquement un unique accélérateur reconfigurable à gros grains réalisant plusieurs fonctions qui étaient, au départ, éclatées dans plusieurs architectures flots de données décrites via notre outil de haut niveau.

Une perspective intéressante serait maintenant de mettre en place une fusion de *patterns* à plusieurs niveaux. Aujourd'hui les composants mututalisables d'un *pattern* doivent être absolument identiques, par exemple deux processeurs de voisinage peuvent être fusionnés s'ils comportent les mêmes arbres de calculs sur le voisinage. On pourrait imaginer de fusionner des composants de même type, mais n'embarquant pas les mêmes unités de calculs internes. Deux types de fusion seraient donc mis en place, une fusion à gros grains dont le rôle serait d'assurer la mutualisation de composants de même type tels que des processeurs de voisinage ou des ALUs et une fusion à grain fin qui serait chargée de fusionner les unités de calcul internes des composants mutualisés, comme fusionner l'arbre de calcul d'une érosion et celui d'une dilatation dans le cadre de la mutualisation d'un processeur de voisinage dédié à l'érosion avec un processeur de voisinage dédié à la dilatation.

Une autre perspective est analysée dans le cadre du projet FREIA. Elle consiste à analyser le graphe des dépendances des différents traitements exécutés sur le pipeline afin de voir quels sont ceux pouvant être réordonnés et regroupés sans modifier le comportement fonctionnel de l'application. L'avantage d'une telle technique est de rentabiliser au mieux la profondeur du pipeline et permet alors de minimiser le nombre d'envois du flot de données. On obtient alors un gain en performance souvent non négligeable même si le développeur de l'application propose à la base un programme dont la structure n'est pas adaptée à

notre accélérateur.

Le prochain chapitre marque une rupture avec ce que nous venons de présenter dans les deux précédents chapitres. Même si le leitmotiv reste le traitement flots de données, nous allons présenter comment, à partir de processeurs vectoriels microcodés, nous pouvons encore améliorer la performance ainsi que la généricité d'un accélérateur exploitant conjointement le parallélisme spatial et temporel.

Chapitre 4

Processeurs VLIW vectoriels

La mise en place de processeurs flots de données, purement dédiés à l'extraction de voisinages, impose de faire des choix quant aux différents calculs réalisables. Si de tels choix ne sont pas connus à priori, on peut alors envisager une structure de calcul plus souple de type processeurs *Very Large Instruction Word*, VLIW. Ces processeurs exploitent le parallélisme au niveau instructions et permettent d'exécuter différentes étapes d'un programme en parallèle en repoussant une partie de la gestion du pipeline dans le compilateur. L'avantage de ces derniers vis-à-vis des architectures dites superscalaires et de supprimer du design tout le contrôle matériel pouvant être réalisé à la compilation laissant ainsi plus de surface dans le circuit pour les unités de calcul.

Le parallélisme au niveau des instructions peut être aussi remplacé ou complété par un parallélisme au niveau des données, car les opérations réalisées en traitement d'images, lorsqu'elles sont parallélisables, consistent à répéter le même noyau de calcul sur un jeu de données assez vaste. Ainsi les processeurs SIMD, *Single Instruction stream Multiple Data stream*, selon la taxonomie de Flynn [29], sont des processeurs exécutant une même instruction sur plusieurs données. On en distingue principalement deux types, les structures multiprocesseurs exécutant en parallèle les mêmes instructions et les structures vectorielles exploitant un parallélisme plus fin au niveau des unités de calcul en travaillant avec des vecteurs de données souvent de grandes tailles.

Les opérateurs de traitement de voisinage tirent un très bon parti du parallélisme et l'on observe souvent un rendement, énoncé avec la loi d'Amdahl [5], de l'ordre de 0.95. En d'autres termes, les opérations utilisant des structures SIMD bénéficient d'une forte accélération et le parallélisme au niveau des données est très bien exploité. Les processeurs VLIW étant eux aussi très bien utilisés, il est donc tentant d'augmenter massivement la quantité d'unités de calcul pour accélérer les traitements. En pratique l'accroissement des unités de calcul a de très fortes répercussions sur la largeur des instructions et sur le nombre de ports de la file de registres, augmentant fortement le nombre d'interconnexions dans le circuit. Envisager un processeur VLIW standard avec une très grande quantité d'unités de calcul est donc problématique et plusieurs voies existent : on peut tout d'abord considérer une grande quantité de processeurs VLIW en parallèle exécutant tous la même instruction [44], on peut également augmenter le nombre de clusters du processeur et enfin, on peut envisager de mettre en place des unités de calcul vectoriel au sein d'un processeur VLIW afin de disposer d'un fort degré de parallélisme en limitant le nombre

de ports indépendants sur la file de registres. Toutes ces solutions ont un impact fort sur la vitesse et l'efficacité des calculs et sont parfois contraignantes, car elles peuvent imposer des degrés de parallélisation plus ou moins efficaces. C'est la raison pour laquelle nous allons, dans une première partie, exprimer d'un point de vue algorithmique quelles performances et quelles efficacités pouvons-nous attendre d'un parallélisme à grain moyen vis-à-vis d'un parallélisme à grain fin. Ces considérations vont ainsi nous permettre, dans une seconde partie, de présenter et de justifier notre architecture de processeurs VLIW vectoriels pouvant être utilisée à plusieurs niveaux de granularité.

4.1 Avant propos

L'exécution d'instructions en parallèle n'est pas un domaine nouveau de la microélectronique, les processeurs superscalaires représentent une évolution des processeurs classiques exécutant une instruction par cycle. On peut alors trouver deux types de processeurs superscalaires, les processeurs de type CISC, *Complex Instruction Set Computer*, et les processeurs de type RISC *Reduced Instruction Set Computer*.

L'apparition des processeurs RISC, marque un pas en avant dans la standardisation des architectures microprogrammées en réduisant le nombre d'instructions et en garantissant leur exécution avec un même nombre de cycles contrairement aux processeurs CISC. Ceci a eu pour effet de modifier en profondeur les architectures en réalisant des pipelines de longueurs fixes, les unités de contrôle ont pu être simplifiées au profit d'unités de calcul plus performantes et de fréquences plus élevées. La tâche du compilateur change également puisque dans le cas d'un jeu d'instructions CISC, le compilateur doit être capable de détecter et d'utiliser au mieux des instructions complexes qui ne sont pas toujours orthogonales. La standardisation introduite avec les processeurs RISC allège donc la tâche du compilateur, car le jeu d'instructions est largement simplifié.

La mise en place de processeurs superscalaires en introduisant, au sein de processeur RISC ou CISC, des mécanismes matériels visant à exécuter des instructions en parallèle a permis d'augmenter de façon significative les performances. En effet ces processeurs sont capables de rechercher dans un paquet d'instructions quelles sont celles pouvant être exécutées en parallèle afin d'utiliser au mieux toutes les ressources du processeur. D'autres mécanismes ont été ajoutés tels que la prédiction de branchement ou encore l'amélioration de la gestion des hiérarchies des mémoires caches. Tous ces mécanismes ont un coût et la surface du circuit qui leur est réservée est importante et même souvent plus que celle dédiée aux unités de calcul.

Un changement idéologique encore plus marqué que la transition des processeurs CISC vers RISC a permis de faire émerger un nouveau concept permettant l'exécution d'instructions en parallèle tout en conservant un maximum de surface sur le circuit pour les unités de calcul. Cette philosophie appelée VLIW permet de décrire explicitement le parallélisme au niveau des instructions en laissant au programmeur ou au compilateur la bonne gestion du parallélisme. Ceci est rendu possible en considérant non plus un mot d'instructions, mais des paquets d'instructions qui sont décodés en même temps à chaque cycle et qui autorisent le fonctionnement en parallèle des différentes unités du processeur. On remarque sur la figure 4.1 la différence majeure entre un processeur superscalaire et un processeur

VLIW. Pour le premier, l'ordonnancement des instructions et leur exécution en parallèle se fait dynamiquement dans le circuit tandis que dans le cas du second, cet ordonnancement est réalisé statiquement lors de la compilation.

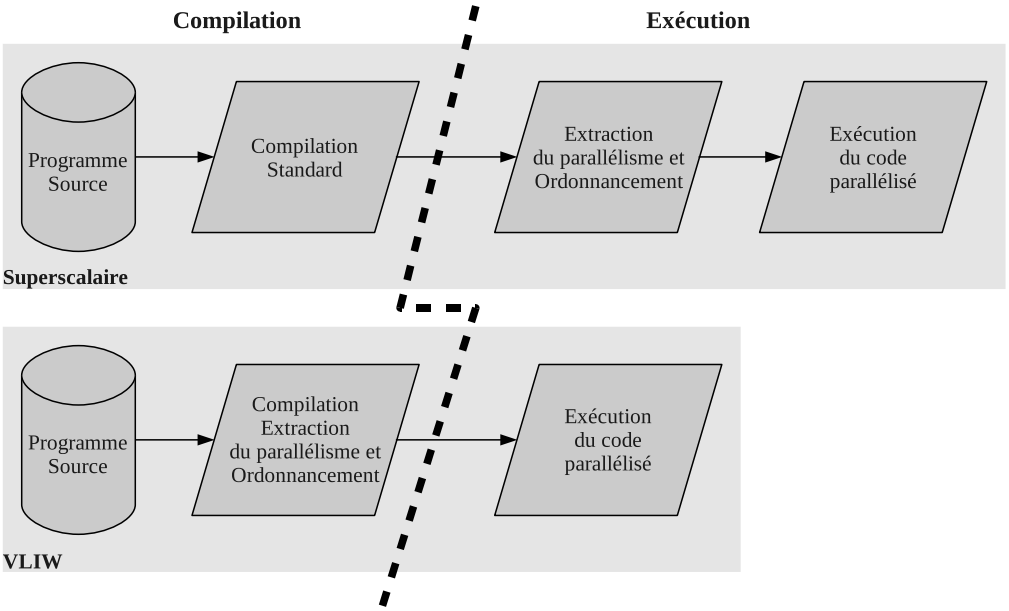


FIG. 4.1: Les étapes de compilation et d'exécution de code sur architecture Superscalaire et VLIW

Historiquement, les processeurs VLIW étaient considérés comme trop coûteux en transistors ainsi qu'en connexions internes et les compilateurs n'étaient pas suffisamment efficaces dans le domaine du calcul. Aujourd'hui un grand nombre d'avancées ont été réalisées et les techniques de compilation ont permis de combler l'écart entre une mise en œuvre C d'un algorithme et une mise en œuvre assembleur du même algorithme par un expert. Ces processeurs sont maintenant matures et la société Texas Instrument propose des architectures dédiées au traitement, non plus uniquement du signal, mais multimédia [34] délivrant une puissance d'environ 5 GOPS à 600 MHz et permettant par exemple de réaliser de la compression vidéo H.264 [73].

Les machines SIMD sont à l'origine des machines composées d'un certain nombre de processeurs exécutant la même instruction sur des données différentes grâce à des canaux de communications externes. Ces machines, telles que le Cray X-MP, pouvaient manipuler les données sous forme de vecteurs d'une centaine d'éléments et étaient capables de virtualiser les opérations en fonction du nombre d'unités de calcul. Avec l'arrivée des instructions Altivec des processeurs IBM PowerPC et des instructions MMX/SSEx des processeurs Intel Pentium, la notion de SIMD peut être vue différemment. On considère maintenant des unités de calcul capables de travailler sur des vecteurs de données, par exemple 16 données 8 bits, afin d'utiliser au maximum la largeur des mémoires caches et des unités de calculs. On fait référence à cette sous-classe d'architecture SIMD comme étant des instructions micro-SIMD car opérant avec des jeux de données plus réduits.

Depuis maintenant quelques années, avec l'augmentation des densités d'intégrations, de nouveaux circuits apparaissent intégrants tous les éléments de calcul des machines SIMD

sur la même puce. On constate l'apparition de processeurs purement vectoriels tels que le processeur *VIRAM* [42] qui permet de travailler avec de grands vecteurs pouvant aller jusqu'à 128 éléments. Le nombre d'éléments des vecteurs était dimensionné pour être plus important que le nombre d'unités de calcul, donnant l'opportunité de diminuer la bande passante des instructions, de masquer plus simplement les temps de latence mémoire et de permettre de changer les ressources matérielles pour de nouvelles générations de circuits sans modifier lourdement le programme. Ce processeur vectoriel est très intéressant dans le cadre des systèmes embarqués, car l'exploitation d'un parallélisme au niveau des données est moins coûteuse que l'exploitation d'un parallélisme au niveau des instructions comme on le trouve dans les processeurs supercalaires et les processeurs VLIW [43].

On remarque également l'apparition de processeurs orientés *stream* tels que le processeur *Imagine* [38] qui a inspiré le design du processeur *Storm*, présenté en figure 4.2, de la société *Stream Processors*. La dernière génération de ce processeur multimédia [39] est composé de 80 unités de calcul micro-SIMD, sur des entiers ne dépassant pas 32 bits, organisé en 16 modules VLIW d'ordre 5 et permet d'atteindre une puissance de 512 GOPS à 800 Mhz pour 34 millions de transistors.

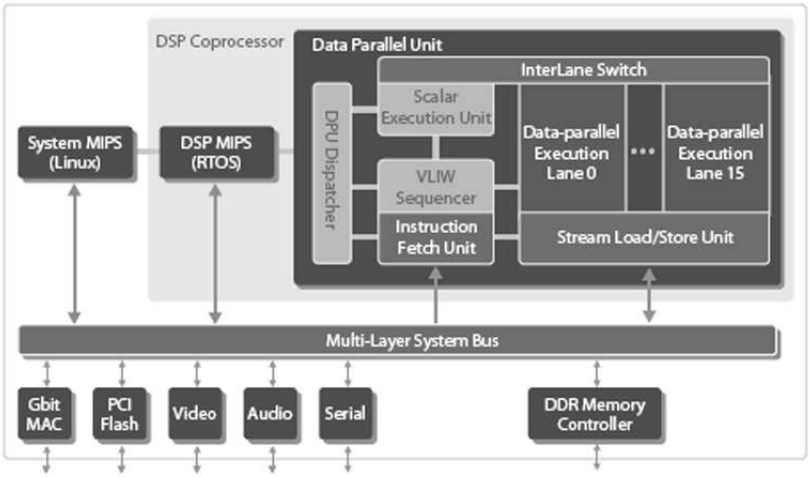


FIG. 4.2: Vue fonctionnelle du processeur Storm de la société Stream Processors

D'autres circuits existent avec une structure SIMD composée d'un grand nombre d'éléments de calculs décodant la même instruction. Le processeur *IMAPCar* [65], *Integrated Memory Array Processor for Car*, de la société NEC, figure 4.3, représente bien cette catégorie de circuit dédiée au traitement d'images embarqué. Il est composé de 128 processeurs VLIW en parallèle décodant tous la même paquet d'instructions afin d'atteindre une puissance de calcul de l'ordre de 100 Gops à 100 Mhz pour une consommation de l'ordre de 2W. On peut citer également le processeur *Xetal II* [1] de la société *NXP* délivrant une puissance de calcul également de l'ordre de 100 Gops pour une consommation réduite de 600mW. Ces circuits sont toutefois peu évolutifs, car ils sont mis en place au sein d'un ASIC. Il existe également des architectures reposant sur le même principe général de fonctionnement, mais dont la cible est cette fois un FPGA avec les avantages en terme de reconfigurations et d'évolutions que cela confère. Ainsi le processeur SIMD *ter@pix* [14] de

la société *THALES* délivre une puissance de 20Gops au sein d'un FPGA de taille moyenne : le Virtex4 SX55.

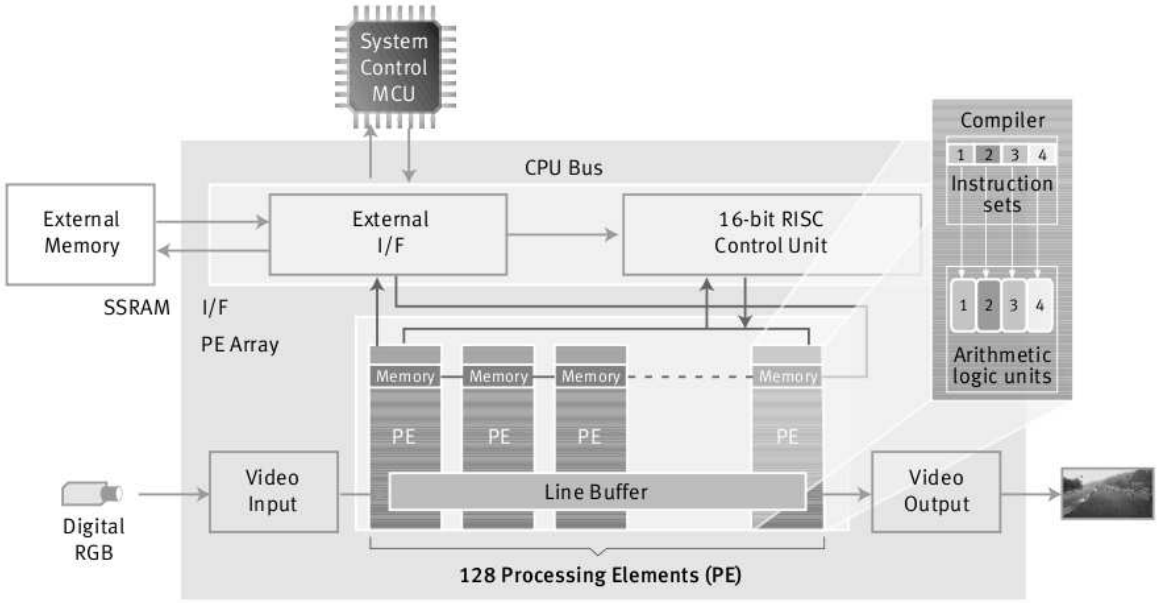


FIG. 4.3: Vue fonctionnelle du processeur NEC IMAPCar

Nous pouvons également parler d'une autre machine SIMD appelé la maille associative [26]. Cette machine SIMD, dont les ressources de calcul (PE) sont répartis en une grille bidimensionnelle, est très performante pour le traitement d'images et particulièrement pour la morphologie mathématique. Elle se compose, dans sa première version, d'une matrice de 512×512 PE, un par pixel, communiquant via un réseau en logique asynchrone en maille carrée. Par exemple, des opérations telles que des reconstructions géodésiques et des lignes de partage des eaux peuvent être réalisées en quelques microsecondes. Toutefois, la surface de cette machine comporte plus de 600 millions de transistors et une virtualisation a été proposée pour permettre d'affecter N pixels à un PE. Avec $N = 1024$, la taille du circuit se trouve réduite de 25% ce qui est assez faible, contenu de la perte de performance d'un facteur 30 environ. Il faut toutefois relativiser cette diminution de la vitesse de calcul, car un opérateur tel que la ligne de partage des eaux reste encore de l'ordre de la milliseconde. Il est encore difficile aujourd'hui de déployer une maille associative 256×256 même virtualisée par 1024 dans des circuits logiques programmables.

Les cartes graphiques présentent de nos jours un grand intérêt pour le monde du calcul haute performance et il est bon de procéder à une petite description de leurs fonctionnements. En effet même si pour le moment le marché visé est celui des calculateurs débarqués, nous commençons à observer l'émergence du marché mobile des accélérateurs 3D et nous assisterons peut-être à la même dérive déjà observée avec les circuits provenant du monde du jeu vidéo. À l'origine, les architectures dédiées à l'affichage de scènes en 3D étaient construites autour de deux types d'accélérateurs : l'un dédié au calcul du pavage de l'espace par des triangles et l'autre au traitement des textures. Chacune de ces étapes se faisait dans des processeurs spécialisés et il était difficile de déterminer un ratio fixe entre les deux

types de processeurs. En effet si l'on considère un pavage avec de gros triangles, les processeurs correspondants sont souvent inactifs alors que les processeurs dédiés aux calculs des textures sont surchargés. On observe également la situation duale avec de petits triangles, les processeurs alors chargés du pavage sont surchargés alors que ceux dédiés aux calculs des textures sont souvent inactifs.

Il est naturel que les accélérateurs graphiques aient évolué vers une architecture unifiée où chaque processeur peut calculer les textures et réaliser le pavage de l'espace par des triangles. Avec une telle architecture, il est maintenant possible d'utiliser les cartes graphiques pour d'autres calculs, scientifiques principalement, et les constructeurs fournissent des outils de développement à cet effet ainsi que des circuits spécialisés tels que TESLA [49], dépourvus de sorties vidéos.

Ainsi, les circuits graphiques de la société *NVidia* ont évolué vers une architecture massivement parallèle en considérant plusieurs unités de calcul SIMD, figure 4.4.

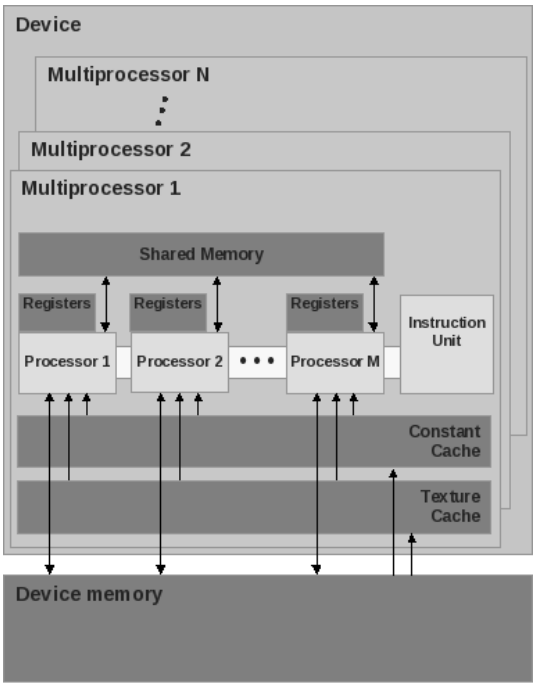


FIG. 4.4: Architecture des multiprocesseurs des circuits nVidia

Ce circuit est donc un assemblage de multiprocesseurs indépendants, figure 4.5, équipés chacun de 8 processeurs généralistes (appelés SP), qui effectuent toujours une même opération à la manière d'une unité SIMD, et de 2 processeurs spécialisés (appelés SFU).

Un multiprocesseur se sert de ces 2 types de processeurs pour exécuter des instructions sur des groupes de 32 éléments. Chacun de ces éléments est appelé *thread*, et un groupe de 32 *threads* forment un *warp*. Chaque unité multiprocesseur est en mesure d'exécuter 24 *warps*, c'est-à-dire 768 *threads* sans coûts d'ordonnancement puisque des mécanismes matériels existent afin de sauvegarder et restaurer les contextes. On dispose ainsi d'une architecture SIMT, *single-instruction, multiple-thread*, où il est nécessaire de découper un calcul avec de bonnes possibilités de parallélisation en un très grand nombre de *threads* afin de maximiser les débits des unités de calculs. Cette découpe en *threads* permet facilement,

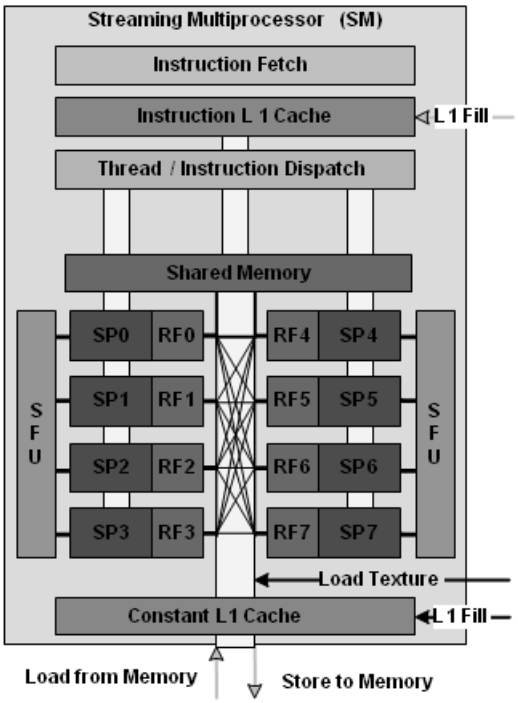


FIG. 4.5: Architecture d'un multiprocesseur NVidia

avec presque aucune modification du code source, d'atteindre des vitesses de calculs plus importantes lorsqu'une nouvelle génération de circuit voit le jour embarquant un plus grand nombre d'unités multiprocesseur. Aujourd'hui, un circuit Geforce 8 GTX permet d'exécuter 12288 *threads* pour 16 multiprocesseurs fonctionnant à une fréquence de 1.5 GHz. Toutefois, la taille et la consommation du circuit sont assez rédhibitoires pour une utilisation embarquée puisque le circuit comporte pas moins de 681 millions de transistors sur $470mm^2$ pour une consommation de l'ordre de 150W.

L'association d'IBM, de Sony et de Toshiba a permis de créer le processeur CELL dédié à la fois au monde du calcul et à celui des jeux vidéo, domaines finalement assez proches au vu de ce que nous avons pu dire précédemment.

Comme le montre la figure 4.6, on retrouve une architecture multiprocesseur composée d'un maître et de plusieurs esclaves. Les processeurs ne possèdent pas une unité d'exécution des instructions dans le désordre pour des contraintes de surface, obligeant le programmeur ou le compilateur à procéder à un ordonnancement optimal des instructions. Le processeur central PPE, *PowerPC Processing Element*, est un processeur conventionnel avec un accès à la mémoire centrale via deux niveaux de caches et pouvant exécuter des instructions SIMD de type *AltiVec*.

Les huit cœurs additionnels, appelés *Synergystic Processing Elements* ou SPE, sont constitués de deux entités : la mémoire locale (LS) de 256 Ko et une unité de calcul SIMD se rapprochant de l'*AltiVec*. Cette dernière dispose d'accès mémoire extrêmement rapides tandis que l'accès à la mémoire principale doit se faire obligatoirement via une requête de transfert asynchrone. Ce processeur, dans sa dernière version fondue en 2006, atteint à 3.2 Ghz une puissance de calcul crête de 206 GFLOPS avec des flottants codés sur 32 bits.

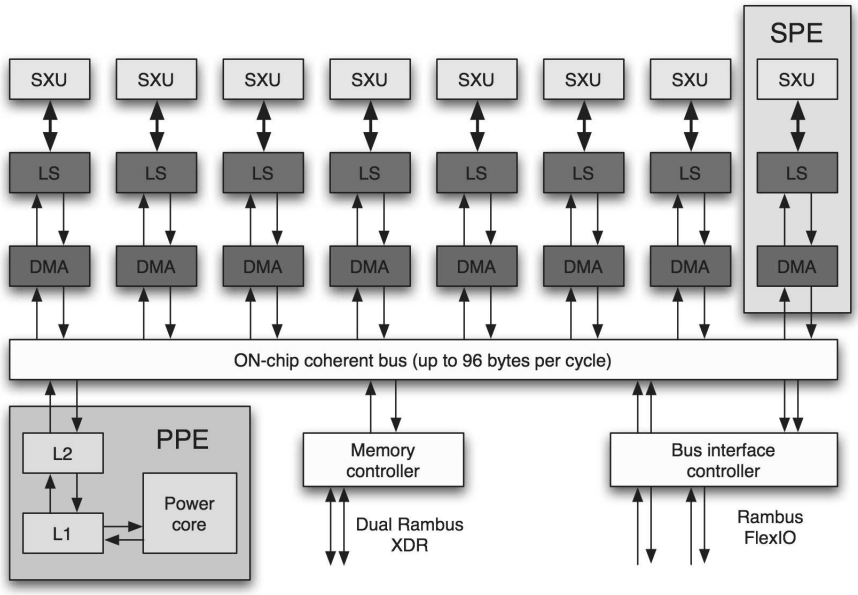


FIG. 4.6: Architecture du circuit Cell d'IBM/Sony/Toshiba

La société Intel se lance aujourd'hui dans le secteur du calcul de rendu 3D et du calcul haute performance avec un circuit nommé *Larrabee* [68] dont une vue fonctionnelle est présentée en figure 4.7.

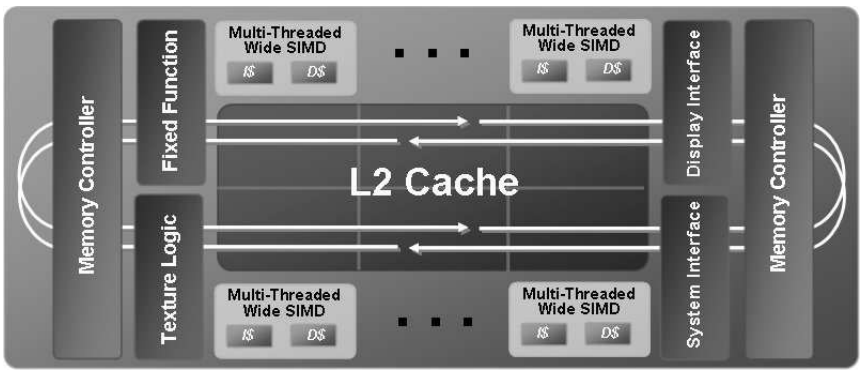


FIG. 4.7: Architecture du circuit Larrabee d'Intel

Ce circuit peut être considéré comme un circuit hybride entre un processeur spécialisé pour le rendu 3D et un multiprocesseur généraliste, car les processeurs de ce circuit utilisent une architecture compatible x86 ainsi qu'une hiérarchie de cache avec un contrôle de cohérence et de larges unités de calculs SIMD. Les coeurs x86 du circuit Larrabee sont plus simples que ce que l'on peut trouver dans des processeurs actuels tels que les *core 2 duo* et sont dépourvus d'unités d'exécutions des instructions dans le désordre. Chacun des cœurs du circuit *Larrabee* peut exécuter simultanément jusqu'à quatre *threads* et dispose d'unités SIMD de 512 bits gérant des vecteurs composés de 16 éléments flottants 32 bits ce qui est quatre fois plus larges que les unités de calculs SSE(x) que l'on retrouve dans les stations

de travail. Un bus full duplex de 512 bits avec une topologie en anneau permet d'assurer la communication entre le cache et les cœurs , mais également entre coeurs. Ce circuit dispose, dans une première version en 45 nm, de 32 cœurs et peut consommer jusqu'à 300W pour une puissance de 2 TFLOPS.

La problématique de construction et de gestion matérielle et logicielle des multicœurs pour l'embarqué devient primordiale et des solutions existent aujourd'hui autres que celles poussées par Intel ou NVidia. En particulier la société Tilera propose un circuit [83], provenant du projet RAW [81] du *Massachusetts Institute of Technology*, composé de 64 processeurs VLIW associés via un réseau sur puce et composé de quatre contrôleurs mémoires. Ce circuit est présenté en figure 4.8 et se veut multi domaines, bien que le cœur des applications soit orienté vers les télécommunications avec des aspects vidéo et multimédia. En effet chacun des cœurs est en mesure de faire fonctionner un noyau Linux ce qui peut paraître assez lourd dans le cadre de notre problématique des architectures flots de données pour le traitement d'images.

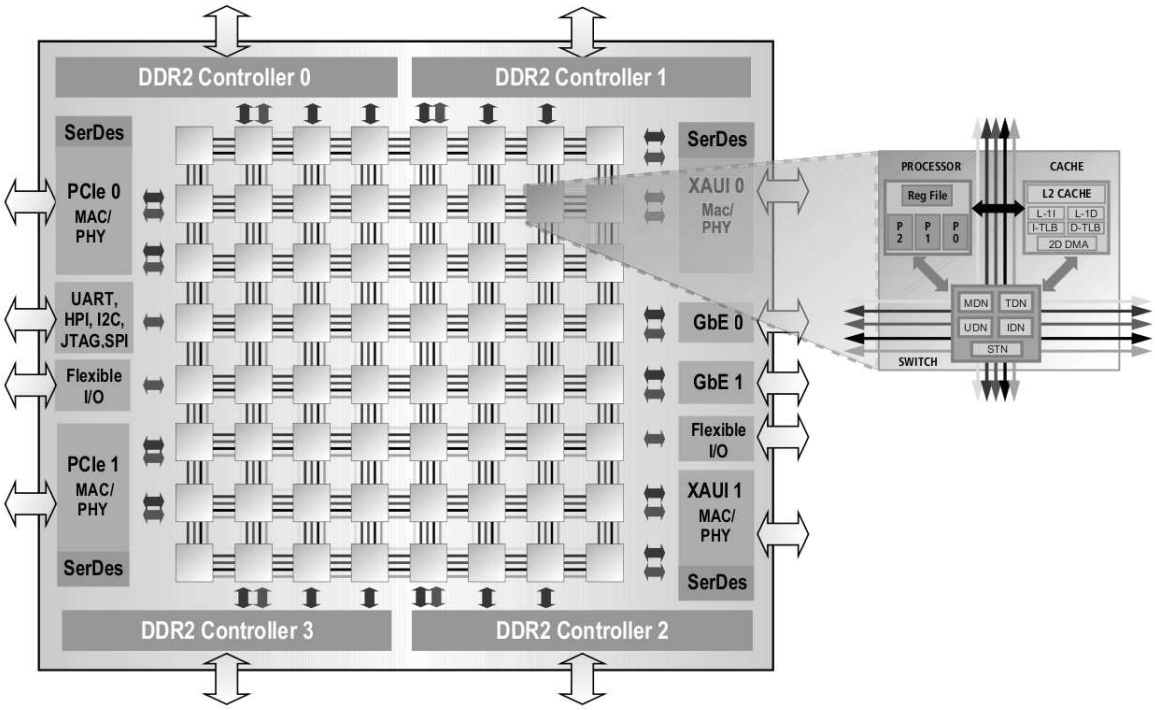


FIG. 4.8: Architecture du circuit MPSoC Tilera

Il ressort de toutes ces architectures qu'une certaine uniformisation est en marche. Les circuits pour les calculateurs à hautes performances sont aujourd'hui concurrencés par les systèmes dédiés aux jeux vidéo qui finalement nécessitent de fortes puissances de calcul dans de lourds formats de données tel que le flottant 32/64 bits. On assiste donc à une rationalisation des coûts de développement par la création de circuits servant à la fois de base pour des calculateurs téraflopique, voire même pétaflopique et pour les systèmes dédiés aux jeux vidéo. Le secteur de l'embarqué reprend souvent sur une même puce les architectures hautes performances que l'on pouvait trouver il y a quelques années sur de gros calculateurs et l'on peut raisonnablement se dire que ce transfert de technologie deviendra

de plus en plus courant. C'est la raison pour laquelle nous avons envisagé la création d'un cœur vectoriel léger dédié au traitement d'images qu'il nous est possible de répliquer en une grande quantité au sein du même circuit. Ce groupe de processeurs vectoriels se positionne alors comme un accélérateur que l'on pourrait venir brancher dans un SoC un peu à la manière de ce que nous avons pu présenter dans le chapitre précédent, mais avec une plus grande souplesse vis-à-vis des opérations de traitement d'images réalisables.

4.2 Considérations algorithmiques

4.2.1 Avant propos

La mise en œuvre parallèle et efficace d'applications repose sur des choix algorithmiques et plusieurs granularités peuvent être mises en place avec pour chacune d'entre elles certains avantages et inconvénients que nous détaillerons ci-après. La mise en place d'un parallélisme spécifique est généralement contrainte par l'architecture cible sur laquelle fonctionne l'application. Nous essayerons dans cette sous-section de présenter deux aspects différents du parallélisme exploitable pour le calcul d'opérations régulières de traitement d'images en essayant de s'abstraire des contraintes matérielles lorsque cela est possible.

4.2.2 Parallélisation à grain moyen

Il ressort de différentes architectures embarquées détaillées précédemment que la notion de machine SIMD est principalement exploitée dans le sens originel du terme, c'est-à-dire en utilisant plusieurs processeurs fonctionnant en parallèle et décodant un même paquet d'instructions. Dans le cadre du traitement d'images, les architectures de type IMAPCar ne peuvent travailler qu'avec une taille prédéterminée d'image, contraignant les développeurs, dans le cas d'image plus importante, de découper l'image en imagettes se recouvrant sur une certaine plage de pixels. La figure 4.9 montre ce principe pour une image découpée en 25 imagettes dont les tailles utiles varient en fonction de la position dans l'image ainsi qu'en fonction de la taille de la zone de recouvrement. Cette dernière est donc directement fonction du nombre d'opérations de voisinage ainsi que de la taille du noyau ou de l'élément structurant considéré.

Cette découpe pose problème puisqu'il n'est alors plus possible de simplement réaliser un certain nombre d'opérations de voisinage successivement sur une même imagette si les zones de recouvrement ne sont pas suffisamment larges. Pour que les bords soient calculés correctement, il faut soit augmenter la taille des zones de recouvrement, soit réaliser une opération de voisinage sur toutes les imagettes avant de poursuivre avec une autre. Il est bien évidemment possible de combiner ces deux méthodes pour réaliser sur une imagette autant d'opérations de voisinage que le recouvrement le permet, les autres opérations seront calculées une fois toutes les imagettes traitées.

La figure 4.10 présente une caricature de ce qui peut se passer lorsque l'on itère, sur une même imagette, 15 érosions avec des éléments structurants carrés 3×3 alors que les bandes de recouvrement sont prévues pour une unique érosion avec ce même élément structurant. On remarque alors sur cette figure que les bords traités sur chacune des imagettes sont faux.

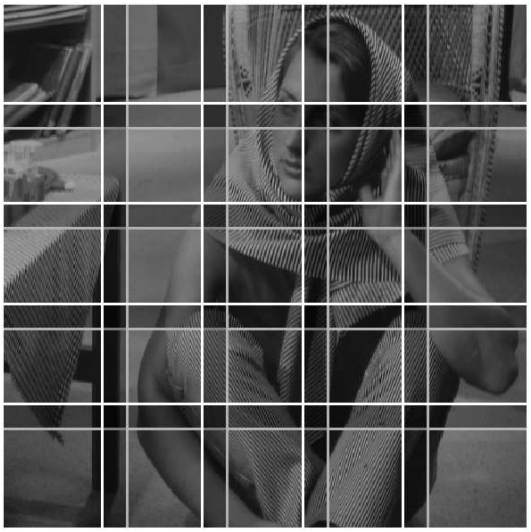


FIG. 4.9: Zones de recouvrement nécessaires au traitement d'une image par découpage en imagettes

L'exemple présenté ici est quelque peu caricatural, mais ce problème se pose tout de même assez rapidement, comme dans le cas d'ouvertures morphologiques, ou encore dans le cadre de reconstructions géodésiques. L'augmentation de la taille des zones de recouvrement provoque une augmentation du nombre d'imagettes et donc une augmentation du temps de calcul. Prenons l'exemple suivant : en considérant une image de dimension 506×506, nous pouvons mettre en place une découpe de l'image en 16 imagettes de dimensions 128×128 avec une zone de recouvrement de largeur 2 pixels. Cette zone de recouvrement permet de réaliser une opération de voisinage, telle qu'une érosion, avec un élément structurant de rayon unitaire. Si nous voulons réaliser deux érosions successivement dans chaque imagette, il faut élargir la zone de recouvrement à quatre pixels provoquant, dans notre exemple, une augmentation du nombre de ces dernières à traiter, passant ainsi de 16 à 25. La figure 4.11 illustre ce point et montre que l'ajout d'une opération de voisinage par imagette provoque l'augmentation en nombre de ces dernières à cause des zones de recouvrement devant être élargies.

La figure 4.12 présente l'efficacité d'un calcul d'une ou plusieurs opérations de voisinage réalisées au niveau imagette. Cette courbe permet de mesurer la quantité de calcul réalisé plusieurs fois ou effectué sur des données non pertinentes, par exemple hors de l'image. Trois tailles d'imagettes sont considérées et pourraient correspondre à trois architectures SIMD composées de 64, 128 ou 256 éléments de calcul. À chaque augmentation du nombre d'imagettes, on remarque dans le même temps une baisse de l'efficacité des calculs, car les imagettes contiennent moins de données utiles. Ce phénomène est accentué sur les bords car, si l'ajout d'une opération entraine une augmentation du nombre d'imagettes, la taille utile de ces dernières au bord de l'image est alors minimale et l'efficacité chute fortement. Ce phénomène est d'autant plus marqué que les imagettes ont une taille importante (cf courbes en pointillés gris clair de la figure 4.12 pour un nombre d'opérations égale à 7).

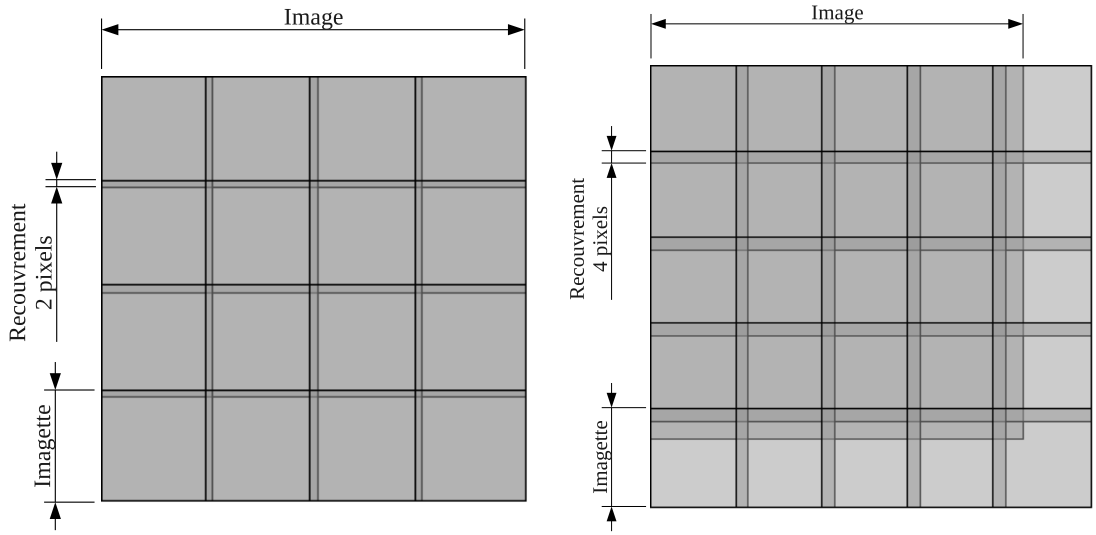
Ces derniers graphiques montrent qu'il est difficile d'atteindre l'utilisation la plus efficace d'une architecture reposant sur un traitement au niveau imagette. Plusieurs paramètres entrent en ligne de compte, comme la taille des images, la taille des imagettes et

(a) Image originale

(b) Zones de recouvrement bien ajustées

(c) Zones de recouvrement trop faibles

FIG. 4.10: Influence de la taille des zones de recouvrement lors de calculs d'opérations de voisinage sur des imagettes



(a) Une opération par imagette (b) Deux opérations par imagette

FIG. 4.11: Nombre d'imagettes en fonction du nombre d'opérations à réaliser

le nombre d'opérations à réaliser au sein d'une même imagette. Il est difficile de toujours utiliser au mieux l'architecture et on observe le plus souvent une efficacité autour de 60 %, car il est difficile d'adapter la taille des imagettes ainsi que le nombre d'opérations de voisinage pour des raisons architecturales.

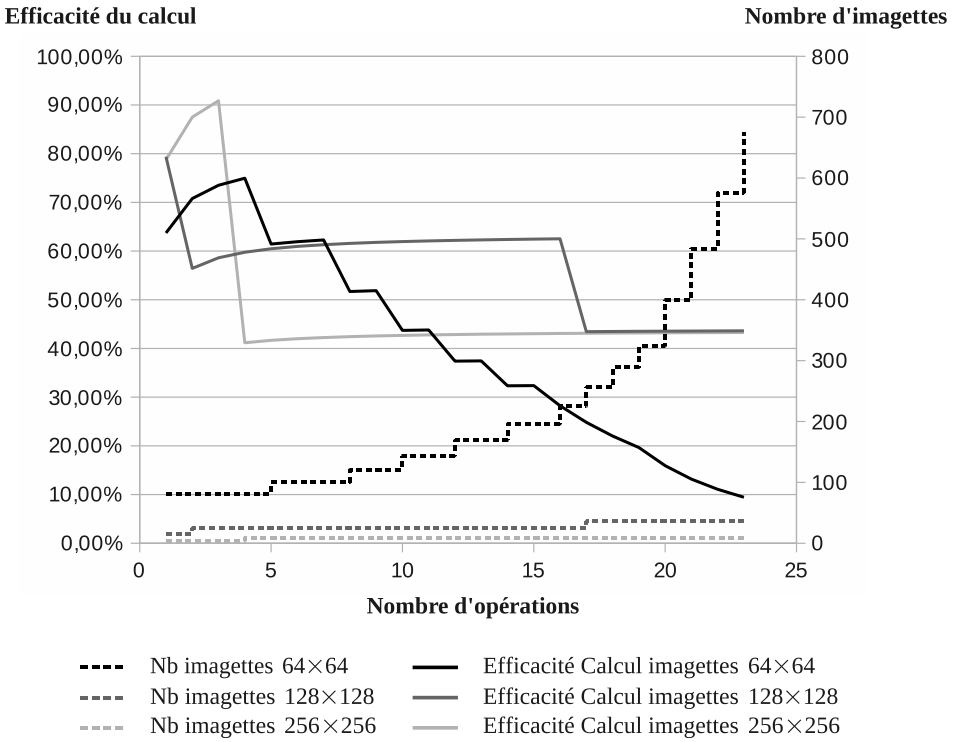
4.2.3 Parallélisation à grain fin

Les architectures vectorielles ou micro-SIMD permettent aussi de mettre en place une parallélisation de type imagette avec les contraintes que nous avons exposées précédemment et qui dans notre cas s'exprime de manière très forte du fait de la petite taille des jeux d'instructions micro-SIMD imposant souvent le traitement d'imagette d'une taille maximale de 16×16 pixels.

