UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INGÉNIERIE

Par

ALAIN HORÉ

TRAITEMENT DES IMAGES BIDIMENSIONNELLES À L'AIDE DES FPGAs

1er décembre 2005

Bibliothèque Paul-Émile-Boulet

UQAC Université du Québec à Chicoutimi

Mise en garde/Advice

Afin de rendre accessible au plus grand nombre le résultat des travaux de recherche menés par ses étudiants gradués et dans l'esprit des règles qui régissent le dépôt et la diffusion des mémoires et thèses produits dans cette Institution, **l'Université du Québec à Chicoutimi (UQAC)** est fière de rendre accessible une version complète et gratuite de cette œuvre.

Motivated by a desire to make the results of its graduate students' research accessible to all, and in accordance with the rules governing the acceptation and diffusion of dissertations and theses in this Institution, the **Université du Québec à Chicoutimi (UQAC)** is proud to make a complete version of this work available at no cost to the reader.

L'auteur conserve néanmoins la propriété du droit d'auteur qui protège ce mémoire ou cette thèse. Ni le mémoire ou la thèse ni des extraits substantiels de ceux-ci ne peuvent être imprimés ou autrement reproduits sans son autorisation.

The author retains ownership of the copyright of this dissertation or thesis. Neither the dissertation or thesis, nor substantial extracts from it, may be printed or otherwise reproduced without the author's permission.

RÉSUMÉ

Les images sont de plus en plus utilisées dans de nombreuses disciplines (télédétection, médecine, etc.) car elles fournissent des informations précieuses sur les scènes ou les objets filmés. En médecine, depuis quelques années, les images sont devenues des éléments indispensables à l'établissement d'un bon diagnostic.

Quelquefois, il apparaît qu'une image brute ne fournit pas tous les détails nécessaires à une bonne analyse, elle doit être retouchée de diverses manières : c'est le traitement des images. De nombreux algorithmes ont été mis au point pour effectuer des traitements d'images; certains présentent une complexité algorithmique élevée, ce qui fait que le temps de calcul pour traiter une image peut vite devenir exorbitant (cas des filtres, redimensionnement) dans les architectures monoprocesseurs. Afin d'accélérer le travail des radiologues dans l'analyse des images, nous avons recours au calcul parallèle.

Les FPGAs (Field-Programmable Gate Arrays) sont des circuits électroniques parallèles qui permettent aujourd'hui de développer des applications de plus en plus performantes en vitesse d'exécution et gourmandes en ressources matérielles. Dans le cadre de notre travail de recherche, nous mettons en œuvre, sur FPGA, quelques algorithmes de traitement d'images et nous étudions la faisabilité des traitements en temps réel (idéalement, 20 traitements par seconde ou 1 image toutes les 50 ms).

Le langage de programmation que nous utilisons dans notre étude est psC. Nos travaux permettent de tester la puissance de ce langage dans le déploiement d'applications sur FPGA et ils contribuent à son amélioration.

REMERCIEMENTS

"Le résultat d'un travail n'est que l'aboutissement de nombreux efforts et de beaucoup de persévérances". Chaque réussite cache derrière elle une pléthore de personnes qui se sont illustrées par leurs aides et conseils. Qu'il me soit permis d'adresser mes sincères remerciements aux personnes et organismes qui ont contribué à l'aboutissement de ce travail, notamment :

- L'Agence Canadienne de Développement International (ACDI) à travers le Programme Canadien de Bourses de la Francophonie (PCBF), pour le financement complet de mes études de Maîtrise.
- Mes directeurs de recherche, Messieurs Yves Chiricota et Luc Morin, pour les connaissances transmises et pour avoir accepté de diriger notre travail de bout en bout.
- Monsieur Rachid Beguenane, professeur à l'UQAC, pour ses directives, conseils et contribution à nous fournir un environnement de travail stable.
- Mes conseillères pédagogiques du PCBF, Mesdames Élise Tousignant et Jovette Chouinard, pour leurs guides, aides, conseils, réconforts et encouragements durant toute ma formation.
- Les ingénieurs et programmeurs de Novakod Technologies, Messieurs Marc Lamontagne, François Blackburn, Jean-Michel Fortin, Patrick Latour et Pierre-Jean Boulianne, pour leur soutien technique lors de la phase pratique de mes travaux de recherche.
- Tous ceux qui, de près ou de loin, ont contribué au succès de notre formation et à l'aboutissement de nos travaux.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