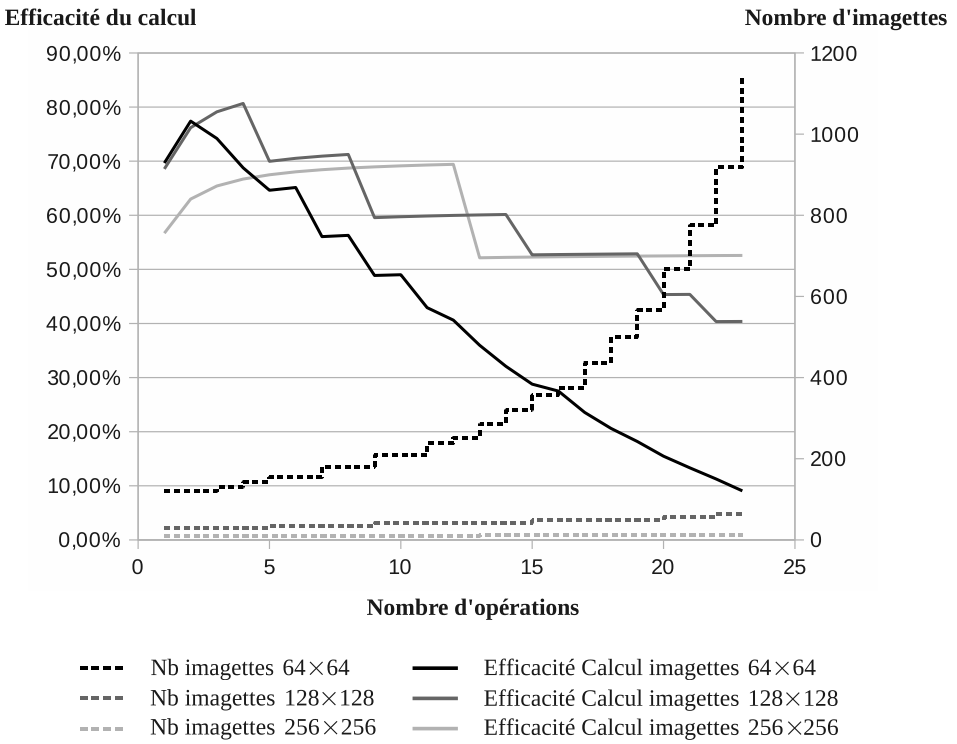
Il existe cependant une autre manière de paralléliser les calculs en considérant cette fois les lignes entières de l'image, on supprime ainsi les problèmes de recouvrement des imagettes tout en évitant une certaine redondance dans les calculs.

Les opérations de voisinages sont réalisées en procédant à des décalages des lignes de droite à gauche et vice versa pour accéder aux différents voisins de l'élément structurant. On extrait ainsi tous les voisinages d'une ligne de pixels et il est ensuite possible de procéder aux calculs sur ces derniers. La figure 4.13 montre comment extraire les voisinages 3×3 d'une ligne de pixels en réalisant six décalages et trois recopies de lignes. On obtient donc 9 tableaux comportant les pixels correspondant à un même voisin parmi tous les voisinages extraits. Il est alors possible d'appliquer un noyau de calcul sur ces tableaux de façon vectorielle comme si on appliquait un calcul sur un voisin dans le cas scalaire. On peut par exemple appliquer des coefficients d'un filtre sur chacun des éléments des tableaux et ensuite effectuer l'addition membre à membre de tous les tableaux pour obtenir le résultat d'une convolution sur une ligne de l'image.

Pour mettre en place ce principe d'extraction parallélisé des voisinages, il faut être en



(a) Image 506x506



(b) Image 720x576

FIG. 4.12: Efficacité du calcul d'opérations de voisinages sur des architectures SIMD avec découpage en imagettes

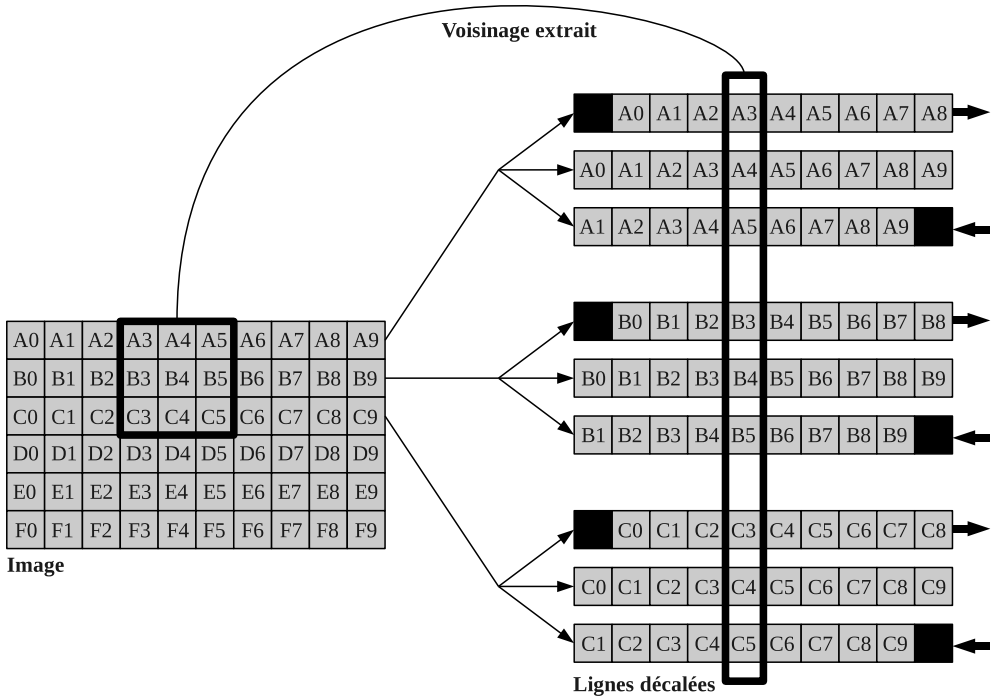


FIG. 4.13: Extraction du voisinage par plusieurs décalages de lignes

mesure de réaliser des opérations de décalages à gauche et à droite. Il faut aussi pouvoir calculer des opérations arithmétiques et logiques sur chacun des pixels des lignes extraites en utilisant le jeu d'instructions vectorielles qui permet dans ce cas de calculer ces opérations par paquets de pixels. Par exemple, dans le cadre des instructions Intel SSE2, il est possible de calculer un infimum en chargeant deux paquets de 16 pixels codés sur 8 bits. Une fois que nous disposons de telles instructions, il est simple de les empaqueter dans des fonctions permettant de travailler sur les lignes de l'image comme nous le faisons sur les vecteurs. Une opération de voisinage sur une image se déroule telle que décrite dans l'algorithme 8, en supposant que nous disposons des fonctions suivantes (utilisant en interne des instructions vectorielles) :

- *LitLigne(I,n)* : Lit la ligne n de l'image I
- *EcritLigne(S,n,L)* : Écrit la ligne L à la position n de l'image S
- *DécaleDroite(L,vbrd)* : Renvoie la ligne L décalée d'un pixel vers la droite. Le pixel inséré à la valeur *vbrd*
- *DécaleGauche(L,vbrd)* : Renvoie la ligne L décalée d'un pixel vers la gauche. Le pixel inséré à la valeur *vbrd*
- *Opération(L1,L2)* : Renvoie le résultat d'une opération calculée sur les pixels deux à deux des lignes $L1$ et $L2$. Une opération pouvant être par exemple un minimum, un maximum, une multiplication, une addition, ...

L'algorithme se déroule en trois étapes et nous supposons travailler avec un élément structurant de k lignes et l colonnes. Il faut en premier lieu exécuter un prologue qui permet d'amorcer le calcul en traitant les $\lfloor \frac{k}{2} \rfloor$ premières lignes, car elles sont impactées par le bord NORD. Dans notre exemple, où $k = 3$ et $l = 3$, il faut uniquement traiter la première ligne dans le prologue puisque l'élément structurant que nous considérons est inscrit dans

Algorithme 8 Opération de voisinage avec une architecture micro-SIMD

```
ENTRÉES: I, M, vbrd
SORTIES: S

  /* Prologue */

  D00 = LitLigne(I,0)
  D01 = DécaleDroite(D00,vbrd)
  D02 = DécaleGauche(D00,vbrd)

  D10 = LitLigne(I,1)
  D11 = DécaleDroite(D10,vbrd)
  D12 = DécaleGauche(D10,vbrd)

  /* Noyau de calcul du prologue */
  tmp1 = Opération(D00,D01)
  tmp2 = Opération(D02,D10)
  tmp3 = Opération(D11,D12)
  tmp1 = Opération(tmp1,tmp2)
  tmp1 = Opération(tmp1,tmp3)

  EcritLigne(S,0,tmp1)

  /* État permanent */

  Pour x=2 à M-1 Faire
    D20 = LitLigne(I,x)
    D21 = DécaleDroite(D20,vbrd)
    D22 = DécaleGauche(D20,vbrd)

    /* Noyau de calcul de l'état
    permanent */
    tmp1 = Opération(D00,D01)
    tmp2 = Opération(D02,D10)
    tmp3 = Opération(D11,D12)
    tmp4 = Opération(D20,D21)
    tmp4 = Opération(D22,tmp4)
    tmp1 = Opération(tmp1,tmp2)
    tmp2 = Opération(tmp3,tmp4)
    tmp1 = Opération(tmp1,tmp2)
    EcritLigne(S,x-1,tmp1)

    /* Vieillissement des données */

    tmp1 = D00
    D00 = D10
    D10 = D20
    D20 = tmp1

    tmp1 = D01
    D01 = D11
    D11 = D21
    D21 = tmp1

    tmp1 = D02
    D02 = D12
    D12 = D22
    D22 = tmp1

  Fin pour

  /* Épilogue */

  /* Noyau de calcul de l'épilogue */

  tmp1 = Opération(D00,D01)
  tmp2 = Opération(D02,D10)
  tmp3 = Opération(D11,D12)
  tmp1 = Opération(tmp1,tmp2)
  tmp1 = Opération(tmp1,tmp3)

  EcritLigne(S,M-1,tmp1)
```

un carré de taille 3×3. Une fois le calcul amorcé, le régime permanent est atteint et l'on peut traiter les lignes de l'indice $\lfloor \frac{k}{2} \rfloor$ à $M - \lfloor \frac{k}{2} \rfloor$. Le traitement comprend la lecture et le décalage à gauche et à droite de la ligne x de l'image, le calcul avec tous les voisins de la ligne $x-1$, le stockage du résultat et enfin le vieillissement des lignes décalées D_{kl} pour le prochain tour de boucle. On termine l'algorithme par un épilogue qui permet de terminer les calculs en traitant les $\lfloor \frac{k}{2} \rfloor$ dernières lignes de l'image qui sont au bords SUD de l'image et pour lesquelles il faut un traitement spécialisé pour des raisons analogues au traitement des premières lignes dans le prologue.

Cet algorithme effectue, à chaque étape, tous les calculs sur tous les voisins. Si certaines propriétés de réduction du nombre de calculs existent par rapport aux opérations mises en place, il n'est pas obligatoire de refaire à toutes les itérations de boucle tous les calculs. En effet au lieu de mettre en place un vieillissement des données sur les lignes décalées, on peut très bien le mettre en place sur certains calculs intermédiaires.

Pour réaliser avec cet algorithme plusieurs opérations de traitement d'images dans la même boucle de traitement, il est juste nécessaire de rallonger le prologue et l'épilogue de manière à correctement synchroniser les opérations. Bien sûr, il est important de disposer de suffisamment de mémoire pour stocker tous les décalages de lignes ou dans une moindre mesure tous les calculs intermédiaires inhérents aux opérations mises en jeu. Il existe donc une limitation physique, intrinsèque à chaque architecture, vis-à-vis du nombre d'opérations réalisables en une passe de calcul.

Il est possible de diminuer le nombre d'opérations nécessaires aux calculs d'opérations de voisinage en imaginant disposer d'instructions plus spécialisées qui permettraient de combiner les opérations de décalage avec les opérations arithmétiques et logiques. Une telle architecture logicielle avait été déployée il y a quelques années dans la bibliothèque de traitement d'images Micromorph [35]. Dans le cas de la morphologie mathématique on pourrait ainsi disposer des opérations atomiques suivantes prenant en entrée un vecteur de pixels Ve et renvoyant un vecteur de pixels Vs :

- Décalage à droite et supremum : $Vs = rshift(Ve) \vee Ve$
- Décalage à gauche et supremum : $Vs = lshift(Ve) \vee Ve$
- Décalage à droite et infimum : $Vs = rshift(Ve) \wedge Ve$
- Décalage à gauche et infimum : $Vs = lshift(Ve) \wedge Ve$

Il faut bien entendu que l'unité de calcul réalisant ces opérations dispose d'un registre permettant de réintroduire pour la prochaine étape le pixel décalé, un peu à la manière d'un bit de retenue dans les processeurs généralistes. Dans le cadre du filtrage linéaire, on pourrait également imaginer mettre en place des instructions analogues permettant de réaliser un décalage d'un vecteur de pixels et la multiplication des éléments d'un vecteur par un scalaire afin d'appliquer à la volée les coefficients du noyau de convolution.

L'efficacité du parallélisme avec un processeur micro-SIMD ou vectoriel, réalisant des opérations de voisinages directement sur les lignes entières, est donc assez facilement prévisible, car toujours maximal dès lors que la largeur de l'image est multiple de la largeur du jeu d'instructions vectorielles. Ceci est dû au fait qu'il n'y a pas de calcul redondant dans la méthode de traitement des lignes complètes de l'image quelque soit le nombre d'opérations réalisées à la volée. Cette efficacité reste toutefois assez théorique puisqu'elle est détachée des contraintes mémoires.

Si l'on considère l'efficacité du calcul comme étant le ratio entre le nombre de pixels à

traiter en théorie et le nombre de pixels traités en pratique, on peut tracer la courbe de la figure 4.14. En effet comme expliqué précédemment l'efficacité est maximale uniquement lorsque la taille d'une ligne est multiple de la taille des vecteurs manipulés par les instructions vectorielles, dans le cas contraire les vecteurs extraits de l'image en fin de ligne ne sont pas pleinement utilisés et des calculs sont réalisés, d'une certaine façon, *en trop*.

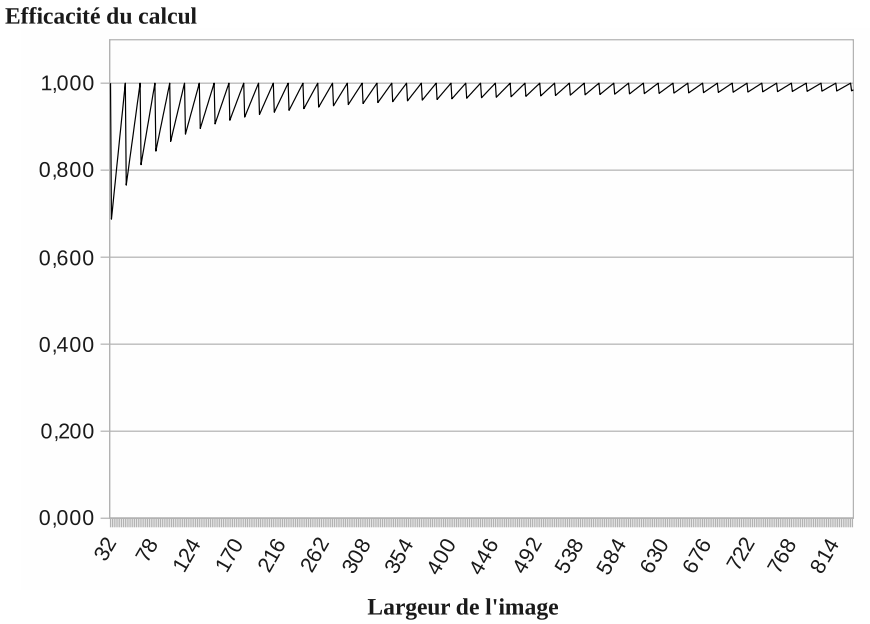


FIG. 4.14: Efficacité du traitement d'opérations de voisinage sur une architecture micro-SIMD de taille 16

On comprend alors mieux pourquoi ce type de jeu d'instructions est limité à une taille de vecteur relativement faible pour qu'en cas de manipulation d'un petit jeu de données, l'efficacité ne soit pas trop mauvaise. En pratique on considère que pour avoir une efficacité des calculs dans le pire cas de l'ordre de 80%, il faut que la taille des vecteurs soit au maximum égale au quart de la taille du jeu de données, ce qui permet tout de même d'atteindre de bonnes vitesses de traitement. De plus, certaines architectures, telles que certains processeurs VLIW, sont capables de masquer les opérations d'accès à la mémoire pendant les calculs améliorant ainsi l'efficacité.

Les architectures embarquables disposant d'un jeu d'instructions micro-SIMD ou vectorielles ne sont pas légion et sont souvent trop limitées pour du calcul intensif. En effet, la plupart de ces processeurs n'ont souvent pas été conçus spécifiquement pour fonctionner avec ce jeu d'instructions, mais ont plutôt subi une modification a posteriori pour le mettre en œuvre. On trouve dans la majorité des cas des processeurs disposant d'unités scalaires travaillant avec des données sur N bits et des unités de calcul vectoriel travaillant avec k éléments de $\frac{N}{k}$ bits. C'est le cas par exemple du processeur ATMEL AVR32, un processeur 32 bits équipé d'instructions micro-SIMD, ne pouvant travailler qu'avec des vecteurs de quatre données 8 bits ou 2 données 16 bits.

Aujourd'hui la taille des images dépasse largement la taille 512×512 et il peut être intéressant d'augmenter sensiblement la largeur des instructions vectorielles. La société Intel à d'ailleurs fait l'annonce de son nouveau jeu d'instructions AVX [36], *Advanced Vector*

Extensions, qui pourront évoluer dans le futur jusqu'à des tailles de 1024 bits contre 128 pour le SSE(x).

4.2.4 Bilan

Ces considérations algorithmiques nous permettent de mieux comprendre quelles attentes avoir en terme d'efficacité d'un parallélisme à grain fin par rapport à un grain moyen en tenant compte des applications à réaliser et de leur complexité. Chacune des méthodes à ses avantages, à savoir une bonne efficacité dans les calculs pour le grain fin, et peut-être une plus grande vitesse de traitement pour le grain moyen si une architecture SIMD massive est utilisée. Toutefois, ces méthodes algorithmiques peuvent être complémentaires plutôt que concurrentes, car il est possible de traiter avec un parallélisme à grain fin le contenu de chaque imagette. On aborde alors des structures de calcul où le parallélisme existe à plusieurs niveaux et donne ainsi la possibilité de disposer d'une puissance de calcul au téra opérations par seconde. A titre d'exemple, si nous considérons le découpage d'une image en 32 imagettes et que nous travaillons sur toutes les imagettes en même temps avec des instructions vectorielles de largeur 128 à une fréquence de 400 MHz il est possible d'atteindre une puissance de calcul brute de l'ordre de 1.6 TOPS en supposant disposer de mémoires dont la bande passante est suffisamment élevée.

4.3 Structure du processeur VLIW vectoriel

4.3.1 Vue générale

L'objectif principal de ce processeur est d'exploiter le parallélisme à plusieurs niveaux, car il n'est pas possible d'augmenter la puissance de calcul uniquement en mettant en place des instructions vectorielles ou uniquement avec un grand nombre d'unités SIMD très larges. En effet, la taille des images que nous souhaitons utiliser impose, en quelque sorte, une limite supérieure que nous ne pouvons pas dépasser en terme de parallélisme exploitable. Il faut donc en plus d'un parallélisme des unités de calcul considérer une autre forme de parallélisme. On peut d'une part rester au niveau imagette et essayer de vectoriser les calculs ou travailler avec plusieurs imagettes à la fois. On peut d'autre part estimer qu'une application n'est pas composée uniquement d'une opération de voisinage et mettre ainsi en place un parallélisme temporel en construisant un flot de processeurs vectoriels. C'est cette seconde option que nous avons choisie et qui nous permettrait d'exploiter une partie des outils de haut niveau développés dans le chapitre 3.

La figure 4.15 montre la structure du processeur que nous avons mise en place. Il s'agit d'un processeur VLIW composé de deux clusters. Le premier est dédié au calcul scalaire et à la gestion des boucles et le second est dédié au calcul vectoriel. Chacun de ces clusters dispose d'une file de registres adaptée a la largeur des unités de calcul ainsi que de connexions vers l'unité de gestion mémoire. Cette dernière, outre le fait d'adresser des mémoires statiques où il est possible de stocker quelques lignes d'une image, permet de transmettre et de recevoir un flux de vecteurs de pixels, vers ou depuis un autre processeur, au travers de files d'attente d'une taille assez faible.

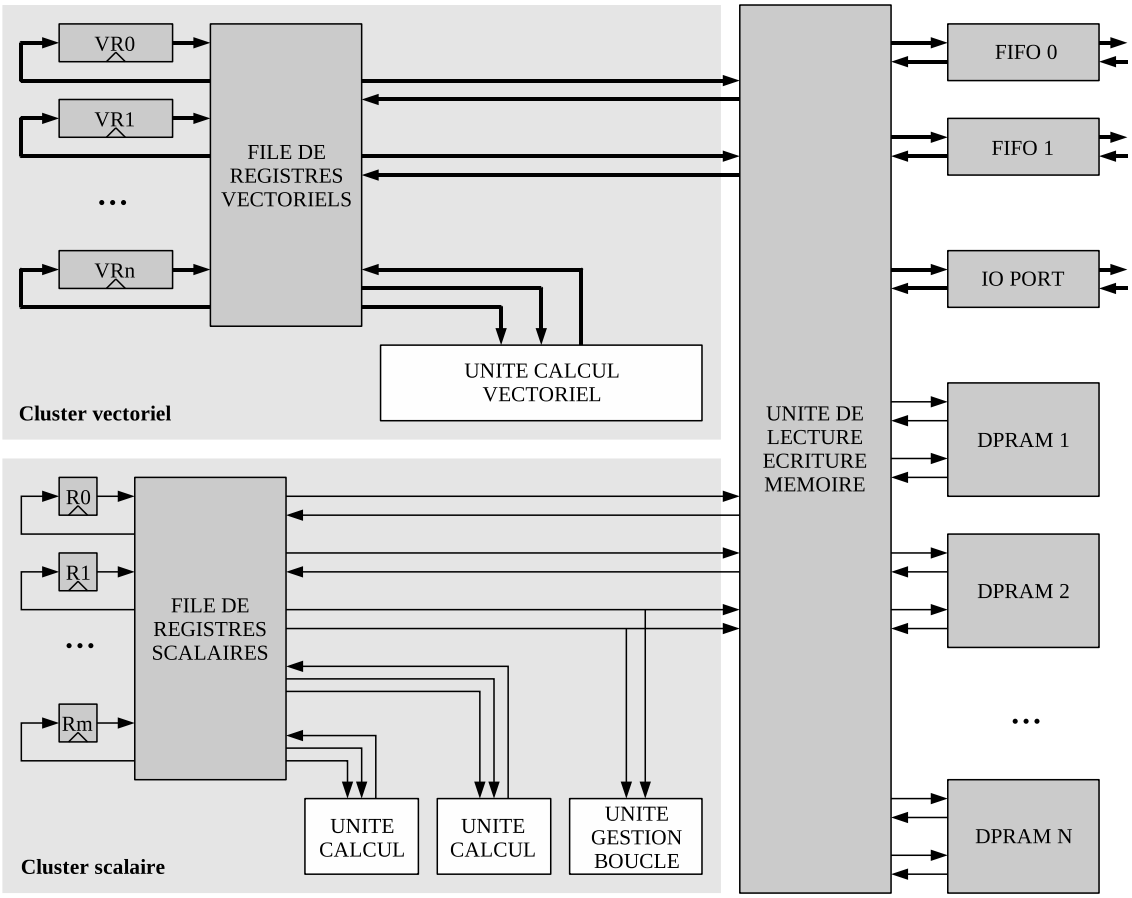


FIG. 4.15: Vue fonctionnelle du processeur VLIW SIMD

L'objectif de ces processeurs est double, à la fois traiter chaque opération de manière vectorielle et traiter plusieurs étapes de l'application dans un même flot, c'est-à-dire exploiter un parallélisme temporel. La topologie des interconnexions est légèrement moins critique que dans le cadre de l'association de processeurs de voisinage décrite dans le chapitre 3, car la versatilité des calculs réalisables ici confère plus de souplesse au déploiement d'une application sur une topologie d'interconnexions statiques. En effet, l'objectif ici est de disposer d'un maximum d'opérations dans chacun des processeurs afin de les utiliser au maximum de leurs possibilités. On utilisera alors une topologie simple de processeurs, comme par exemple celle présentée en figure 4.16, et qui n'a pas besoin d'être aussi riche que ce que nous avons pu décrire dans le chapitre précédent. Nous verrons plus loin dans ce chapitre que d'autres topologies peuvent être mises en place pour disposer de traitements à des granularités différentes.

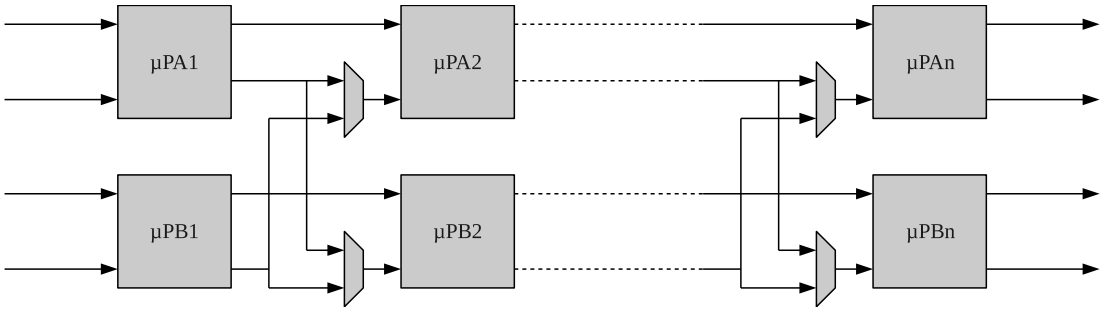


FIG. 4.16: Un exemple d'interconnexion des processeurs dans un flot de n étages

L'objectif de ce processeur n'est pas d'être complètement figé par la structure donnée dans les figures citées plus haut. Trois hiérarchies de parallélisme sont exploitables. Tout d'abord, la largeur de l'unité de calcul vectoriel est paramétrable afin d'améliorer les performances à un grain assez fin. Ensuite, le nombre d'unités de calcul et de ports sur la file de registres vectoriels peuvent être augmenté afin d'améliorer le parallélisme entre les instructions et pouvoir traiter plus efficacement différentes opérations de voisinage, améliorant ainsi le parallélisme à un grain plutôt intermédiaire. Enfin, la possibilité de distribuer les étapes d'une application sur un flot de processeurs nous garantit un dernier niveau d'exploitation du parallélisme non plus spatial, mais temporel, et cette fois à un grain assez élevé.

4.3.2 Vue détaillée

4.3.2.1 Files de registres

Deux files de registres sont utilisées dans ce processeur, l'une dédiée au cluster scalaire et l'autre dédiée au cluster vectoriel. Toutefois, seule la largeur des registres, leur nombre ainsi que le nombre de ports peut varier. On peut donc tout à fait procéder à une description générique de la file de registres et en instancier une différente pour chacun des clusters. La figure 4.17 présente une telle file de registres, on remarque la présence de signaux de validation qui permettent de valider ou non une écriture. Ces signaux sont importants, par

exemple dans le cas d'une erreur de prédiction de branchement afin de ne pas valider la dernière étape du pipeline : le *writeback*.

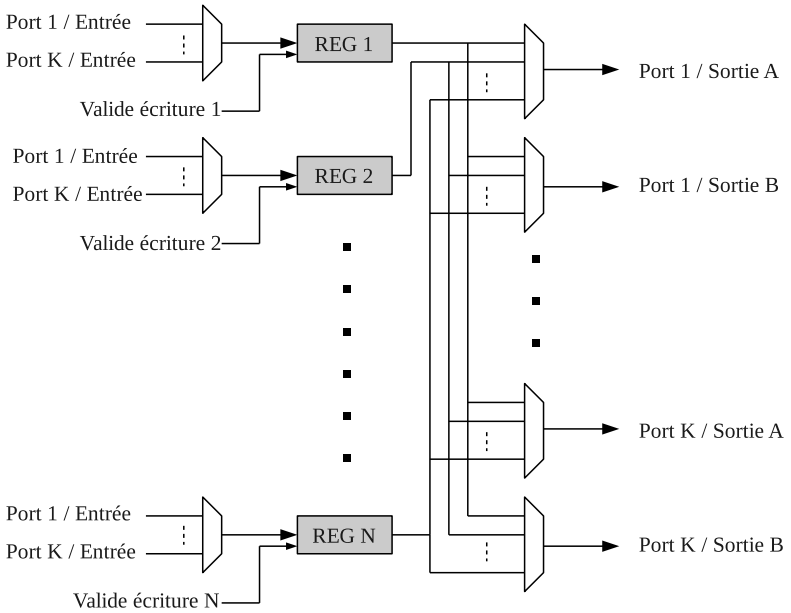


FIG. 4.17: File de registres du processeur VLIW vectoriel à K ports et N registres

On observe également la présence d'un grand nombre d'interconnexions entre les registres et les différents multiplexeurs assurant le bon aiguillage sur les différents ports d'entrée ou de sortie. En effet, le nombre de ports à une très forte influence sur la taille de la file de registres puisque cette dernière augmente de manière quadratique lorsque le nombre des ports croît. Cette complexité est presque exclusivement due au nombre de connexions à réaliser vers tous les ports et se matérialise par une file de registres beaucoup plus grosse et donc une cadence plus faible. La figure 4.18 présente cet état de fait et est surtout valide dans le cadre de la réalisation d'un ASIC. De nos jours le nombre d'interconnexions au sein du circuit à un impact beaucoup plus important que le nombre des registres et il est intéressant, contrairement à il y a une vingtaine d'années, de limiter le nombre de ports plutôt que le nombre de registres.

En pratique on peut sans problème utiliser une file de registres avec une douzaine de ports sans trop contraindre le design du circuit. Toutefois, nous souhaitons exploiter un parallélisme temporel en raccordant plusieurs processeurs en flot, il est alors intéressant de minimiser la taille du circuit. En effet, un grand nombre de ports implique un grand nombre d'unités de calcul ou de raccordement divers ce qui est assez contradictoire avec notre objectif de minimiser la taille du processeur afin d'en disposer plusieurs sur le circuit. De plus, la mise en place d'unités de calcul vectoriel est justement là pour minimiser le nombre de ports et le nombre d'unités de calcul indépendantes, car nous partons du principe que le même noyau de calcul est appliqué à tous les pixels de l'image.

Nous avons présenté, dans la vue générale du processeur, le cluster de calcul vectoriel comme disposant d'une unité de calcul avec quatre ports sur la file de registres de manière à faire simultanément un calcul vectoriel et deux accès mémoires à des adresses différentes.

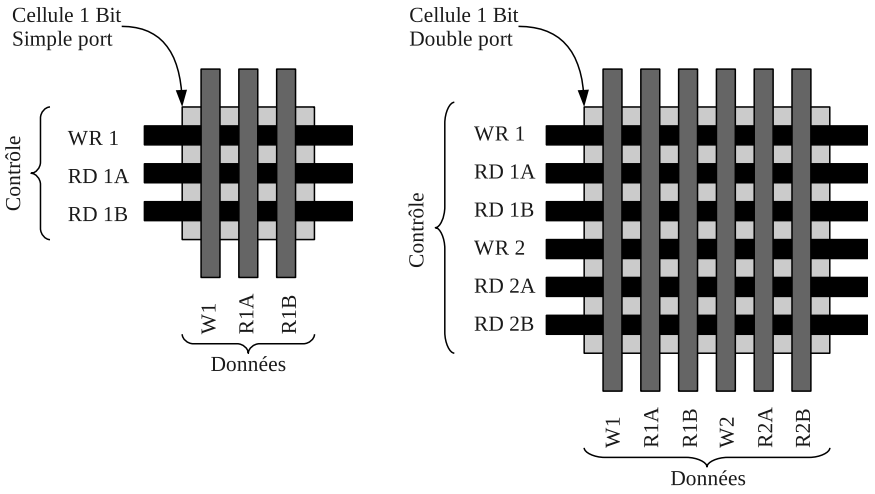


FIG. 4.18: Organisation d'une cellule un bit d'une file de registres multiport

On peut tout à fait partager les liens de communication mémoire avec une unité de calcul supplémentaire pour disposer de deux unités sans augmenter la taille de la file de registres. Cependant, l'utilisation de cette unité de calcul se fera au détriment d'un accès mémoire et peut donc s'avérer inutile si la localité des calculs n'est pas bonne. Il convient donc de bien analyser le groupe d'applications à réaliser pour savoir si ce genre d'optimisation est bénéfique.

4.3.2.2 Unités de calcul

Nous détaillerons principalement ici l'unité de calcul vectoriel, car l'unité de calcul scalaire est tout à fait standard et est utilisée majoritairement dans le cadre de compteurs de boucles et dans la génération d'adresses. Il est toutefois possible de lui ajouter quelques instructions micro-SIMD afin de travailler avec de mini vecteurs composés par exemple de 4 éléments 8 bits ou encore de 32 éléments 1 bit si l'unité scalaire fonctionne en 32 bits. Ces instructions micro-SIMD peuvent s'avérer utiles lorsque l'on souhaite manipuler des images dont les pixels sont stockés sous forme de bits afin de ne pas travailler avec des vecteurs trop grands dans le cluster vectoriel. En effet si le cluster vectoriel fonctionne avec des vecteurs composés de 16 à 32 éléments de 32 bits, un vecteur de pixels codés sur un bit comporte de 512 à 1024 éléments. Ceci peut être parfois très intéressant pour des opérations binaires de voisinage, mais également handicapant dans d'autres opérations telles que des rotations d'images. Nous verrons cet aspect en détail un peu plus loin dans la section 4.4.2 traitant des opérations réalisables par notre processeur.

L'unité de calcul vectoriel, présentée en figure 4.19, se décompose en deux blocs, tout d'abord un bloc permettant de réaliser un décalage à droite ou à gauche du vecteur donné en premier opérande. On trouve ensuite l'unité opérative qui réalise un calcul sur les deux opérandes ou uniquement entre le premier vecteur et ce même vecteur décalé à gauche ou à droite d'un élément.

La mise en place d'une unité de décalage en amont des calculs est justifiée, car elle permet de faciliter les calculs sur des voisinages. En effet nous réalisons l'extraction du

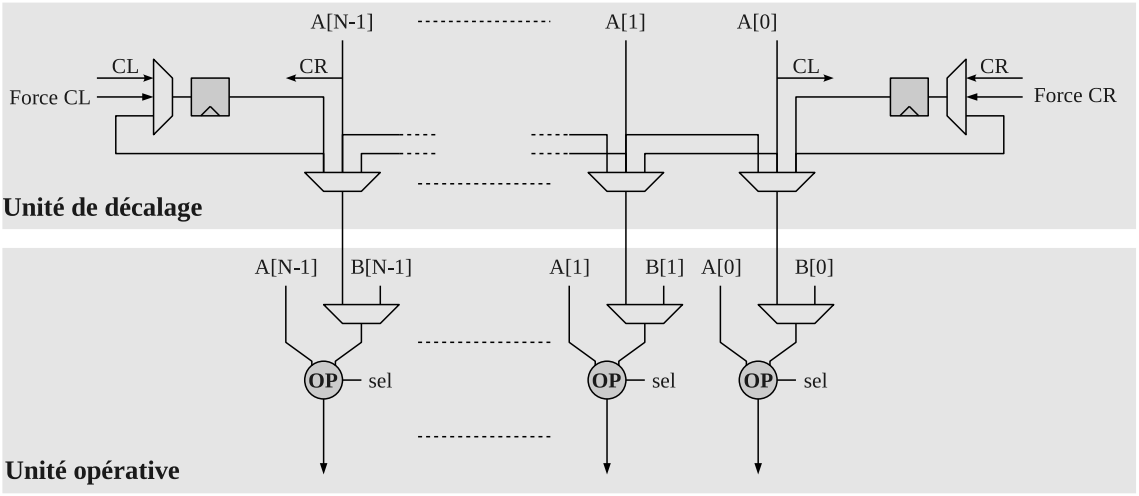


FIG. 4.19: Unité de calcul vectoriel du processeur VLIW

voisinage en procédant à des décalages de lignes comme le montre la figure 4.13 et nous pouvons dorénavant réaliser cette opération sur des vecteurs, c'est-à-dire des paquets de pixels contigus appartenant à une même ligne. Il est bon de pouvoir réaliser cette opération en même temps que le calcul sur le voisinage, car ces vecteurs ainsi décalés peuvent être directement pris en compte dans les calculs sans à avoir à être stockés en mémoire. Les registres recevant les signaux CL et CR permettent, à la manière d'une retenue, de conserver le pixel éjecté lors d'un décalage à droite ou à gauche, pixel qui sera utile pour le calcul au cycle suivant. Ces registres de retenues peuvent être forcés de manière à simuler la valeur du bord hors de l'image lors du traitement du premier vecteur d'une ligne.

L'unité opérative prend en charge les calculs vectoriels suivants :

- Opérations logiques : et, ou, ou exclusif
- Opérations arithmétiques : addition (saturée ou non), soustraction (saturée ou non) et multiplication sous certaines conditions (contraintes de surface)
- Opérations morphologiques : infimum et supremum
- Opérations de déplacement conditionnel
- Opérations de conversion des formats de données et de copie
- Opérations de test des éléments de deux vecteurs

L'opérateur de multiplication est optionnel puisque nous nous limitons à la morphologie mathématique et nous n'utilisons pas d'éléments structurants volumiques. Il est possible d'envisager de câbler cet opérateur puisque l'unité de calcul, grâce à l'unité de décalage, est en mesure de traiter tous types de convolution. De plus, si la cible pour la mise en œuvre du processeur est un FPGA, il est recommandé d'utiliser les blocs DSP câblés de ces derniers pour réaliser l'unité opérative et ainsi bénéficier, entre autres, de l'opération de multiplication sans utiliser de ressources supplémentaires.

Le format des données au sein d'un vecteur correspond normalement au format des données utilisées dans le cluster scalaire. Il peut être intéressant de disposer d'opérations fonctionnant sur un format de donnée plus réduit. On imagine très bien, en supposant l'unité scalaire et les éléments des vecteurs codés sur 32 bits, que l'on dispose d'opérations travaillant sur 8 ou 16 bits. Généralement les imageurs dépassent rarement ce format de

données. Il est alors nécessaire de prendre également en compte cette remarque au niveau de l'unité de décalage pour supporter plusieurs formats de données. On peut même pousser le raisonnement plus loin en donnant l'opportunité aux unités de décalage de supporter un seul bit par élément du vecteur, ce qui permettrait de traiter, avec des opérations logiques simples, le cas de la morphologie mathématique binaire de manière extrêmement efficace.

4.3.2.3 Unités de lecture et écriture mémoire

L'unité de lecture et écriture mémoire a pour but d'une part, de permettre le stockage et la relecture de données scalaires ou vectorielles en mémoire locale et d'autre part, d'accéder aux FIFO de communication assurant l'interconnexion avec d'autres processeurs.

La figure 4.20 montre une vue simplifiée de cette unité en omettant volontairement les FIFO ainsi que leur port IO. Nous reviendrons plus en détail sur ces composants un peu plus loin dans le chapitre.

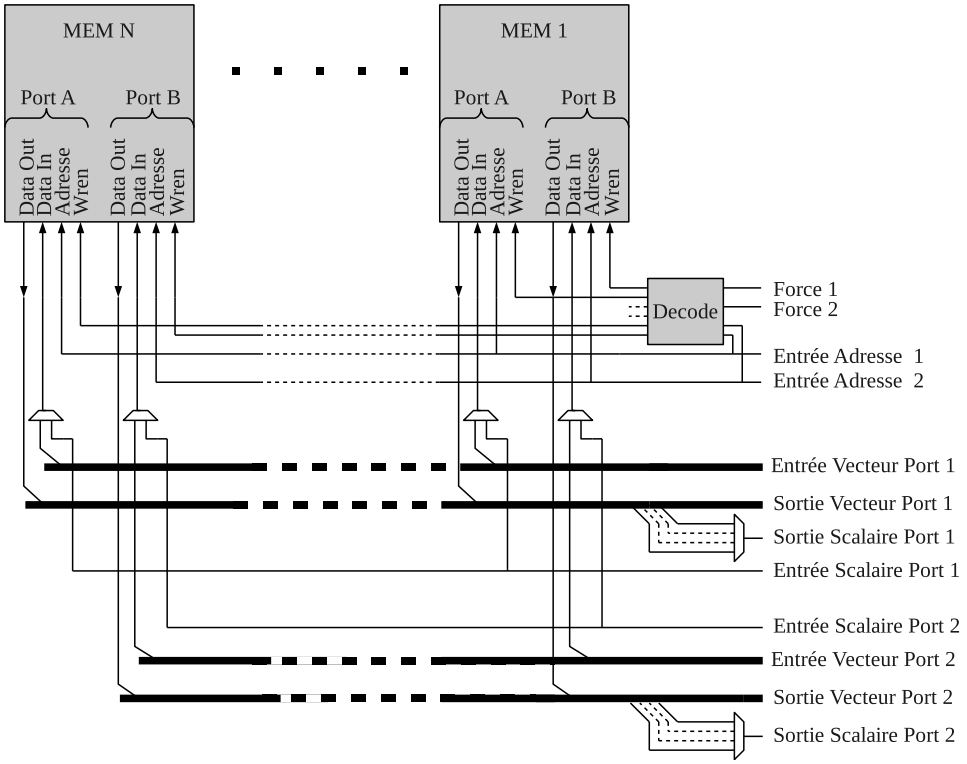


FIG. 4.20: Unité de lecture et écriture mémoire (vue simplifiée)

Cette unité se compose de N mémoires dont la largeur des données correspond à la taille d'un élément d'un vecteur ainsi qu'à la largeur des mots utilisés dans l'unité scalaire, la valeur typique étant 32 bits. Le nombre N de mémoires est donc directement lié à la taille des vecteurs. Cette unité se raccorde à la fois sur la file de registres du cluster vectoriel et sur la file de registres du cluster scalaire. Il est possible de réaliser un accès par port et chacun des ports se voit partagé exclusivement entre un accès vectoriel et un accès scalaire. Les multiplexeurs situés avant l'entrée des données des mémoires permettent de sélectionner si une donnée scalaire ou vectorielle doit être prise en compte. Dans le

cas d'une écriture mémoire scalaire, la même donnée est présentée aux mêmes ports de toutes les mémoires, mais la validation d'une unique mémoire est assurée grâce au bloc de décodage. Ce dernier génère donc tous les signaux de validation en écriture et peut donc soit activer un unique port d'une mémoire en écriture, soit activer les mêmes ports de toutes les mémoires pour écrire une donnée vectorielle via les signaux *force1*, *force2*. Les entrées d'adresses sont directement reliées au cluster scalaire ce qui confère une bonne souplesse dans la manipulation de pointeurs pour lire et écrire des données en mémoire.

La taille des mots mémoire est conçue pour contenir une donnée scalaire et la gestion des adresses permet d'accéder à tous les mots mémoire. Un vecteur en mémoire s'étale donc sur N adresses, où N correspond au nombre d'éléments dans un vecteur, et les accès sont obligatoirement alignés pour simplifier cette unité. La figure 4.21 présente l'organisation de la mémoire et montre comment accéder aux périphériques comme les FIFO.

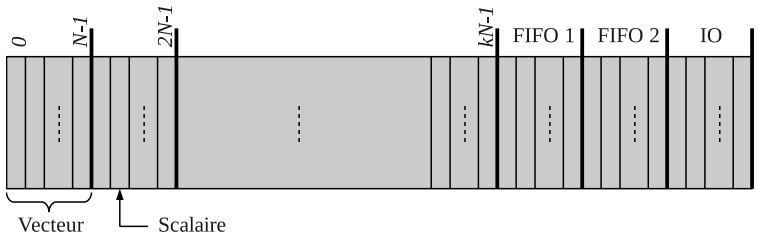


FIG. 4.21: Mapping mémoire du processeur VLIW vectoriel

Un aspect que nous n'avons pas exposé dans le schéma de l'unité est de pouvoir réaliser de l'adressage indirect vectoriel. En effet, les adresses des mémoires peuvent être contrôlées soit de manière globale pour accéder *normalement* aux vecteurs en mémoire statique, soit de manière indépendante pour réaliser un accès différent dans chacune des mémoires en un seul cycle. Ce mode d'adressage assez ésotérique a pour but principal de réaliser des histogrammes d'images de manière parallèle. En effet, on peut ainsi réaliser N histogrammes partiels de l'image pour ensuite les réunifier grâce à l'unité scalaire et transmettre ainsi le résultat complet.

L'accès au FIFO se fait simplement en écrivant ou en lisant à une adresse précise. Matériellement, cela se fait simplement en les raccordant aux entrées et sorties vectorielles de l'unité et en ajoutant quelques sorties de validation à l'unité de décodage.

Les IO sont accessibles depuis l'espace d'adressage et peuvent être utilisées dans le but de passer des informations globales relatives à certaines constantes à utiliser dans le programme par le processeur. On peut donc imaginer qu'un système hôte communique avec les processeurs via ces IO pour les informer de la taille des images ou pour démarrer les traitements.

4.3.2.4 Unité de gestion de boucles et branchements

Afin de simplifier la gestion des boucles, une unité spécialisée est dédiée à la manipulation du PC, *compteur de programme* en vue de mettre en œuvre des boucles sans avoir de problèmes de prédiction de branchements. En effet, la manipulation des lignes avec des vecteurs de grandes tailles va minimiser le nombre des itérations d'une boucle. Il risque ainsi

de se produire assez souvent des erreurs de branchement et un nombre assez important de cycles risque d'être perdu ce qui diminue le rendement des calculs.

Considérons l'exemple où l'on souhaite éroder une ligne de 512 pixels sur un processeur vectoriel travaillant avec k éléments simultanément. L'élément structurant que nous considérons est un segment de longueur 2, et l'opération consiste à calculer l'infimum d'un pixel avec son voisin pour tous les pixels de la ligne. Le programme en pseudo assembleur est donnée à l'algorithme 9.

Algorithme 9 Pseudo-langage machine d'une érosion d'une ligne de pixels

```
/* Prologue */
vecload v1,FIFO0
vecload v1,FIFO0; rshiftinf v2,v1,BORDSVAL

/* État permanent */
Pour r0=0 to 512/k-2 Faire
  vecload v1,FIFO0; rshiftinf v2,v1,C; vecstore v2, FIFO1; add r0,r0,1
Fin pour

/* Épilogue */
rshiftinf v2,v1; vecstore v2, FIFO1
vecstore v2, FIFO1
```

Chacune des lignes du programme comporte une ou plusieurs instructions exécutées en parallèle. Pour éroder correctement une ligne de pixels, il est nécessaire de mettre en place un pipeline logiciel [22] [82] [45] qui dans notre cas consiste à réordonnancer les accès mémoires et les calculs dans la boucle de traitement afin de pouvoir exploiter le parallélisme au niveau des instructions de notre processeur tout en disposant d'un code plus compact à l'intérieur de la boucle. Le prologue permet de charger les deux premiers vecteurs et de commencer le premier calcul, calcul qui pourra être envoyé à la FIFO en sortie dans l'état permanent pendant que les vecteurs suivants sont traités. L'épilogue permet d'effectuer le dernier calcul et d'envoyer à la FIFO les deux derniers vecteurs.

La boucle *for* de cet exemple est volontairement peu détaillée puisque nous avons la possibilité de la mettre en œuvre soit via l'unité de gestion de boucle, soit de manière purement logicielle en utilisant une instruction de comparaison et de branchement.

Étudions tout d'abord ce qui se passerait avec une approche classique par comparaison et branchement. Plusieurs possibilités s'offrent à nous. Une première, si nous désirons simplifier la vie du programmeur ou simplifier la tâche du compilateur, est de disposer d'une unique instruction réalisant à la fois le test et le branchement. Toutefois, ce principe est assez inefficace et fait perdre des cycles. La figure 4.22 présente pour un pipeline à cinq étages (*Fetch, Decode, Execute, Memory access, Write back*) comment est affecté ce dernier par une instruction de branchement regroupant le test et le saut. Les instructions *nop* représentent les instructions qui pourraient être exécutées bien que le saut soit effectué. Il convient alors de toujours mettre un certain nombre de *nop* après une telle instruction de branchement, nombre variant en fonction de la profondeur du pipeline.

Une seconde possibilité serait de découper l'instruction de test et de branchement en deux instructions séparées afin de pouvoir réaliser le test, intercaler quelques opérations

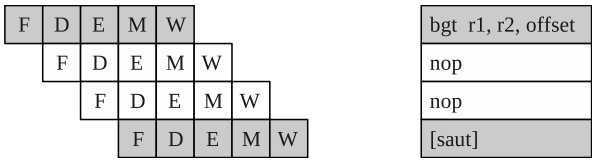


FIG. 4.22: Vue d'un pipeline à 5 étages lors de l'exécution d'une instruction de branchement monolithique

utiles et seulement ensuite lancer l'instruction de branchement qui effectuera un saut en fonction du résultat du test. On minimise alors le nombre d'instructions *nop* puisque le branchement peut être réalisé plus tôt dans le pipeline du fait qu'il n'est plus nécessaire de calculer le résultat du test au niveau du branchement. Toujours avec le même type de pipeline que dans le cadre de la figure 4.22, la figure 4.23 présente la façon dont le pipeline est affecté par un découpage en deux de l'instruction de test et de branchement. Toutefois, si nous n'arrivons pas à mettre deux instructions entre le test et le saut cette solution s'avère assez mauvaise comparée à la solution avec une seule instruction pour le test et le branchement.

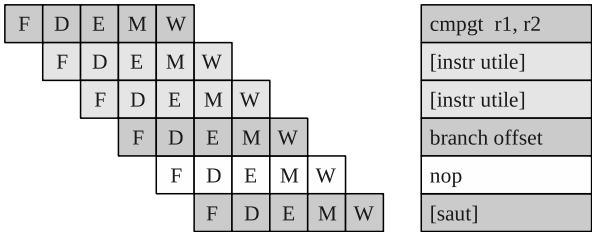


FIG. 4.23: Vue d'un pipeline à 5 étages lors de l'exécution d'une instruction de branchement en deux étapes

Ainsi, dans le cas de notre programme d'érosion, le nombre de cycles nécessaires à l'exécution de la boucle en considérant l'incrémentation du compteur, le test, le saut et l'exécution du cœur de la boucle, occupe $5(k-2)$ cycles. En effet l'incrémentation du compteur de la boucle peut être regroupée avec le paquet d'instructions réalisant une étape de l'érosion.

Maintenant que nous avons étudié le cas où nous souhaitions mettre en œuvre la boucle par des instructions classiques de branchement, il est intéressant d'observer quel gain peut nous apporter une unité de gestion des boucles. En effet, une telle unité prend en charge complètement l'incrémentation du PC et supprime toutes les instructions de gestion de la boucle telles que les comparaisons, les branchements et l'incrémentation du compteur. Ce type d'unité peut bien évidemment fonctionner uniquement dans le cadre d'une boucle dont la condition d'arrêt dépend d'un compteur incrémenté à chaque étape. Dans notre exemple, le nombre de cycles machine nécessaires à la réalisation de la boucle est $k-2$ cycles. On observe donc un réel gain de performance par rapport à un système de branchement classique, bien qu'il soit important de mentionner que cet écart se réduirait si d'avantages d'instructions utiles étaient présentes dans le cœur de la boucle.

La structure de l'unité de gestion de boucle, figure 4.24, repose sur la manipulation du PC en fonction d'arguments donnés en entrée, à savoir l'adresse de fin de la boucle et le nombre d'itérations. L'adresse du début de la boucle étant déduite de la valeur du PC lors

de la configuration de la boucle. Le fonctionnement de cette unité repose sur l'initialisation matérielle d'un compteur à zéro et sur l'incrémentation automatique de ce dernier à chaque fois que le PC est égal à l'adresse de fin de boucle. Il faut également prendre en compte un aspect important qui est de pouvoir mettre en place des boucles imbriquées. C'est le rôle de la pile de sauvegarder les trois paramètres d'une boucle en cours d'exécution, lorsqu'une nouvelle boucle imbriquée dans la première doit être mise en place. Une fois cette boucle imbriquée terminée, la pile permet la restauration des arguments de la première boucle, lui permettant de poursuivre son exécution normalement. La profondeur de la pile est un paramètre permettant de spécifier le nombre maximum d'imbrications de boucles. Une profondeur de 8 niveaux dans la pile est souvent suffisante dans le cadre du traitement d'images.

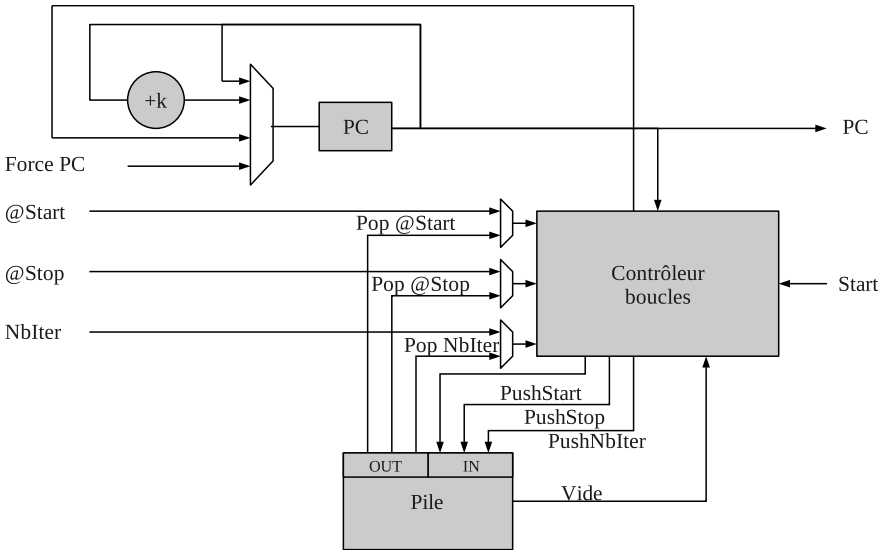


FIG. 4.24: Structure fonctionnelle de l'unité de gestion des boucles

4.3.2.5 Parallélisme des instructions

Le jeu d'instructions de ce processeur est relativement standard et les instructions sont regroupées par paquet de huit pour former un bundle VLIW. La structure du bundle est présentée dans le tableau 4.1.

MEM IO	MEM IO	MEM IO	MEM IO	VEC ALU	ALU	ALU	CTRL

TAB. 4.1: Structure du bundle d'instructions

La description des différents champs est donnée ci-après :

– MEM IO : le processeur est capable de réaliser quatre opérations simultanées avec l'unité de gestion mémoire. On peut par exemple effectuer deux accès aux FIFO et deux accès mémoires dans un même cycle. Il faut bien faire attention au fait que seulement deux instructions simultanées peuvent accéder à la mémoire statique du processeur avec des adresses différentes. Le seul moyen de réaliser quatre accès dans

la mémoire statique est de lancer une écriture et une lecture sur un port avec la même adresse. On peut ainsi sur chaque port lire une donnée avant qu'elle ne soit écrasée.
- VEC ALU : contrôle les opérations réalisées par l'unité de calcul vectoriel. Elle prend en charge, en plus des opérations arithmétiques et logiques déjà détaillées précédemment, les copies des registres vectoriels.
- ALU : deux opérations scalaires peuvent être réalisées simultanément. Ces unités de calcul sont principalement là pour gérer la mise à jour des adresses lors des accès en mémoire, pour réaliser des tests et pour calculer l'adresse du PC en cas de branchement.
- CTRL : champs de configuration de l'unité de gestion des boucles et de mise en place des instructions de synchronisation des FIFO. La mise en place d'une boucle requiert l'adresse de fin de la boucle et le nombre de répétitions, l'adresse de début de la boucle étant déduite de la valeur du PC lorsque la boucle est mise en place. Les instructions de synchronisation FIFO permettent de bloquer le PC tant que la FIFO concernée est pleine ou vide selon qu'un accès en écriture ou lecture est effectué.

4.3.2.6 Interconnexions

Le mécanisme de communication externe du processeur est important puisqu'il permet de mettre en place un système puissant composé de plusieurs processeurs raccordés les uns aux autres. On peut alors réaliser plusieurs opérations, exécutées normalement séquentiellement, sous forme d'un flot de calcul en exploitant un parallélisme temporel.

Chaque processeur dispose de deux FIFO d'entrée et de deux signaux de sortie qui seront raccordés aux FIFO d'entrée de différents processeurs comme le montre la figure 4.25. Les flèches fines symbolisent le transit des données scalaires et les flèches épaisses symbolisent le transit des données vectorielles. Si les vecteurs sont vraiment trop larges vis-à-vis du circuit cible, il est possible de multiplexer de manière temporelle les transferts et compenser cette perte de temps en utilisant deux domaines d'horloges, un pour le processeur et un autre beaucoup plus élevé pour les transferts entre processeurs. On peut également envisager un transfert à l'extérieur du circuit par le biais de communications série et parallèle rapides. On peut citer par exemple le bus HyperTransport [75] offrant une bande passante théorique de 41.6 Go/s dans sa troisième version. Un tel type d'interconnexion fait penser à la machine Transputer [6], présentée au chapitre 2.

Le débit maximal des communications est directement lié au format des vecteurs employés ainsi qu'à la fréquence interne du processeur. On peut atteindre un débit de 12.8 Go/s avec des vecteurs de 32 éléments de 32 bits et une fréquence de fonctionnement de 100 Mhz.

Le nombre de processeurs utilisés et la topologie des interconnexions mises en place a un impact moins critique sur les opérations de traitement d'images réalisables que dans le cadre des opérateurs flots de données décrits dans les chapitres 2 et 3. En effet, la possibilité de regrouper plusieurs opérations de voisinage ainsi que plusieurs opérations arithmétiques au sein d'un même processeur simplifie le déploiement d'une application. Toutefois, il faut bien veiller au fait que le processeur le plus chargé, c'est-à-dire avec la cadence la plus faible en terme d'envoi de données sur le bus d'interconnexion, imposera sa cadence à tout le circuit. Il est impératif de procéder à un découpage homogène de l'application sur tous les processeurs.

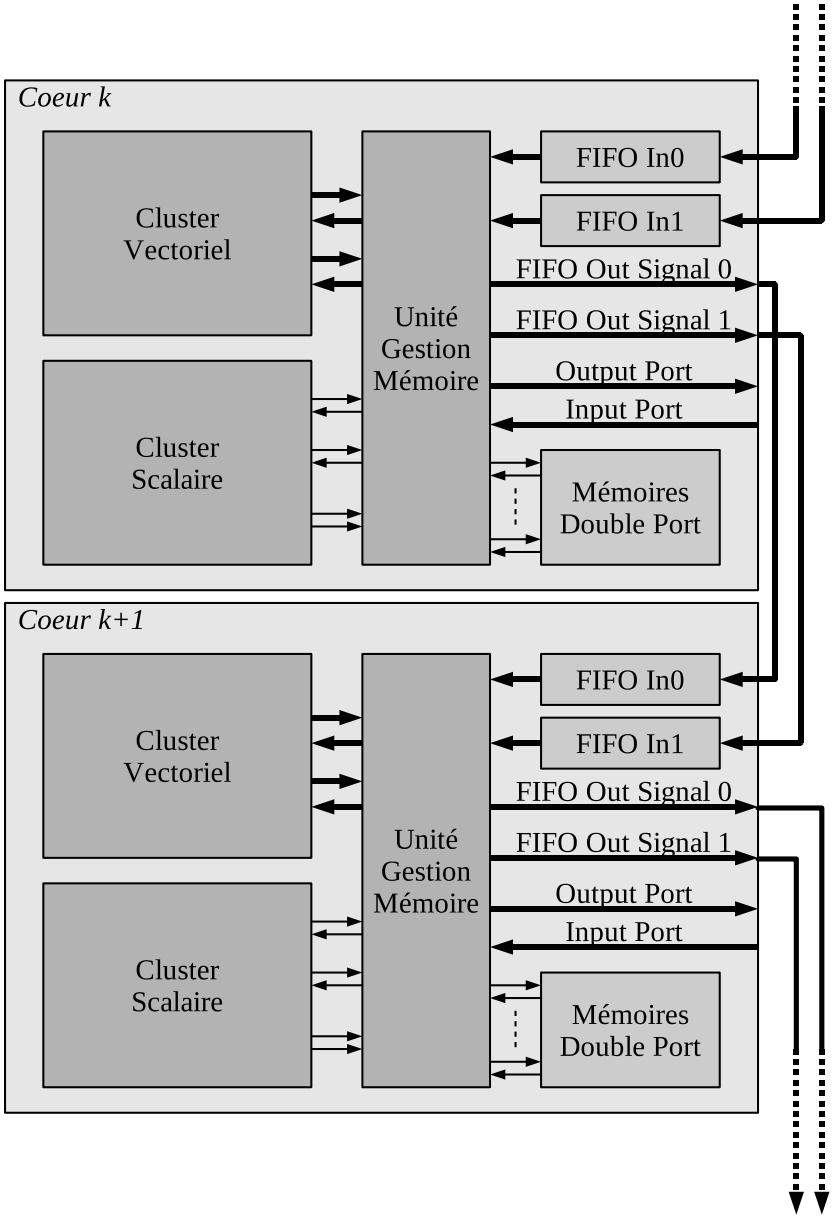


FIG. 4.25: Raccordement des cœurs vectoriels via des FIFO

Le système d'interconnexion peut être employé pour travailler à trois niveaux de granularité. Nous avons effectivement parlé jusqu'à présent d'un parallélisme au niveau ordonnancement des opérations de traitement d'images impliquant de travailler avec des processeurs principalement raccordés en série, figure 4.25. Nous pouvons également mettre en place un parallélisme à grain fin en éclatant une opération sur plusieurs processeurs associés en série ou en parallèle. Par exemple, dans le cadre d'une opération de voisinage, il est possible d'utiliser un processeur par groupe de voisins. Mais nous pouvons aussi passer à un grain moyen en considérant une association des processeurs en parallèle en traitant sur chacun d'entre eux une portion différente de l'image avec la même opération.

La figure 4.26 présente un tel système où l'on opère un gradient sur chaque portion d'une image. Il faut bien entendu considérer des zones de recouvrement à cause de l'opération de voisinage. Nous avons choisi ici de découper l'image juste dans le sens de la hauteur afin de simplifier la tâche des DMA acheminant les pixels vers les différents processeurs. Toutefois, si des DMA 2D sont utilisés, le découpage en imagette peut être réalisé à la fois en hauteur et en largeur.

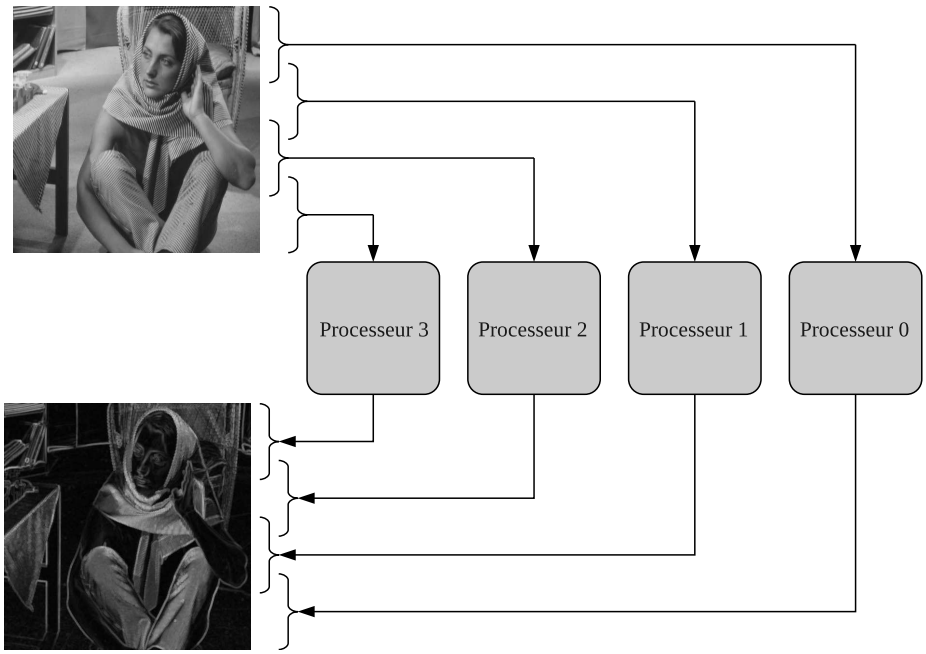


FIG. 4.26: Exploitation des processeurs en parallèle

Afin de pouvoir réaliser les calculs à la fois dans un mode gros grain et grain moyen, on peut envisager la topologie d'interconnexion de la figure 4.27. En utilisant les quatre entrées vers les processeurs $\{A0, A1, A2, A3\}$ on peut exécuter la même opération sur quatre portions de l'image et obtenir ainsi un traitement à grain moyen. Les données sont ensuite transmises horizontalement vers la série de processeurs Bn qui pourront à leur tour réaliser une opération sur les quatre portions de l'image et ainsi de suite. Comme nous avons pu l'expliquer dans la section 4.2, si nous souhaitons réaliser plusieurs opérations successivement sur les imagettes il est nécessaire de prendre en compte une zone de recouvrement épaisse pour supporter toutes les opérations de voisinage réalisées dans les processeurs An, Bn, Cn et Dn. Cette structure est fort intéressante, car elle respecte les recommandations

émises dans la section 4.2. En effet, si nous travaillons en exploitant le traitement de quatre imagettes en parallèle nous pouvons chaîner qu'un nombre très réduit d'opérations, tandis que si nous travaillons directement au niveau image, le flot de processeurs peut être beaucoup plus long.

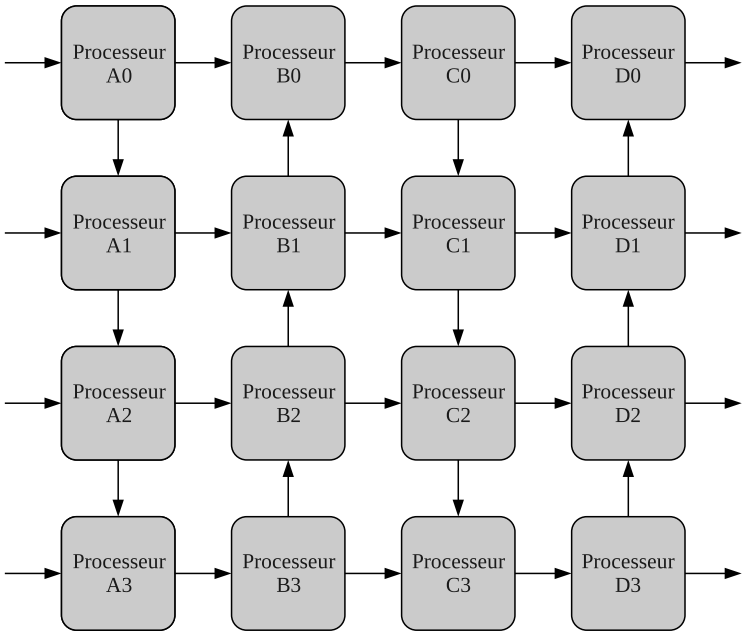


FIG. 4.27: Topologie d'interconnexion permettant l'exploitation des processeurs à moyen et gros grain

Le traitement à gros grains est obtenu en envoyant l'image à traiter dans le processeur $A0$, ce dernier renvoie les premiers résultats disponibles immédiatement au processeur $A1$, qui à son tour transmettra ces premiers résultats au processeur $A2$ et ainsi de suite jusqu'à ce que l'image complètement traitée sorte par le processeur $D0$. On obtient ainsi la séquence de traitement suivante : $A0$, $A1$, $A2$, $A3$, $B3$, $B2$, $B1$, $B0$, $C0$, $C1$, $C2$, $C3$, $D3$, $D2$, $D1$, $D0$.

La figure 4.28 illustre comment utiliser les diverses connexions en fonction de la granularité des opérations à réaliser. L'intérêt de basculer d'une granularité à l'autre est de pouvoir mettre en place un pipeline court pour des opérations spécifiques ou un pipeline profond dans le cadre d'opérations bien précises telles que des filtres alternés séquentiels ou encore une chaîne de traitement assez statique comportant beaucoup d'opérations simples.

4.3.2.7 Programmation des processeurs

Nous n'avons pas encore parlé de la mémoire des instructions du processeur. Cette mémoire est séparée de la mémoire des données et dispose d'une largeur correspondant à un mot d'instructions VLIW. Mais ce n'est pas tant la géométrie de cette mémoire ou même sa taille qui nous préoccupe, mais plutôt les moyens d'accès mis en œuvre pour programmer efficacement un réseau de processeurs.

A chaque lancement d'un flot d'opérations, si ce dernier est différent du flot précédent, il est nécessaire de charger les programmes à exécuter dans les processeurs. En considérant la

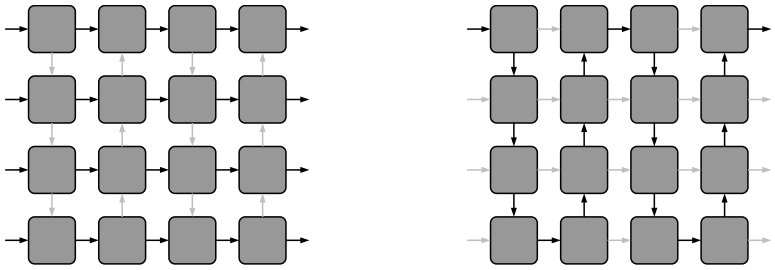


FIG. 4.28: Exemple de topologie d'interconnexions pour un traitement à moyen et gros grain

topologie présentée en figure 4.27, il faut programmer séquentiellement les seize processeurs ce qui peut être peu efficace au vu des opérations à réaliser. En effet, il se peut que tous les processeurs ou certains sous-ensembles remplissent la même fonction et disposent donc des mêmes instructions. Il serait judicieux de prévoir plusieurs groupes de processeurs dont les mémoires d'instructions puissent être adressés en même temps.

Nous prévoyons de raccorder simplement les mémoires sur un bus en liaison avec le système hôte et dont le décodage des adresses, spécifiant normalement l'accès à une mémoire particulière, serait géré de manière logicielle par un registre. On peut de cette manière choisir les mémoires qui doivent être programmées avec le même microcode, et disposer ainsi de groupes de mémoires définis par l'utilisateur.

4.4 Exemples d'opérations réalisables

4.4.1 Avant propos

Maintenant que nous avons détaillé la structure du processeur ainsi que certaines topologies d'interconnexions, nous allons voir comment réaliser quelques opérations de traitement d'images et principalement de morphologie mathématique. Nous n'allons pas ici donner les programmes détaillés au niveau assembleur, mais plutôt présenter la manière de paralléliser les opérations sur notre paquet de processeurs vectoriels. Nous supposons disposer d'un nombre limité de processeurs, associés comme présenté à la figure 4.27, nous permettant à la fois de disposer d'un flot court de processeurs avec une forte capacité de calcul parallèle ou bien d'un flot profond de processeurs dont les possibilités de chaînage d'opérations sont élevées.

Les algorithmes présentés ci-dessous s'appuient en partie sur une bibliothèque d'opérations basées sur le traitement des lignes. Un exemple d'une telle opération en pseudoassembleur a été donné à l'algorithme 9 et permet d'éroder un pixel et son voisin sur une ligne d'une image. Nous supposons, de la même manière, disposer d'opérations nous permettant de :

- recevoir et envoyer une ligne à travers les canaux de communication d'un processeur
- éroder ou dilater une ligne avec un élément structurant segment de taille 3 et centré

– calculer des opérations arithmétiques telles que l'infimum ou le supremum de deux lignes sur les pixels pris deux à deux
– décaler des lignes
– trier deux à deux les pixels de deux lignes
– chercher le minimum ou le maximum d'une ligne
– sommer les pixels d'une ligne

4.4.2 Rotation d'images

Les rotations d'images sont des opérations simples dans l'absolue, mais pouvant être gourmandes en temps de calcul et difficile à mettre en œuvre dans des architectures de type flot de données, car elles nécessitent un accès aléatoire à tous les pixels de l'image.

Un processeur VLIW vectoriel peut, en fonction de la quantité de mémoire présente et de la taille des images, travailler sur des images binaires dans leur totalité pourvu que les pixels soient réellement codés sur un bit. On prendra par exemple un processeur ayant une ALU scalaire sur 32 bits, travaillant avec des vecteurs de 16 éléments 32 bits et disposant d'une mémoire totale de 32 ko. Il est alors possible avec un tel processeur de stocker une image binaire d'une taille pouvant aller jusqu'à 2^{20} pixels, comme une image 1024×1024.

Si nous désirons réaliser des rotations ou des symétries sur une image en niveau de gris et qu'il est nécessaire dans certains cas de disposer de l'image dans sa totalité dans la mémoire du processeur, il est possible de réaliser ces opérations en découpant l'image par plans de bits. Ainsi, une rotation d'image 8 bits sera réalisée en découpant le problème en huit rotations d'images codées sur un bit.

La figure 4.29 présente une vue générale permettant de mieux cerner le fonctionnement d'opérations de traitement d'images par plans de bits. Il faut bien évidemment que les traitements soient indépendants d'un plan de bit à l'autre et la figure présente une opération de rotation de 33° d'une image codée sur 8 bits.

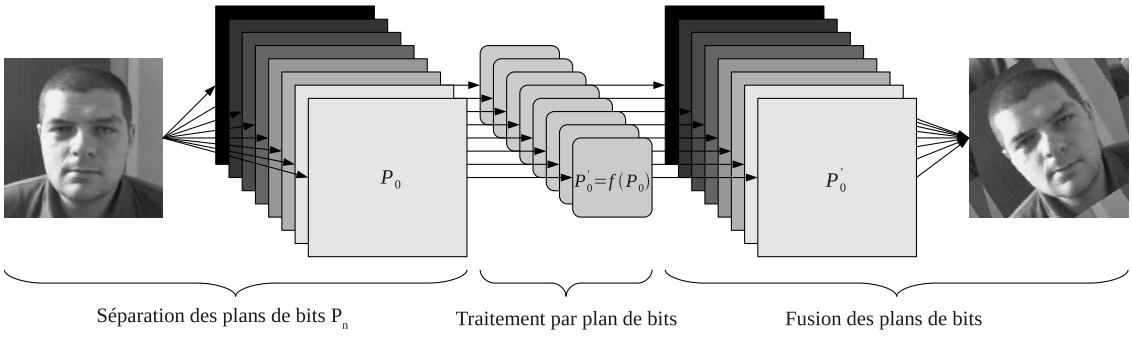


FIG. 4.29: Chaîne de traitement d'images sous forme de plans de bits

Les images envoyées au premier processeur du réseau sont dans le format 8 bits, ce dernier doit donc sélectionner un bit parmi les 8 à traiter et construire des vecteurs de 512 pixels (en effet, nous disposons de 16 éléments 32 bits dans un vecteur) à transmettre au processeur suivant. Une fois les traitements réalisés par un certain nombre de processeurs, il convient de recoder l'image 1 bit sous 8 bits en affectant à chaque pixel une puissance de deux correspondant à la position du bit sélectionné dans le premier processeur. On

réitère ensuite toutes ces opérations sept fois pour obtenir huit images qu'il faudra ensuite fusionner afin d'obtenir le résultat final. Chacune des huit images étant codée par une unique puissance de deux, la fusion est simplement réalisée par une opération de *ou* logique.

La figure 4.30 montre comment est réalisée la fusion des données. On utilise le réseau de processeurs en parallèle afin de grouper quatre images binaires en une image 8 bits. Il faut encore répéter cette opération une seconde fois pour grouper les quatre dernières images binaires dans une seconde image 8 bits. Une troisième étape est alors nécessaire pour fusionner les deux images 8 bits partiellement reconstruites afin de disposer de l'image 8 bits complète. Les deux premières étapes utilisent les processeurs *An* en parallèle pour conditionner les pixels sous forme de bits en pixels sous forme d'octets. Les processeurs *Bn* opèrent les *ou* logiques pour construire l'image 8 bits reconstruite partiellement. La troisième étape utilise le flot de processeurs en série pour associer les deux images partielles en une image complète et permet d'ordonnancer encore quinze opérations derrières.

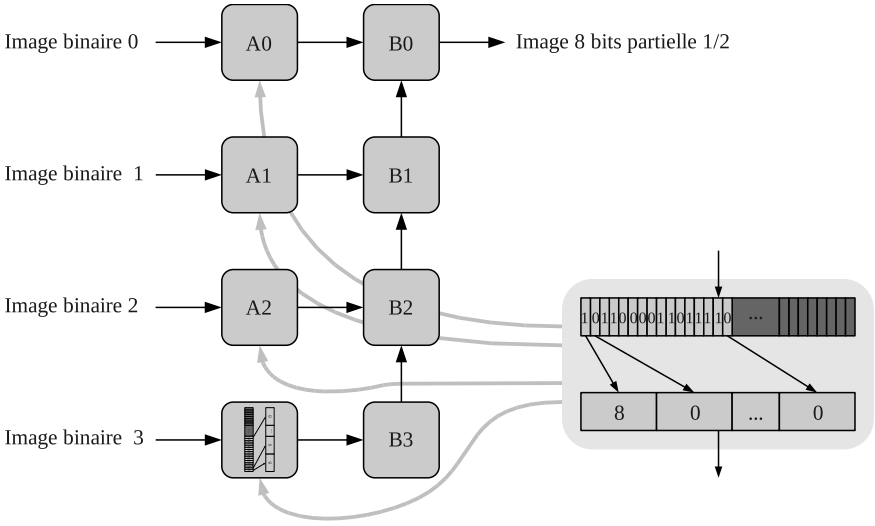


FIG. 4.30: Fusion d'images binaires vers une image 8 bits

Maintenant que nous avons vu comment séparer et regrouper les différents plans de bits d'une image, nous allons voir quelles opérations sont réalisables sur ces derniers. Nous allons détailler en particulier les cas de la rotation d'un angle de $\frac{\pi}{2}$ et de la rotation d'un angle arbitraire grâce à différents décalages des colonnes et des lignes d'une image. Nous allons également présenter des méthodes de traitement par décomposition des plans de bits dans notre réseau de processeurs.

4.4.2.1 Rotation par pas de $\pm\frac{\pi}{2}$

Les rotations d'images d'un angle de $\pm\frac{\pi}{2}$ en utilisant des instructions SIMD s'opèrent de la même manière que les transpositions de matrices et peuvent s'opérer par plan de bit. Ces opérations très bien détaillées par Brambor [15], reposent sur l'utilisation d'opérations dites de *shuffling*, qui ont pour but de mélanger deux vecteurs.

Nous allons dans un premier temps étudier le cas de la transposition de blocs de données et nous placer dans un cas où les vecteurs ont une taille réduite afin de faciliter la

compréhension de la méthode. Prenons des vecteurs contenant 8 éléments ainsi que les instructions de mélange suivantes :

- *shuffleLo8* : prend deux vecteurs de huit éléments et mélange les quatre premiers éléments des vecteurs
- *shuffleHi8* : prend deux vecteurs de huit éléments et mélange les quatre derniers éléments des vecteurs
- *shuffleLo4* : prend deux vecteurs de quatre éléments et mélange les deux premiers éléments des vecteurs
- *shuffleHi4* : prend deux vecteurs de quatre éléments et mélange les deux derniers éléments des vecteurs
- *shuffleLo2* : prend deux vecteurs de deux éléments et mélange les premiers éléments des vecteurs
- *shuffleHi2* : prend deux vecteurs de deux éléments et mélange les derniers éléments des vecteurs

Les vecteurs ont tous la même taille et lorsque nous parlons d'un vecteur contenant deux éléments nous entendons que la taille des éléments et quatre fois plus importante que la taille des éléments des vecteurs disposant de huit éléments. La figure 4.31 montre comment ses instructions fonctionnent.

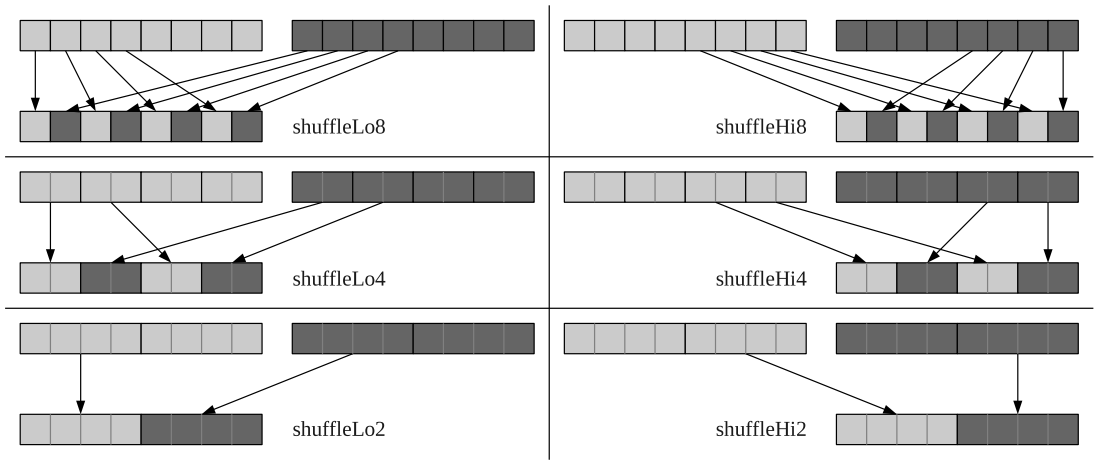


FIG. 4.31: Instructions de mélange de vecteurs

Un bon ordonnancement de ces instructions permet de transposer un bloc de données en trois étapes composées de huit opérations. La complexité de cet algorithme en termes du nombre d'instructions de mélange est exactement $N \cdot log(N)$ où N correspond au nombre de lignes et de colonnes du bloc de données et correspond également au nombre d'éléments d'un vecteur dans le processeur.

La figure 4.32 présente les différentes étapes de calcul de la transposition et l'on remarque que l'on utilise une paire d'instructions de mélange par étape. Si les vecteurs comportaient 64 éléments, il serait donc nécessaire, pour transposer un bloc, de disposer de six paires d'instructions, chacune utilisable dans une des six étapes du calcul.

Maintenant que nous savons comment transposer un bloc par pas de $\frac{\pi}{2}$ il suffit, pour transposer une image entière, de réaliser une transposition de manière standard, mais en

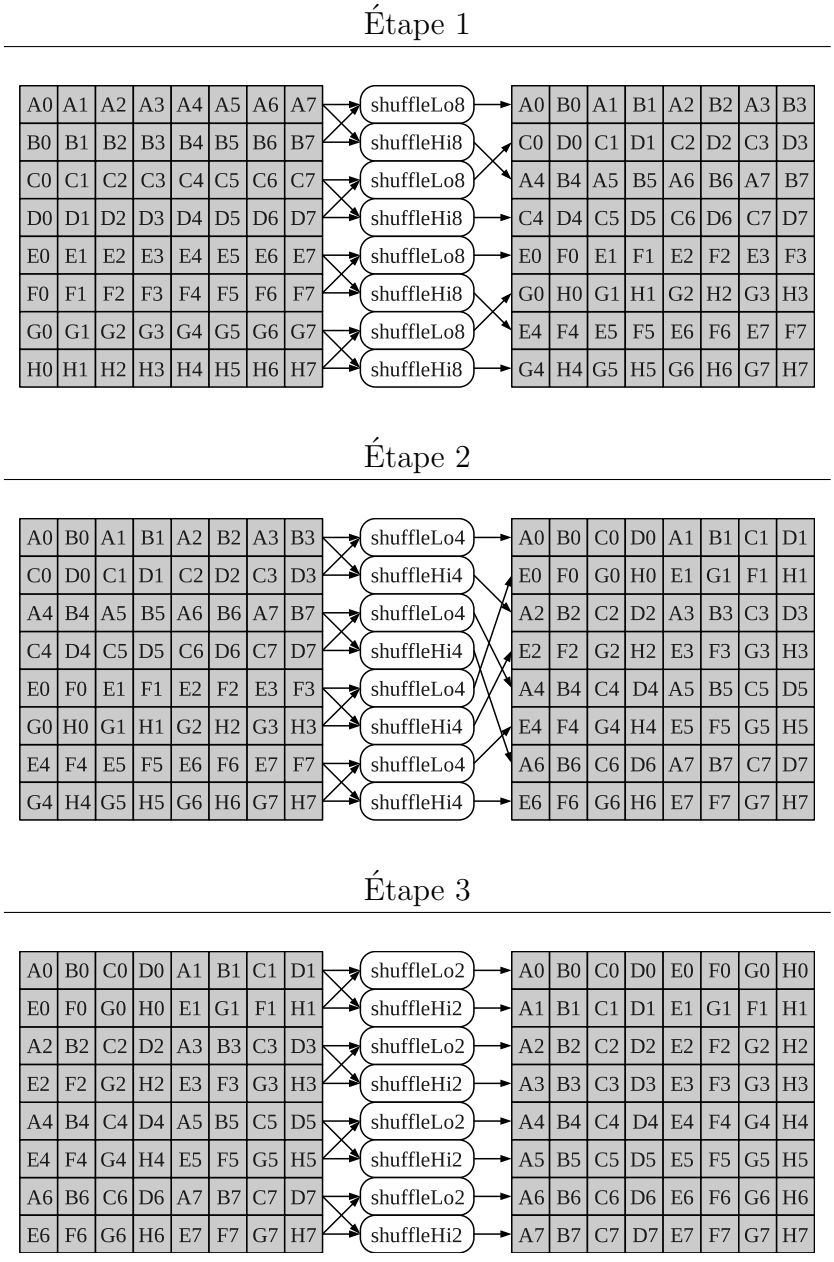


FIG. 4.32: Étapes d'une transposition d'un bloc de pixels 8×8

ne considérons plus comme élément atomique un pixel, mais un bloc de données transposé au préalable.

Les opérations de rotations se déroulent exactement comme une transposition et il est juste nécessaire de procéder à quelques ajustements vis-à-vis du sens horaire ou trigonométrique de la rotation. Une rotation de $+\frac{\pi}{2}$ peut être réalisée en permutant, à l'issue de la dernière étape du calcul de la transposition, la première ligne avec la dernière, la seconde avec l'avant-dernière et ainsi de suite. A l'inverse une rotation de $-\frac{\pi}{2}$ est obtenue en permutant les lignes du bloc avant le traitement.

Nous avons remarqué la relation existant entre le nombre d'étapes pour réaliser une transposition avec la taille du bloc et donc avec la taille d'un vecteur. Chaque étape nécessite deux instructions spécialisées de type *shuffle* et si le nombre d'éléments au sein d'un vecteur est important, on risque de devoir disposer d'un nombre assez élevé de ces instructions. Dans le cas binaire qui nous préoccupe dans cette section, les données manipulées par le cluster vectoriel comportent un grand nombre de pixels puisque ces derniers sont codés sur un bit et l'on peut en pratique observer une taille de 512 pixels binaires par vecteur. A titre d'exemple, avec cette taille de vecteur, il serait nécessaire de prévoir neuf paires d'instructions de type *shuffle*. De plus, il devient difficile de traiter des images dont la largeur est inférieure à 512 pixels et il faudra envisager la mise en place d'un padding. Si nous ne voulons pas ou nous ne pouvons pas mettre en place ces instructions dans le cluster vectoriel, nous pouvons exploiter l'unité scalaire qui travaille généralement sur 32 bits et qui peut donc manipuler 32 pixels binaires en mode micro-SIMD. Il suffirait alors d'ajouter au cluster scalaire cinq paires d'instructions de type *shuffle* pour procéder à des transpositions de blocs 32×32 et gérer plus simplement des images de tailles plus modestes. Le cluster scalaire étant composé de deux unités de calcul le niveau de parallélisme en mode binaire reste élevé.

4.4.2.2 Rotation par décalages

Les rotations d'images d'un angle arbitraire [61] sont basées uniquement sur le décalage des lignes et des colonnes. C'est une méthode simple où aucune interpolation n'est réalisée, mais qui peut être décomposée pour être traitée par plans de bits.

On décompose la rotation, d'angle α avec $\alpha \in [-\frac{\pi}{2}, +\frac{\pi}{2}]$ et de centre (c_x, c_y) avec $c_x \in [0, M-1]$ et $c_y \in [0, N-1]$, en trois étapes : un décalage des lignes, un décalage des colonnes et enfin un dernier décalage des lignes. Il est nécessaire de mettre en place deux opérations différentes, l'une dédiée au décalage des lignes et l'autre dédié au décalage des colonnes. Ces opérations peuvent être réalisées dans le même flot de données puisque les processeurs travaillent par plans de bits et sont donc en mesure de conserver un plan complet d'une image en mémoire locale.

L'opération de décalage des lignes, présentée en figure 4.33, est décomposée en deux étapes, l'une avant le décalage des colonnes et l'autre après afin de préserver au mieux la topologie de l'image. Le décalage des lignes étant réalisé en deux étapes, chaque étape devra donc utiliser un angle $\frac{\alpha}{2}$ comme référence.