CHAPITRE 1 :

INTRODUCTION

1.1. Problématique

Une image peut se définir comme la représentation plus ou moins fidèle d'une personne ou d'une chose par le truchement de la photographie, du dessin, etc. Les images sont de plus en plus utilisées dans de nombreuses disciplines (télédétection, médecine, météorologie, ...) car elles fournissent des informations précieuses et décisionnelles sur les scènes ou les objets filmés. En médecine, voir à l'intérieur du corps sans l'ouvrir a toujours fasciné l'homme. Les débuts de l'imagerie médicale commencent avec la découverte des rayons X en 1895 par l'Allemand Wilhelm Röntgen **[1]**. Depuis quelques années, les images médicales sont devenues des éléments indispensables à l'établissement d'un diagnostic précis. Les techniques d'acquisition sont de plus en plus sophistiquées :

- ✓ Les rayons X;
- ✓ Les ultrasons (l'échographie utilise ce principe);
- ✓ La résonance magnétique nucléaire (détection des tumeurs, images de la colonne vertébrale et de la moelle épinière).

Dans de nombreux cas, il apparaît qu'une image filmée brute ne fournit pas tous les détails nécessaires à une bonne interprétation ou analyse; dans d'autres cas, il pourrait s'avérer utile de ne voir que certains points ou éléments spécifiques d'une image en négligeant les autres. Pour ce faire, l'image doit être retouchée de diverses manières afin de répondre à nos besoins; cela s'appelle le traitement des images. De nombreux algorithmes ont été mis au point au fil des années pour effectuer des opérations sur les images **[11]**, dans le but d'y faire ressortir des détails plus ou moins cachés. Parmi ces algorithmes, nous

pouvons citer la convolution, le filtrage, le zoom, etc. Pour la plupart des algorithmes de traitement, le nombre d'opérations à effectuer dépend de la taille des images. Certains algorithmes présentent une complexité algorithmique élevée qui a pour conséquence d'accroître les temps d'exécution sur ordinateur. Sur les architectures monoprocesseurs, le traitement des images en temps réel (nous entendons par temps réel la capacité de traiter 20 images par seconde, soit 1 image toutes les 50 ms; cela permet de conserver les effets de la persistance rétinienne grâce à laquelle notre œil fait d'une succession d'images une animation) peut rapidement s'avérer laborieux lorsque la taille de l'image est grande (les données d'une image sont généralement représentées sous la forme d'un tableau ou d'une matrice à deux dimensions, une dimension représentant les lignes et une autre les colonnes. La taille d'une image représente le nombre total de données, c'est le produit du nombre de lignes par le nombre de colonnes). Dans le cas de la convolution par exemple (il s'agit de l'un des algorithmes couramment utilisés en traitement d'images), la formulation mathématique discrète est :

Étant donnée une matrice x de taille $M \times N$, un masque de convolution w de taille $K \times L$, la convolution y de x par w est la matrice de taille $(M-K+1)\times(N-L+1)$ dont les éléments sont calculés par la formule $y[i,j]=\sum_{s=1}^{k}\sum_{t=1}^{l} w[s,t]\times x[i+s-1,j+t-1]$.

Le calcul de chaque élément $y[i,j]$ nécessite $K \times L$ multiplications et $K \times L-1$ additions. Pour un masque de convolution de taille 3×3, cela se résume à 9 multiplications et 8 additions. Le calcul de la convolution, dans le cas d'une image de taille 1600×1200, nécessiterait environ 17 280 000 multiplications et 15 360 000 additions. D'une manière

générale, il est extrêmement difficile, voire impossible à l'heure actuelle, d'effectuer de telles opérations en temps réel sur un monoprocesseur. Cette réalité a pu être constatée par IMAGEM, entreprise de développement de logiciels pour le visionnement d'images de radiologie à des fins de diagnostic. Afin d'accélérer le travail des cliniciens et radiologues dans l'analyse des images, il a été convenu de s'orienter vers des solutions, des architectures matérielles favorables au calcul en temps réel des opérations de traitement d'images coûteuses en temps de calculs. La voie qui s'ouvre pour répondre à ce besoin est le calcul parallèle **[3]**.