Lorsque $\alpha > 0$, le décalage des lignes se fait à gauche lorsque $l < yc$ et à droite lorsque $l > yc$ et inversement si $\alpha < 0$. L'expression du décalage k d'une ligne d'indice l s'écrit de la manière suivante :

$$\forall l \in [0, N-1], k(l) = \begin{cases} (y_c - l)\tan(\frac{\alpha}{2}), & \text{si } l < y_c \\ (l - N + 1 - y_c)\tan(\frac{\alpha}{2}), & \text{si } l > y_c \\ 0, & \text{sinon} \end{cases}$$

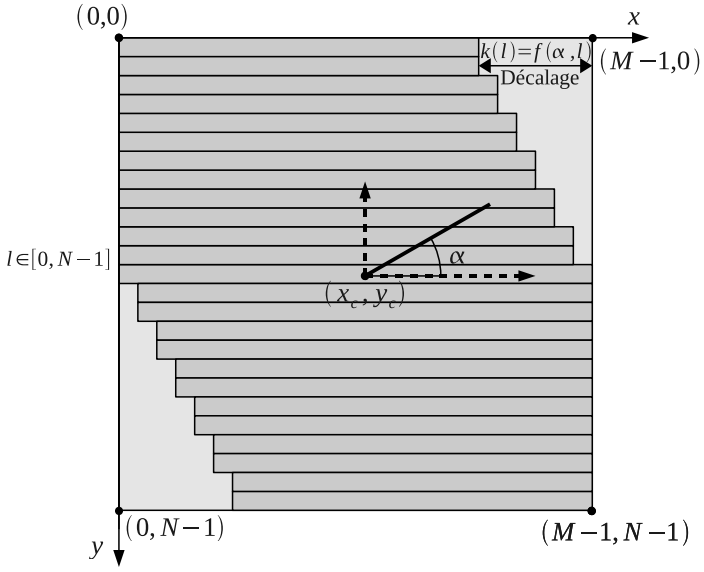


FIG. 4.33: Rotation par décalage : étape de traitement des lignes de l'image

L'opération de décalage des colonnes est présentée en figure 4.34 et l'expression du décalage k' d'une colonne d'indice c s'écrit de la manière suivante :

$$\forall c \in [0, M-1], k'(c) = \begin{cases} (x_c - c)\sin(\alpha), & \text{si } c < x_c \\ (c - M + 1 - x_c)\sin(\alpha), & \text{si } c > x_c \\ 0, & \text{sinon} \end{cases}$$

L'opération de décalage des lignes est simple à mettre en œuvre et peut se réaliser de manière parallélisée en utilisant les instructions de décalage du cluster vectoriel ou avec les instructions micro-SIMD du cluster scalaire. Un problème se pose pour réaliser vectoriellement l'opération de décalage des colonnes à cause du chargement des vecteurs à partir des lignes d'une image. En effet, chaque colonne subit un décalage différent et il est impossible dans l'état actuel du processeur de réaliser cette opération directement à l'aide d'instructions vectorielles. Il faut soit travailler de manière scalaire et traiter les pixels un à un, soit procéder à une transposition de l'image afin de se replacer dans un contexte favorable aux unités de calcul vectoriel ou micro-SIMD.

Le déploiement d'une opération de rotation d'image avec un angle $\alpha \in [-\frac{\pi}{2}, +\frac{\pi}{2}]$ est réalisé en trois passes successives dans notre réseau de processeurs. Les deux premières passes, basées sur une configuration présentée en figure 4.35, permettent de sélectionner quatre plans de bits parmi ceux d'une image 8 bits source, de réaliser les rotations de chaque plan indépendamment et de réunir ces quatre résultats de poids faibles au sein d'une image 8 bits un peu à la manière de ce qui a été présenté en figure 4.30. En réalisant

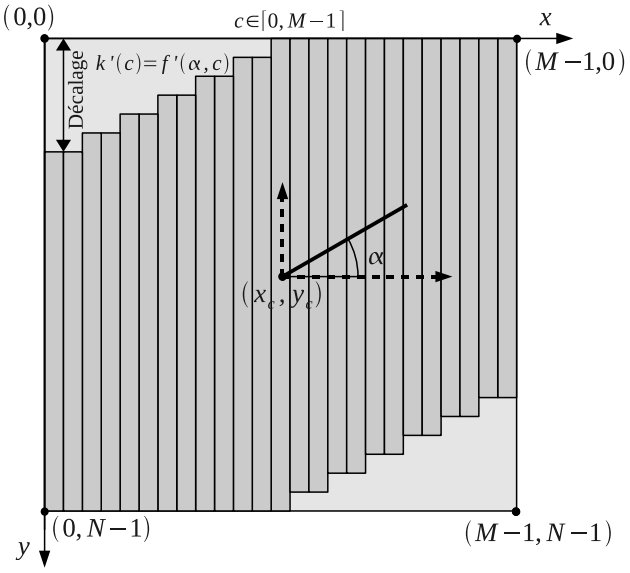


FIG. 4.34: Rotation par décalage : étape de traitement des colonnes de l'image

cette opération une deuxième fois on obtient une seconde image 8 bits comportant les quatre autres plans de poids forts pivoté d'un angle α. Il est nécessaire ensuite de mettre en place une troisième passe afin de réunir les deux images partielles pour obtenir l'image finale. Cette dernière opération peut se faire avec un seul processeur en multiplexant l'envoi des deux images ligne par ligne. Nous pouvons *masquer* le temps de traitement de cette dernière passe en ordonnançant les calculs utilisant le résultat de la rotation dans les quinze processeurs restants.

Les images présentées en figure 4.36 sont un exemple des différentes étapes nécessaires à la rotation d'une image.

4.4.3 Mesures

Les opérations de mesures sont des opérateurs prenant en entrée une image et renvoyant une information réduite telle que le volume global de cette dernière. Nous distinguerons ici deux types d'opérations : le calcul d'un paramètre scalaire et le calcul de l'histogramme.

4.4.3.1 Mesure d'un paramètre global scalaire

La mesure d'un paramètre global d'une image avec des instructions vectorielles repose sur des mesures partielles qui doivent être, à l'issue du traitement de tous les vecteurs, regroupées pour obtenir le résultat complet.

Lorsque les valeurs mesurées sont du même format que les pixels de l'image, la mesure est faite simplement comme présenté en figure 4.37 où l'on calcule dans deux vecteurs N minima et N maxima sur l'image, N étant le nombre d'éléments dans un vecteur. Chacun des éléments contient un résultat partiel qu'il faut ensuite traiter un à un pour obtenir les résultats complets. On utilise alors l'unité scalaire du processeur pour réaliser cette dernière opération.

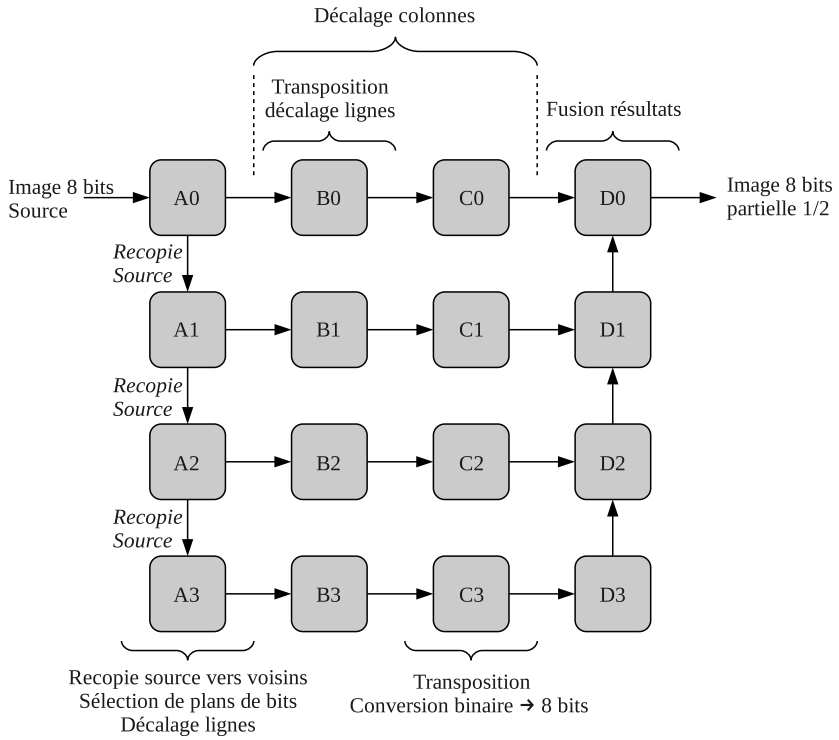


FIG. 4.35: Déploiement sur le réseau de processeurs d'une étape de rotation d'images par décalages en lignes et en colonnes

La mesure de paramètres dont le format est différent de celui employé pour le codage des pixels nécessite quelques instructions supplémentaires par rapport à ce que nous avons pu décrire dans le cas de la mesure du minimum et du maximum. En effet, il est nécessaire de convertir les éléments du vecteur vers un autre format de données disposant, dans la plupart de cas, d'un nombre de bits plus important. L'unité de calcul vectoriel du processeur dispose d'instructions de mélange de vecteurs permettant une telle conversion du format des données.

La figure 4.38 présente un exemple de calcul du volume d'une image. Le volume étant la somme de tous les pixels, il est obligatoire d'accumuler le volume dans une variable disposant d'une capacité de codage beaucoup plus importante que celle employée pour le stockage d'un pixel. Si l'on considère que les pixels sont codés sur 8 bits et que l'on travaille dans le pire cas, c'est-à-dire avec une image où toutes les données sont à 255, un volume stocké sur 32 bits permet de manipuler des images jusqu'à 2^{24} pixels, soit plus de 16 millions de valeurs. Ce chiffre est en dessous de la vérité puisque le volume n'est pas calculé sur une unité scalaire, mais sur une unité vectorielle travaillant avec N éléments 32 bits. En réalité chaque élément contient une somme partielle pouvant être le résultat de l'addition de 16 millions de pixels. Si l'unité scalaire regroupant les données peut gérer les débordements de retenue, il est possible en réalité de calculer le volume sur une image comportant $N \cdot 2^{24}$ pixels ce qui représente dans notre exemple en figure 4.38 plus de 64 millions de pixels.

(a) image originale (b) premier décalage des lignes avec $\frac{\alpha}{2}$

(c) décalage des colonnes avec α (d) second décalage des lignes avec $\frac{\alpha}{2}$

FIG. 4.36: Exemple de rotation d'angle $\alpha = \frac{\pi}{8}$ d'une image par décalage

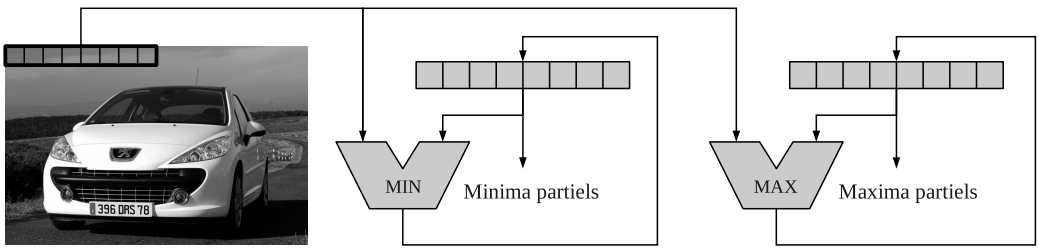


FIG. 4.37: Mesure du minimum et maximum global d'une image à l'aide d'opérations vectorielles

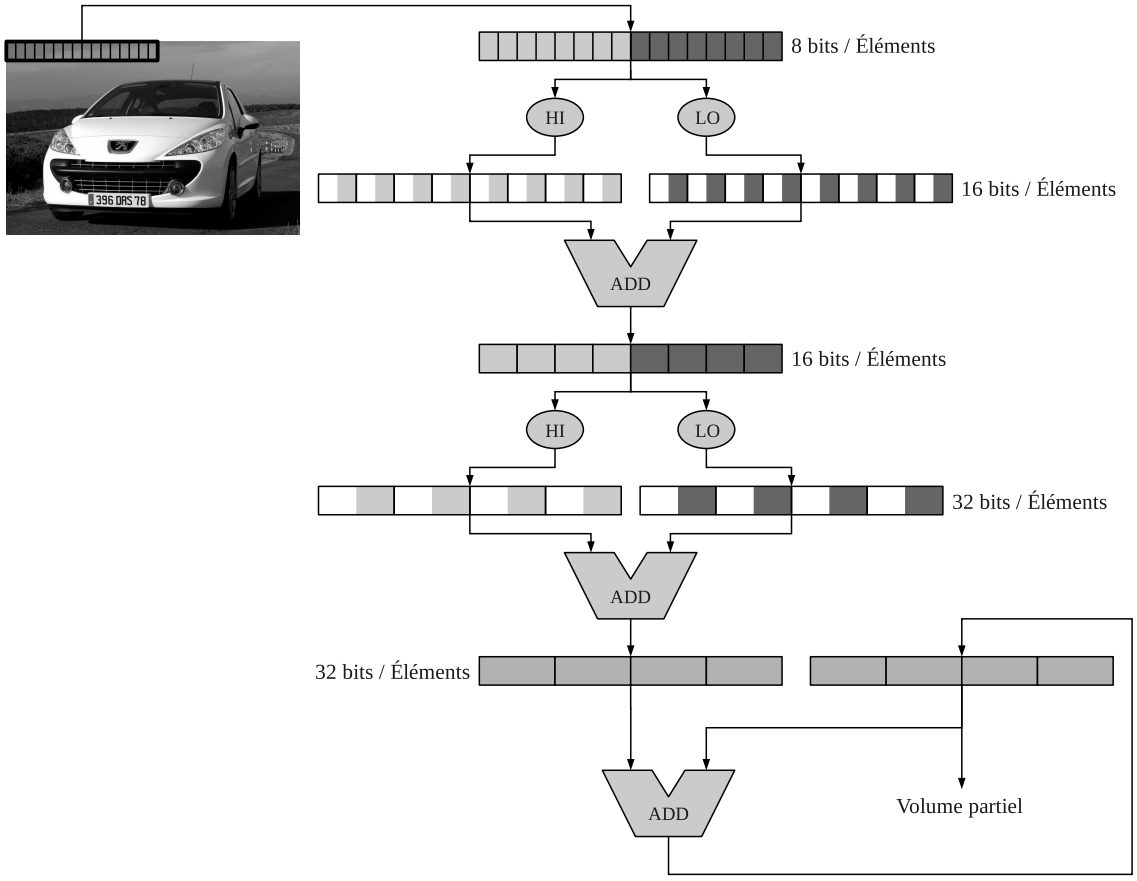


FIG. 4.38: Mesure du volume d'une image à l'aide d'opérations vectorielles

4.4.3.2 Calcul de l'histogramme

Le calcul de l'histogramme de manière vectorielle est grandement facilité par l'utilisation du mode d'adressage indirect vectoriel dont nous avons parlé dans la section 4.3.2.3. Nous nous plaçons dans un cas standard où l'unité scalaire fonctionne sur 32 bits et où un vecteur contient N éléments codés sur 32 bits. Les pixels sont codés en niveaux de gris sur 8 bits de sorte que l'histogramme puisse être contenu dans la mémoire locale. Le processeur dispose donc de N mémoires de largeur 32 bits composées chacune au minimum de 256 mots où l'on pourra calculer en parallèle N histogrammes partiels de l'image.

On effectue dans un premier temps une phase d'initialisation à zéro des N histogrammes, puis la mise à jour des histogrammes se fait de la manière suivante : on charge $4 \cdot N$ données 8 bits dans un vecteur, on convertit ce vecteur de pixels 8 bits en quatre vecteurs de pixels 32 bits et on procède à un accès indirect vectoriel pour mettre à jour les N histogrammes. En mettant en place un pipeline logiciel, il est possible de mettre à jour en quelques cycles N histogrammes en parallèle. Pour terminer le calcul, il faut réunifier les N histogrammes en employant le cluster scalaire, cette étape consommant environ $256 \cdot N$ cycles.

La figure 4.39 illustre les étapes du calcul des histogrammes partiels à partir d'une image en niveau de gris 8 bits et en considérant 4 éléments 32 bits par vecteurs. Quatre accès indirects vectoriels sont donc nécessaires pour mettre à jour quatre histogrammes à partir de 16 pixels.

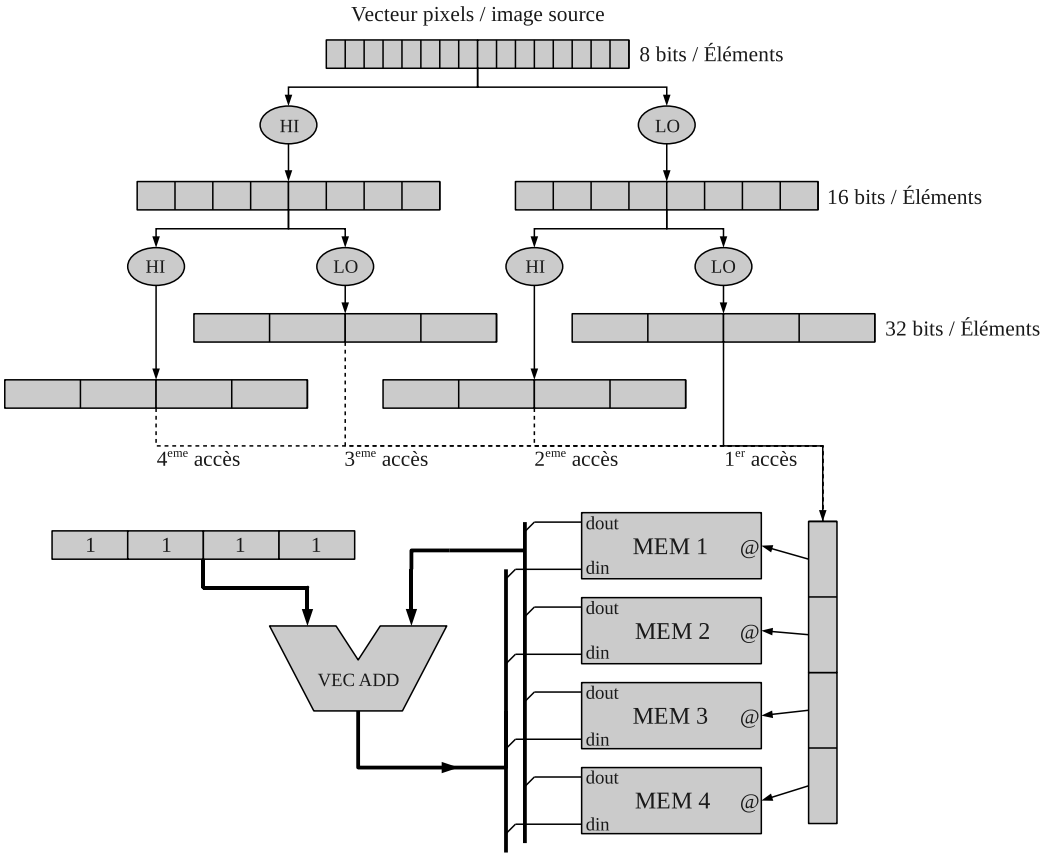


FIG. 4.39: Calcul d'un histogramme à l'aide d'opérations vectorielles

Le contenu de la mémoire vu par l'unité scalaire est présenté en figure 4.40. On se rend compte qu'il est simple de calculer l'histogramme complet de l'image puisque nous retrouvons la distribution d'un niveau de gris donné des N histogrammes partiels dans un vecteur en mémoire locale.

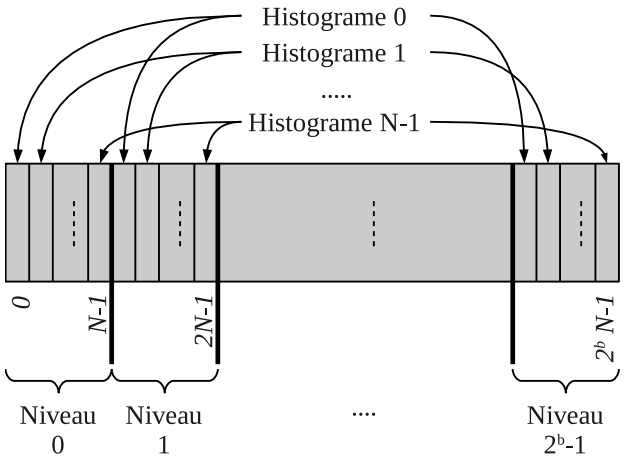


FIG. 4.40: Stockage des histogrammes partiels en mémoire locale provenant d'images codées sur b bits

Le mode d'adressage décrit ici pourrait être utilisé pour disposer d'une autre méthode d'extraction de voisinage en découpant une image en bandes et où chaque banque mémoire interne disposerait uniquement d'un voisin spécifique à une topologie [59]. Dans le cas de la maille 8-connexe, où l'on considère un voisinage de rayon 1, il est nécessaire que le processeur vectoriel dispose de 9 mémoires pour stocker chacun des voisins et le pixel central. Ce mode d'adressage et assez particulier et nous allons présenter dans les prochaines sections comment les opérations de voisinages sont réalisées avec notre architecture.

4.4.4 Opérateurs morphologiques basiques

Nous traitons dans cette section la mise en œuvre au sein de nos processeurs de quelques opérations de base de morphologie mathématique. Nous allons d'abord traiter deux cas particuliers des filtres de rang, à savoir l'érosion et la dilatation. Même si ces opérations peuvent être faites à partir du système de tri des filtres de rang, il existe des manières bien plus rapides pour les mettre en place aussi bien en termes algorithmiques qu'en termes matériels. Nous reviendrons ensuite sur les filtres de rang avant de donner quelques exemples de compositions possibles d'opérateurs de base pour réaliser des gradients, des filtrages ou bien encore des amplifications de contraste.

4.4.4.1 L'érosion et la dilatation

Nous n'aborderons ici que le cas de l'érosion. Il suffit de remplacer dans les opérations les infima par des suprema pour traiter le cas de la dilatation. Nous avons déjà détaillé dans l'algorithme 8 la manière d'extraire un voisinage 3×3, mais nous ne faisions aucune hypothèse quand à la nature des calculs réalisés. En effet dans le cadre d'opérations

de morphologie mathématique et en particulier pour l'érosion avec un élément structurant composé de k lignes et l colonnes, certains calculs peuvent être conservé d'une ligne à l'autre. Le principe est d'éroder la ligne d'indice i nouvellement lue avec un élément structurant $1 \times k$, et de conserver les $l-1$ dernières érosions ainsi calculées. En calculant l'infimum de la ligne venant d'être érodée avec les $l-1$ dernières, on obtient l'érosion de la ligne d'indice $i - l/2$ avec un élément structurant $l \times k$.

Un exemple réalisant une érosion 3×3, avec la méthode que nous venons de décrire, est présenté dans l'algorithme 10. Les fonctions *recoitLigne* et *envoieLigne* font référence à l'utilisation des FIFO du processeur pour recevoir et transmettre des lignes sous forme de vecteurs de pixels. La fonction *erodeLigne* érode une ligne en utilisant les instructions dédiées de décalage avec infimum. La boucle peut être mise en place grâce à l'unité de gestion des boucles puisque cette dernière ne dépend que d'un compteur incrémenté à chacune des itérations.

Algorithme 10 Érosion d'une image optimisée pour un élément structurant 3×3

```
/* Prologue */
E = recoitLigne()
A = erodeLigne(E)
E = recoitLigne()
B = erodeLigne(E)

S = infLigne(A,B)
envoieLigne(S)

/* État permanent */
Pour x=2 to M-1 Faire
  E = recoitLigne()
  C = erodeLigne(E)
  A = infLigne(A,B)
  S = infLigne(A,C)
  envoieLigne(S)

  tmp = &A
  &A = &B
  &B = &C
  &C = tmp
Fin pour

/* Épilogue */
S = infLigne(A,B)
envoieLigne(S)
```

Le cas de l'optimisation en maille hexagonale est un peu plus complexe, mais repose toujours sur le même principe, à savoir conserver des calculs sur les lignes déjà traitées durant les étapes précédentes. La figure 4.41 indique quels calculs peuvent être conservé en fonction de la parité des lignes. En effet, comme nous avons déjà pu l'évoquer précédemment, la maille hexagonale est juste simulée en changeant d'éléments structurants en fonction de la parité des indices des lignes.

Derrière les optimisations décrites pour maille carrée se cachent les propriétés de décomposition des éléments structurants. Il est en effet possible de réaliser une érosion $l \times k$ en la décomposant en deux érosions $l \times 1$ et $1 \times k$ ce qui d'une part, simplifie la mise en œuvre de l'algorithme et d'autre part, réduit la quantité de calcul. Par exemple si $k = l$ on passe d'une complexité égale à k^2 à une complexité égale à $2k$. D'autres décompositions existent et ont des propriétés fortes en termes de réduction du nombre de calcul. On peut citer par exemple la décomposition logarithmique [77] [78] qui vise, lors d'utilisation d'éléments structurants ayant une taille importante, à diminuer fortement le nombre de calculs par une décomposition particulière des éléments structurants.

La figure 4.42 montre la décomposition logarithmique qu'il est possible d'obtenir pour

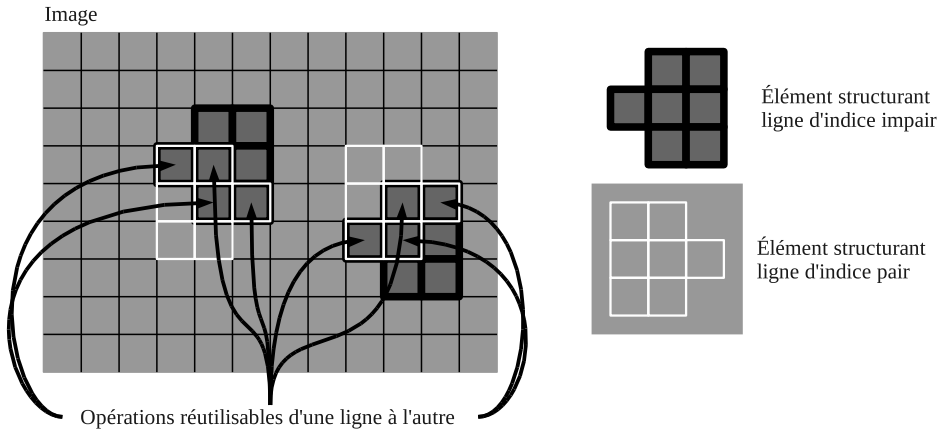


FIG. 4.41: Réutilisation des calculs dans le cadre d'opération de morphologie de base avec un élément structurant hexagonal

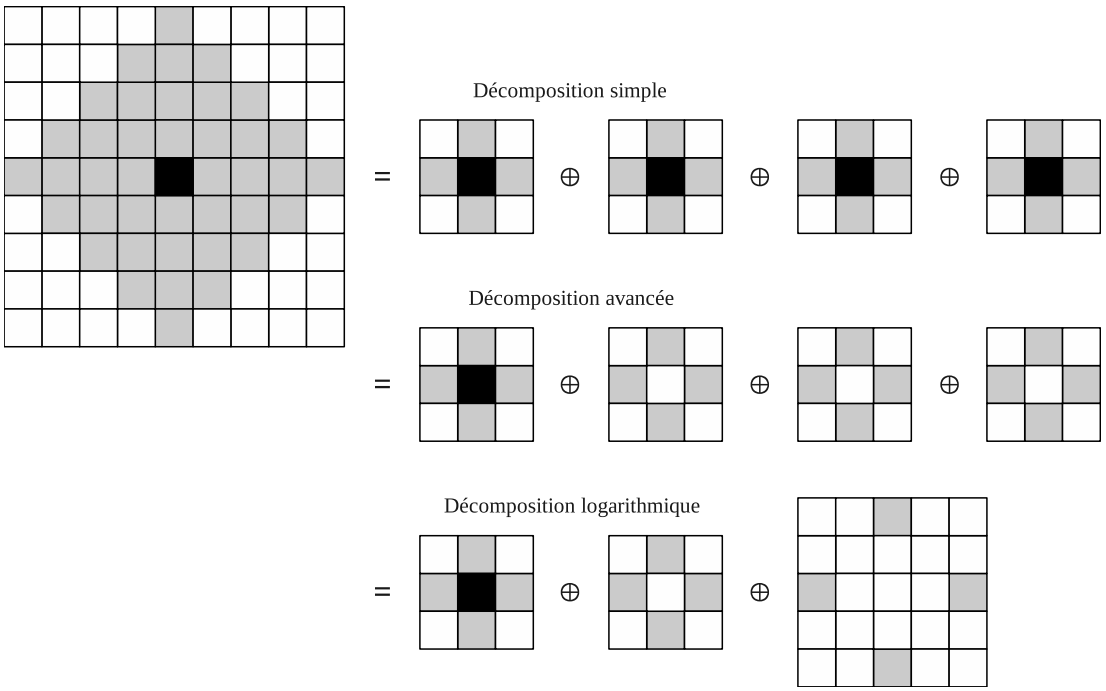


FIG. 4.42: Exemple de décomposition logarithmique d'un élément structurant losange de rayon 4

un élément structurant de type losange de rayon 4. Sans décomposition il serait nécessaire d'effectuer 40 opérations pour obtenir l'érodé d'un voisinage en un point, avec la décomposition simple on tombe à 16 opérations, avec la décomposition avancée on peut encore économiser trois opérations. Finalement avec la décomposition logarithmique il est juste nécessaire d'effectuer 10 opérations avec pour chacune seulement quatre voisins. Cette dernière décomposition est donc très avantageuse, mais présente un inconvénient, les éléments structurants, même s'ils disposent uniquement de quatre voisins comme dans le cas des décompositions simples et avancées, ne sont pas de rayons unitaires. Cela signifie que dans nos processeurs il sera nécessaire de conserver un plus grand nombre de lignes de l'image pour opérer des calculs sur des voisinages plus étendus. Il sera donc difficile de mettre en place plusieurs opérations de voisinage au sein du même processeur en utilisant un pipeline logiciel pour des problèmes de contention mémoire.

4.4.4.2 Filtres de rang

Les cas particuliers de l'érosion et de la dilatation étant maintenant traités, nous pouvons détailler le fonctionnement général des filtres de rang au sein de nos processeurs. Ces filtres reposent sur un tri des pixels et nous avons déjà largement abordé les différentes optimisations en fonction du nombre de valeurs à trier dans le chapitre 2. Nous avons donc à notre disposition trois méthodes de tri.

Une première méthode consiste alors à extraire, pour une ligne de pixels donnée, les N voisins dans N tableaux de la taille d'une ligne, comme présenté à la figure 4.13 ou à l'algorithme 8. Les éléments à trier sont donc les pixels de mêmes indices au sein des N tableaux. Nous pouvons alors utiliser un tri à bulle vectoriel en utilisant les instructions *min* et *max* vectorielles pour réaliser tous ces tris. Les valeurs au bord de l'image pourront être simulées par le système du damier dont nous avons démontré l'efficacité dans le chapitre 2, il suffit lors du décalage des lignes de l'image, pour extraire les différents tableaux de voisinages, d'insérer alternativement des pixels noir ou blanc afin de biaiser un minimum le tri sur les bords.

Une seconde méthode consiste à utiliser le même principe d'extraction des tableaux de voisinages, mais les calculs réalisés sur ces derniers bénéficieront de structures de tris optimales connues pour certaines tailles de voisinage [33] et qui peuvent soit fournir uniquement la médiane dans des versions élaguées soit le tri complet du jeu de données. Un exemple est donné dans l'algorithme 11 où nous mettons en place un tri optimisé et élagué de cinq valeurs pour calculer uniquement la valeur médiane. Cet algorithme permet de calculer un filtre médian en considérant un élément structurant de type croix de rayon unitaire. La fonction *triLigne* tri les pixels deux à deux de deux lignes, c'est-à-dire qu'elle réalise le tri entre les pixels de mêmes indices en se basant sur les opérations vectorielles d'infimum et de supremum. La fonction *majDamier* se contente de modifier la première ou la dernière valeur des tableaux de voisinages pouvant être en dehors de l'image, il s'agit juste ici de mettre en place l'émulation du damier autour de l'image.

Enfin, une dernière méthode fonctionne en réutilisant les tris des lignes précédentes [70], un peu à la manière de ce que nous avons pu décrire pour le calcul de l'érosion. Il est donc nécessaire de modifier en conséquence l'extraction des tableaux de voisinages, puisque le vieillissement des données est maintenant appliqué sur les lignes partiellement triées plutôt

Algorithme 11 Filtre médian optimisé pour un élément structurant croix de rayon 1

```
/* Prologue */
D00 = recoitLigne()
D01 = DécaleDroite(D00,0)
D02 = DécaleGauche(D00,0)

D10 = recoitLigne()
D11 = DécaleDroite(D10,255)
D12 = DécaleGauche(D10,255)

/* Noyau de calcul du prologue */
tmp1,tmp2 = triLigne(D00,D01)
tmp3,tmp4 = triLigne(D02,D11)
tmp3 = supLigne(tmp2,tmp3)
tmp3,tmp4 = triLigne(tmp3,tmp4)

envoieLigne(tmp3)

/* État permanent */
Pour x=2 to M-1 Faire
  D20 = recoitLigne()
  D21 = DécaleDroite(D20,0)
  D22 = DécaleGauche(D20,0)

  /* Noyau de calcul de l'état
  permanent */
  tmp1,tmp2 = triLigne(D01,D10)
  tmp4,tmp5 = triLigne(D12,D21)
  tmp2,tmp3 = triLigne(tmp2,D11)
  tmp1,tmp2 = triLigne(tmp1,tmp2)
  tmp4 = supLigne(tmp1,tmp4)
  tmp3 = infLigne(tmp3,tmp5)
  tmp2,tmp3 = triLigne(tmp2,tmp3)
  tmp2 = supLigne(tmp4,tmp2)
  tmp2 = infLigne(tmp2,tmp3)

  envoieLigne(tmp2)

  /* Vieillissement des données */
  tmp1 = D00
  D00 = D10
  D10 = D20
  D20 = tmp1

  tmp1 = D01
  D01 = D11
  D11 = D21
  D21 = tmp1

  tmp1 = D02
  D02 = D12
  D12 = D22
  D22 = tmp1

  majDamier(x%2,D10,D12) ;

Fin pour

/* Épilogue */

/* Noyau de calcul de l'épilogue */

tmp1,tmp2 = triLigne(D01,D10)
tmp3,tmp4 = triLigne(D11,D12)
tmp3 = supLigne(tmp2,tmp3)
tmp3,tmp4 = triLigne(tmp3,tmp4)

envoieLigne(tmp3)
```

que sur les tableaux de voisinages eux-mêmes.

4.4.4.3 Gradients

Nous pouvons définir le gradient de trois façons, le gradient supérieur, le gradient inférieur et le gradient épais. Ils correspondent respectivement à la soustraction entre l'image dilatée et l'image originale, à la soustraction entre l'image originale et l'érodée et à la soustraction entre la dilatée et l'érodée. Plusieurs possibilités s'offrent à nous pour mettre en œuvre cette opération. On peut très bien travailler à un grain élevé pour calculer le gradient épais en utilisant deux processeurs en parallèle envoyant les résultats de l'image érodée et dilatée à un troisième chargé de la soustraction des deux flux. On peut également travailler à un grain beaucoup plus fin pour optimiser les ressources de calculs. En effet une fois les tableaux de voisinages extraits, il est relativement simple de calculer à la fois le supremum et l'infimum de ces derniers pour ensuite réaliser la soustraction produisant le gradient épais.

Ces choix dépendent largement des autres étapes du calcul, car bien souvent une application ne se compose pas uniquement d'un gradient et il convient de regarder la dépendance des images intermédiaires de l'application. Il apparaitra alors certainement une solution qui s'imposera soit pour des raisons de performances, soit pour des contraintes de places, les processeurs étant en nombre limité.

4.4.4.4 Autres opérateurs composés

D'une manière générale, les opérations composées peuvent être réalisées dans la même optique que le gradient précédemment décrit. On peut soit travailler avec une granularité assez élevée en affectant une opération par processeur, soit essayer de mettre un maximum d'opérations dans un même processeur et donc travailler à un grain plus fin.

Certaines opérations simples telles que des ouvertures ou fermetures morphologiques ne nécessitent généralement qu'un unique passage dans le flot de processeurs ce qui laisse le choix de la granularité à mettre en place en fonction des autres opérations de l'application. Toutefois, nous avons vu dans le chapitre précédent que plus un flot de processeurs est profond, plus il est difficile à employer. Le fait de pouvoir changer la granularité des traitements avant le déclenchement d'une opération est donc un atout certain, car si on remarque qu'il n'est pas possible d'utiliser au mieux le flot de processeurs dans son ensemble il est alors possible de le raccourcir pour traiter plusieurs parties de l'image en parallèle sous forme d'imagettes.

Les opérations plus gourmandes en étapes de calcul sont, en revanche, plus simples à distribuer sur un flot profond de processeurs puisqu'elles sont très souvent décomposables en un nombre fini d'opérations élémentaires. On peut par exemple citer les filtres alternés séquentiels qui nécessitent souvent la mise en place en série de plusieurs dizaines d'érosions et de dilatations. Il est alors intéressant de chercher à mettre le plus possible d'opérations par processeur pour essayer de minimiser le nombre de passes à réaliser dans le flot de processeurs.

4.4.5 Opérateurs géodésiques

Les opérateurs géodésiques sont construits en reprenant les opérations d'érosions et de dilatations et en appliquant la définition standard telle que nous avons pu la voir dans le chapitre 2. A titre de rappel, on définit une érosion géodésique comme étant l'érosion du marqueur f avec un élément structurant de rayon 1, suivie d'un supremum avec le masque g :

$$\varepsilon_g^{(1)}(f) = \varepsilon^{(1)}(f) \vee g \text{ et } f \geq g$$

De manière duale, la dilatation géodésique est définie de la façon suivante :

$$\delta_g^{(1)}(f) = \delta^{(1)}(f) \wedge g \text{ et } f \leq g$$

Ces calculs, que nous qualifierons d'opérations géodésiques simples, peuvent être réalisés à l'aide d'une seule opération de voisinage. Nous pouvons déployer directement ces formules au sein d'un processeur : il suffit de modifier l'algorithme d'érosion ou de dilatation pour qu'il prenne l'image masque comme paramètre supplémentaire. Le processeur réalise donc une érosion ou une dilatation et le résultat est contrôlé avec le masque en réalisant soit un supremum soit un infimum. L'image masque et l'image marqueur sont envoyées ligne par ligne chacune leur tour afin de multiplexer les flux de données.

On peut également déployer ces opérations en décomposant le calcul dans plusieurs processeurs, un dédié au calcul d'une opération de voisinage du type dilatation ou érosion et un autre dédié au calcul d'une simple opération arithmétique dyadique de type infimum ou supremum. Cette solution est peu intéressante, car d'une part, le processeur réalisant l'opération de voisinage est bien plus chargé que le processeur réalisant l'opération arithmétique ce qui induit une sous-utilisation de ce dernier et d'autre part, car la topologie d'interconnexions que nous avons choisie implique d'utiliser des processeurs uniquement dans un but de routage des flux vidéos ce qui réduit la quantité de processeurs utilisables pour ordonnancer plusieurs opérations géodésiques. Si le choix d'une telle structuration des calculs est motivé par une plus grande vitesse de traitement, il est préférable de déployer l'érosion ou la dilatation géodésique dans un unique processeur et de travailler en parallèle sur plusieurs imagettes. Les processeurs seront beaucoup mieux utilisés et les calculs beaucoup plus rapides.

Passons maintenant aux opérations de reconstruction géodésique. Ces opérations, que nous qualifierons de complexes, nécessitent de réaliser séquentiellement un nombre non connu a priori d'opérations géodésiques simples. En effet, une reconstruction géodésique arrive à stabilité lorsque le volume de l'image ne varie plus, il est donc impossible de savoir à l'avance le nombre d'itérations nécessaires. Cette nature dynamique des calculs nous contraint d'envisager le pire cas en travaillant avec un flot de processeurs le plus long possible, mettant donc de côté le travail sur imagette. Il est nécessaire de mettre en place dans les deux derniers processeurs du flot de calcul la mesure du volume sur les résultats de chaque opération géodésique simple. Ces résultats seront dirigés vers les ports IO du processeur afin d'être lu par le système hôte après réception de l'image résultat. Si ces volumes sont égaux, la reconstruction géodésique est terminée et il est alors possible de passer à une autre opération. Si ces volumes sont différents, il faut alors poursuivre les calculs en reprenant notre image résultat et en réitérant un flot de traitement à travers tous les processeurs.

La figure 4.43 présente une reconstruction géodésique par dilatation avec comme marqueur une image en niveaux de gris sur 8 bits et comme masque une image binaire codée également sur 8 bits. On constate qu'après 16 opérations simples dans notre flot de processeurs, la reconstruction n'est toujours pas terminée, il faut donc réinjecter le résultat partiel en entrée du système pour continuer les étapes de ce calcul. Il faudra, pour que cette reconstruction soit terminée, pas moins de 18 passes dans le flot de processeurs. On comprend mieux alors l'intérêt de disposer d'un pipeline profond pour ce genre d'opérations.

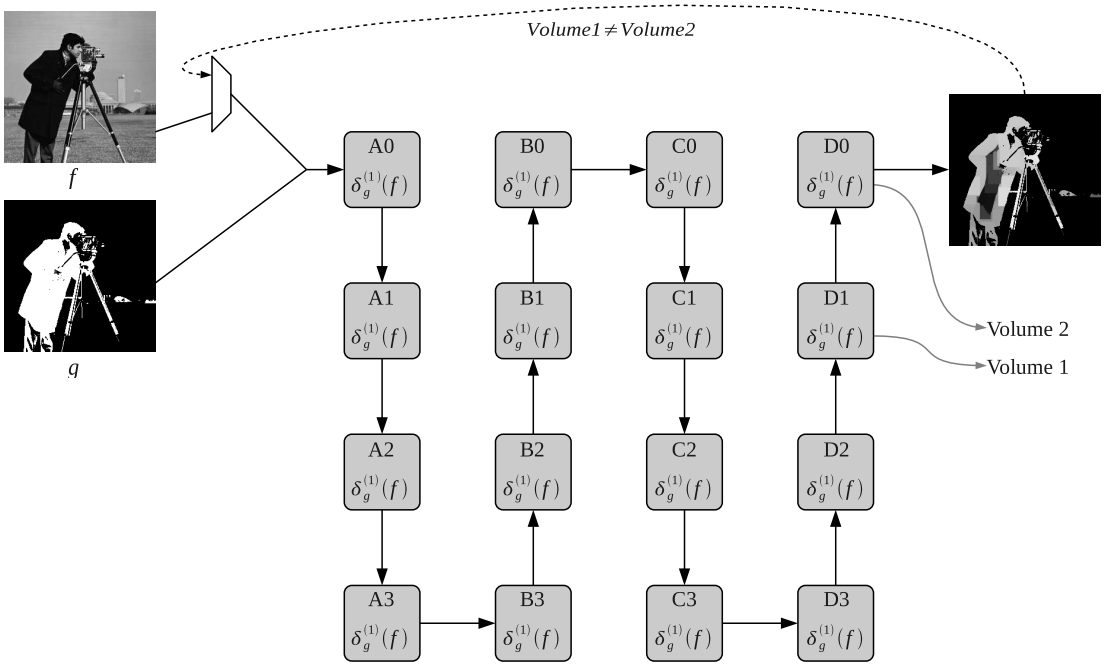


FIG. 4.43: Exemple de reconstruction géodésique par dilatation mise en place sur le réseau de processeurs

On peut également, pour ce type d'opérations complexes, mettre en place des calculs dits récursifs dans le même esprit que le calcul de la transformation distance exposée dans le chapitre 2 qui nous permettait de diminuer le nombre de passes nécessaires aux calculs. Brambor [15] a défini de manière vectorielle comment procéder à des opérations récursives dépendant du sens de parcours de l'image. Il propose, par exemple, une décomposition en quatre itérations une étape d'une reconstruction géodésique. Chacune des étapes privilégie une des quatre directions possibles : de haut en bas, de bas en haut, de droite à gauche et de gauche à droite. Ces directions correspondent en fait à un sens de parcours de l'image.

Toutefois, les opérations à réaliser s'organisent pour deux d'entre elles sur les lignes de l'image et pour deux autres sur les colonnes. L'exploitation du parallélisme dans notre processeur étant basée sur les lignes de l'image, le traitement en colonnes doit comporter une étape de rotation de l'image de $\frac{\pi}{2}$ avant calcul sur les lignes et une étape de rotation $-\frac{\pi}{2}$ après calcul. Toutes ces opérations supplémentaires cassent la possibilité de paralléliser temporellement les calculs et nécessitent l'amorçage d'une quinzaine de flots de calculs dans notre réseau de processeurs pour juste réaliser une itération. Même si une itération récursive propage un plus grand nombre de pixels qu'une itération standard, il faut compter en moyenne huit itérations récursives pour réaliser une reconstruction géodésique, soit en

moyenne 120 amorçages de flots de calculs dans notre réseau.

Il faut garder à l'esprit que pour 120 passes dans notre réseau de processeurs associés en séries, il est possible de calculer 1920 itérations standards d'une reconstruction géodésique, ce qui est dans la grande majorité des cas largement suffisants pour atteindre la stabilité.

4.4.6 Opérateurs morphologiques à supports variables dans l'espace

Ces opérateurs, présentés au chapitre 3, permettent de réaliser des opérations de traitement d'images en utilisant un voisinage pouvant être différent pour chaque pixel. On utilise alors une image *carte* dont le niveau de gris indique, pour le pixel correspondant dans l'image à filtrer et pour une topologie prédéfinie du voisinage, quelle taille d'éléments structurants il faut choisir [48] [8] [7]. Nous nous limitons ici au traitement avec un seul degré de liberté, c'est-à-dire que nous faisons varier uniquement la taille du voisinage par homothétie. Nous ne cherchons pas pour le moment comment mettre en œuvre des opérations où la forme et l'orientation pourraient également changer dans l'espace.

Ce type d'opération peut être réalisée en décomposant les éléments structurants en éléments structurants ayant un rayon unitaire. En choisissant cette décomposition, nous nous limitons aux opérations d'érosion et de dilatation, car les autres filtres de rang ne peuvent être décomposés de la même manière.

On peut alors réaliser ces opérations comme une suite d'opérations géodésiques où la carte correspond à un masque binaire partout où elle est différente de zéro, on soustrait à chaque itération sur cette dernière une valeur correspondant aux nombres d'opérations de voisinages réalisées avec un élément structurant unitaire. Le traitement s'arrête lorsque le volume de la carte est nul, signifiant ainsi qu'aucune opération ne reste à réaliser. Il convient bien évidemment que la soustraction soit en réalité une soustraction saturée à zéro.