L'idée à la base du parallélisme est celle d'exécuter simultanément des instructions, des opérations ou des tâches mutuellement exclusives (c'est-à-dire, possédant une certaine indépendance). Il s'agit simplement d'essayer de refléter le mode de fonctionnement de l'homme qui fait autant que possible plusieurs tâches en même temps. Le but du parallélisme est donc de permettre des gains de temps dans l'exécution des calculs sur ordinateur.

Depuis plusieurs années, les concepts et techniques autour du parallélisme ont fait leur chemin. Les stratégies comme le pipeline, le parallélisme de données ou de contrôle ne sont plus nouvelles **[9, 10, 20]**. Cependant, des études sont toujours en cours pour faciliter l'extraction du parallélisme dans les algorithmes séquentiels (dans le cas du traitement des images, les algorithmes séquentiels sont bien connus) **[5, 21]**. Dans le cadre de notre travail de recherche, nous devons mettre en œuvre des algorithmes parallèles de traitement d'images cruciaux en temps afin de faciliter le traitement en temps réel des images. Les

principaux algorithmes retenus sont le redimensionnement (zoom), la convolution, la brillance, le contraste, le fenêtrage, l'effet gamma.

En ce qui concerne le langage de programmation ciblé pour la mise en œuvre des solutions algorithmiques parallèles, nous utilisons psC, un langage parallèle pur en perpétuelle évolution qui offre des possibilités de simulation avant le déploiement matériel. La plate-forme d'exécution finale de nos solutions est un circuit FPGA (Field-Programmable Gate Array) de la compagnie Gidel.

1.2. Méthodologie

Les travaux de recherche ont commencé par une revue approfondie de la littérature scientifique en rapport avec le sujet de notre étude. Les points suivants ont été abordés :

- ✓ Étude de quelques algorithmes usuels de traitement d'images (convolution, filtre linéaire, zoom, segmentation, etc.).
- ✓ Méthodes de calcul parallèle (plus spécifiquement les méthodes systoliques **[15]**);
- ✓ Traitement d'images (bidimensionnelles) à l'aide du calcul parallèle;
- ✓ Utilisation des FGPA pour le traitement d'images **[6, 22, 25]**;

Cette littérature nous a permis de ne pas partir de zéro dans la résolution du problème à résoudre, elle nous a apporté une aide très précieuse dans la voie à suivre pour atteindre nos objectifs.

La suite du travail a été faite en collaboration avec IMAGEM et NOVAKOD. Il s'agissait dans un premier temps de déterminer les algorithmes de traitement d'images bidimensionnelles à mettre en œuvre, puis de les mettre en œuvre par la suite pour une

exécution sur FPGA. Il convient de noter que nous nous intéressons aux versions parallèles les plus rapides de ces algorithmes (nous gardons à l'esprit que la vitesse d'exécution est le principal facteur à l'origine de notre travail de recherche). Ces algorithmes ont ensuite été testés en mode simulation avec psC pour s'assurer qu'ils s'exécutent normalement et fournissent des résultats fiables en concordance avec ceux attendus. Les tests ont été effectués sur des images bitmap au format 8 bits (un pixel de l'image est représenté sur 8 bits); dans ces tests, les transferts de pixels dans les composants de calculs ont été simulés et nous avons observé en sortie les images et pixels produits à des fins de fiabilité.

Après les phases de tests et simulations, la prochaine étape a été la définition d'une méthode pour le transfert effectif des données (pixels d'images) entre les différents composants matériels intervenant dans le processus final de traitement des images. Par la suite, une étude du FPGA retenu pour l'exécution des algorithmes a été abordée. Cette étude permettait d'identifier les contraintes liées à la fréquence d'horloge, la taille de la mémoire, les types d'accès à la mémoire, l'espace physique occupé par les principaux opérateurs arithmétiques et logiques (addition, multiplication, inégalité, etc.).

L'étape suivante a consisté à mettre en œuvre effectivement les algorithmes sur FPGA. Des expérimentations ont été alors réalisées pour étudier les résultats des calculs et les améliorer de manière à avoir un système fiable et stable de traitement d'images.