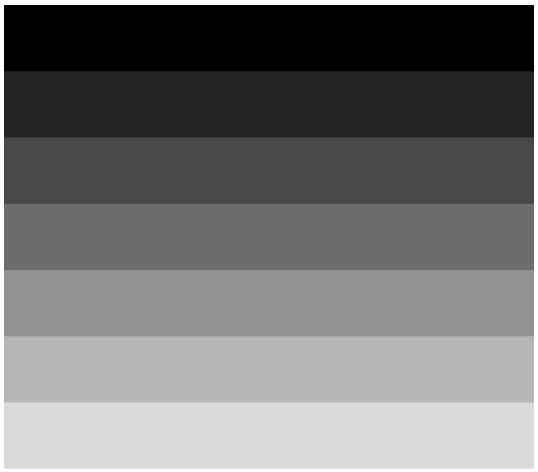
Un processeur doit donc recevoir l'image à calculer ainsi que la carte des éléments structurants. Comme dans le cas de la géodésie, nous avons choisi de multiplexer l'envoi des lignes de la carte avec l'envoi des lignes de l'image afin de n'utiliser qu'un seul processeur par étape afin de maximiser les capacités de traitement du réseau de processeurs. Chaque processeur doit être en mesure de calculer une opération de voisinage avec un élément structurant de taille unitaire. Il doit ensuite valider grâce à la carte et pour chaque pixel, si nous devons appliquer cette opération de voisinage où conserver la valeur originale, cette opération pouvant être réalisée vectoriellement avec des instructions de test de vecteurs et de déplacement conditionné par le résultat du test. Il reste ensuite à réaliser une soustraction saturée de la carte afin de retrancher la valeur correspondant au rayon de l'élément structurant employé dans l'opération de voisinage, qui est ici unitaire.

Nous pouvons avec notre réseau de processeurs traiter, en un flot de calcul avec un élément structurant unitaire par processeur, une image dont les éléments structurants peuvent varier de 1×1 à 33×33.

La figure 4.44 présente un exemple d'utilisation d'un élément structurant variable dans l'espace afin d'opérer des calculs en tenant compte d'une certaine perspective dans l'image. On remarque bien que la dilatation en haut de l'image a un impact moins important que dans le bas de l'image. En effet la carte à été construite de manière à utiliser des éléments

structurants de rayon 1 en haut de l'image jusqu'à un rayon 7 en bas de l'image.

(a) Image originale



(b) Carte des rayons des éléments structurants

(c) Image résultat après dilatation

FIG. 4.44: Dilatation avec un élément structurant variable dans l'espace

La construction de la carte est ici complètement subjective, mais pourrait être construite à partir d'autres opérations de voisinages, comme un gradient par exemple [48].

4.4.7 Quasi-distance

La quasi-distance [12] est aux images en niveaux de gris ce que la distance est pour les images binaires. Nous avons déjà décrit cet opérateur dans le chapitre 3 et il peut être également déployé sur notre réseau de processeurs.

Cette opération se décompose en deux étapes, une première permettant le calcul de l'image associée et de l'image indicatrice et une seconde permettant la régularisation de

l'indicatrice afin de supprimer le phénomène de distance perchée que nous avons déjà abordé. C'est l'image indicatrice qui nous est utile ici et le calcul de chaque étape fait appel à un ensemble d'opérations simples telles que des érosions, soustractions, remplacement de pixels et nécessite un grand nombre d'itérations comme dans le cadre des opérations géodésiques.

Afin d'obtenir l'image associée et l'indicatrice, il suffit alors de mettre en œuvre dans tous les processeurs, associés en série, la brique élémentaire du calcul décrite dans l'algorithme 4 et de lancer autant de passes que nécessaire. Le critère d'arrêt étant établi sur la base de la stabilité du volume du résidu, il est impératif que les processeurs soient en mesure de calculer un volume. Il en va de même pour l'étape de régularisation de l'image indicatrice, on programme les processeurs pour fonctionner en série avec la même brique de calcul décrite à l'algorithme 5. Le critère d'arrêt est lui aussi fondé sur la stabilité du volume de l'indicatrice en cours de régularisation.

4.4.8 SKIZ isotrope

Le SKIZ isotrope [10], opération déjà décrite dans le chapitre 2, est une opération visant à calculer la zone d'influence de différents constituants d'une image binaire et pouvant être utilisé dans le calcul de la ligne de partage des eaux. Cette opération est itérative et est composée, à chaque étape, d'une dilatation d'une image binaire labelisée suivie d'une étape de régularisation afin de bloquer les zones où les labels sont en collisions. La taille de la dilatation à un impact sur la taille de la frontière séparant les zones, et nous nous plaçons dans le cas de dilations de rayon unitaire.

La figure 4.45 présente sur un exemple simple les deux étapes de l'algorithme, on constate qu'à l'issue de la dilatation les formes ayant un label plus élevé empiètent sur les formes de labels inférieurs. L'étape de régularisation corrige cet effet en bloquant les dilatations là où une frontière entre les formes est minimale, c'est-à-dire là où la frontière correspond au rayon de la dilatation.

Nous avons déjà détaillé le fonctionnement des opérations d'érosion et de dilatation au sein des processeurs vectoriels, nous allons donc voir comment mettre en place l'étape de régularisation au sein de ces derniers. Cette étape prend en entrée l'image binaire labellisée f et l'image dilatée $g = \delta(f)$ et renvoie l'image h régularisée. Pour tous les indices p des pixels de l'image, h peut alors s'écrire :

$$h(p) = \begin{cases} 0, & \text{si } (f(p) \neq g(p)) \wedge (f(q) \neq 0) \wedge (f(q) \neq g(p)) \\ g(p), & \text{sinon} \end{cases}, \text{ avec } q \in \{N_f(p) \cup p\}$$

On remarque que pour calculer la régularisation il faut disposer à la fois de l'image binaire labellisée et de l'image dilatée, ce qui nous oriente immédiatement vers l'utilisation d'un seul processeur pour réaliser la dilatation-régularisation. En effet la dilatation nécessite de procéder à une extraction du voisinage de l'image source et la régularisation nécessite d'une part d'avoir à disposition les voisinages de l'image source et d'autre part, de disposer du résultat de la dilatation. Il est donc possible de calculer toutes ces étapes dans un même processeur.

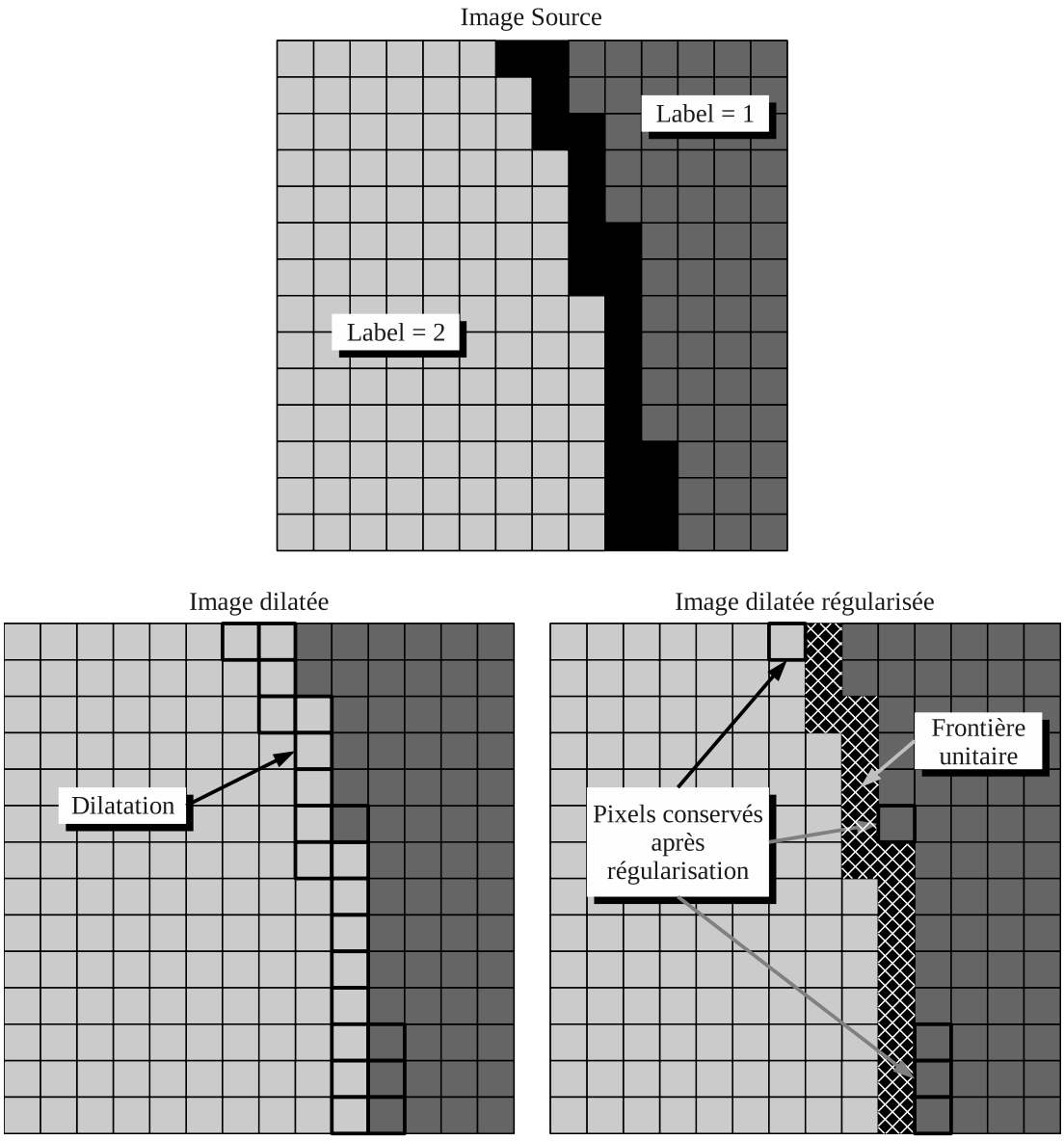


FIG. 4.45: Étapes du SKIZ isotrope : dilatation et régularisation

Un peu comme les opérations géodésiques, le SKIZ isotrope nécessite un grand nombre d'itérations de dilatation-régularisation. Nous pouvons ainsi déployer dans notre réseau de processeurs arrangés en série, seize itérations à condition de ne pas oublier de mettre en place le calcul du volume juste avant la sortie des deux derniers processeurs pour disposer ainsi d'un critère nous informant si un flot de calcul supplémentaire est nécessaire ou non.

La difficulté concernant la mise en œuvre de l'opération de régularisation consiste à réaliser les tests sur le voisinage de f de manière vectorielle. L'algorithme 12 décrit le noyau de calcul à mettre en place pour un élément structurant carré unitaire et suppose que tous les voisinages correspondant aux pixels d'une ligne sont extraits dans neuf tableaux. L'algorithme renvoie alors la ligne dilatée et régularisée. En répétant ce calcul sur toutes les lignes, on obtient une itération du SKIZ isotrope.

Algorithme 12 Noyau de calcul en état permanent d'une itération du SKIZ isotrope avec un élément structurant carré unitaire

Entrées: D_{ij} avec $i \in [0,2], j \in [0,2]$
Sorties: δ_{reg}

```
/* Dilatation */
tmp0 = infLigne(D00, D01)
tmp0 = infLigne(tmp0, D02)
tmp1 = infLigne(D10, D11)
tmp1 = infLigne(tmp1, D12)
tmp2 = infLigne(D20, D21)
tmp2 = infLigne(tmp2, D22)
δreg = infLigne(tmp0, tmp1)
δreg = infLigne(δreg, tmp2)

/* Régularisation */
cmp1 = compareLigne(D11, δreg)

Pour i ∈ [0,2], j ∈ [0,2] Faire
  cmp2 = compareLigne(Dij, 0)
  cmp3 = compareLigne(Dij, δreg)
  cmp3 = ouLigne(cmp3, cmp2)
  cmp3 = ouLigne(cmp3, cmp1)
  δreg = etLigne(cmp3, δreg)
Fin pour
```

4.5 Conclusion

Le processeur présenté dans ce chapitre permet d'utiliser des vecteurs de fortes tailles et est principalement construit pour travailler sur quelques lignes d'une image même si des techniques de séparation des plans de bits d'une image permettent d'accéder à une image binaire dans sa totalité en mémoire locale.

L'unité de calcul vectoriel est intéressante, car elle permet de simplifier la gestion du parallélisme au niveau des instructions et il est alors juste nécessaire de correctement ordonnancer les accès mémoire avec cette unité. En effet, nous savons qu'en traitement d'image, les mêmes calculs sont toujours itérés sur tous les pixels de l'image, il est alors inutile de disposer d'un grand nombre d'unités de calcul indépendantes qui saturerait la file de registres en connexions.

La gestion de la mémoire locale est simple puisque les deux clusters du processeur ont accès à cette dernière et peuvent ainsi s'échanger des données. Il serait certainement intéressant de prévoir un lien plus rapide entre les deux clusters pour permettre à l'unité scalaire de modifier un élément dans un vecteur. Nous avons préféré laisser de côté cette possibilité afin de ne pas alourdir le circuit avec des connexions supplémentaires, soit entre les deux files de registres, soit entre les unités de calcul vectoriel et scalaire.

La possibilité de disposer d'un flot de processeurs via un mécanisme de communication asynchrone nous permet de travailler à plusieurs niveaux de granularité de façon simple et performante. De plus, ce système nous laisse encore d'autres possibilités que nous n'avons pas explorées, notamment la possibilité de placer des points d'entrées de flux de pixels à n'importe quel étage du réseau de processeurs pour faciliter certains ordonnancements de calculs. Une piste également très prometteuse serait de faire évoluer le mécanisme de connexion entre les cœurs vectoriels en employant un NoC type mesh [57], ce qui aurait pour effet d'augmenter le nombre de chemins de données du système et permettrait par la même de spécifier simplement les points d'entrée et de sortie des différents flux d'images.

Un processeur de taille moyenne composé d'une unité scalaire sur 32 bits et d'une unité vectorielle sur 512 bits avec une fréquence de 100 Mhz nous permet d'atteindre une puissance de calcul brute de 6.4 GOPS lorsqu'on travaille avec des images 8 bits. Pour disposer d'une plus grande puissance de calcul, nous avons alors la possibilité, soit d'élargir les vecteurs, ce qui nous permettrait d'atteindre une puissance de 12.8 GOPS avec des vecteurs de 1024 bits, soit de raccorder en parallèle et en série un grand nombre de processeurs. Une structure composée de 16 processeurs de 512 bits fonctionnant à 100 Mhz délivre une puissance brute de 102 GOPS. On se rend alors bien compte des possibilités d'évolution de notre processeur ce qui nous permet à la fois de cibler des circuits FPGA ayant des ressources modestes ou alors des circuits ASIC avec de fortes possibilités d'intégrations.

Une grande puissance de calcul n'est pas suffisante pour réaliser des applications de traitement d'images en temps réel. Il faut également disposer de mémoires rapides permettant d'acheminer les pixels à traiter rapidement au réseau de processeurs. Il est nécessaire qu'un système hôte performant orchestrant, via des DMA et des mémoires rapides, l'envoi et la réception d'images vers notre accélérateur. Un parallélisme massif est difficile à alimenter, car le flux de pixels peut être très important. La capacité de notre réseau de processeurs à exploiter un parallélisme à la fois spatial et temporel permet de relâcher les contraintes qui pèsent sur le débit mémoire en entrée et en sortie de l'accélérateur tout en conservant une puissance de calcul élevée.

Les diverses applications proposées dans ce chapitre montrent l'efficacité de la solution et la souplesse d'utilisation grâce d'une part, au principe de traitement de voisinages par décalage de lignes au sein d'un processeur vectoriel et d'autre part, au réseau de processeurs supportant des topologies de déploiements d'applications assez différentes. Il reste tout de même une certaine difficulté pour l'utilisateur devant programmer chacun des processeurs spécifiquement pour chaque étape d'une application. Une perspective serait d'utiliser le système de génération automatique de flots d'opérateurs matériels pour le traitement d'images, présenté dans le chapitre 3, non plus pour générer une description matérielle d'un flot d'opérateurs, mais pour générer le micro logiciel exécuté par chaque processeur du réseau.

Le prochain chapitre va nous permettre d'analyser finement les performances de notre machine VLIW vectorielle mais également les performances des pipelines de processeurs de voisinage détaillés aux chapitres précédents. Des politiques de comparaison sont définis mettant à la fois en avant des possibilités de parallélisme spatial et temporel. Elles permettent de tester équitablement nos architectures avec des processeurs multicœurs et des accélérateurs graphiques que l'on retrouve aujourd'hui dans le commerce.

Chapitre 5

Comparaison des différentes architectures

Dans les précédents chapitres, nous avons présenté différentes architectures dédiées au traitement d'images avec pour chacune d'elles des degrés différents de programmabilité. En effet, nous avons commencé par décrire dans le chapitre 2 la structure d'un processeur de voisinage flot de données qui est à la fois limité par le type d'opération câblé et par la topologie du voisinage, ses deux paramètres étant décrits souvent matériellement.

Dans le chapitre 3, nous avons donc cherché à produire, par un assemblage de processeurs de voisinage, de multiplexeurs et d'unités arithmétiques et logiques, une architecture flot de données capable de calculer un plus grand nombre d'opérations morphologiques. Une telle structure dispose de paramétrages plus avancées, car il est possible de sélectionner à la fois les chemins de données à mettre en place et également les paramètres de calculs des différents composants du pipeline.

Enfin, dans le chapitre 4, nous avons proposé une architecture de processeur beaucoup plus généraliste disposant d'instructions vectorielles larges afin d'allier performance et généricité. Ce dernier système permet de réaliser efficacement un grand nombre d'opérations de traitement d'images en réécrivant simplement du microcode pour les différents processeurs qui composent l'architecture. Une telle souplesse permet également de viser des applications basées sur d'autres corpus théoriques que la morphologie mathématique.

Le leitmotiv de ces trois chapitres n'a donc cessé d'être la recherche d'un maximum de programmabilité tout en sacrifiant au minimum les performances, mais nous pouvons nous demander quelles sont celles que nous pouvons espérer atteindre pour chaque degré de liberté ainsi gagné. Nous proposons ici, au travers d'une politique de tests préalablement définie, de comparer les architectures précédemment décrites à deux systèmes couramment employés dans le domaine du calcul scientifique, à savoir les processeurs multicœurs et les accélérateurs 3D.

5.1 Définition et cibles des tests

5.1.1 Architectures cibles

Le choix des architectures est déterminant et nous avons décidé de comparer les architectures présentées dans les chapitres précédents à deux systèmes très différents et répandus afin de mieux cerner les performances atteintes. Nous avons retenu les systèmes suivants :

- Processeur Intel Q9550 : Processeur quadricœur équipé de 12 Mo de cache L2 et cadencé à 2.83 Ghz.
- Carte graphique NVidia Geforce GTX 260 équipée de 216 processeurs élémentaires cadencés à 1.2 Ghz et disposant de 896 Mo de mémoire DDR3.
- Un pipeline à 8 étages de processeurs de voisinage cadencé à 100 Mhz tel que proposé à la fusion des données provenant de plusieurs images à l'annexe A et dont l'extraction du voisinage se trouve être parallélisée par quatre (figure 2.51).
- Un système multicœurs VLIW vectoriel cadencé à 200 Mhz et synthétisé avec des vecteurs de 256 bits (figure 4.15) et dont l'agencement entre les huit cœurs se fait via un anneau bidirectionnel.

Les caractéristiques détaillées des architectures sont données en fin du chapitre aux tableaux 5.7 et 5.8.

5.1.2 Sélection des tests d'évaluation des performances brutes

Nous considérons deux types d'opérations, les opérations nécessitant un voisinage, telles que les érosions/dilatations, et les opérations n'en nécessitant pas, telles que les opérations arithmétiques et logiques. Nous choisissons également un groupe d'images dont le nombre de pixels varie de 4 kilopixels à 16 mégapixels afin d'observer des limitations architecturales et le cas échéant afin d'en connaître la cause. De manière à mieux cerner ces limitations, nous utilisons deux formats de codage des pixels, un format sur 8 bits et un format sur 16 bits.

Les opérations de traitement d'images retenues sont les suivantes :

- Érosion : $g = \varepsilon_{3\times3}(f)$
- Érosion : $g = \varepsilon_{5\times5}(f)$
- Érosion : $g = \varepsilon_{7\times7}(f)$
- Érosions 3×3 réalisées deux fois : $g = \varepsilon_{3\times3} \circ \varepsilon_{3\times3}(f)$
- Érosions 3×3 réalisées trois fois : $g = \varepsilon_{3\times3} \circ \varepsilon_{3\times3} \circ \varepsilon_{3\times3}(f)$
- Addition : $g = f + h$

L'enchaînement de plusieurs opérations nous permet d'observer l'impact de l'acheminement des données à traiter vers le système de traitement. En effet, il nous est possible de mesurer la différence de traitement entre réaliser N fois une opération et donc envoyer N fois l'image à traiter ou fusionner au cœur de la boucle de traitement N fois le même calcul en n'envoyant qu'une seule fois l'image.

Les résultats des tests sont présentés majoritairement sous forme de nombre de cycles par pixel résultat produit en sortie de l'opérateur. Ce ratio permet ainsi de disposer d'une base de comparaison indépendante de la fréquence de fonctionnement d'un système.

5.2 Performances brutes des plateformes de tests

5.2.1 Circuits dédiés à l'accélération 3D

Comme nous avons pu l'introduire dans le chapitre 4 les accélérateurs 3D sont aujourd'hui de véritables systèmes de calcul composés d'un grand nombre d'unités de traitement. Nous utilisons pour ces tests une carte Geforce GTX 260 composée de 216 processeurs élémentaires regroupés en 27 multiprocesseurs de flux, nommés *SM*. Comme nous l'avons déjà énoncé, ces SM regroupent huit processeurs élémentaires à la manière d'une machine SIMD, c'est-à-dire qu'ils décodent tous la même instruction. Chaque SM est multithreadé et gère et exécute matériellement 768 threads sans coûts supplémentaires d'ordonnancement. L'objectif du développeur est de découper son jeu de données en quantité suffisante pour alimenter tous les threads de chaque SM. Dans notre cas, avec la Geforce GTX260, le nombre optimal de threads est 20736.

Plusieurs implémentations d'opérations de morphologie mathématique existent sur ces circuits. On peut citer par exemple la bibliothèque GpuCV [4] qui propose des spécialisations d'opérateurs d'OpenCV reposant sur l'utilisation du standard OpenGL/GLSL ou sur l'API Cuda proposée par NVidia [49]. Il ressort des tests de cette bibliothèque que l'API proposée par CUDA est plus performante que l'utilisation de l'OpenGL/GLSL, toutefois cette bibliothèque propose principalement des implémentations avec des images RGBA 8 bits. Un pixel est donc codé sur 32 bits, simplifiant l'implémentation des opérateurs, mais complexifiant leur utilisation. En outre, il est généralement d'usage et plus performant d'appliquer des traitements différents selon les canaux ou de proposer une relation d'ordre pour ces derniers afin d'éviter l'apparition de fausses couleurs dues au traitement indépendant des différents canaux de l'image.

On peut également citer le projet OpenVidia qui propose un outil[60] de test d'opérateurs de traitement d'images en utilisant un GPU via la bibliothèque CUDA et permet l'emploi d'images codées sur 8 et 16 bits. Nous avons donc exécuté notre sélection d'opérations avec cet outil, et un échantillon des résultats est proposé dans le tableau 5.1.

Le tableau 5.1 nous montre dans un premier temps que la mise en place d'un grand nombre de microthreads de calculs, exploités par les différents SM du GPU, est onéreuse puisque le temps de traitement d'une image allant de 64×64 à 256×256 ne varie presque pas. On observe également, dans un second temps, que les coûts de transfert des images depuis la mémoire centrale vers la mémoire du GPU, et vice versa, sont très élevés. Il faut cependant garder à l'esprit que ces coûts de transfert sont dépendants de la structure du système hôte et qu'ils peuvent varier grandement.

Les graphiques présentés aux figures 5.1, 5.2, 5.3, 5.4 présentent l'évolution du nombre de cycles par pixel produit en fonction du nombre de pixels à traiter. Ces courbes sont construites sur la base du temps de calcul pour une opération donnée en tenant compte ou pas du transfert depuis/vers la mémoire centrale de l'hôte. On remarque que le nombre de cycles par pixel évolue de la même manière quelque soit le calcul réalisé et montre donc à quel point les GPU sont sensibles au volume de données à traiter plutôt qu'au type de calcul réalisé (dans une certaine mesure).

La figure 5.5 montre l'évolution du nombre de cycles par pixels dans le cas où des opérations sont fusionnées ou non. On entend ici par fusion le fait d'enchaîner plusieurs

Op. / Images	**Érosion** 3×3				**Érosion** 5×5			
	8 bits		16 bits		8 bits		16 bits	
	C*	CT**	C	CT	C	CT	C	CT
4096×4096	2424	20034	3519	37317	5442	23079	6152	42322
2048×2048	633	5498	896	10408	1375	6190	1529	10550
1024×1024	186	2072	247	3070	383	2453	407	3065
512×512	72	1185	89	1368	134	1349	134	1412
256×256	38	986	41	936	55	992	55	952
128×128	36	933	34	861	52	942	41	866
64×64	35	904	33	819	53	820	40	828

Op. / Images	**Érosion** 7×7				**Addition**			
	8 bits		16 bits		8 bits		16 bits	
	C	CT	C	CT	C	CT	C	CT
4096×4096	9106	27076	9233	42951	877	18213	1604	35515
2048×2048	2329	6875	2332	12734	248	5257	428	94340
1024×1024	620	2493	613	3323	93	1809	140	2922
512×512	205	1301	189	1430	50	989	59	1258
256×256	74	1017	70	926	32	805	35	837
128×128	73	950	49	782	31	751	30	761
64×64	72	770	48	750	30	727	29	723

*Temps sans transferts mémoires hôte. **Temps avec transferts mémoires hôte.

TAB. 5.1: Temps de calcul en microseconde de quelques opérations de traitement d'images sur un GPU Geforce GTX 260

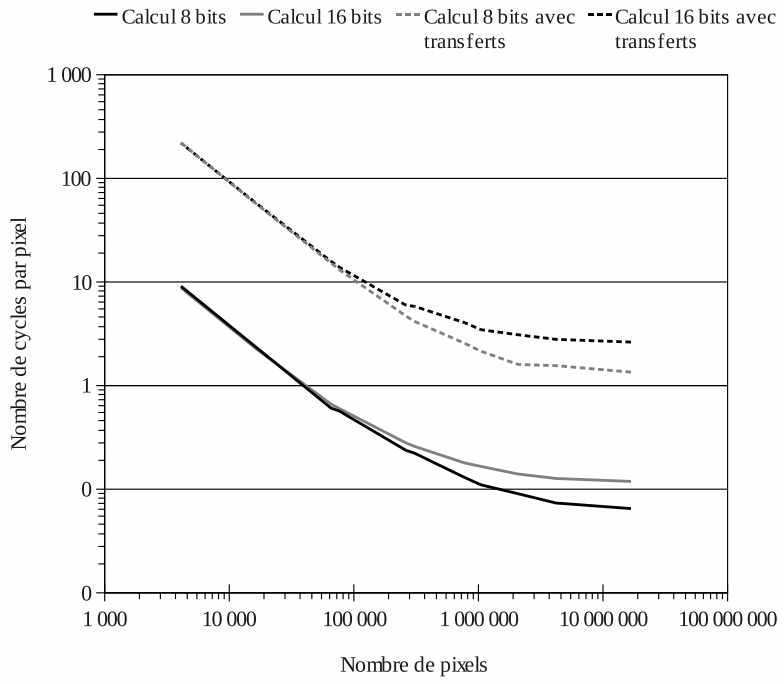


FIG. 5.1: Nombre de cycles par pixel pour produire une addition de deux images sur un GPU NVidia GTX 260

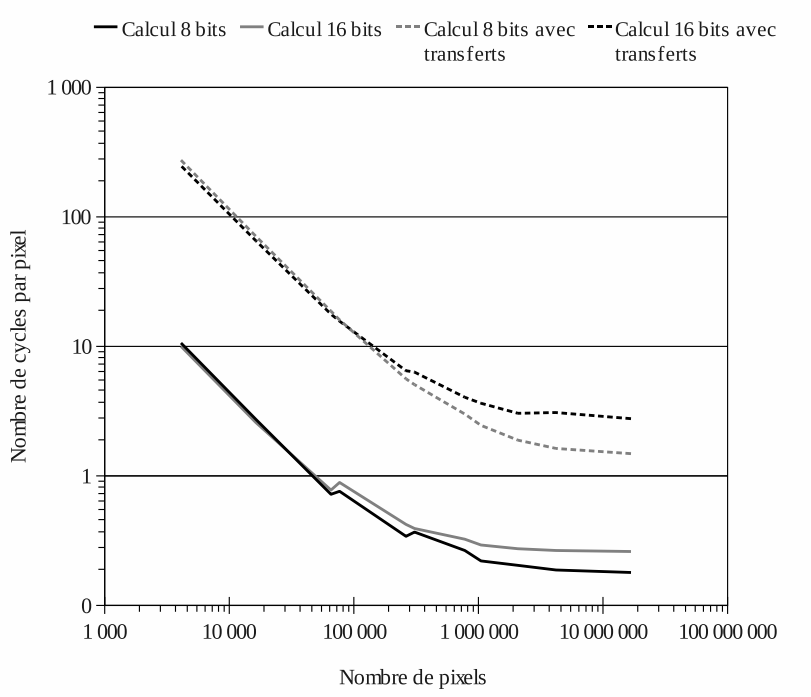


FIG. 5.2: Nombre de cycles par pixel pour produire une érosion avec un élément structurant 3×3 sur un GPU NVidia GTX 260

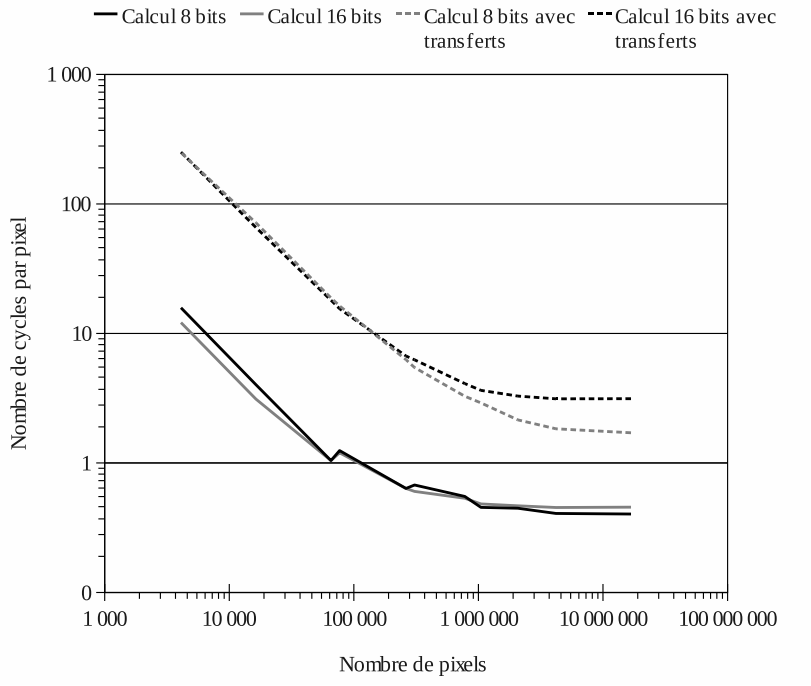


FIG. 5.3: Nombre de cycles par pixel pour produire une érosion avec un élément structurant 5×5 sur un GPU NVidia GTX 260

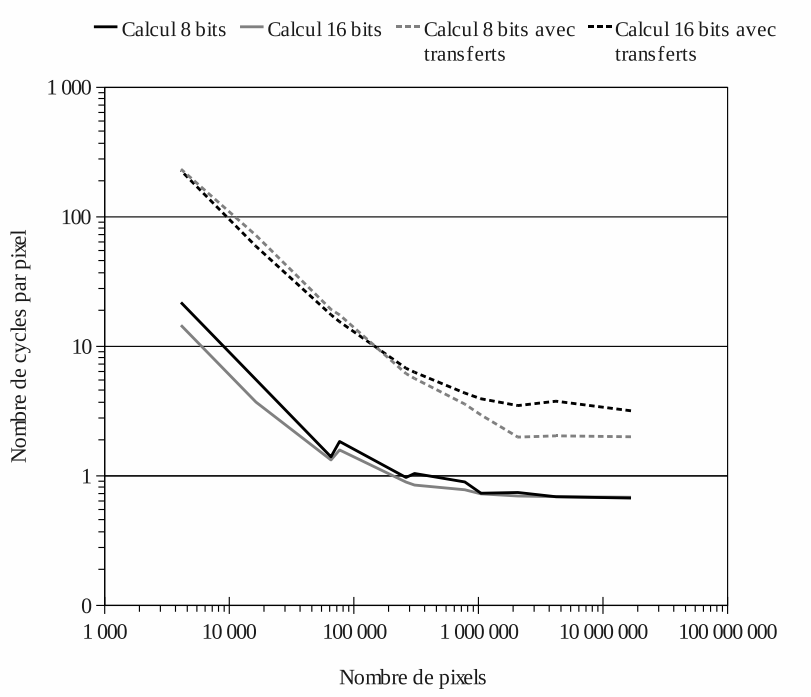


FIG. 5.4: Nombre de cycles par pixel pour produire une érosion avec un élément structurant 7×7 sur un GPU NVidia GTX 260

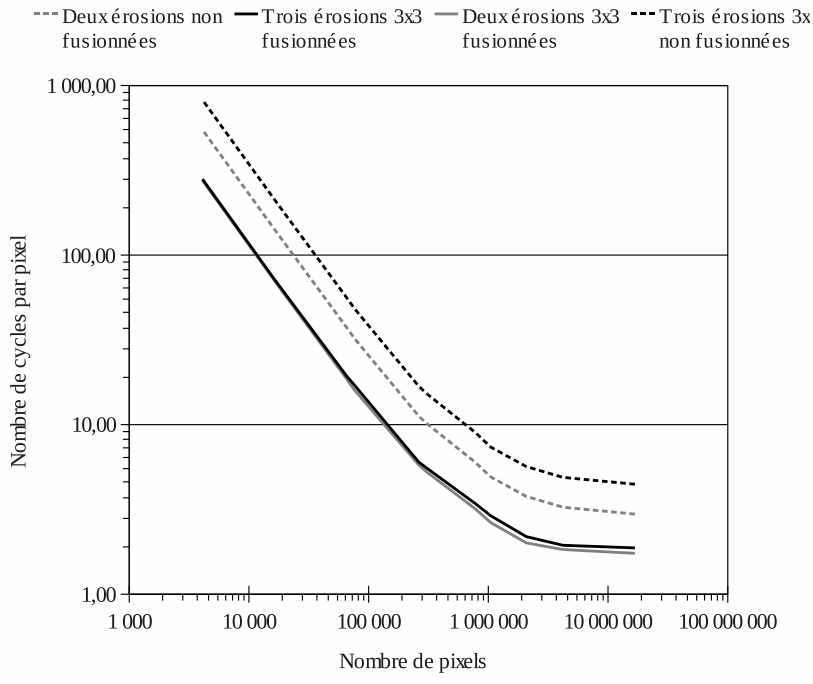


FIG. 5.5: Nombre de cycles par pixel pour produire deux et trois érosions 8 bits 3×3 fusionnées au sein du même coeur de traitement sur un GPU NVidia GTX 260

opérations dès lors que l'image à traiter se trouve dans le GPU, on peut même parler dans ce cas de fusion à gros grains puisque l'on ne modifie pas le cœur de traitement des opérateurs. Cette figure nous montre donc l'intérêt de faire un maximum de traitements dans le GPU afin d'éviter les aller-retour vers la mémoire. Aller-retour qui peuvent toutefois être nécessaires dans le cas où des calculs ne peuvent être réalisés que sur le système hôte du GPU. Il faudrait dans ce cas essayer de réordonnancer les opérateurs pour minimiser les transferts d'images entre GPU et hôte.

Nous considérerons dorénavant les temps de calcul sur GPU sans prendre en compte les transferts mémoires depuis et vers le système hôte qui sont dépendants des performances de ce dernier.

5.2.2 Processeur généraliste multicœur

5.2.2.1 Aspects généraux

Les processeurs généralistes sont aujourd'hui contraints à une évolution horizontale plutôt que verticale, la course à la fréquence n'étant plus aujourd'hui le cheval de bataille des différents fondeurs. Nous assistons à une croissance forte à la fois en termes de nombre de cœurs, mais également en termes de largeur d'instructions vectorielles. Le processeur Sandy bridge d'Intel, futur remplaçant du processeur Westmere (lui-même une réduction du Nehalem), en est un bon exemple puisqu'il embarque entre 4 et 8 cœurs et qu'il dispose d'instructions AVX dont les vecteurs sont deux fois plus larges que ceux des instructions SSEx.

Afin d'observer au mieux le fonctionnement de ce type de processeur, nous proposons quatre modes de fonctionnement :

- monocœur sans instructions SSEx
- monocœur avec instructions SSEx
- multicœur sans instructions SSEx
- multicœur avec instructions SSEx

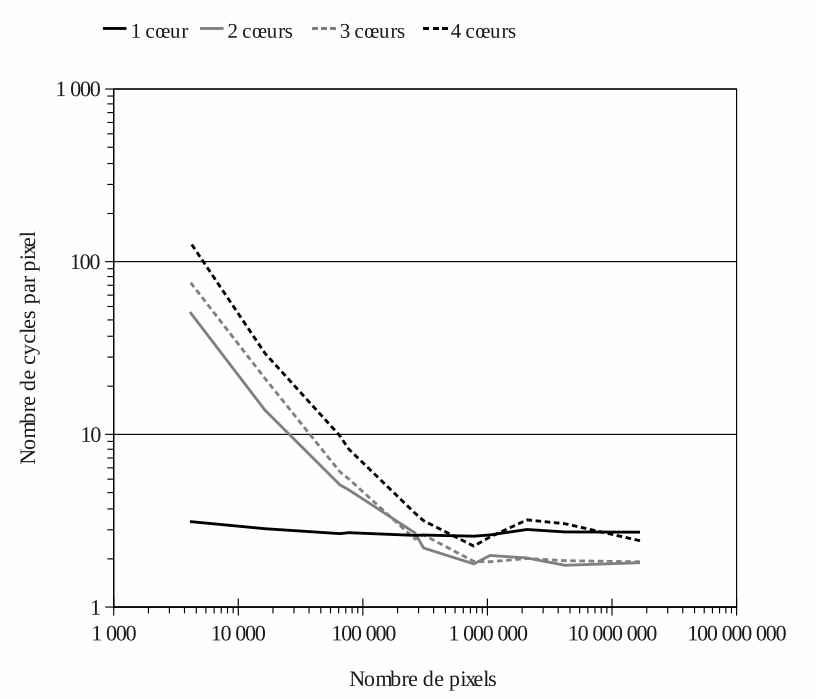
Les opérations utilisant les instructions vectorielles fonctionnent de la même manière que les opérations de traitement d'images portées sur un cœur VLIW vectoriel (cf. chapitre 4). Par exemple, dans le cas d'une érosion, nous extrayons le voisinage en réalisant différents décalages sur les lignes de l'image afin de pouvoir utiliser les instructions vectorielles entre deux lignes ainsi décalées sans problèmes d'alignement mémoire.

La réalisation d'une opération sur plusieurs cœurs se fait en attribuant à chacun de ces derniers une bande de pleine largeur de l'image. Les données sont donc découpées en autant de bandes que de cœurs et des zones de recouvrement sont à prévoir si des opérations de voisinages sont réalisées. Le parallélisme est alors exploité en instanciant dans chaque cœur un processus léger, *thread*, prenant en paramètre une bande différente de l'image.

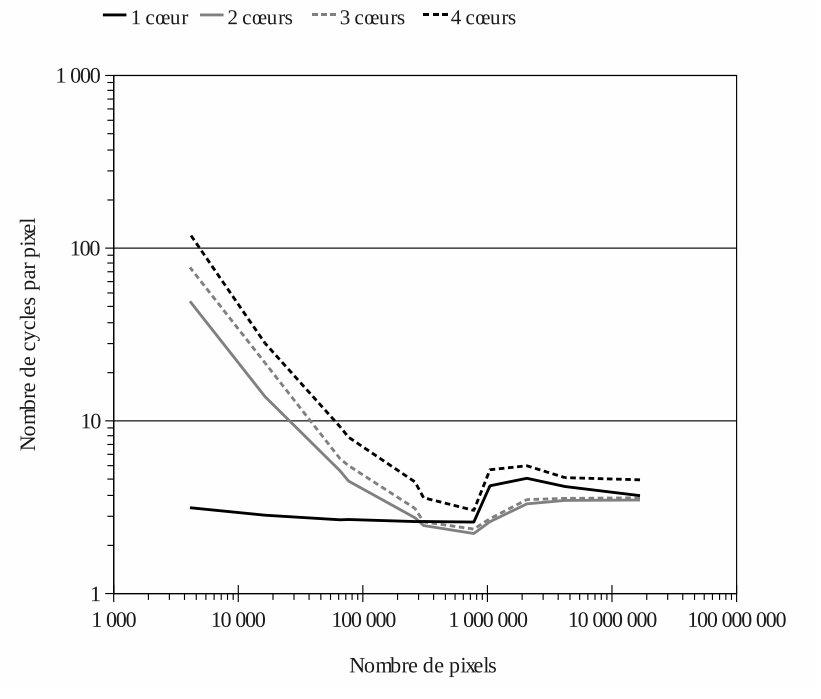
5.2.2.2 Tests sans instructions SIMD

Un échantillon de résultats de tests est présenté dans le tableau 5.2.

Nous présentons dans les figures 5.6, 5.7, 5.8, 5.9 les performances du processeur Q9550 avec les traitements précédemment sélectionnés et en utilisant 1, 2, 3 ou 4 coeurs. Les instructions vectorielles ne sont pas employées ici.

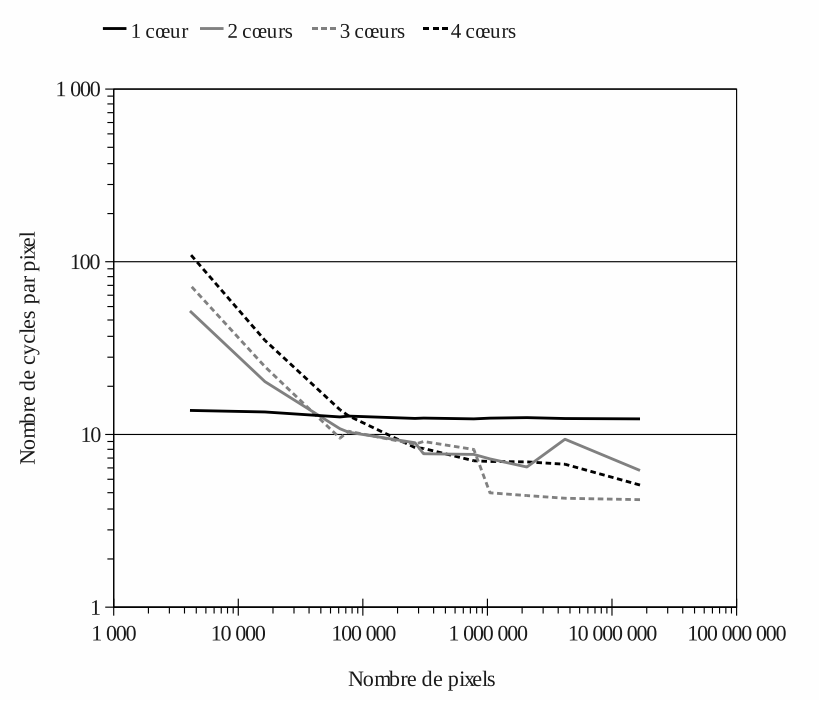


(a) 8 bits par pixel

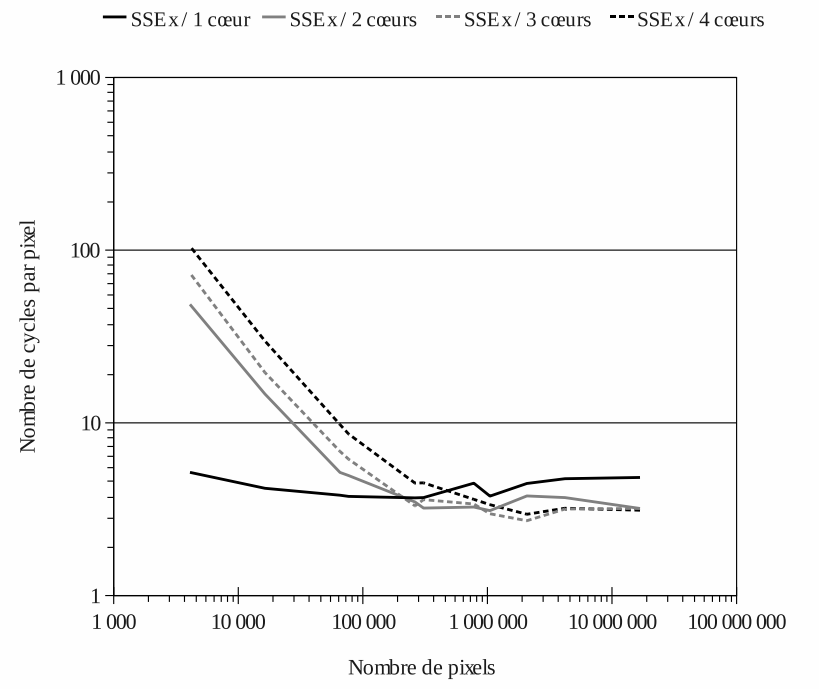


(b) 16 bits par pixel

FIG. 5.6: Nombre de cycles par pixel pour produire une addition avec 1, 2, 3, et 4 cœurs sur un processeur Intel Q9550 sans SSEx

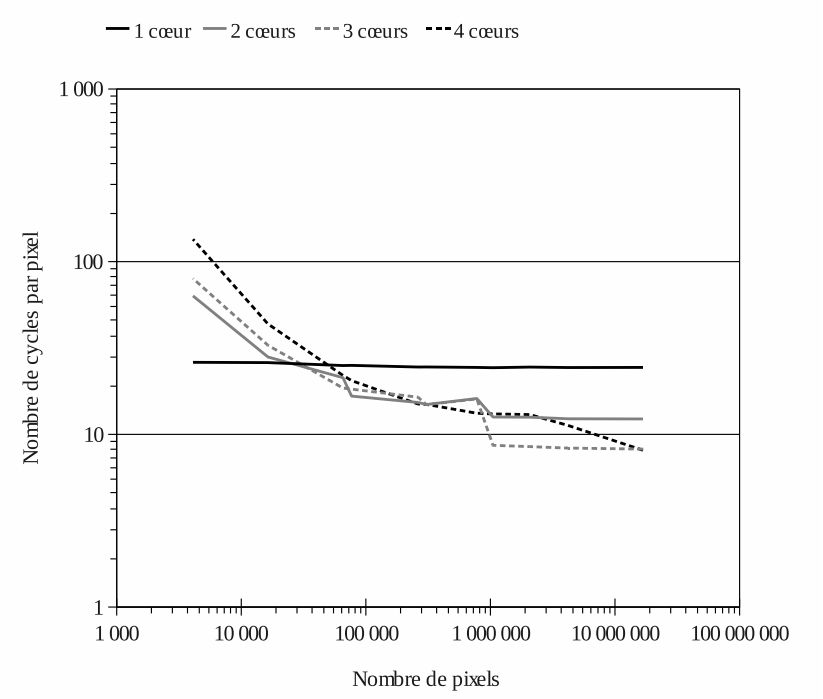


(a) 8 bits par pixel

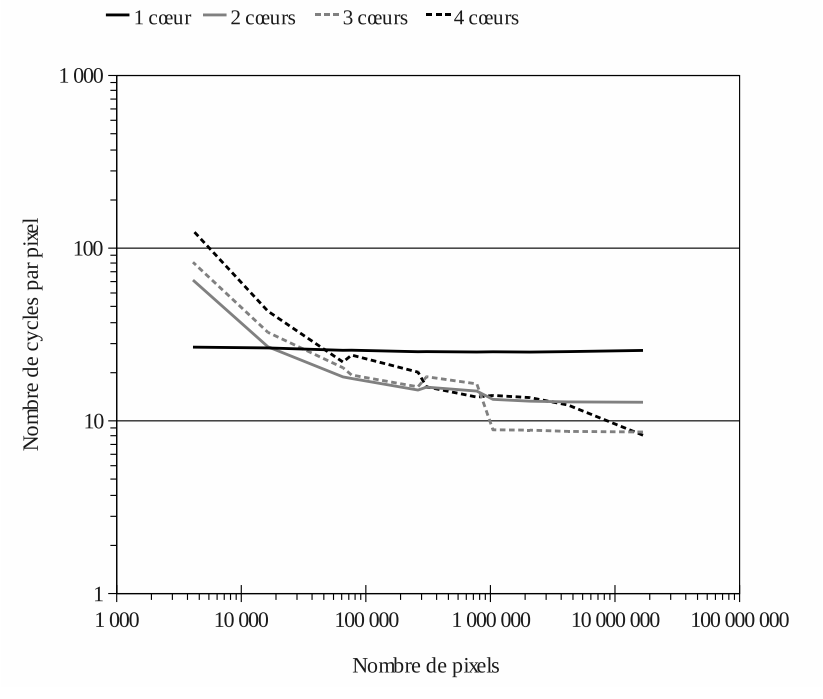


(b) 16 bits par pixel

FIG. 5.7: Nombre de cycles par pixel pour produire une érosion 3×3 avec 1, 2, 3, et 4 cœurs sur un processeur Intel Q9550 sans SSEx

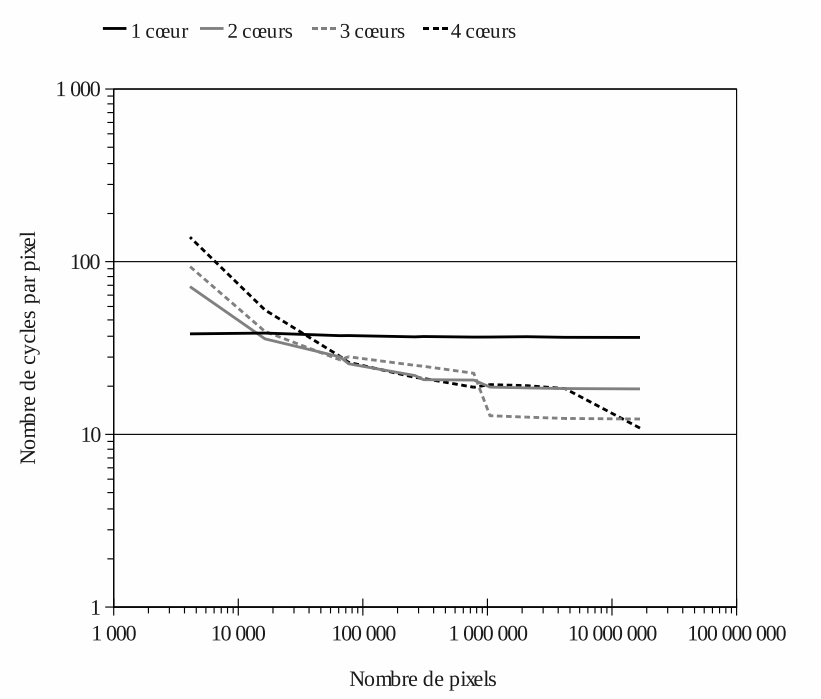


(a) 8 bits par pixel

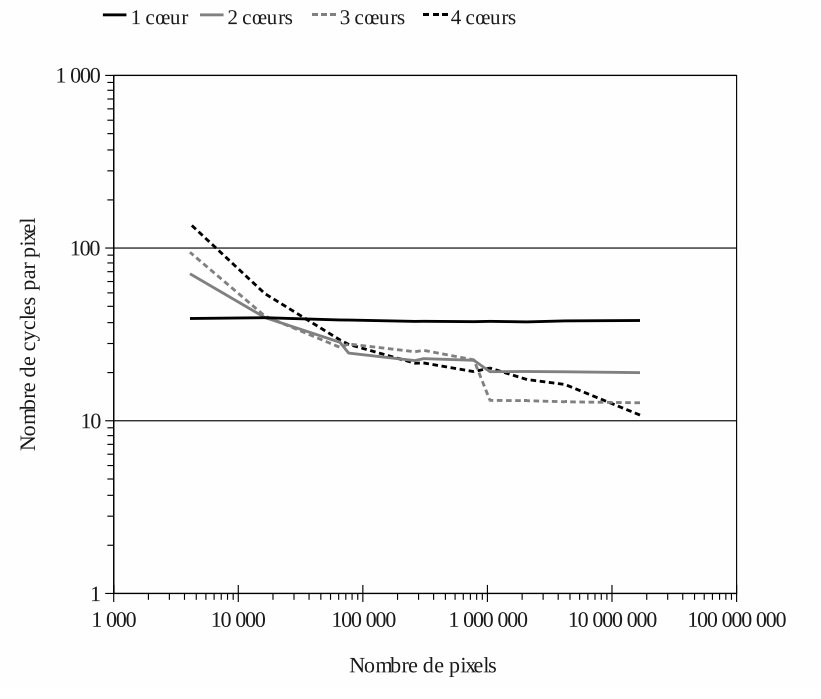


(b) 16 bits par pixel

FIG. 5.8: Nombre de cycles par pixel pour produire une érosion 5×5 avec 1, 2, 3, et 4 cœurs sur un processeur Intel Q9550 sans SSEx



(a) 8 bits par pixel



(b) 16 bits par pixel

FIG. 5.9: Nombre de cycles par pixel pour produire une érosion 7×7 avec 1, 2, 3, et 4 cœurs sur un processeur Intel Q9550 sans SSEx

Op. / Images	**Érosion** 3×3				**Érosion** 5×5			
	8 bits		16 bits		8 bits		16 bits	
	1C*	2C**	1C	2C	1C	2C	1C	2C
4096×4096	73054	36719	76934	38810	144910	72871	151786	76049
2048×2048	18350	13919	18983	9711	36183	18268	37354	19100
1024×1024	4611	2678	4750	2513	9020	4674	9319	4941
512×512	1148	830	1183	881	2276	1418	2330	1398
256×256	292	250	301	249	580	493	593	416
128×128	78	117	79	112	150	162	153	155
64×64	19	75	20	74	37	91	38	94

Op. / Images	**Érosion** 7×7				**Addition**			
	8 bits		16 bits		8 bits		16 bits	
	1C	2C	1C	2C	1C	2C	1C	2C
4096×4096	216140	108826	225988	112724	16106	10718	21902	20664
2048×2048	54106	27321	56217	28498	4038	2590	6181	5137
1024×1024	13580	6969	13965	7136	970	737	1560	966
512×512	3402	2031	3488	2068	241	249	242	254
256×256	864	651	888	660	61	118	61	119
128×128	223	207	229	229	16	80	16	80
64×64	55	103	56	102	4	74	4	71

*Utilise un cœur. **Utilise deux cœurs.

TAB. 5.2: Temps de calcul en microsecondes de quelques opérations de traitement d'images sur un processeur Intel Q9550 sans SSEx

On remarque d'une manière générale que lorsque le nombre de cycles par pixel est faible, la mise en œuvre, en mode multicœurs, d'une opération n'est intéressante qu'à partir d'un nombre important de pixels à traiter.

Cette dernière observation est assez logique puisque les processeurs multicœurs ne sont pas dimensionnés pour, qu'au même moment, tous les cœurs puissent disposer d'une pleine bande passante mémoire. Nous pouvons observer ce phénomène dans d'autres cas de figure et notamment lorsque trois ou quatre cœurs sont utilisés, nous remarquons alors peu de différence en termes de nombre de cycles par pixel et nous remarquons même parfois des performances moins bonnes lorsque quatre cœurs sont utilisés.

La figure 5.10 montre l'évolution du nombre de cycles par pixels lorsque deux ou trois opérations sont fusionnées ou non au sein d'un même thread. Ces résultats ont été obtenus en utilisant trois cœurs et en employant deux politiques de calcul. La première, dite fusionnée, consiste à itérer sur un cœur en charge d'une bande de l'image les différentes érosions, alors que la seconde, dite non fusionnée, consiste à réaliser une érosion multicœur pour ensuite l'enchaîner autant de fois que voulu. On observe sur cette figure qu'il y a peu d'intérêt à fusionner des calculs au niveau imagette. En comparant cette figure avec la figure 5.7 on remarque que la fusion est intéressante uniquement dans un cas, c'est à dire pour des images de moins de 100 000 pixels, il est de toute façon préférable d'utiliser une implémentation monocœur d'une érosion que l'on itère ensuite plusieurs fois. En effet, il ne faut pas oublier que l'utilisation de plusieurs cœurs introduit des temps de préparation des données et de création des threads non négligeables.

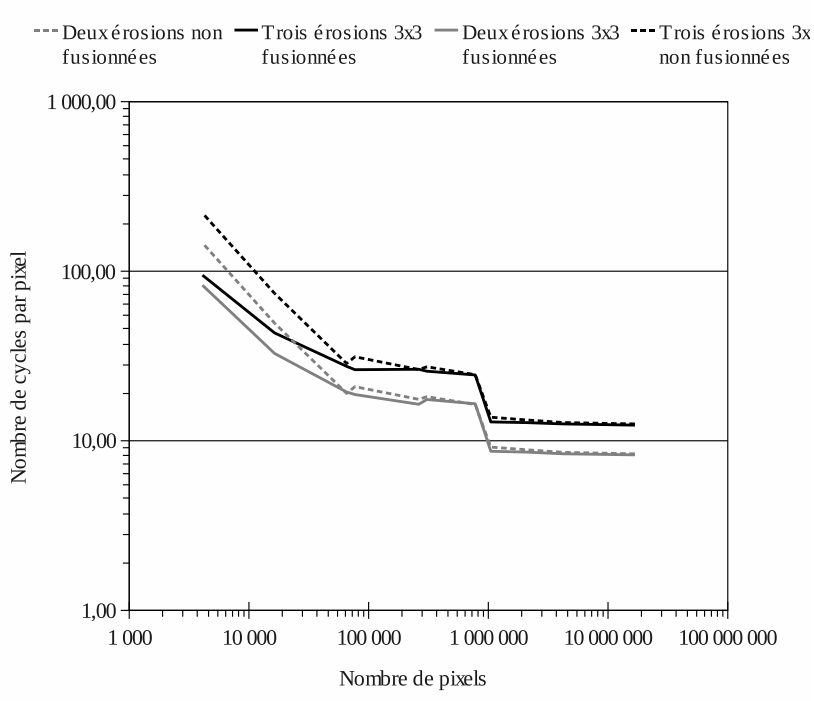


FIG. 5.10: Nombre de cycles par pixel pour produire deux et trois érosions 3×3 8 bits fusionnées au sein du même coeur de traitement sur un processeur Intel Q9550 sans SIMD.

5.2.2.3 Tests avec instructions SIMD

Un échantillon des résultats des tests est présenté dans le tableau 5.2. Les figures 5.11, 5.12, 5.12, 5.12 permettent d'observer le nombre de cycles par pixel en fonction du nombre de pixels à traiter pour des opérateurs multicœurs avec SSEx. On remarque en premier lieu le même surcoût que lors de la mise en place d'opérateurs multicœurs sans SSEx lorsque de petites images sont employées. Toutefois, la rentabilité des opérations utilisant à la fois le multicœur et les instructions SSEx se situe à un seuil, en terme de nombre de pixels, beaucoup plus important que lorsque les instructions vectorielles ne sont pas employées.

Cette rentabilité ne peut d'ailleurs jamais être atteinte si le nombre de *load/store* vectoriels est important par rapport aux instructions vectorielles de calcul. Le cas de l'addition d'images, figure 5.11, illustre bien ce phénomène et révèle par la même occasion une incapacité du circuit Q9550 à pouvoir alimenter en données vectorielles tous les cœurs du circuit. Cette incapacité est compréhensible, puisque dans le cadre de l'architecture Core 2, il faudrait un débit mémoire de l'ordre de 88 Go/s sur chaque cache de niveau 2. Pour les tailles d'images que nous utilisons, le débit réel du cache L2 varie entre 10 Go/s et 50 Go/s [67].

L'utilisation de trois cœurs comme dans le cas non SIMD semble être la meilleure stratégie de parallélisation. Il serait intéressant de réaliser ces mêmes tests sur une architecture Nehalem, car on trouve un cache L2 propre à chaque cœur et un cache L3 commun à tous. L'agencement des cœurs par deux sur chaque cache L2 dans l'architecture Core 2 quadricœurs semble ne pas être optimal dans le cas du traitement d'image qui est très gourmand en bande passante mémoire.

Contrairement au cas n'utilisant pas les instructions SIMD, la fusion d'opérateurs sur les différentes bandes de l'image, chacune traitée par un cœur, peut être plus performant qu'itérer plusieurs fois un calcul non fusionné, mais multicœur. La figure 5.15 présente ce cas de figure, et l'on observe bien que la fusion d'opérateurs est toujours plus intéressante en termes de nombre de cycles par pixels. Toutefois, si l'on compare avec le graphique de la figure 5.12 il est préférable d'itérer plusieurs fois une érosion 3×3 monocœur pour des images ayant moins d'un million de pixels pour ensuite basculer sur une version fusionnée utilisant trois cœurs.

Op. / Images	**Érosion** 3×3				**Érosion** 5×5			
	8 bits		16 bits		8 bits		16 bits	
	1C*	2C**	1C	2C	1C	2C	1C	2C
4096×4096	12062	9586	28666	18954	18800	12483	51917	29161
2048×2048	3002	2380	7047	5480	4666	3402	12878	8791
1024×1024	452	691	1397	1150	856	762	2955	2099
512×512	115	157	341	323	224	214	680	597
256×256	32	99	88	119	62	124	176	190
128×128	10	73	24	85	20	77	48	104
64×64	4	65	7	70	8	72	15	79

Op. / Images	**Érosion** 7×7				**Addition**			
	8 bits		16 bits		8 bits		16 bits	
	1C	2C	1C	2C	1C	2C	1C	2C
4096×4096	26393	15128	74490	40442	10365	11099	20726	21486
2048×2048	6663	4449	18773	9964	2675	2781	5295	5657
1024×1024	1309	998	4092	2831	159	327	819	1031
512×512	331	303	1018	740	40	118	74	154
256×256	93	134	265	280	11	88	20	97
128×128	29	90	72	128	3	74	5	78
64×64	11	72	22	82	1	73	1	75

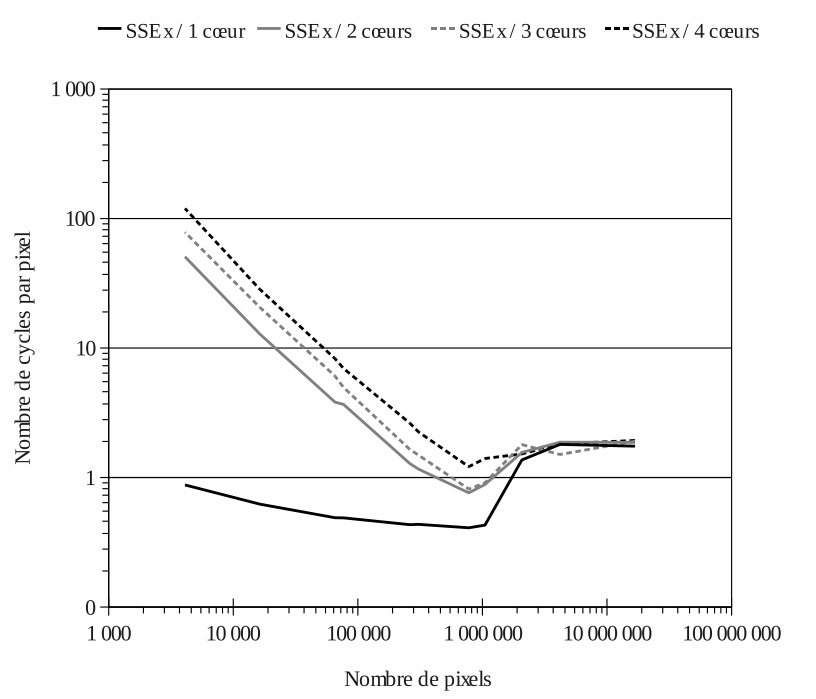
*Utilise un cœur. **Utilise deux cœurs.

TAB. 5.3: Temps de calcul en microsecondes de quelques opérations de traitement d'images sur un processeur Intel Q9550 avec SSEx

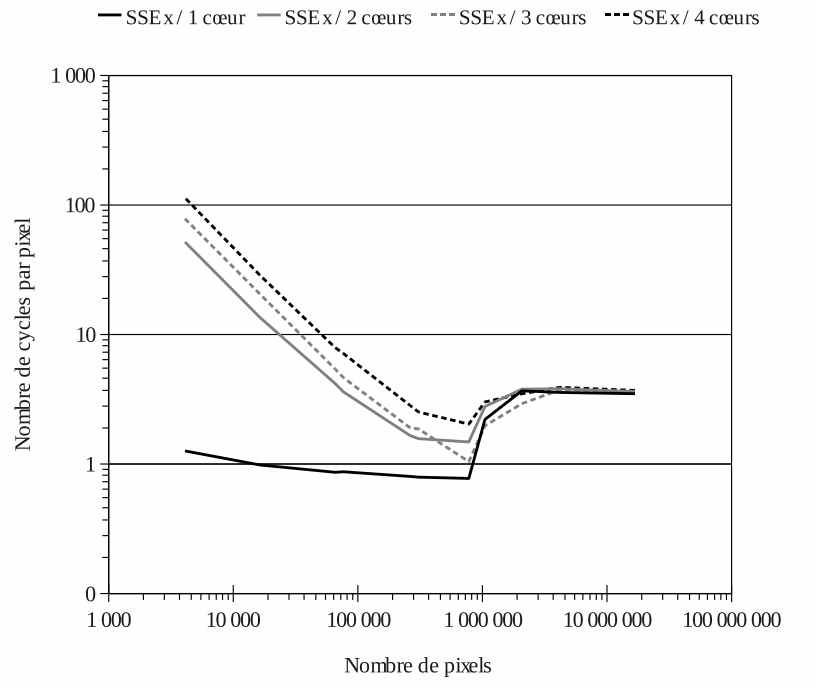
5.2.2.4 Synthèse des performances

D'une manière générale, ces quelques tests nous montrent à quel point il devient difficile d'optimiser une application sur un processeur multicœur superscalaire. Un grand nombre de mécanismes de gestion de cache, de prédictions de branchement et autres mécanismes viennent compliquer la tâche de description d'un parallélisme efficace par le programmeur. On remarque, au travers des différents graphiques, quelques tendances et notamment qu'il est souvent préférable d'utiliser les instructions vectorielles en utilisant qu'un cœur pour des images dont la résolution maximale varie entre 640×480 et 1024×1024. Au-delà, il est opportun d'utiliser une mise en œuvre multicœur de l'opérateur avec des instructions SSEx tout en ordonnançant les opérations sur les bandes de l'image.

A titre d'information, le tableau 5.4 montre l'accélération obtenue en utilisant des opérateurs avec les instructions SSEx par rapport aux mêmes opérateurs ne les mettant pas en œuvre, le tout dans un contexte monocœur ou double cœur.

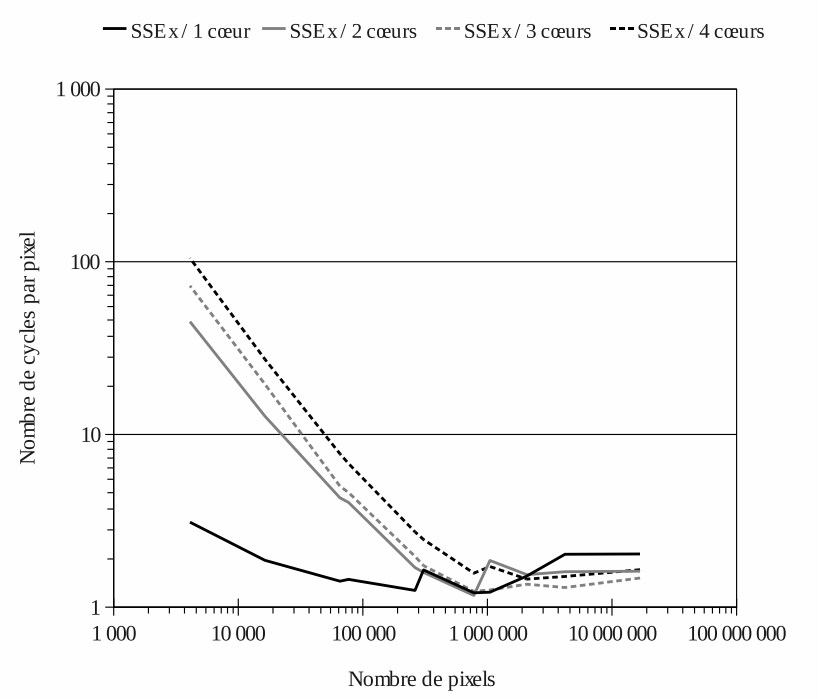


(a) 8 bits par pixel

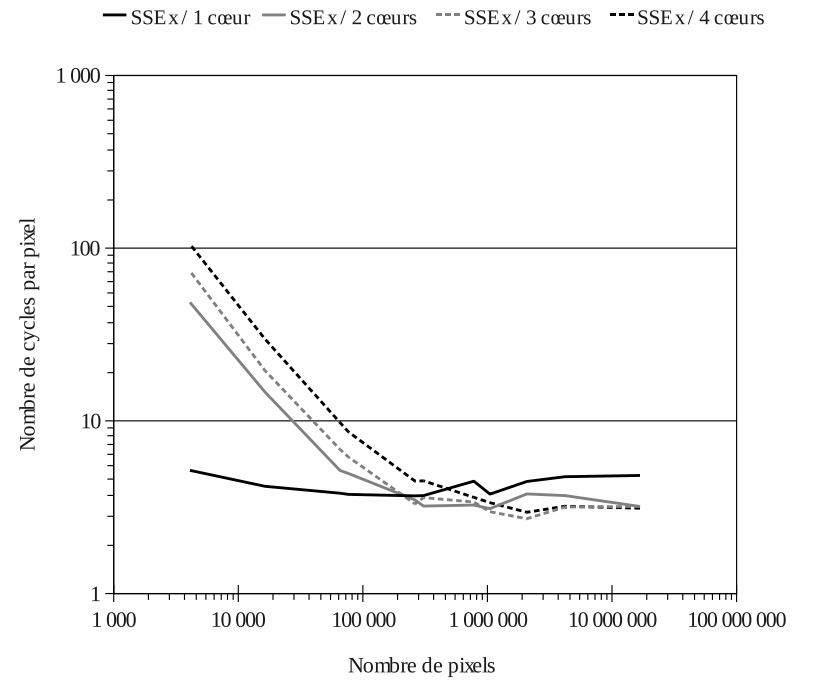


(b) 16 bits par pixel

FIG. 5.11: Nombre de cycles par pixel pour produire une addition avec 1, 2, 3, et 4 cœurs sur un processeur Intel Q9550 avec SSEx

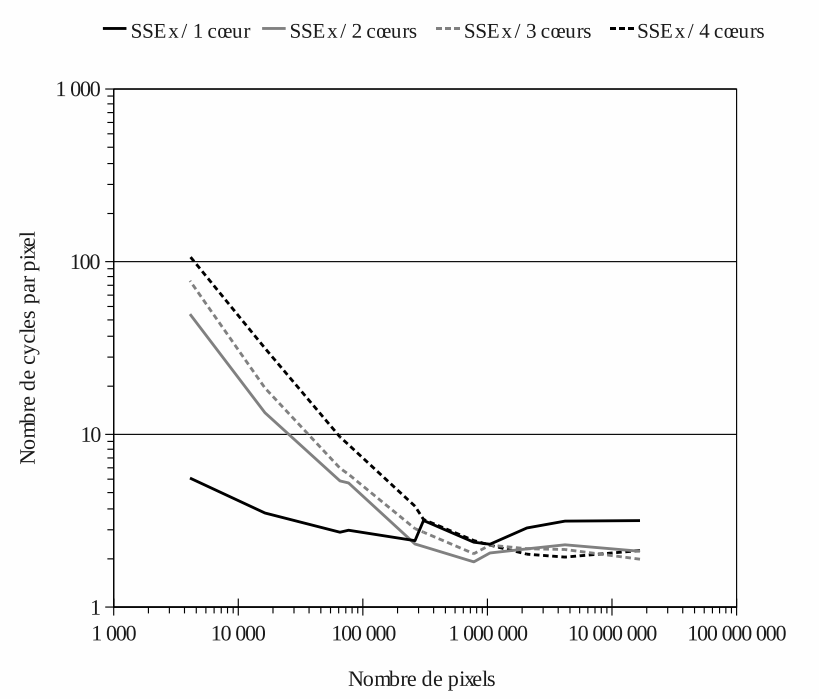


(a) 8 bits par pixel

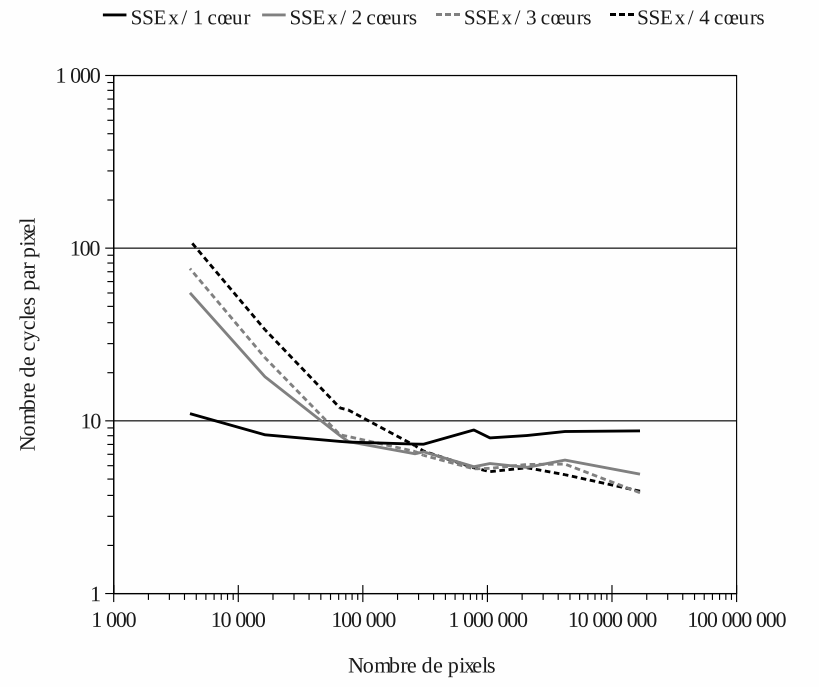


(b) 16 bits par pixel

FIG. 5.12: Nombre de cycles par pixel pour produire une érosion 3×3 avec 1, 2, 3, et 4 cœurs sur un processeur Intel Q9550 avec SSEx

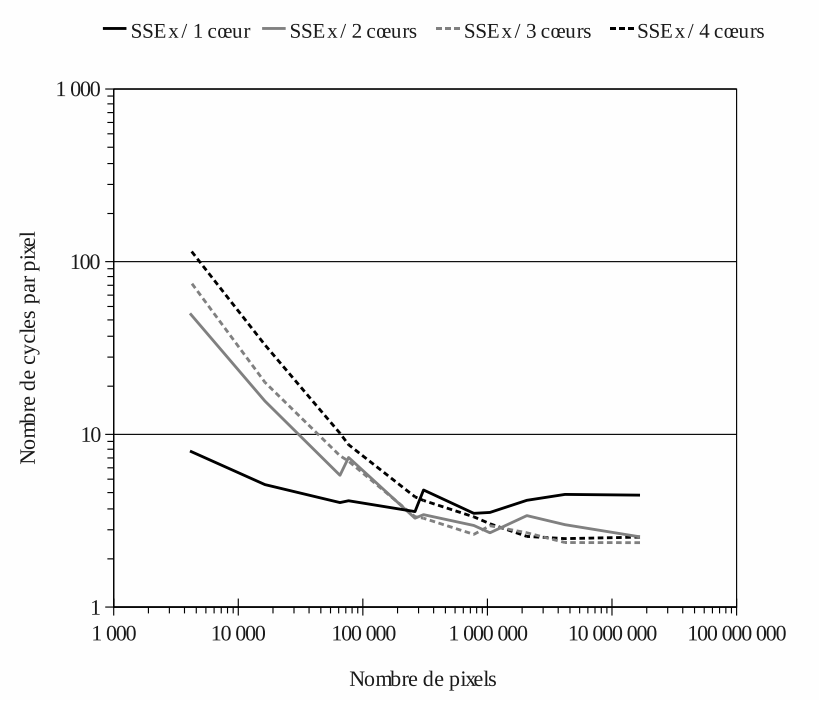


(a) 8 bits par pixel

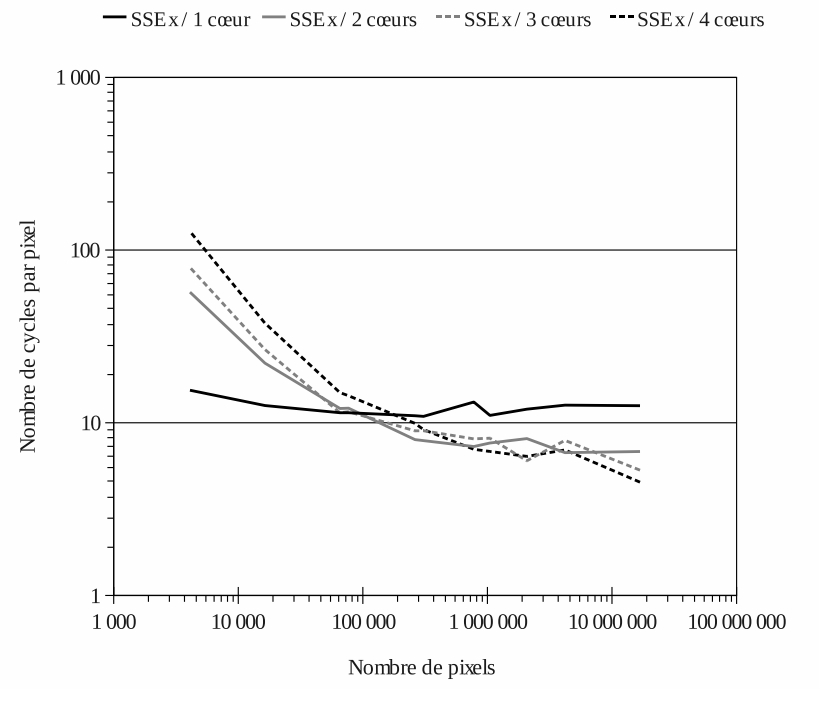


(b) 16 bits par pixel

FIG. 5.13: Nombre de cycles par pixel pour produire une érosion 5×5 avec 1, 2, 3, et 4 cœurs sur un processeur Intel Q9550 avec SSEx



(a) 8 bits par pixel



(b) 16 bits par pixel

FIG. 5.14: Nombre de cycles par pixel pour produire une érosion 7×7 avec 1, 2, 3, et 4 cœurs sur un processeur Intel Q9550 avec SSEx

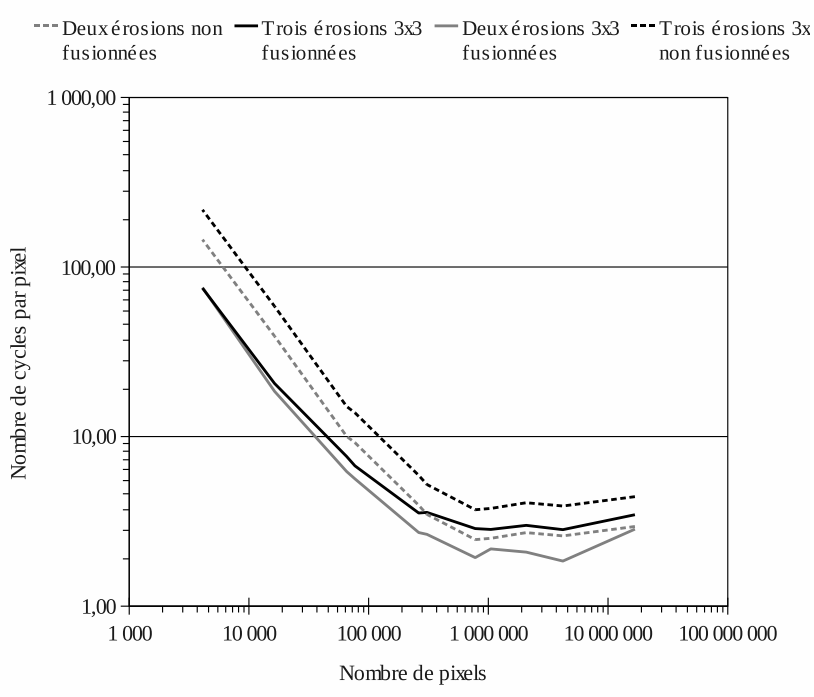


FIG. 5.15: Nombre de cycles par pixel pour produire deux et trois érosions 8 bits 3×3 fusionnées au sein du même cœur de traitement sur un processeur Intel Q9550 avec SSEx

Op. / **Images**	**Érosion** 3×3				**Érosion** 5×5			
	8 bits		16 bits		8 bits		16 bits	
	1C*	2C**	1C	2C	1C	2C	1C	2C
4096×4096	6.05	3.83	2.68	2.04	7.70	5.84	2.92	2.60
2048×2048	6.11	5.85	2.69	1.77	7.75	5.37	2.90	2.17
1024×1024	10.20	3.88	3.40	2.19	10.54	6.13	3.15	2.35
512×512	9.98	5.29	3.47	2.73	10.16	6.63	3.43	2.34
256×256	9.12	2.53	3.42	2.09	9.35	3.98	3.37	2.19
128×128	7.80	1.60	3.29	1.32	7.50	2.10	3.19	1.49
64×64	4.75	1.15	2.86	1.06	4.62	1.26	2.53	1.19
Op. / **Images**	**Érosion** 7×7				**Addition**			
	8 bits		16 bits		8 bits		16 bits	
	1C	2C	1C	2C	1C	2C	1C	2C
4096×4096	8.19	7.19	3.03	2.79	1.55	0.97	1.06	0.96
2048×2048	8.12	6.14	2.99	2.86	1.51	0.93	1.17	0.91
1024×1024	10.37	6.98	3.41	2.52	6.10	2.25	1.90	0.94
512×512	10.28	6.70	3.43	2.79	6.02	2.11	3.27	1.65
256×256	9.29	4.86	3.35	2.36	5.55	1.34	3.05	1.23
128×128	7.69	2.30	3.18	1.79	5.33	1.08	3.20	1.03
64×64	5.00	1.43	2.55	1.24	4.00	1.01	4.00	0.95

*Utilise un cœur. **Utilise deux cœurs.

TAB. 5.4: Accélération obtenue sur un processeur Intel Q9550 pour une implémentation SSEx d'une érosion 7×7 et d'une addition de deux images par rapport à une implémentation C standard

5.2.3 Pipeline de processeurs de voisinage SPoC

Les processeurs de voisinage présentent des performances assez atypiques puisqu'ils traitent les données dès que les premiers pixels de l'image lui sont envoyés. Leur seule limite réside principalement dans la bande passante des mémoires dans lesquelles sont stockées les images sources et destinations. Ces processeurs peuvent être facilement montés à des fréquences importantes puisque l'on peut aisément casser les chemins critiques en ajoutant des registres au sein de l'architecture. Ces registres influent juste sur la latence, qui de toute manière est assez importante puisque la sortie du processeur est valide uniquement lorsque l'envoi de la deuxième ligne de l'image source a commencé.

Comme énoncé dans l'annexe A nous utilisons pour chaque processeur de voisinage un extracteur parallélisé couplé à quatre unités de calculs du voisinage produisant quatre pixels érodés ou dilatés par cycle. Chaque processeur du pipeline réalise donc 32 min ou max par cycle et chaque ALU située après dans le flot est capable de traiter les vecteurs de quatre pixels en provenance de deux processeurs de voisinage en parallèle. Cette structure permet d'atteindre une puissance de calcul totale de 108,8 GOPS pour un pipeline SPoC à 8 étages fonctionnant à 200 Mhz soit 13,6 GOPS par étage.

Ce pipeline n'est en mesure de traiter qu'un seul format de données. Si les pixels de l'image sont codés sur 16 bits et qu'un traitement sur 8 bits est nécessaire, aucun mécanisme n'est réellement prévu à part celui de compléter automatiquement les pixels 8 bits en 16 bits.

Comme dans le cas des GPU, nous ne prendrons pas en compte les temps de transferts depuis et vers l'hôte et nous supposons disposer d'une interface mémoire de 3,2 Go/s nous permettant de lire deux images et écrire deux images 8 bits en même temps. Cette bande passante mémoire est standard pour des barrettes DDR2 PC2-3200 d'entrée de gamme. Si des images 16 bits sont utilisées et que l'on souhaite conserver une extraction du voisinage parallélisée par quatre, il est alors nécessaire de doubler la bande passante mémoire. Cette option est largement réalisable et il faudrait utiliser une mémoire DDR2 PC2-6400. Une alternative au traitement de données 16 bits avec une bande passante mémoire de 3,2 Go/s serait d'utiliser des extracteurs de voisinage uniquement parallélisés par deux, mais aurait pour effet de multiplier par deux les temps de traitement.

Les performances brutes pour quelques opérations tests sélectionnées sont données dans le tableau 5.5. On remarque que, quel que soit le calcul réalisé, les temps de calcul ne varient pas pour une taille d'image fixe. En effet, une érosion 5×5 est réalisée en enchaînant deux érosions 3×3 et la structure même du pipeline SPoC nous oblige à circuler à travers les six étages restant pour sortir les données. De la même manière, l'érosion 3×3 ne consomme que le premier étage, mais il est pourtant nécessaire de passer à travers les sept autres en réalisant un "NOP" sur le flux pixels. C'est la raison pour laquelle les temps de calcul sont stables lorsqu'un seul passage dans le pipeline est nécessaire.

Le tableau nous montre également que le nombre de cycles par pixel est relativement stable et augmente lentement lorsque la taille des images diminue à cause de la latence du pipeline. Il est tout à fait normal que le nombre de cycles par pixel soit égal à 0,25 puisque nous avons parallélisé par quatre les extracteurs dans les processeurs de voisinage, de sorte qu'ils soient capables de produire quatre pixels résultats par coup d'horloge.

Op. / Images	**Érosion** 3×3		**Érosion** 5×5		**Érosion** 7×7		**Addition**	
	Temps*	CPP**	Temps	CPP	Temps	CPP	Temps	CPP
4096×4096	21012	0.250	21012	0.250	21012	0.250	21012	0.250
2048×2048	5263	0.251	5263	0.251	5263	0.251	5263	0.251
1024×1024	1321	0.252	1321	0.252	1321	0.252	1321	0.252
512×512	332	0.254	332	0.254	332	0.254	332	0.254
256×256	84.5	0.258	84.5	0.258	84.5	0.258	84.5	0.258
128×128	21.8	0.266	21.8	0.266	21.8	0.266	21.8	0.266
64×64	5.8	0.283	5.8	0.283	5.8	0.283	5.8	0.283

*Temps en microsecondes **Nombre de Cycle Par Pixel.

TAB. 5.5: Performances des opérations tests avec extracteur parallélisé par quatre sur l'accélérateur SPoC

5.2.4 Processeur VLIW vectoriel

Le tableau 5.6 présente les performances obtenues sur le pool d'opérations sélectionnées en considérant des images sur 8 et 16 bits par pixel. On remarquera le coût de traitements d'images codées sur 16 bits par rapport à celles codées sur 8. En effet, les vecteurs ayant un nombre de bits fixe, le nombre d'éléments au sein de ces derniers est deux fois moindre dans le cas d'instructions 16 bits vis-à-vis d'instructions impliquant des vecteurs composés d'éléments 8 bits.

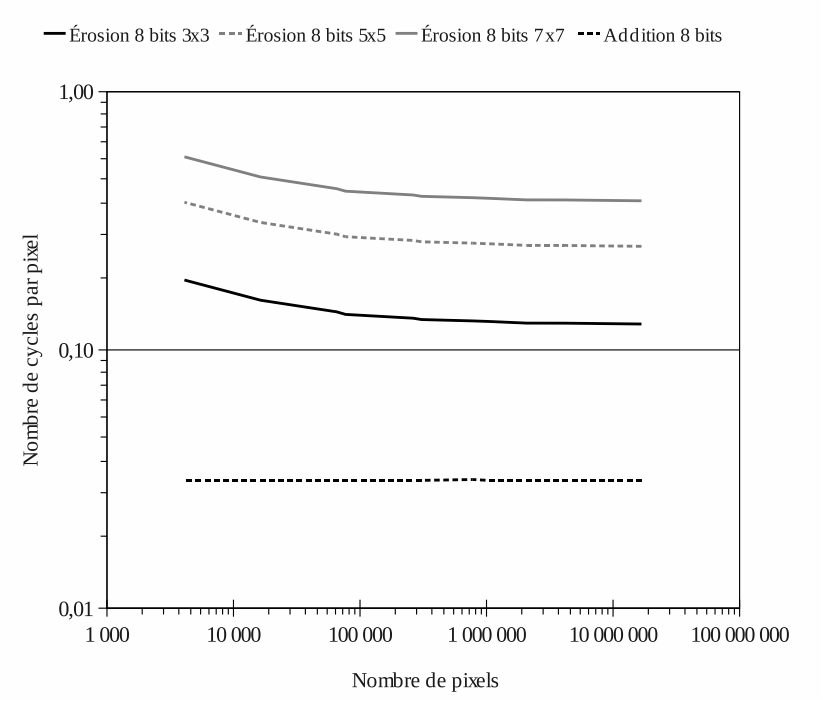
Le débit mémoire nécessaire afin d'alimenter en données un processeur VLIW vectoriel est de l'ordre de 6,2 Go/s ce qui est du même ordre de grandeur que le débit mémoire nécessité par le pipeline SPoC à fréquence égale. On obtient une puissance de calcul théorique sur un cœur VLIW de l'ordre de 8 GOPS pour des opérations 8 bits et 4 GOPS pour des opérations 16 bits. Cette puissance de calcul peut être largement augmentée en raccordant en pipeline plusieurs cœurs ou bien en augmentant la taille des vecteurs.

Ce processeur nous permet de mettre en place notre principe d'érosion par décalage des lignes via l'unité de calcul vectoriel capable de décaler un vecteur avant de réaliser un calcul, le tout en un seul cycle, ce qui nous permet d'obtenir de meilleurs temps de calcul pour une puissance de calcul théorique inférieure à celle d'un étage du pipeline de SPoC.