La dernière phase de notre travail a été de comparer les résultats obtenus à ceux prévus dans des évaluations théoriques et vérifier si les temps observés étaient favorables au calcul en temps réel. Nous avons aussi comparé nos résultats aux performances observées chez les

monoprocesseurs afin de faire ressortir les gains. Enfin, les comparaisons des résultats ont été suivies par des phases d'analyse et d'interprétation.

CHAPITRE 2 :

LES FPGAs

2.1. Introduction

Le présent chapitre décrit les FPGAs ainsi que les principaux composants et éléments qui les caractérisent. Nous abordons également les différents processus de conception avec les FPGAs (conception schématique, conception abstraite) ainsi que leurs avantages et inconvénients. Pour terminer le chapitre, nous décrivons brièvement le processus de conception adopté dans le cadre de nos travaux de recherche et présentons les architectures reconfigurables et la technologie Novakod.

2.2. Description des FPGAs (Field-Programmable Gate Arrays)

Les FPGAs, apparus en 1985 sous l'initiative de l'entreprise Xilinx, sont des composants électroniques entièrement reconfigurables pouvant être reprogrammés afin d'accélérer notablement certaines phases de calculs.

L'architecture générale d'un FPGA est la suivante **[24]**:

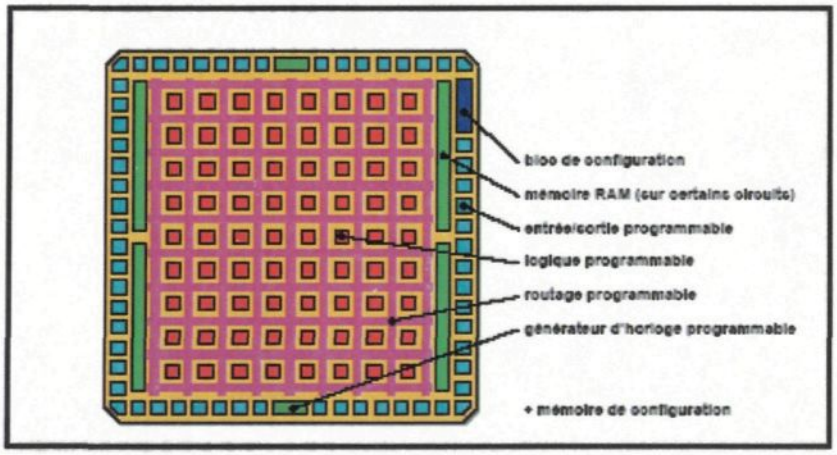

Figure 1 : Architecture générale d'un FPGA

- ✓ Les entrées/sorties peuvent être de diverses natures: signaux utilisateurs, signaux d'horloge (l'horloge définit la cadence des calculs), signaux de test, etc.
- ✓ Les blocs logiques programmables possèdent deux types principaux de composantes: les portes logiques configurables (pour faire des fonctions logiques assez simples à plusieurs entrées) et les bascules servant de mémoire élémentaire de 1 bit.
- ✓ Le routage programmable permet la connexion des blocs logiques entre eux.
- ✓ Le générateur d'horloge permet la distribution des horloges (plusieurs divisions de la fréquence d'horloge sont possibles) partout où elles sont nécessaires.

Un des avantages des FPGAs est leur grande souplesse qui permet de les réutiliser aisément dans des algorithmes différents en un temps court **[7]**. Le progrès de ces technologies permet aujourd'hui de faire des composants toujours plus rapides et à plus haute intégration, ouvrant ainsi la voie au développement d'applications importantes (traitement du signal, codage/décodage, cryptage/décryptage, reconnaissance des formes, etc.).

Le deuxième avantage qu'offrent les FPGAs est la possibilité de les utiliser facilement dans les chaînes systoliques pour paralléliser des calculs critiques. Nous rappelons que la programmation systolique consiste à découper les calculs en blocs différents et à les placer au sein d'un pipeline de données (les résultats des blocs précédents servant d'entrée aux suivants); en d'autres termes, le calcul systolique est une forme de parallélisme géométrique dans lequel les flots de données traversent continuellement une matrice de processeurs ou d'éléments de calculs.