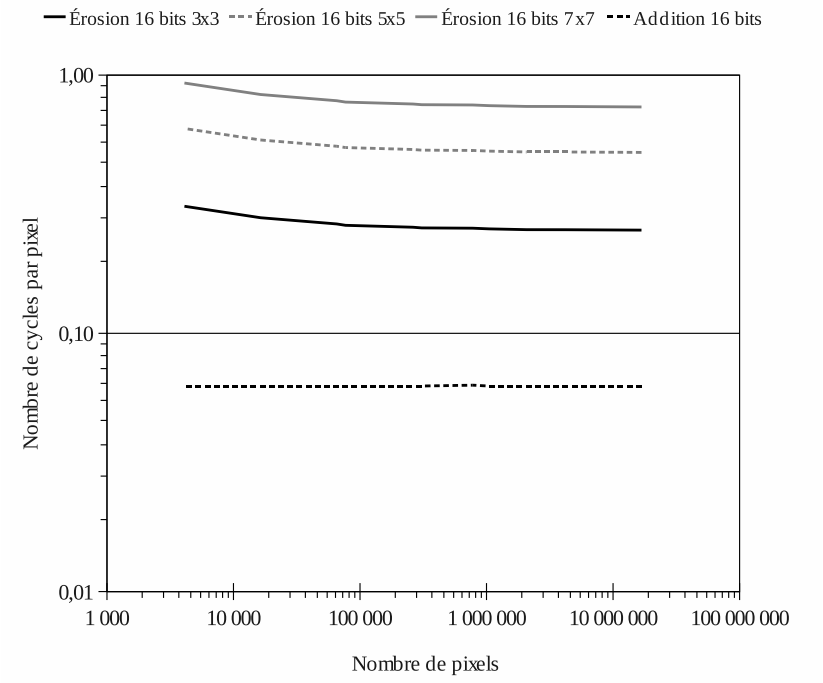
Op. / Images	**Érosion** 3×3		**Érosion** 5×5		**Érosion** 7×7		**Addition**	
	8 bits	16 bits	8 bits	16 bits	8 bits	16 bits	8 bits	16 bits
4096×4096	10566	21050	21132	42101	31699	63152	2621	5242
2048×2048	2661	5282	5323	10565	7985	15847	655	1310
1024×1024	675	1330	1351	2661	2026	3991	163	327
512×512	173	337	347	675	521	1012	40	81
256×256	46	86	92	173	138	260	10	20
128×128	12	22	25	45	38	68	2,5	5,12
64×64	3,8	6,3	7,6	12	11	19	0,6	1,2

TAB. 5.6: Temps de calcul en microsecondes de quelques opérations de traitement d'images sur le processeur VLIW Vectoriel

La figure 5.16 nous permet d'observer le nombre de cycles par pixel produit en sortie des opérateurs tests. On observe que les performances sont très facilement prédictibles, d'une part, car aucune gestion du cache n'est mise en place et d'autre part, grâce au générateur de boucle évitant toutes les erreurs liées à la prédiction de branchement.

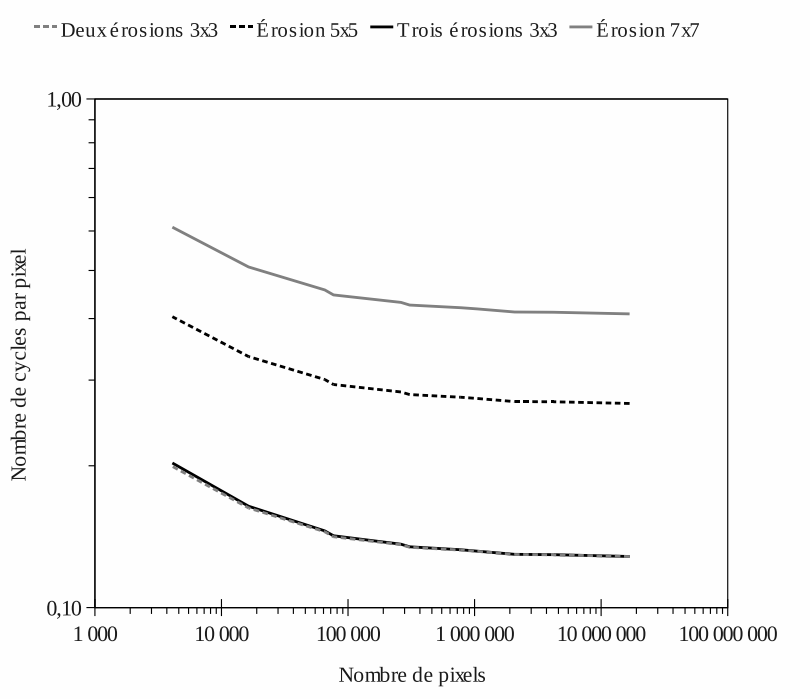


(a) 8 bits par pixel

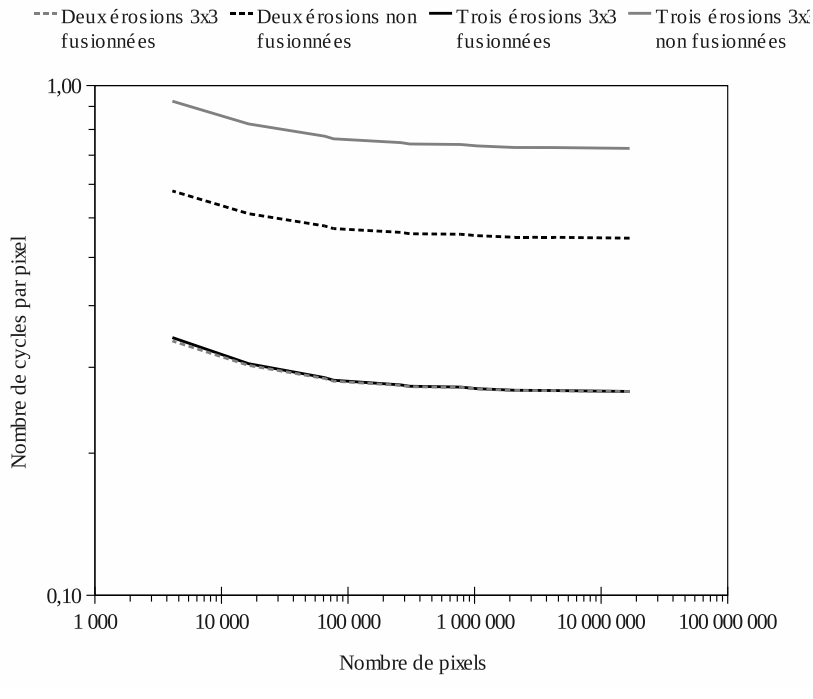


(b) 16 bits par pixel

FIG. 5.16: Nombre de cycles par pixel pour réaliser les opérations de test sur le processeur VLIW vectoriel



(a) 8 bits par pixel



(b) 16 bits par pixel

FIG. 5.17: Nombre de cycles par pixel pour produire deux ou trois érosions 3×3 regroupées sur un cœur VLIW ou réparties respectivement sur deux ou trois cœurs en pipeline

Nous remarquons à la figure 5.17, tout comme dans le cadre du pipeline SPoC, l'intérêt du pipeline de processeur VLIW vectoriel. En effet, le chaînage de plusieurs opérations permet un gain de temps non négligeable grâce des aller-retour vers la mémoire hôte évités ainsi qu'au fait de pouvoir amorcer les calculs à un étage dès la disponibilité des premiers résultats de l'étage précédent.

5.3 Synthèse des performances brutes

Après avoir décrit précisément les performances intrinsèques de chaque machine cible réalisant nos opérateurs de tests, nous allons proposer un comparatif des performances de ces différentes architectures. Nous essayons de nous placer dans un cas favorable à toutes afin de disposer d'un comparatif reflétant les points forts de chaque système.

Les calculs sont réalisés, pour chaque accélérateur, dans la configuration suivante :

- les transferts hôte accélérateur sont mis de côté le cas échéant
- le processeur Intel Q9550 utilise les instructions SIMD
- le processeur Intel Q9550 est considéré comme deux architectures cibles, une cible utilisant un seul cœur et une cible utilisant trois cœurs (nous avons montré précédemment qu'il était préférable d'utiliser trois cœurs sur ce système plutôt que quatre).
- le comportement des calculs en 8 et 16 bits étant assez similaire, nous prenons en compte uniquement les calculs sur 8 bits.

La figure 5.18 présente le nombre de cycles par pixel pour réaliser une addition de deux images. Ce comparatif nous permet d'atteindre souvent la puissance de calcul théorique d'une architecture et l'on observe la performance du processeur vectoriel qui atteint 0,031 cycle par pixel quelque soit la taille de l'image. Le pipeline SPoC est lui aussi constant, mais avec un nombre de cycles par pixel égal à environ 0,250. Les autres architectures sont pénalisées principalement pour les petites images, à cause du surcoût engendré par la gestion de threads et pour les grosses images, à cause des défauts de cache de l'architecture. En effet l'addition ne bénéficie pas d'une bonne localité des données puisque le nombre de *load/store* confondu est plus important que le nombre d'instructions de calcul.

Différents calculs d'érosions sont proposés aux figures 5.19, 5.20, 5.21 et les figures 5.22, 5.23 présentent ces mêmes calculs, mais réalisés de manière fusionnée. Le cas du pipeline SPoC est quelque peu à part puisque nous ne pouvons faire autrement que de fusionner des érosions de rayon unitaire pour en obtenir de rayon supérieur. Une fois de plus, nous remarquons les bonnes performances du processeur VLIW vectoriel même dans les cas non fusionnés bien qu'il soit préférable de l'utiliser dans ce mode.

Tous ces comparatifs doivent bien sûr être relativisés par les aspects de surface et de consommation liés à chacune des architectures. Les tableaux 5.7 et 5.8 synthétisent ces différents points et montrent la pertinence de notre approche, car d'une part, la consommation de nos accélérateurs ainsi que le nombre de ressources logiques mises en œuvre sont largement inférieures à celles caractérisant les processeurs généralistes et les GPU et d'autre part, les temps de calcul, même si nos systèmes ne sont que dans des FPGA, sont loin d'être mauvais et même parfois meilleurs.

Notre objectif de gain en souplesse entre les différents accélérateurs tout en minimisant la perte de performance est donc atteint. Nous sommes maintenant en mesure, avec notre

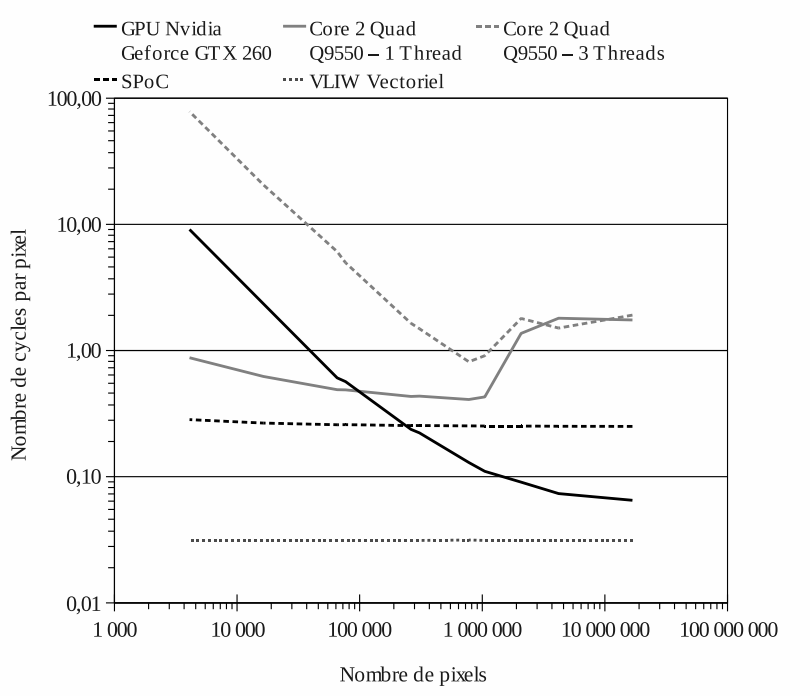


FIG. 5.18: Comparatif du nombre de cycles par pixel pour réaliser une addition d'images avec les différentes architectures sélectionnées

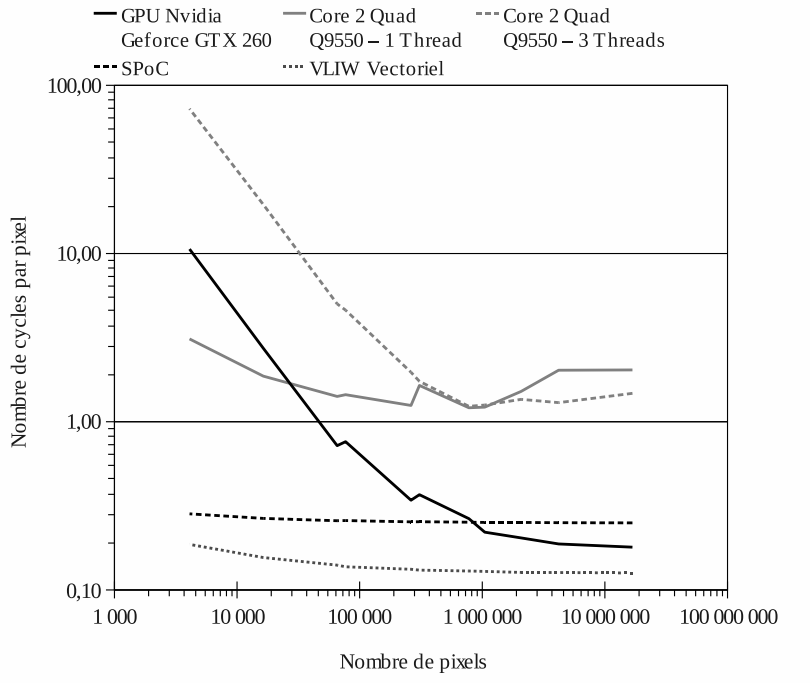


FIG. 5.19: Comparatif du nombre de cycles par pixel pour réaliser une érosion 3×3 avec les différentes architectures sélectionnées

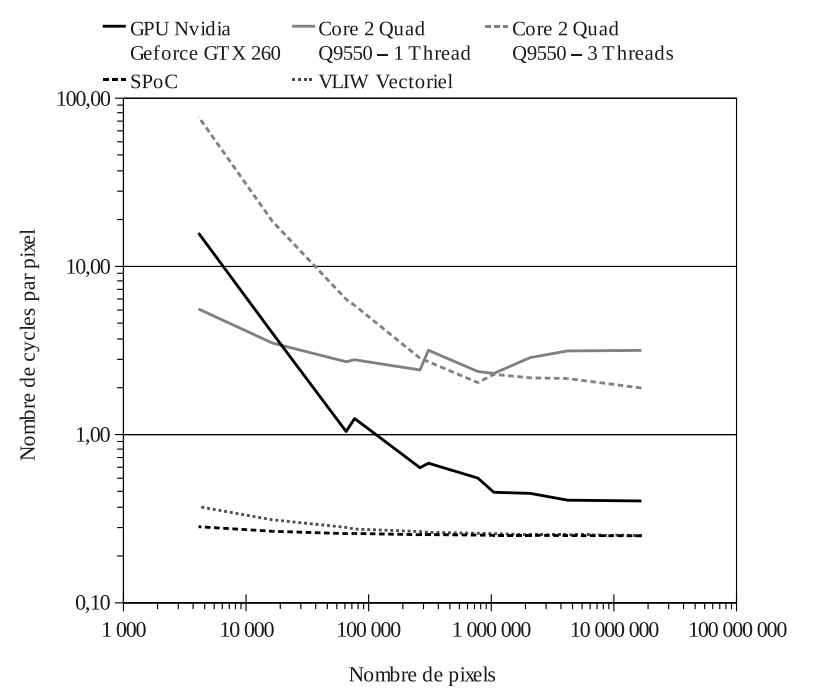


FIG. 5.20: Comparatif du nombre de cycles par pixel pour réaliser une érosion 5×5 avec les différentes architectures sélectionnées

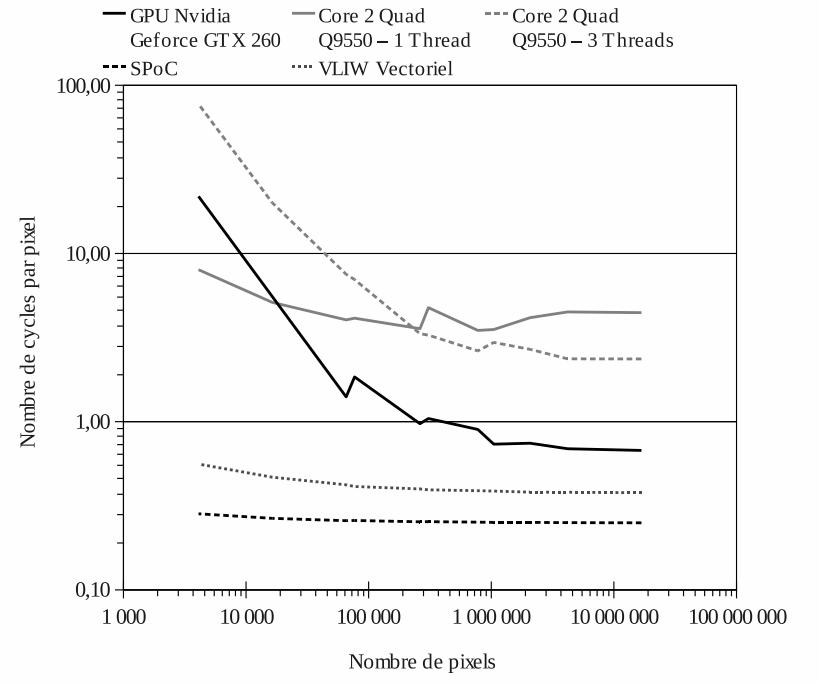


FIG. 5.21: Comparatif du nombre de cycles par pixel pour réaliser une érosion 7×7 avec les différentes architectures sélectionnées

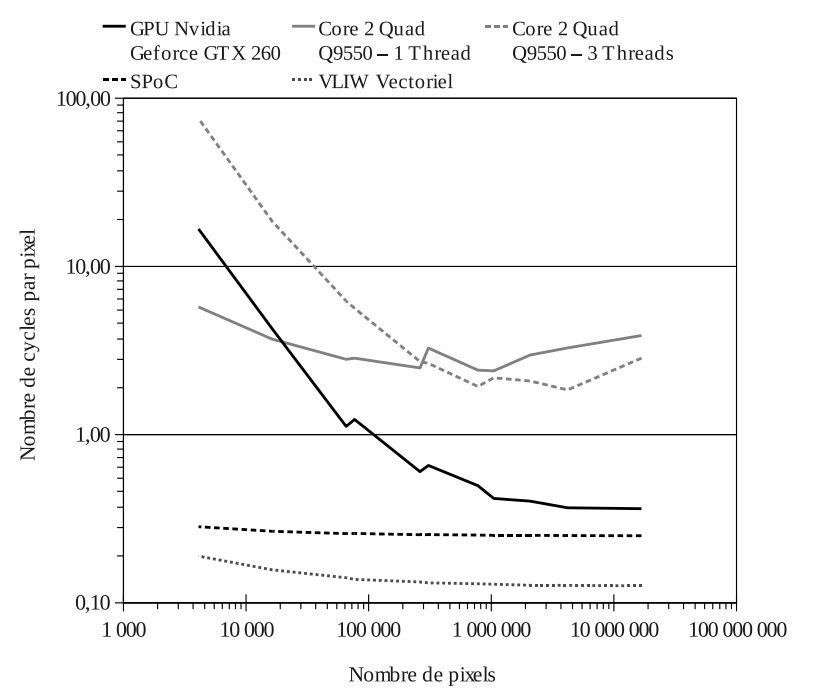


FIG. 5.22: Comparatif du nombre de cycles par pixel pour enchaîner deux érosions 3×3 avec les différentes architectures sélectionnées

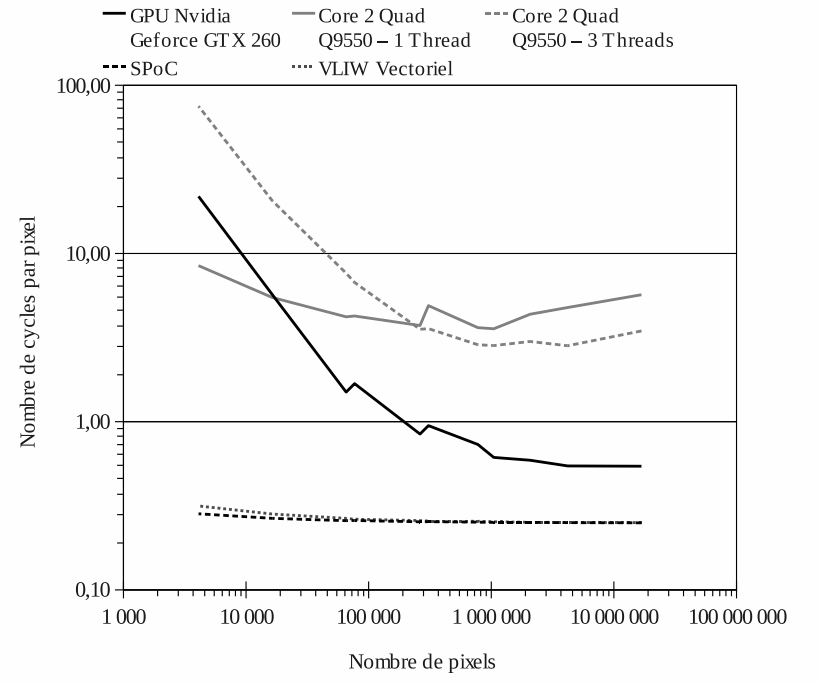


FIG. 5.23: Comparatif du nombre de cycles par pixel pour enchaîner trois érosions 3×3 avec les différentes architectures sélectionnées

		Un étage SPoC	Cœur VLIW Vectoriel
Slices	Regs	1574	2960
	LUTs	1675	7583
BRAM (36 kbits)	Data	32	16
	Instr	-	8
DSP48e		-	70
Fréquence Max		375 MHz	200 MHz
Consommation à 200MHz		≈ 5 W	≈ 10 W

TAB. 5.7: Ressources matérielles d'un Virtex-5 utilisées par le pipeline SPoC ou par le processeur VLIW Vectoriel

	Intel Q9550
Nombre de Cœurs	4
Taille du Cache L2	2×6 Mo
Fréquence interne	2,83 Ghz
Bande passante ext.	$10,2$ Go/s
Technologie	45 nm
Transistors	≈ 800 Millions
Consommation	≈ 95 W

	NVidia GTX 260
Processeur de flux	216
Fréquence interne	1.24 Ghz
Bande passante ext.	111.9 Go/s
Technologie	55 nm
Transistors	≈ 1,4 Milliards
Consommation	≈ 150 W

TAB. 5.8: Ressources matérielles utilisées par les différents circuits testés

système multicœur vectoriel, d'effectuer une grande quantité de calculs tout en disposant d'une bonne efficacité alliant le parallélisme spatial, grâce aux instructions vectorielles, et le parallélisme temporel inspiré par le pipeline de processeur SPoC.

5.4 Application test : segmentation de la route

5.4.1 Description de l'application

Une application de segmentation de la route est proposée afin d'observer le comportement des machines cibles dans un cas réel d'utilisation. Cette application est basée sur le calcul de la ligne de partage des eaux (LPE) décrite dans le chapitre 2 où un exemple est donné à la figure 2.39.

L'objectif de cette application est de localiser la chaussée sur une image prise depuis une voiture en déplacement. Les deux marqueurs proposés nous permettent de segmenter la chaussée du reste de l'image grâce au calcul de la LPE. Afin d'obtenir une meilleur segmentation, la LPE est calculée sur le gradient de l'image source, puis filtrée par une fermeture. Nous concentrerons notre comparaison sur le calcul de la LPE entre les différentes machines cibles citées précédemment.

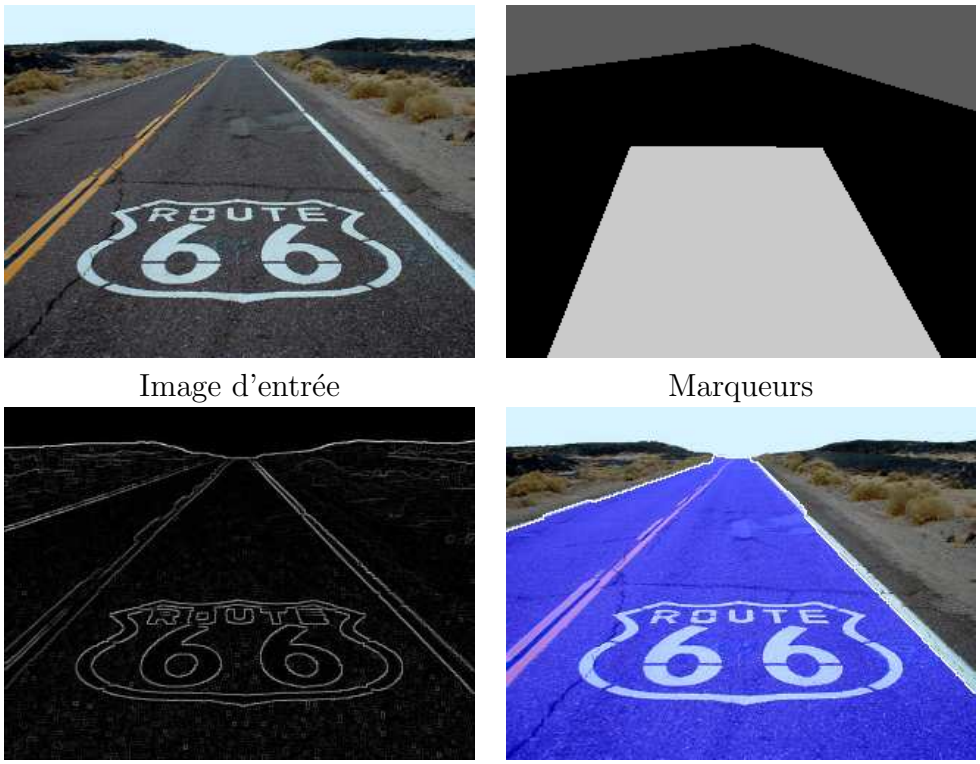


Image d'entrée | Marqueurs

Gradient | Ligne de partage des eaux

FIG. 5.24: Images de l'application test de segmentation de la route

5.4.2 Mise en œuvre de la LPE

Processeur généraliste multicœur Deux méthodes de calcul de la LPE peuvent être envisagées sur un processeur généraliste. La première consiste à employer une structure en file d'attente hiérarchique (FAH) [53] [10] permettant de propager l'information uniquement où cela est nécessaire. Cette méthode est difficilement parallélisable efficacement sur un système multicœur.

La seconde méthode consiste à utiliser la mise en œuvre standard de la LPE en effectuant à chaque niveau de gris, un SKIZ isotrope. Comme nous avons déjà pu le voir, les itérations d'un SKIZ isotrope sont spatiallement parallélisables avec une efficacité assez bonne selon le volume de données à traiter ainsi qu'en fonction de l'utilisation ou non d'instructions vectorielles SSEx.

Circuit dédié à l'accélération 3D Le grand nombre des unités parallèles de calcul des GPGPU rend la mise en place d'une file d'attente hiérarchique impossible. On privilégie alors le déploiement de la LPE par le SKIZ isotrope. Ainsi l'image masque étiquetée et l'image gradient sont transférées au sein de la mémoire du GPU dans le but de calculer la LPE en minimisant les aller-retour avec le mémoire de l'hôte. Le GPU est alors en charge de réaliser un SKIZ isotrope à chaque niveau de gris.

Pipeline de processeurs de voisinage SPoC Le pipeline SPoC permet de réaliser efficacement huit itérations d'un SKIZ isotrope en un seul transfert de l'image. La ligne de partage des eaux est donc mise en œuvre via l'enchaînement de SKIZ pour chaque niveaux de gris, tout comme sur un GPGPU, mais en exploitant un parallélisme temporel.

Toutefois, le nombre d'itérations d'un SKIZ pour un niveau de gris donné n'est pas connu à priori. Cela se traduit de la manière suivante sur un pipeline : lorsque le nombre d'itération d'un SKIZ est inférieur à la profondeur du pipeline, il n'est pas possible de réaliser plusieurs SKIZ de niveaux de gris croissant dans le pipeline. Il faut donc au minimum autant de transfert dans le pipeline que de niveaux de gris à traiter.

Les figures 5.25 et 5.26 présentent, à partir de l'exemple de la figure 5.24, cette particularité du pipeline. Avec notre image test, nous remarquons qu'une profondeur optimale du pipeline SPoC se situe autour de 10 étages afin de réaliser moins de 200 transferts dans le pipeline.

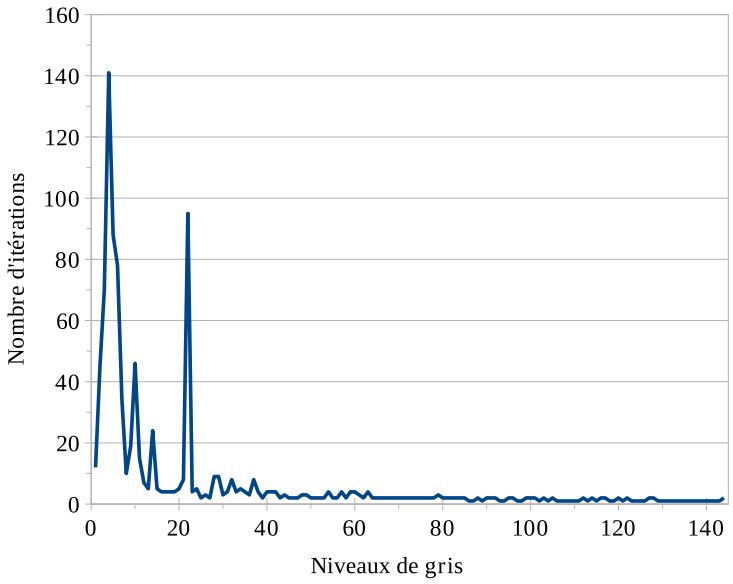


FIG. 5.25: Nombre des itérations de SKIZ pour chaque niveau de gris de l'image gradient source

Processeurs VLIW vectoriel Avec un chaînage linéaire unidirectionnel, les processeurs vectoriels peuvent travailler exactement comme le pipeline SPoC avec des résultats similaires, mais aussi avec les mêmes limitations. Nous avons donc choisi d'utiliser leur souplesse associée à un réseau linéaire bidirectionnel afin d'exploiter un parallélisme spatial.

Nous proposons donc de distribuer l'image masque et l'image gradient source entre les huit cœurs. Les figures 5.27 et 5.28 présentent l'utilisation du SoC embarquant les processeurs vectoriels linéairement chaînés.

5.4.3 Résultats

Le tableau 5.29 présente les résultats obtenus pour chacune des plate-formes.

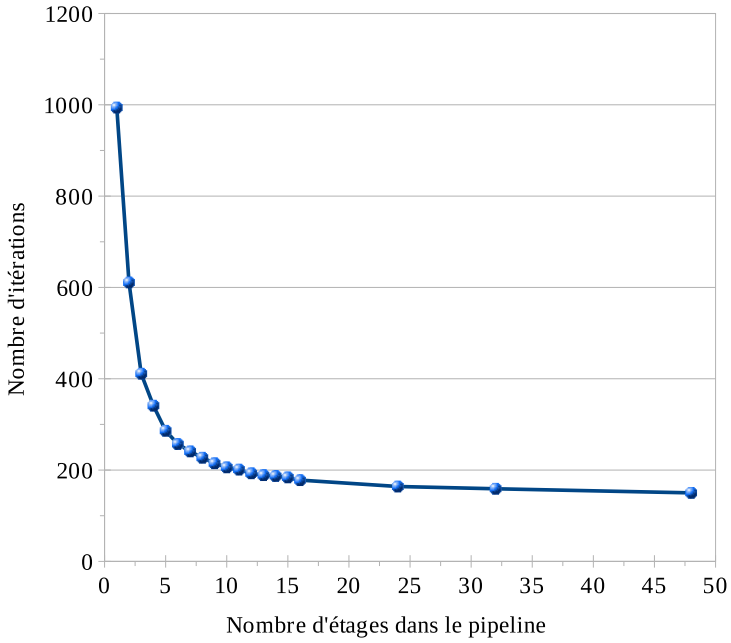


FIG. 5.26: Nombre de transfert d'images vers le pipeline en fonction de la profondeur de ce dernier

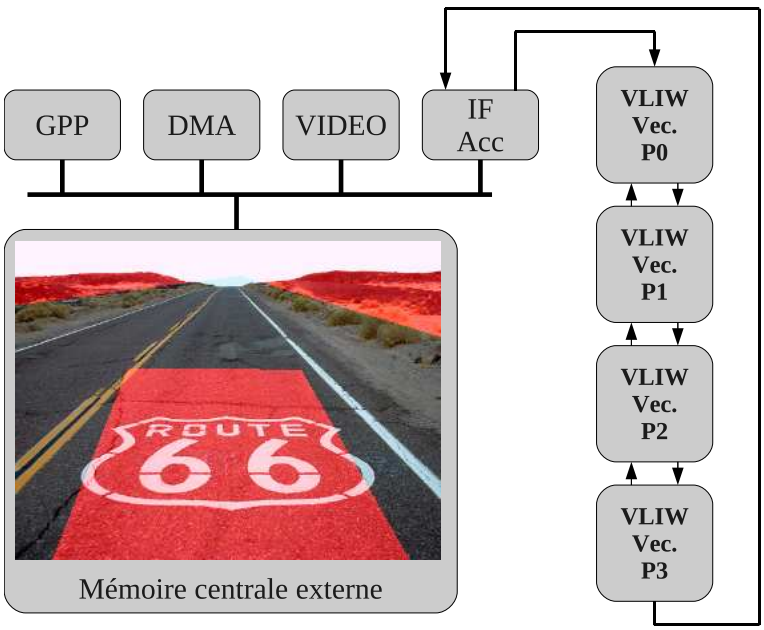


FIG. 5.27: Vue générale du SoC incorporant quatre processeurs vectoriels

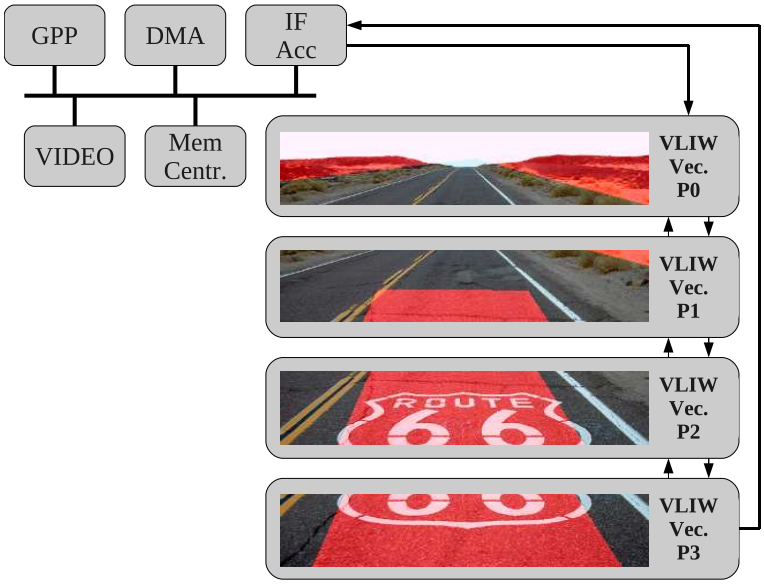


FIG. 5.28: Distribution recouvrante des données entre les différents cœurs vectoriels du SoC pour la mise en œuvre de la LPE par SKIZ isotrope

Machine	Temps	CPP	GOPS [a]	GOPS [a]/ Watt
SPoC [b]	21.7ms	1059.5	28.1	4.02
Multicoeurs VLIW [b]	6.7ms	325.6	91.1	9.11
GPGPU NVidia GTX260	1.3ms	394.5	469.8	3.14
Intel E5530 - SKIZ [b]	82.1ms	48046.8	7.43	0.09
Intel E5530 - FAH [c]	8.7ms	5094.6	70.2	0.87

[a]Nombre de giga-opérations par seconde
[b]Mise en œuvre de la LPE via des SKIZ
[c]Mise en œuvre de la LPE via des files d'attentes hiérarchiques

FIG. 5.29: Comparaison des temps de calcul et du nombre de cycle par pixel pour produire la LPE utilisé dans l'application de segmentation de la route

On remarque à première vue les très bonnes performances des GPGPU mais il faut garder à l'esprit qu'ils consomment 150W à 250W rendant leur densité de performance assez mauvaise.

Les processeurs généralistes ont des performances moyennes pour une consommation de 80W à 150W. On observe que les files d'attentes hiérarchiques sont encore efficaces de nos jours, mais l'orientation des fabricants d'augmenter le nombre des cœurs devrait, dans les années à venir, changer cet état de fait et rendre une mise en œuvre de la LPE par des SKIZ de plus en plus performante.

Les processeurs de voisinage sont limités par le pipeline temporel, mais traitent l'image en temps réel pour une consommation inférieure à 7W dans un FPGA. Avec une image distribuée dans tous les cœurs, les processeurs vectoriels sont très performants, consomment moins de 10W dans un FPGA et présentent un nombre de cycles par pixel inférieur à celui du GPGPU.

Au travers de cette exemple très dimensionnant, nous avons démontré une nouvelle fois que nos architectures sont capables de réaliser des traitements intensifs dans un cadre embarqué, alliant performances, souplesses et une consommation énergétiques raisonnable.

Chapitre 6

Conclusion et perspectives

6.1 Conclusion générale

Nous avons présenté, dans ce mémoire, plusieurs architectures dont le but premier est de réaliser efficacement des opérations de morphologie mathématique. Nous avons choisi de poursuivre une voie différente de celle empruntée par les algorithmiciens recherchant l'économie de calcul. Nous avons alors développé, sur des machines parallèles, des mises en œuvre d'opérations basiques afin de disposer d'un fort degré d'optimisation. Ces opérations peuvent ensuite être itérées plusieurs milliers de fois afin de déployer des applications complexes tout en conservant les aspects temps réel associés. L'objectif est double, car nous avons voulu dans un premier temps, conserver la performance des processeurs de voisinage flot de données et dans un second temps disposer d'architectures souples pouvant mettre en œuvre des algorithmes variés.

6.1.1 Première partie : processeurs de voisinage flots de données

Nous avons décrit précisément, dans ce chapitre, les différents noyaux de calcul utilisables avec des processeurs de voisinage. Cette description est certes matérielle, mais peut donner lieu à des mises en œuvre logicielles. Nous pouvons citer, par exemple, les moyens optimaux de trier un petit nombre fixe de valeurs connues à priori. L'étude des phénomènes de bord, pour chaque opération, est assez complète et permet de mettre en évidence le problème afin d'éviter de propager des erreurs lorsque des opérations morphologiques simples sont itérées dans des applications plus complexes.

Une nouvelle structure d'extraction du voisinage a été proposée et permet de paralléliser finement les calculs sans consommer de mémoire supplémentaire. L'accélération des calculs est donc directement proportionnelle au nombre de pixels que l'on peut transmettre en parallèle à un processeur de voisinage.

Une évolution des architectures récursives dédiées au traitement morphologique avec des éléments structurants de type segment a été proposée. Celle-ci permet un gain substantiel de mémoire synthétisée sur le circuit et cette évolution peut tout aussi bien être mise en pratique dans le cadre d'un déploiement logiciel afin de minimiser la consommation mémoire.

Nous nous sommes attachés, dans ce chapitre, à réhabiliter les processeurs de voisinage en dépoussiérant leur structure et en les dotant des nouvelles avancées en matière de morphologie mathématique. Ce type de processeur représente assez bien le couteau suisse du morphologue dans la mesure où il permet l'exécution de tous les calculs de base indispensable à la réalisation d'applications complexes.

6.1.2 Deuxième partie : chaînage de processeurs de voisinage

Les processeurs de voisinage, malgré leur performance, sont parfois trop rigides pour être facilement employés dans un grand nombre d'applications. Il est cependant très simple, de par leur nature flot de données, de les raccorder en pipeline afin de construire des systèmes plus complexes.

Nous avons détaillé dans ce chapitre l'intérêt d'exploiter un pipeline, non pas comme une ressource matérielle directement déployée en sortie de capteur, mais plutôt comme un coprocesseur flot de données de traitement d'images.

Nous avons donc cherché à montrer à quel point la morphologie mathématique se mariait bien avec les processeurs de voisinage. En effet, les outils proposés de description de pipeline permettent très simplement de décrire une application assez dynamique tout en disposant d'une forte puissance de calcul. Celle-ci pouvant facilement dépasser la centaine de milliards d'opérations par seconde au sein d'un FPGA.

Plusieurs directions ont été présentées afin de s'approcher de notre double objectif : performance et généricité. Nous avons alors détaillé une structure de pipeline appelée SPoC présentée à l'annexe A et déployée dans le cadre du projet PICS. Cette structure met en œuvre un pipeline à deux entrées composées de processeurs de voisinage et avec la possibilité de spécifier à l'exécution, les différentes opérations et chemins de données de chaque étage. Cependant, un pipeline statique ne peut convenir à toutes les applications et nous avons décidé de proposer un outil capable de générer et fusionner simplement des descriptions de haut niveau de pipeline afin de générer automatiquement des architectures reconfigurables à gros grains.

Les technologies de reconfiguration permettent aujourd'hui d'utiliser au mieux toute la puissance de calcul délivrée par les processeurs de voisinage. Toutefois, il est important, dans le cas d'applications très dynamiques, de mettre en place des outils logiciels capables de programmer efficacement des pipelines profonds. Une piste explorée par le projet de recherche FREIA explore, entre autres pistes, l'optimisation de la gestion logicielle du pipeline afin de maximiser son utilisation en fonction de sa topologie et des calculs à réaliser dans l'application.

6.1.3 Troisième partie : processeurs VLIW Vectoriels

Ce chapitre propose une remise à plat des processeurs flot de données en utilisant des processeurs ayant la capacité d'embarquer du microcode à la manière d'un processeur RISC afin de laisser une liberté totale quant aux opérations réalisables sur le ou les flux pixels. Une telle évolution seule présenterait certes une forte généricité, mais réduirait fortement la puissance de calcul. C'est la raison pour laquelle nous avons ajouté à ces processeurs des

unités de calcul vectoriel larges et optimisées pour l'extraction du voisinage en proposant des méthodes d'accès aux voisins par des vecteurs de pixels.

Nous avons ensuite présenté une structure multicœurs basée sur notre description de processeur afin d'étudier, grâce à quelques présentations de portage d'opérateurs, la pertinence de notre solution à la fois en termes de domaines d'applications réalisables, mais également en termes d'efficacité de calcul.

La topologie des interconnexions employées n'est pas figée et nous sommes capables d'adresser plusieurs niveaux de granularité dans les calculs. Il est aisé d'exploiter simultanément plusieurs granularités de parallélisme d'une part, en découpant une application ou un algorithme sur plusieurs cœurs VLIW et d'autre part, en considérant le traitement de chaque cœur via des opérations vectorielles.

6.1.4 Quatrième partie : comparaisons des différentes architectures

Le comparatif présenté dans ce chapitre est assez crucial, car il permet d'établir des indicateurs de performance en nombre de cycles par pixel par rapport à des processeurs multicœurs généralistes et des GPU, ces architectures étant maintenant couramment employées dans le domaine du calcul scientifique. Nous avons donc pu, au travers de tests simples, mais bien définis, rechercher les points forts et les limitations de toutes ces machines.

Ce comparatif nous a ainsi permis de mettre en évidence l'intérêt de l'approche SPoC qui permet de disposer d'un pipeline aux performances prédictibles et efficaces dès que des fusions d'opérations s'avèrent nécessaires. Notre circuit multicœur vectoriel n'est pas en reste et présente des performances meilleures que SPoC tout en proposant un généricité accru des opérations réalisables. Nous avons donc atteint notre objectif d'allier performance et généricité et les machines multicœurs VLIW vectorielles sont un bon moyen d'adresser les problématiques de morphologie mathématique et plus généralement de traitement d'images.

6.2 Perspectives

Les techniques de description et de fusion des pipelines, proposés par l'outil développé au chapitre 3, peuvent être considérés comme une première étape vers une fusion complètement automatisée. Une perspective intéressante serait maintenant de mettre en place une fusion de *patterns* à plusieurs niveaux de granularité. En effet, nous nous sommes principalement occupés de l'optimisation à gros grains avec la mutualisation de composants identiques dans un pipeline. Il serait également possible, à un grain beaucoup plus fin, de chercher à mutualiser des composants dont les unités de calculs sont matériellement proches comme un processeur calculant une érosion et un autre un gradient. Ceci toujours dans l'optique de créer un pipeline réalisant toutes les fonctions des opérateurs candidats au regroupement en fusionnant les unités de calculs internes.

Nous pourrions également explorer l'utilisation de cet outil de description de pipeline pour, non plus générer du matériel, mais du microcode embarqué dans les cœurs vectoriels. Se pose alors la problématique de fusion des nids de boucles dans la mesure ou un cœur

est capable d'émuler plusieurs étages d'un pipeline de processeurs de voisinage.

Une piste également très prometteuse serait de faire évoluer le mécanisme de connexion entre les cœurs vectoriels en employant un réseau sur puce de type grille bidimensionnelle. Une telle structure nous permettrait à la fois d'envisager un parallélisme massif, où chaque processeur vectoriel aurait en charge une sous-image, mais aussi un mode où plusieurs pipelines pourraient cohabiter pour toujours améliorer le séquencement efficace d'une application. L'exploitation du parallélisme massif et spatial serait un bon moyen de calculer la ligne de partage des eaux, *LPE*. Cette opération étant très séquentielle, la parallélisation la plus efficace sur nos machines consisterait à accélérer chacune des étapes du squelette par zone d'influence dans le but de réaliser une LPE niveau par niveau. Il est aujourd'hui possible de déployer quatre à huit de nos cœurs vectoriels dans un FPGA, chacun travaillant en parallèle sur plusieurs dizaines de pixels. L'image à traiter peut être répartie dans les différents cœurs et les traitements ne nécessitent donc aucune communication externe à la grille. Ainsi, nous estimons être capables de réaliser plusieurs milliers d'itérations de la ligne de partage des eaux au sein d'un FPGA en seulement quelques millisecondes.

Annexe A

Plateforme PICS

A.1 Description de la plateforme

La plateforme que nous décrivons ici a été réalisée dans le cadre du projet PICS, *Programmable imaging with CMOS sensors* [23], qui avait pour but de promouvoir dans le monde de la télédiffusion la technologie des imageurs CMOS ainsi que cette même technologie dans le cadre des caméras intelligentes pour des applications automobiles et de surveillance.

Nous proposons ici un circuit adapté au traitement d'images en provenance d'un imageur CMOS et qui a pour objectif de pouvoir réaliser des applications impliquant la morphologie mathématique dans le contexte des systèmes embarqués pour la vision. Ce système est capable, à une cadence de 30 images par secondes, d'acquérir des images, de les traiter et de retransmettre soit l'image traitée soit une information de toute autre nature. Une vue fonctionnelle du système est proposée en figure A.1.

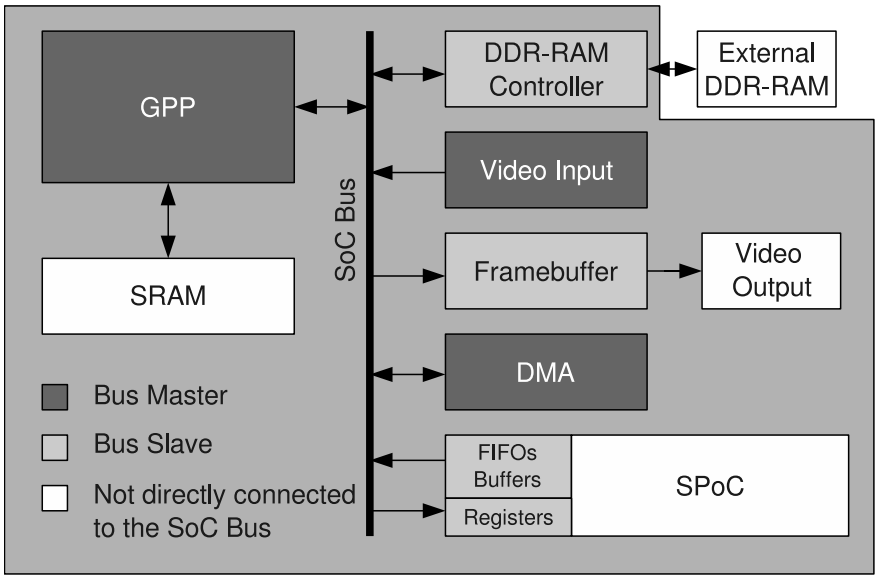


FIG. A.1: un système sur puce dédié au traitement d'images par morphologie mathématique

A.1.1 Structure générale du SoC

Le circuit est composé d'un processeur généraliste (GPP) qui est chargé de contrôler l'ordonnancement de l'application et des communications depuis et vers l'extérieur. Si cela s'avère nécessaire, le GPP peut accéder aux pixels d'une image stockée en DDR-RAM. Ce moyen d'accès est assez coûteux à cause de la latence de la mémoire et doit donc être évité autant que possible. Le processeur généraliste est un processeur standard 32 bits RISC fonctionnant à une fréquence basse (100 MHz) et peut exécuter un mini système d'exploitation en charge des communications réseau. Les opérations sur les images sont déportées dans un pipeline d'opérateurs flots de données appelé SPoC (Several neighborhood Processors On Chip). Afin d'alimenter cet accélérateur en données, des files d'attente de synchronisation et un DMA sont nécessaires afin d'accélérer les temps de traitement en minimisant les temps morts dans le pipeline.

Un composant d'acquisition vidéo a été ajouté au circuit et permet de recevoir un flux vidéo au format PAL et NTSC préalablement numérisé avec un convertisseur analogique numérique externe. La logique d'acquisition vidéo dans le SoC est donc en charge de décoder les trames et de prendre le contrôle du bus principal du circuit, à la manière d'un DMA, pour stocker les images directement dans la mémoire DDR. Une partie optionnelle du circuit est la mémoire vidéo permettant d'afficher une image sur un écran afin d'observer en temps réel les résultats d'une application ou toutes autres informations.

SPoC est optimisé pour minimiser les transferts mémoires entre les différentes étapes d'une application. La structure en pipeline à plusieurs entrées permet, d'une part de mettre en place des opérateurs de morphologie mathématique avec seulement une lecture/écriture d'une image, et d'autre part d'exploiter le parallélisme temporel et spatial pour accélérer les traitements.

Le paramétrage des étages de SPoC est fait en spécifiant une liste d'opérations basiques (érosion, dilatation, soustraction, ...) devant être exécutées à chaque étage ainsi que les différents chemins de données à mettre en place. Tous ces paramètres doivent être stockés dans les registres de configuration avant d'envoyer un flot d'images. Les effets de bord sont automatiquement gérés par l'accélérateur et la seule tâche du GPP est de programmer les registres de SPoC et d'ordonnancer, via le DMA, l'envoi et la réception des données.

Une fois le pipeline configuré, les pixels des images à traiter sont dirigés, via le bus principal du SoC, vers les files d'attente d'entrées du pipeline afin que les calculs puissent commencer. Si deux images sont envoyées vers l'accélérateur, les calculs pourront commencer uniquement lorsque toutes les files d'attente d'entrées contiendront des données. Après une latence correspondant à une ligne par étage dans le pipeline, les pixels résultats apparaissent et sont stockés dans des files d'attente de sorties à chaque cycle. Le GPP programme alors les DMA pour envoyer et stocker progressivement les résultats depuis et vers la mémoire principale.

A.1.2 Le pipeline SPoC

SPoC est composé d'une part, de plusieurs processeurs de voisinage, appelés PoC, associés en parallèle et en série à travers un chemin de données reconfigurable et d'autre part, d'unités arithmétiques et logiques (ALU) programmables. La structure interne de SPoC est proposée en figure A.2. Un PoC travaille avec des noyaux 3×3 et est capable de réaliser

une érosion, une dilatation, un gradient, un médian ou bien encore une convolution, ces deux dernières opérations étant optionnelles, car assez coûteuses en surface.

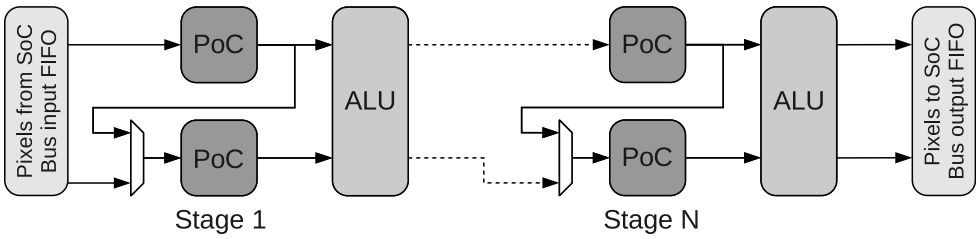


FIG. A.2: Vue interne du pipeline SPoC

La structure en pipeline est intéressante pour la morphologie mathématique, et ce dernier est souvent bien exploité à chacune des étapes d'une application. En outre, SPoC est une architecture MIMD et peut réaliser deux opérations en parallèle sur un même flux ou réaliser deux opérations sur des flux différents. Les multiplexeurs dans le pipeline permettent d'utiliser tous les PoC du circuit en série afin de doubler la profondeur du pipeline lorsqu'un seul flux image est utilisé. Une telle possibilité est adaptée a la mise en place de filtres alternés séquentiels qui consomment une grande quantité d'opérateurs de type érosion/dilatation.

L'ALU, figure A.3, collecte les flux en provenance des PoC disposés en parallèle dans un même étage et configure, avant que les calculs ne démarrent, les différents chemins de données afin de sélectionner les flux devant ressortir de cette dernière. Elle embarque également des registres permettant de mixer les calculs entre un flux et différentes constantes.

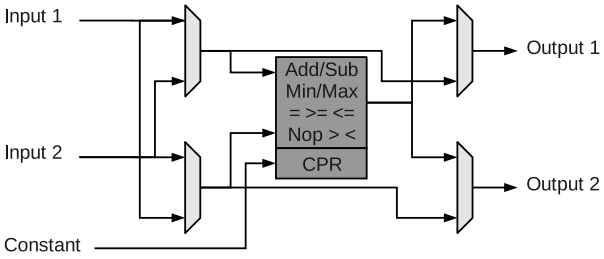


FIG. A.3: Vue interne de l'ALU entre les processeurs de voisinage

L'ALU permet de calculer, entre les flux ainsi qu'avec des constantes, les opérations suivantes : addition, soustraction, addition saturée, soustraction saturée, valeur absolue de la différence, infimum, supremum. Des unités de comparaison ont été introduites afin d'appliquer des seuils sur les flux ou bien encore des recopies conditionnelles. Par exemple, il est possible de comparer les pixels deux à deux de deux flux d'images et de remplacer un pixel, lorsque la comparaison est vraie, par la valeur définie dans un registre de constante.

Les processeurs de voisinage PoC mettent en œuvre une extraction du voisinage parallélisée, présentée en figure A.4 afin d'augmenter la puissance de calcul tout en diminuant la latence des traitements. A chaque cycle, les PoC extraient N voisinages successifs lorsque les données lui sont transmises par paquet de N pixels contigus. La figure A.5 montre une vue simplifiée de l'architecture d'extraction parallèle du voisinage pour des vecteurs de

quatre pixels dans le cadre du traitement d'une image 13×6. Les registres RA1 à RF2 représentent les registres de voisinages et à chaque cycle, quatre pixels sont envoyés à l'entrée du processeur. Chaque PoC est composé de deux lignes à retard, PDL1 et PDL2, pour être capable d'extraire correctement le voisinage (figure A.4). Ces dernières doivent être paramétrées avant les calculs avec la largeur de l'image à traiter. La taille de ces lignes à retard est la même dans le cas de l'extraction parallélisée que dans le cas non parallélisé, la seule différence réside dans la géométrie de ces mémoires puisqu'elles doivent absorber quatre pixels par cycle (dans notre exemple).

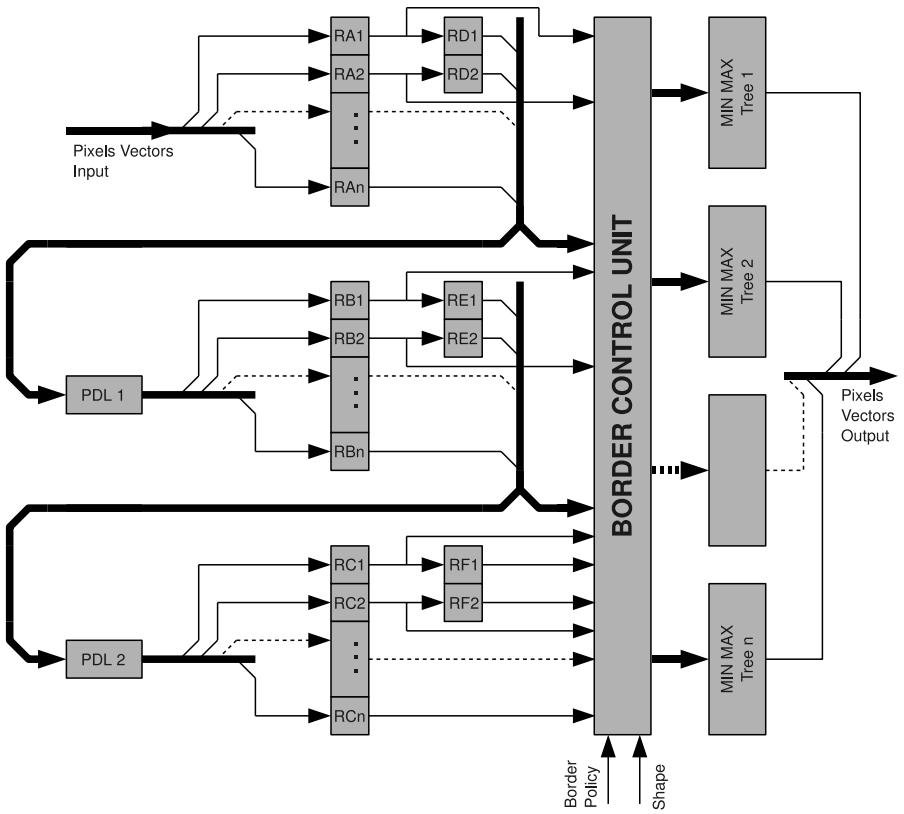


FIG. A.4: Vue fonctionnelle d'un processeur de voisinage parallélisé PoC

Les centres des voisinages sont représentés par les registres {RB2, ..., RBn,RE1} et dans notre exemple, le dernier élément structurant extrait, en considérant un balayage vidéo standard des pixels, est composé des registres {RAn,RBn,RCn,RD1,RE1,RF1,RD2,RE2,RF2} où n représente la taille du vecteur de pixels. Le recouvrement entre le dernier voisinage au cycle k et le premier au cycle $k+1$ est par exemple garanti par les chemins de données entre les registres {RA1,RA2} et {RD1,RD2}.

Certains pixels extraits peuvent être invalidés grâce à l'unité de gestion des bords en fonction de la position de chacun des voisinages dans l'image.

L'unité de gestion des bords est construite en trois étages, le premier est un compteur des lignes et colonnes, le second est dédié à la mise en place de comparateurs pour valider ou non les pixels dans le troisième étage.

L'invalidation des registres de voisinage dépend des calculs qui doivent être réalisés

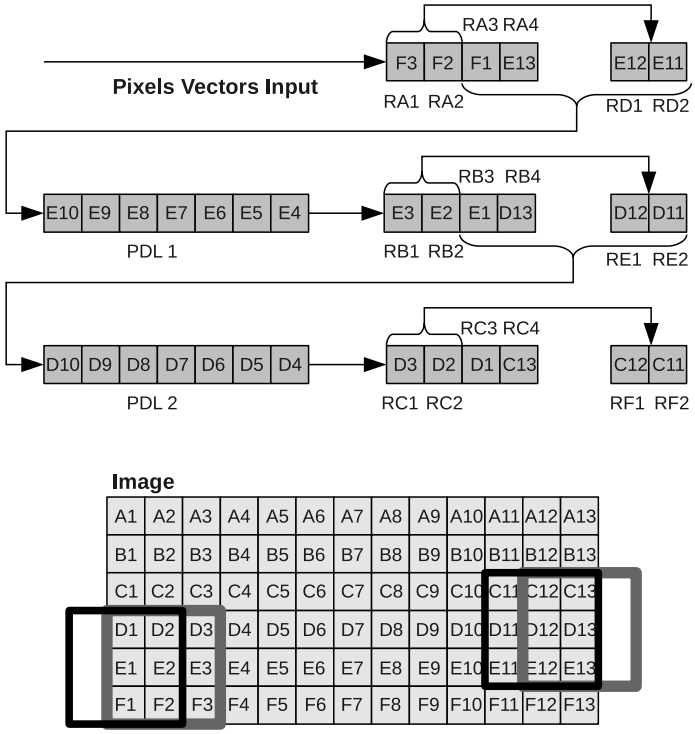


FIG. A.5: Vue simplifiée du fonctionnement d'un processeur de voisinage PoC parallélisé avec un degré quatre pixels

sur l'élément structurant défini par l'utilisateur (carré, croix, hexagone...). Par exemple, dans le cadre d'une érosion, les voisins à invalider doivent être remplacés par la valeur maximale autorisée par le format de codage des pixels. Au contraire, pour une dilatation, les pixels invalidés doivent être remplacés par la valeur minimale autorisée par le format de codage des pixels. Dans le cadre du filtre médian, la politique choisie est de remplacer les voisins invalidés comme si l'image était entourée par un damier. La figure A.6 montre le bon fonctionnement du tri avec un tel damier comme politique de gestion des bords.

Une fois les bords correctement gérés, les pixels sont dirigés vers les arbres de calcul du minimum ou du maximum de chaque voisinage. Ces arbres de calcul, figure A.4, peuvent être remplacés avant synthèse par un arbre de multiplieur et accumulateur pour réaliser des convolutions ou par un tri à bulle complètement déployé matériellement afin de calculer tous les filtres de rang (l'érosion, la dilatation et le médian sont des cas particuliers de filtre de rang). Pour des contraintes de surfaces, nous avons décidé de synthétiser uniquement le premier étage du pipeline avec les unités de tri, car il n'est pas possible de chaîner plusieurs filtres de rang pour en composer un à support plus important.

A.1.3 Modèle de programmation

L'accélérateur SPoC ne peut être directement raccordé à l'unité d'acquisition vidéo, car les différentes opérations d'une application ne peuvent pas dans tous les cas tenir dans un seul pipeline et il est parfois nécessaire de réaliser plusieurs passes avant qu'une opération soit valide. C'est la raison pour laquelle le pipeline est raccordé au bus du SoC pour avoir

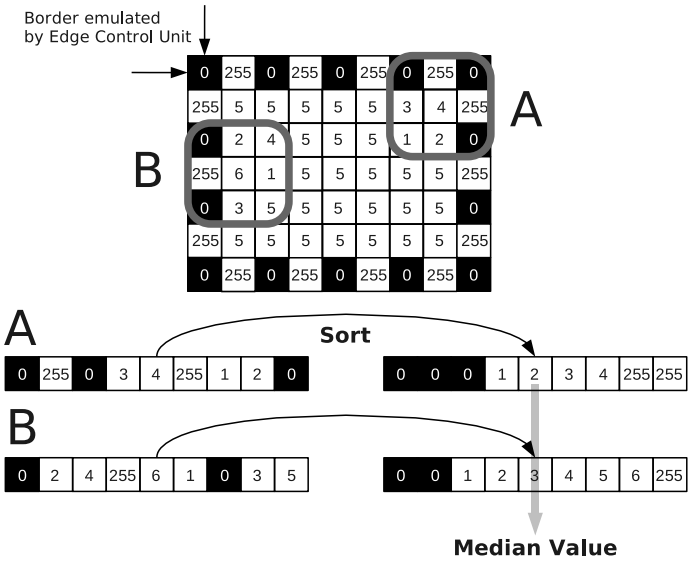


FIG. A.6: Émulation d'un damier au bord de l'image dans le cadre du calcul du filtre médian

la possibilité de traiter une image en plusieurs passes dans le pipeline. Le GPP ne doit pas alimenter le pipeline, car il est trop lent dans les accès mémoire. Le processeur doit juste configurer le DMA pour lire et écrire les différentes files d'attente d'entrées et de sortie du pipeline. Les étapes permettant la gestion du pipeline sont présentées en figure A.7.

A.2 Applications et benchmarks

Deux exemples sont présentés pour tester les performances de la plateforme. La première est une application de vidéo surveillance et la seconde est une application de localisation de plaque d'immatriculation. Les plateformes utilisées pour réaliser les comparatifs de performances sont les suivantes :

- Intel Pentium IV Xeon "Irwindale" à une fréquence de 3 GHz avec 1Mo de cache L2 et utilisant les instructions SSE2
- Trimedia DSP à 320 MHz
- Storm-1 SP16HP-G220 à 700 Mhz [39]

Un étage du pipeline SPoC utilise approximativement 60k portes ASIC sans prendre en compte les mémoires de ligne. Quatre mémoires double port sont nécessaires dans un étage avec pour chacune une taille égale à la taille d'une ligne de l'image. En général, pour des pixels 8 bits et des images de 512 pixels un étage du pipeline requiert 2 ko.

Notre système sur puce est routé avec un pipeline à 8 étages et fonctionne à 100 MHz dans un FPGA Virtex-4 LX 60. Il occupe environ 21000 slices et utilise 64 ko de BRAM, 32 pour le GPP et 32 pour l'accélérateur qui est capable de traiter des images 1024×1024 en 256 niveaux de gris. Le pipeline SPoC peut toutefois être routé à 300 MHz dans un Virtex-4 et 400 MHz dans un Virtex-5.

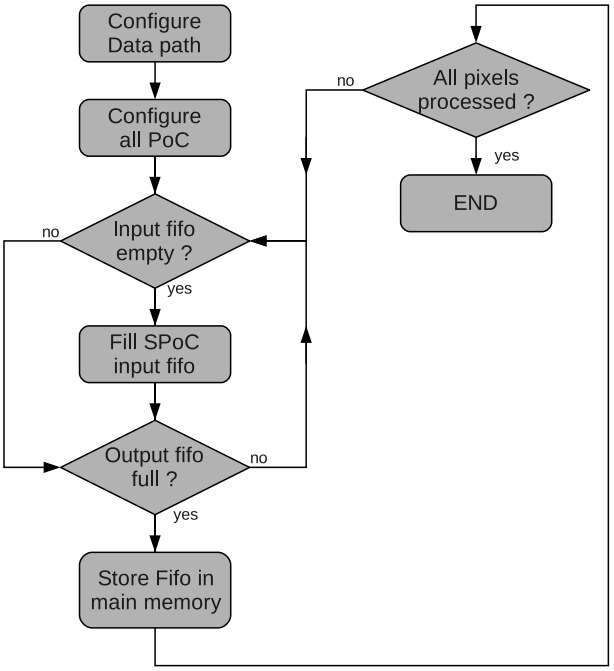


FIG. A.7: Exemple de gestion du pipeline SPoC par un processeur généraliste

A.2.1 Puissance de calcul brute

Nous considérons une image 512 × 512 dont les pixels sont codés sur 8 bits et les opérations mises en place pour les tests sont des érosions avec des éléments structurants dont la taille est croissante.

Arch. / Op.	P4	Trimedia	Storm-1	Notre SoC
érosion 1	0,4	8,2	0,06	0,65
érosion 3	1,3	24	0,18	0,65
érosion 10	4,2	82	0,60	0,65
érosion 16	6,6	131	0,96	0,65
érosion 17	7	140	1,03	1,31

FIG. A.8: Temps de calcul, en millisecondes, de différentes tailles d'érosion pour une image 512 × 512

Le tableau A.8 présente les résultats obtenus pour des érosions dont l'élément structurant à un rayon compris entre 1 et 17 soit une taille variant de 3 × 3 à 35 × 35 en 8-connexité. La 6-connexité a également été testée et les résultats sont assez équivalents à ceux présentés ici. Des érosions dont le rayon est unitaire sont itérées pour obtenir les différentes tailles proposées dans ce test (propriété de composition de l'érosion). Nous observons un temps de calcul croissant pour les différents processeurs, alors que notre SoC, grâce au pipeline SPoC, révèle un temps de traitement constant lorsqu'une seule passe de calcul est nécessaire. En effet, SPoC peut chaîner jusqu'à seize érosions dans le même flot, mais lorsqu'une dix-septième est requise, il est obligatoire de relancer le flux dans le

pipeline une seconde fois ce qui induit un temps de traitement doublé entre une érosion de taille 16 et une érosion de taille 17. On en déduit alors que le principal défi d'une telle architecture est de remplir au mieux le pipeline avec les différentes étapes d'une application afin de maximiser la quantité des opérations réalisées dans le pipeline.

A.2.2 Application de détection de mouvements

La méthode employée pour extraire les zones en mouvement d'une image, prise à partir d'une position fixe, n'est pas basée sur une estimation du flot optique [2] et aucun à priori sur la vidéo n'est pris en compte. L'algorithme employé ici [13] est basé sur le calcul de gradients morphologiques temporels et spatiaux sur le flux vidéo. L'équation ci-dessous permet de faire ressortir les zones en mouvement en calculant les gradients temporels et spatiaux du flux vidéo. La figure A.10 présente un exemple de résultat.

$$mcm^t = \inf(\left|g^{t+1} - g^t\right|, \left|g^t - g^{t-1}\right|)$$

L'avantage principal de cette méthode réside dans la seule nécessité de mémoriser trois images successives du flux vidéo pour en extraire les zones en mouvement. Si des objets à détecter bougent trop lentement vis-à-vis de la fréquence d'acquisition des images, les gradients peuvent non plus être calculée aux temps t^{-1}, t, t^{+1}, mais aux temps t^{-n}, t, t^{+n}.

Une étape de prétraitement est nécessaire avant de lancer la détection de mouvement afin de filtrer l'image pour supprimer le bruit engendré par le capteur CMOS. Ce filtrage est réalisé par un filtre alterné séquentiel (Asf), qui consiste à réaliser une succession d'ouvertures et de fermetures de tailles croissantes [71] sur le flux vidéo.

	P4	**Trimedia**	**Storm-1**	**SPoC**
Temps (ms) mcm	0,7	3,6	0,08	1,14
Temps (ms) Asf	13,2	121	0,8	1,52
Temps (ms) total	13,9	124,6	0,88	2,66
Nombre total de cycles par pixel	543	516	8	3,5

FIG. A.9: Performances de l'application de détection de mouvement

Le tableau A.9 présente les temps de traitement de chaque étape de l'application avec les différentes architectures sélectionnées pour un flux image 320 × 240 en 8 bits. Notre SoC réalise l'application quatre fois plus vite que le Pentium 4 mais 3 fois moins vite que le processeur Storm-1 qui, rappelons-le, fonctionne à une fréquence sept fois supérieure. Si l'on se ramène au nombre de pixels résultats par cycle, notre circuit réalise l'application de manière beaucoup plus efficace que les autres cibles des tests.

A.2.3 Localisation de plaques d'immatriculation

La localisation de plaques d'immatriculation dans une image est basée sur la propriété d'adjacence des caractères alphanumériques la constituant. Une méthode similaire est décrite en [3]. L'algorithme utilise principalement des transformations chapeau haut-de-forme (CHdF) avec des éléments structurants dont la taille correspond à la largeur des caractères

(a) (b)

FIG. A.10: Exemple de détection de mouvement

de la plaque. L'opérateur CHdF peut s'écrire comme étant la soustraction de f avec l'ouverture γ, on parle alors de chapeau haut-de-forme par ouverture. On peut également l'écrire comme la soustraction de la fermeture φ avec l'image initiale f, on parle alors de chapeau haut-de-forme par fermeture. Les formules ci-dessous montrent ces différentes définitions :

$$\gamma(f) = \delta(\varepsilon(f))$$
$$\varphi(f) = \varepsilon(\delta(f))$$
$$CHdF_{\gamma}(f) = f - \gamma(f)$$
$$CHdF_{\varphi}(f) = \varphi(f) - f$$

Une opération de post-traitement est nécessaire afin de supprimer les faux positifs en utilisant des ouvertures horizontales et verticales avec des éléments structurants dépendants de la taille approximative de la plaque recherchée.

Les équations de l'application sont données ci-après et la figure A.11 présente un exemple de détection de plaque.

$$g(f) = \inf(f - \gamma(f), \varphi(f) - f)$$
$$plate(f) = \gamma(threshold(g(f), a))$$

Les performances des différents systèmes sélectionnés, donnés au tableau A.12, sont assez proches de ceux obtenus avec l'application de détection de mouvement.

A.3 Conclusion

Le circuit décrit dans cette annexe est un bon exemple d'architecture réalisable à partir des processeurs de voisinage et permet une mise en œuvre assez efficace des applications proposées. Ces applications sont principalement développées pour montrer les capacités de

(a) (b)

FIG. A.11: Localisation des plaques d'immatriculation

	P4	Trimedia	Storm-1	SPoC
Temps (ms)	11,91	104	0,65	1,95
Nb cycle/pixels	465	433	6	2,5

FIG. A.12: Temps de localisation des plaques d'immatriculation sur différentes machines (en ms)

la plateforme dans des cas concrets sans toutefois être totalement industrialisables dans l'état.

L'accélérateur SPoC complet développe sur un Virtex-4 une puissance de calcul de 54,4 GOPS à 100 MHz dans le cas de traitement d'images 8 bits. La puissance de calcul maximale atteinte pour un PoC est 3,2 GOPS, mais celle-ci peut être grandement améliorée en augmentant la taille des vecteurs de pixels ou en accroissant la fréquence. Toutefois, de tels changements ont une répercussion sur la bande passante mémoire ainsi que sur le bus alimentant le pipeline, il serait donc préférable de prévoir un système d'interconnexion capable de réaliser de gros transferts de données afin d'éviter les problèmes de saturations du bus général du SoC.

Bibliographie

[1] A.A. Abbo, R.P. Kleihorst, V. Choudhary, L. Sevat, P. Wielage, S. Mouy, B. Vermeulen, and M. Heijligers. Xetal-ii : A 107 gops, 600 mw massively parallel processor for video scene analysis. *Solid-State Circuits, IEEE Journal of*, 43(1) :192–201, Jan. 2008.

[2] Kelson R. T. Aires, Andre M. Santana, and Adelardo A. D. Medeiros. Optical flow using color information : preliminary results. In *SAC '08 : Proceedings of the 2008 ACM symposium on Applied computing*, pages 1607–1611, New York, NY, USA, 2008. ACM.

[3] Antonio Albiol, J. Manuel Mossi, Alberto Albiol, and Valery Naranjo. Automatic license plate reading using mathematical morphology. In *Proceedings of the The 4th IASTED International Conference on Visualization, Imaging and Image Processing*, Marbella, Spain, september 2004.

[4] Yannick Allusse, Patrick Horain, Ankit Agarwal, and Cindula Saipriyadarshan. Gpucv : an opensource gpu-accelerated framework forimage processing and computer vision. In *MM '08 : Proceeding of the 16th ACM international conference on Multimedia*, pages 1089–1092, New York, NY, USA, 2008. ACM.

[5] Gene Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. *AFIPS Conference Proceedings*, 30 :483–485, 1967.

[6] Iann M. Barron. The transputer. In *MiniMicro West*, volume 2, pages 1–8, San Francisco, CA, November 1983.

[7] S. Beucher. *Segmentation d'images et morphologie mathématique*. Thèse de doctorat en morphologie mathématique, ENSMP, 1990. 1822.

[8] S. Beucher, J-M. Blosseville, and F. Lenoir. Traffic spatial measurements using video image processing. In *Symposium on Optical and Optoelectronic Engineering*, page 8, Cambridge, Mass., USA, 1-6 Nov. 1987, 1987. 287.

[9] S. Beucher and C. Lantuéjoul. Use of watersheds in contour detection. Int. workshop on image processing, real-time edge and motion detection/estimation, rennes, septembre 1979, CMM/CG Fontainebleau, ENSMP, Novembre 1979.

[10] Serge Beucher. Algorithmes sans biais de la ligne de partage des eaux. Note interne, Centre de Morphologie Mathématique / ENSMP, Février 2002.

[11] Serge Beucher. Transformations résiduelles en morphologie numérique. Version longue du papier présenté à iss, le 5 février 2004, Centre de Morphologie Mathématique / ENSMP, mars 2004. voir http ://cmm.ensmp.fr/ beucher/publi.html.

[12] Serge Beucher. Numerical residues. *Image Vision Comput.*, 25(4) :405–415, 2007.

[13] L. Biancardini, E. Dokladalova, S. Beucher, and L. Letellier. From moving edges to moving regions. In *IbPRIA 2005, Iberian Conference on Pattern Recognition and Image Analysis*, page 10, Estoril, Portugal, June 7-9 2005, 2005.

[14] Philippe Bonnot, Fabrice Lemonnier, Gilbert Edelin, Gerard Gaillat, Olivier Ruch, and Pascal Gauget. Definition and simd implementation of a multi-processing architecture approach on fpga. In *DATE08*, 2008.

[15] Jaromir Brambor. *Algorithmes de la morphologie mathématique pour les architectures orientées flux*. Thèse de doctorat en morphologie mathématique, ENSMP, 2006. Dirigée par Michel Bilodeau.

[16] J. E. Bresenham. Algorithm for computer control of a digital plotter. pages 1–6, 1998.

[17] Philip Brisk, Adam Kaplan, and Majid Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *DAC '04 : Proceedings of the 41st annual conference on Design automation*, pages 395–400, New York, NY, USA, 2004. ACM.

[18] A. Broggi, G. Conte, F. Gregoretti, C. Sansoè, R. Passerone, and L. M. Reyneri. Design and implementation of the paprica parallel architecture. *J. VLSI Signal Process. Syst.*, 19(1) :5–18, 1998.

[19] A. Broggi, G. Conte, F. Gregoretti, C. Sansoè, and L. M. Reyneri. The evolution of the paprica system. *Integr. Comput.-Aided Eng.*, 4(2) :114–136, 1997.

[20] Michael Butts, Anthony Mark Jones, and Paul Wasson. A structural object programming model, architecture, chip and tools for reconfigurable computing. In *FCCM '07 : Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 55–64, Washington, DC, USA, 2007. IEEE Computer Society.

[21] E. Casseau, C. Jego, and E. Martin. Architectural synthesis of digital signal processing applications dedicated to submicron technologies. *Electronics, Circuits and Systems, 2001. ICECS 2001. The 8th IEEE International Conference on*, 1 :535–538 vol.1, 2001.

[22] A. E. Charlesworth. An approach to scientific array processing : The architectural design of the ap-120b/fps-164 family. *Computer*, 14(9) :18–27, 1981.

[23] Christophe Clienti, Serge Beucher, and Michel Bilodeau. A system on chip dedicated to pipeline neighborhood processing for mathematical morphology. In *EUSIPCO-2008, 16th European Signal Processing Conference*, August 2008.

[24] Christophe Clienti, Michel Bilodeau, and Serge Beucher. An efficient hardware architecture without line memories for morphological image processing. In *ACIVS '08 : Proceedings of the 10th International Conference on Advanced Concepts for Intelligent Vision Systems*, pages 147–156, Berlin, Heidelberg, 2008. Springer-Verlag.

[25] Katherine Compton and Scott Hauck. Reconfigurable computing : a survey of systems and software. *ACM Comput. Surv.*, 34(2) :171–210, 2002.

[26] J. Denoulet and A. Merigot. System on chip evolution of a simd architecture for image processing. In *Computer Architectures for Machine Perception, 2003 IEEE International Workshop on*, pages 9 pp.–298, May 2003.

[27] B. Ducourthial and A. Merigot. Parallel asynchronous computations for image analysis. 90(7) :1218–1229, July 2002.

[28] Raffi Enficiaud. *Algorithmes multidimensionnels et multispectraux en Morphologie Mathématique : approche par méta-programmation.* Thèse de doctorat en morphologie mathématique, ENSMP, 2007. Dirigée par Michel Bilodeau.

[29] M. Flynn. Very high speed computing systems. *Proceedings of the IEEE*, 54(12) :1901–1909, 1966.

[30] Antoine Fraboulet and Tanguy Risset. Master interface for on-chip hardware accelerator burst communications. *J. VLSI Signal Process. Syst.*, 49(1) :73–85, 2007.

[31] J. Gil and M. Werman. Computing 2-d min, median, and max filters. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(5) :504–507, 1993.

[32] Andrew S. Glassner. *Graphics Gems.* Academic Press, Inc., Orlando, FL, USA, 1990.

[33] Andrew S. Glassner, editor. *Graphics gems.* Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[34] Jeremiah Golston. Dm642 digital media processor. volume 5022, pages 700–706. SPIE, 2003.

[35] Christophe Gratin. Le logiciel micromorph. Transparents ecole d'eté, Centre de Morphologie Mathématique / ENSMP, 1991. 4330.

[36] Intel. Advanced Vector Extensions, Mar 2008.

[37] Ernest W. Kent, Michael O. Shneier, and Ronald Lumia. Pipe (pipelined image-processing engine). *Journal of Parallel and Distributed Computing*, 2(1) :50 – 78, 1985.

[38] B. Khailany, W.J. Dally, U.J. Kapasi, P. Mattson, J. Namkoong, J.D. Owens, B. Towles, A. Chang, and S. Rixner. Imagine : media processing with streams. *Micro, IEEE*, 21(2) :35–46, Mar/Apr 2001.

[39] B. Khailany, T. Williams, J. Lin, E. Long, M. Rygh, D. Tovey, and W.J. Daly. A programmable 512 gops stream processor for signal, image, and video processing. *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, pages 272–602, Feb. 2007.

[40] J-C. Klein and R. Peyrard. Pimm1, an image processing asic based on mathematical morphology. In *Second Annual IEEE ASIC Seminar and Exhibit*, pages 7.1.1–7.1.4, Rochester, September 25-28, 1989, 1989. 1263 CF L-32/90/MM.

[41] Jean-Claude Klein, Fabrice Lemonnier, Michel Gauthier, and René Peyrard. Hardware implementation of the watershed zone algorithm based on a hierarchical queue structure. In *Proceedings IEEE Workshop on Nonlinear Signal and Image Processing*, pages 859–862, Neos Marmaras, Halkidiki, Greece, June 20-22, 1995, 1995. 5228.

[42] Christoforos Kozyrakis. A media-enhanced vector architecture for embedded memory systems. Technical report, Berkeley, CA, USA, 1999.

[43] Christoforos Kozyrakis and David Patterson. Vector vs. superscalar and vliw architectures for embedded multimedia benchmarks. In *MICRO 35 : Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 283–293, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[44] S. Kyo, T. Koga, and S. Okazaki. Imap-ce : a 51.2 gops video rate image processor with 128 vliw processing elements. *Image Processing, 2001. Proceedings. 2001 International Conference on*, 3 :294–297 vol.3, 2001.

[45] M. Lam. Software pipelining : an effective scheduling technique for vliw machines. *SIGPLAN Not.*, 23(7) :318–328, 1988.

[46] C. Lantuéjoul and S. Beucher. Geodesic distance and image analysis. In *5th ICS*, volume 37 of *Mikroscopie*, pages 138–142, Salzburg, Autriche, Septembre 1979, 1980.

[47] Fabrice Lemonnier and Jean-Claude Klein. Fast dilation by large 1d structuring elements. In *IEEE Workshop on Nonlinear Signal and Image Processing*, pages 479–482, Neos Marmaras, Halkidiki, Greece, June 20-22, 1995, 1995. 5229.

[48] Romain Lerallut, Etienne Decencière, and Fernand Meyer. Image filtering using morphological amoebas. *Image Vision Comput.*, 25(4) :395–404, 2007.

[49] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla : A Unified Graphics and Computing Architecture. *IEEE Micro*, 28(2) :39–55, 2008.

[50] Robert M. Lougheed and David L. McCubbrey. The cytocomputer : A practical pipelined image processor. In *ISCA '80 : Proceedings of the 7th annual symposium on Computer Architecture*, pages 271–277, New York, NY, USA, 1980. ACM.

[51] Georges Matheron. Les nivellements. note manuscrite, CG/CMM Fontainebleau, ENSMP, Février 1997. 6004.

[52] David May and Roger Shepherd. Occam and the transputer. In *Proc. of the IFIP WG 10.3 workshop on Concurrent languages in distributed systems : hardware supported implementation*, pages 19–33, New York, NY, USA, 1985. Elsevier North-Holland, Inc.

[53] F. Meyer. Un algorithme optimal de ligne de partage des eaux. In *Actes 8ème Congrès AFCET Reconnaissance des Formes et Intelligence Artificielle*, pages 847–857, Lyon-Villeurbanne, 25-29 Novembre 1991, 1991. 2422 EX N-10/91/M.

[54] Fernand Meyer. From connected operators to levelings. In Henk J.A.M. Heijmans and Jos B.T.M. Roerdink, editors, *Mathematical Morphology and its Applications to Image and Signal Processing, Proc. ISMM'98*, pages 191–198, Amsterdam, June 1998, 1998. Dordrecht : Kluwer.

[55] Fernand Meyer and Jesus Angulo. Micro-viscous morphological operators. In Gerald Jean Francis Banon, Junior Barrera, Ulisses de Mendonça Braga-Neto, and Nina Sumiko Tomita Hirata, editors, *Proceedings*, volume 1, pages 165–176, São José dos Campos, October 10–13, 2007 2007. Universidade de São Paulo (USP), Instituto Nacional de Pesquisas Espaciais (INPE).

[56] Ethan Mollick. Establishing moore's law. *IEEE Ann. Hist. Comput.*, 28(3) :62–75, 2006.

[57] Fernando Moraes, Ney Calazans, Aline Mello, Leandro Möller, and Luciano Ost. Hermes : an infrastructure for low area overhead packet-switching networks on chip. *Integr. VLSI J.*, 38(1) :69–93, 2004.

[58] N. Moreano, E. Borin, Cid de Souza, and G. Araujo. Efficient datapath merging for partially reconfigurable architectures. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(7) :969–980, July 2005.

[59] Dominique Noguet. *Architectures parallèles pour la morphologie mathématique géodésique*. Thèse de doctorat en microélectronique, Institut Polytechnique de Grenoble, INPG, 1998.

[60] OpenVidia. Cuda Vision Workbench, Feb 2009.

[61] A W Paeth. A fast algorithm for general raster rotation. In *Proceedings on Graphics Interface '86/Vision Interface '86*, pages 77–81, Toronto, Ont., Canada, Canada, 1986. Canadian Information Processing Society.

[62] Alan W. Paeth. Median finding on a 3 x 3 grid. pages 171–175, 1990.

[63] A. Rosenfeld and J.L. Pfaltz. Distance functions on digital pictures. *Pattern Recognition*, 1 :33–61, 1968.

[64] Richard M. Russell. The cray-1 computer system. *Commun. ACM*, 21(1) :63–72, 1978.

[65] S. Kyo S. Okazaki and F. Hidano. Imapcar : A highly parallel integrated memory array processor for in-vehicle image recognition applications. *Proc. ITS World Congress*, pages ID–1744, 2006.

[66] Lorenz A. Schmitt and Stephen S. Wilson. The ais-5000 parallel processor. *IEEE Trans. Pattern Anal. Mach. Intell.*, 10(3) :320–330, 1988.

[67] Robert Schöne, Wolfgang E. Nagel, and Stefan Pflüger. Analyzing cache bandwidth on the intel core 2 architecture. In Christian H. Bischof, H. Martin Bücker, Paul Gibbon, Gerhard R. Joubert, Thomas Lippert, Bernd Mohr, and Frans J. Peters, editors, *PARCO*, volume 15 of *Advances in Parallel Computing*, pages 365–372. IOS Press, 2007.

[68] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee : a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3) :1–15, 2008.

[69] J. Serra. *Image analysis and mathematical morphology*. Academic Press, London, 1982.

[70] J.L. Smith. Implementing median filters in xc4000e fpgas. *Xilinx Application Notes, 44*, 1996.

[71] Pierre Soille. *Morphological Image Analysis : Principles and Applications*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[72] Charles V. Stewart and Charles R. Dyer. Scheduling algorithms for pipe (pipelined image-processing engine). *Journal of Parallel and Distributed Computing*, 5(2) :131 – 153, 1988.

[73] Paola Sunna. Avc/h.264, un système de codage vidéo évolué pour la hd et la sd. Technical report, RAI / CRIT / UER, 2005.

[74] K. Tatas, K. Siozios, and D. Soudris. *A Survey of Existing Fine-Grain Reconfigurable Architectures and CAD tools*. Springer Netherlands, 2007.

[75] Jay Trodden, Don Anderson, and MindShare Inc. *HyperTransport System Architecture*. Addison-Wesley Professional, Boston, MA, USA, 2003.

[76] H. Ueda, K. Kato, H. Matsushima, K. Kaneko, and M. Ejiri. A multiprocessor system utilizing enhanced dsps for image processing. In *Systolic Arrays, 1988., Proceedings of the International Conference on*, pages 611–620, May 1988.

[77] R. van den Boomgaard and D. A. Wester. Logarithmic shape decomposition. In C. Arcelli, L. P. Cordella, and G. Sanniti di Baja, editors, *Aspects of Visual Form Processing*, pages 552–561, Capri, Italy, May 1994. World Scientific Publishing Co., Singapore.

[78] Rein van den Boomgaard and Richard van Balen. Methods for fast morphological image transforms using bitmapped binary images. *CVGIP : Graph. Models Image Process.*, 54(3) :252–258, 1992.

[79] M. van Herk. A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels. *Pattern Recognition Letters*, 13(7) :517–521, 1992.

[80] Luc Vincent. Morphological grayscale reconstruction in image analysis : applications and efficient algorithms. To appear in the ieee transactions on image processing, 1993, 1992. 2846.

[81] E. Waingold, M. Taylor, V. Sarkar, V. Lee, W. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktumi, R. Barua, J. Babb, S. Amarsinghe, and A. Agarwal. Baring it all to software : The raw machine. Technical report, Cambridge, MA, USA, 1997.

[82] Shlomo Weiss and James E. Smith. A study of scalar compilation techniques for pipelined supercomputers. *ACM Trans. Math. Softw.*, 16(3) :223–245, 1990.

[83] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5) :15–31, 2007.

[84] S. Yehia, S. Girbal, H. Berry, and O. Temam. Reconciling specialization and flexibility through compound circuits. pages 277–288, Feb. 2009.

[85] M. Zuluaga and N. Topham. Resource sharing in custom instruction set extensions. *Application Specific Processors, 2008. SASP 2008. Symposium on*, pages 7–13, June 2008.