2.3. Processus de conception

Le processus de conception de systèmes avec les FPGAs passe par l'une des deux méthodes suivantes **[27]** :

- ✓ Une conception schématique : cette approche est la plus laborieuse, la plus difficile pour les concepteurs. Elle consiste à dessiner au niveau logique (portes, connexions) le circuit que l'on veut mettre en œuvre. Cette approche pose des problèmes lorsqu'il faut modifier ou réutiliser une conception existante; on note aussi des difficultés de migration d'une conception d'un constructeur à un autre.
- ✓ Une conception abstraite : elle consiste à utiliser des langages matériels HDL *(Hardware Description Language)* qui sont des langages de programmation de haut niveau permettant de décrire des circuits logiques. Les plus connus de ces HDL sont VHDL et Verilog. Dans ces langages comportementaux, le concepteur décrit, dans des fichiers textuels, les circuits en spécifiant leurs fonctions ou leurs comportements. Par la suite, le code de spécification subit plusieurs traitements (synthèse, simulation, compilation, placement/routage) afin de générer le circuit logique réel qui sera mis en œuvre dans le FPGA. Dans le cas de notre projet de recherche, nous utilisons le langage psC pour la conception de nos circuits. Grâce au compilateur associé à l'environnement de développement du langage, le code est traduit en VHDL; enfin, à l'aide d'outils de la compagnie Altera, le code VHDL passe par des phases de synthèse, compilation, placement et routage avant qu'un fichier binaire soit produit pour pouvoir être exécuté sur FPGA.

2.4. Architectures reconfigurables et technologie Novakod

Une architecture reconfigurable est un circuit intégré composé d'un très grand nombre de blocs ou cellules logiques identiques placées dans une infrastructure de lignes d'interconnexion et dont les fonctions et les interconnexions sont reprogrammables. L'utilisateur peut programmer la fonction de chaque cellule, ainsi que les interconnexions entre les cellules et avec les entrées/sorties du circuit. Les circuits logiques reconfigurables allient les performances des circuits intégrés avec des possibilités de programmation. Ces circuits ont une structure interne (équations logiques et structure d'interconnexion) qui peut être changée dynamiquement. Ils sont aujourd'hui utilisés comme coprocesseurs ou accélérateurs et permettent d'améliorer les performances d'une machine. L'intérêt des systèmes reconfigurables est de pouvoir exploiter le parallélisme inhérent qui apparaît dans de nombreux problèmes algorithmiques. De façon simple, ces systèmes sont composés d'un RPU (Reprogrammable Processing Unit) reprogrammable en une fraction de seconde, d'une mémoire et des unités d'entrée et sortie pour les échanges de données. Ces composants sont souvent mis ensemble sur une même carte qui elle-même peut être montée dans des ordinateurs personnels usuels ou dans des ordinateurs de type industriel : on parle alors d'ordinateur reconfigurable. Il convient de noter que le cœur d'un RPU est un circuit FPGA contenant des milliers d'éléments de calculs ou cellules logiques capables d'effectuer un grand nombre d'instructions en parallèle. Aujourd'hui, la capacité des composants issus de la technologie des architectures reconfigurables est telle que l'on peut prétendre exploiter tout le parallélisme potentiel d'une application, c'est-à-dire dupliquer les

opérateurs en nombre suffisant et optimiser les chemins de données, pour exécuter concurremment des fonctions de calculs, d'entrées-sorties et de mémorisation.

Dans le cas de la technologie Novakod, la combinaison d'un ordinateur et d'un circuit reconfigurable permet d'accroître les performances des applications tout en réduisant les temps et les coûts de développement. Une API (Application Programming Interface) a été développée pour permettre la communication et les échanges de données entre un programme C/C++ et le circuit FPGA chargé d'effectuer les traitements ou calculs sur les données. L'API est contrôlée par un programme C/C++ et a un paramètre sur le fichier binaire d'extension *.rbf* représentant le code exécutable sur FPGA généré par les outils du constructeur Altera à partir du code VHDL issu de la compilation d'un programme psC. Ainsi, une fois un programme écrit avec le langage psC, il est relativement aisé de profiter de la puissance d'exécution des FPGA à partir d'un ordinateur personnel grâce à l'API Novakod.

Un exemple de code psC et C/C++ pour l'addition de 2 nombres est présenté à l'annexe A.

CHAPITRE 3 :

LE LANGAGE psC

3.1. Introduction

Ce chapitre constitue une introduction au langage psC que nous utilisons pour la mise en œuvre d'algorithmes sur FPGAs. Nous y présentons ses origines ainsi que les études menées par rapport à son efficacité et sa rapidité de développement. Nous abordons également les principales caractéristiques du langage (gestion du parallélisme, gestion des composants, etc.) ainsi que son modèle de programmation. Pour terminer le chapitre, nous décrivons le modèle temporel du langage psC et nous présentons ses éléments logiciels (machine virtuelle ou simulateur, environnement de développement).

3.2. Présentation de psC

Désigné originellement sous l'appellation Rodin, puis K3, psC (*Parallel and Synchronous C*) est un langage à la frontière du logiciel et du matériel. Il permet aux programmeurs logiciels de concevoir et déboguer aisément des programmes concurrents et parallèles pouvant être implantés sur des processeurs parallèles. Le langage psC simplifie et facilite la programmation des circuits matériels. Au cours de la session d'automne 2003, à l'Université du Québec à Chicoutimi, des études ont permis de montrer que des étudiants en informatique, sans connaissance des circuits logiques, pouvaient programmer, sans trop de difficultés, des circuits matériels avec psC **[19]**. Pendant l'hiver 2004, des études menées dans la même université ont révélé que le temps de développement avec psC était près de 50% plus court que le temps de développement avec des langages usuels comme VHDL **[19]**.

Le langage psC est complètement parallèle, il permet ainsi l'exécution simultanée de plusieurs instructions ou tâches. De nombreux langages sont des sous-ensembles du langage C (systemC, specC, streamC, Handle-C) auquel on a rajouté quelques instructions pour faire du parallélisme; cependant, ces langages ne permettent pas toujours d'utiliser de façon optimale les ressources des circuits matériels qu'ils adressent, contrairement à psC qui est conçu spécialement pour faire du parallélisme sur de nombreuses variétés d'architectures parallèles. Le langage psC peut être utilisé pour la programmation des systèmes temps réel ou concurrents aussi bien que pour les systèmes reconfigurables. Ces derniers sont de plus en plus en vogue dans l'industrie, les progrès technologiques permettent maintenant de produire des circuits matériels contenant des millions de portes logiques et pouvant remplacer des centaines de CPU, ce qui entraîne des gains en performance énormes (on les retrouve dans le traitement des images, la reconnaissance de la parole, traitement du signal digital, la reconnaissance des formes, etc.). Les domaines d'application potentielle de psC sont ainsi nombreux; l'utilisation de psC pour le traitement des images constitue un des centres d'intérêt de notre projet de maîtrise.

3.3. Caractéristiques

Le langage psC est dédié aux systèmes temps réel et reconfigurables. Quelques caractéristiques de psC sont :

- ✓ Il est un langage parallèle pur avec une syntaxe proche du langage C;
- ✓ Il est fiable, sans gestion dynamique de la mémoire;

- ✓ Il utilise le paradigme composant dans lequel les composants sont complètement indépendants, permettant ainsi le développement aisé de librairies de composants.
- ✓ Il possède un environnement de développement avec possibilité de déboguage des programmes;

3.4. Programmation parallèle orientée évènement (PPOE)

psC est un langage basé sur un nouveau paradigme de programmation : la programmation parallèle orientée évènement (en anglais, *Event Oriented Parallel Programming* (EOPP)).

3.4.1. Trilogie du programmeur [18]

Trois conditions sont requises pour l'existence d'un paradigme de programmation :

- ✓ Il doit y avoir un modèle de programmation pour décrire l'exécution des programmes;
- ✓ Il doit y avoir un langage pour supporter le modèle;
- ✓ Il doit y avoir une plateforme pour l'exécution des programmes.

Le modèle de programmation décrit comment les programmes doivent être exécutés, permettant ainsi aux programmeurs de comprendre le paradigme et de l'appliquer dans la conception d'algorithmes.

Le langage est un moyen d'exprimer ces algorithmes et de supporter le paradigme.

La plateforme est le matériel qui exécute les programmes conformément au paradigme.

La relation entre ces trois concepts est la trilogie du programmeur et s'illustre graphiquement comme suit :

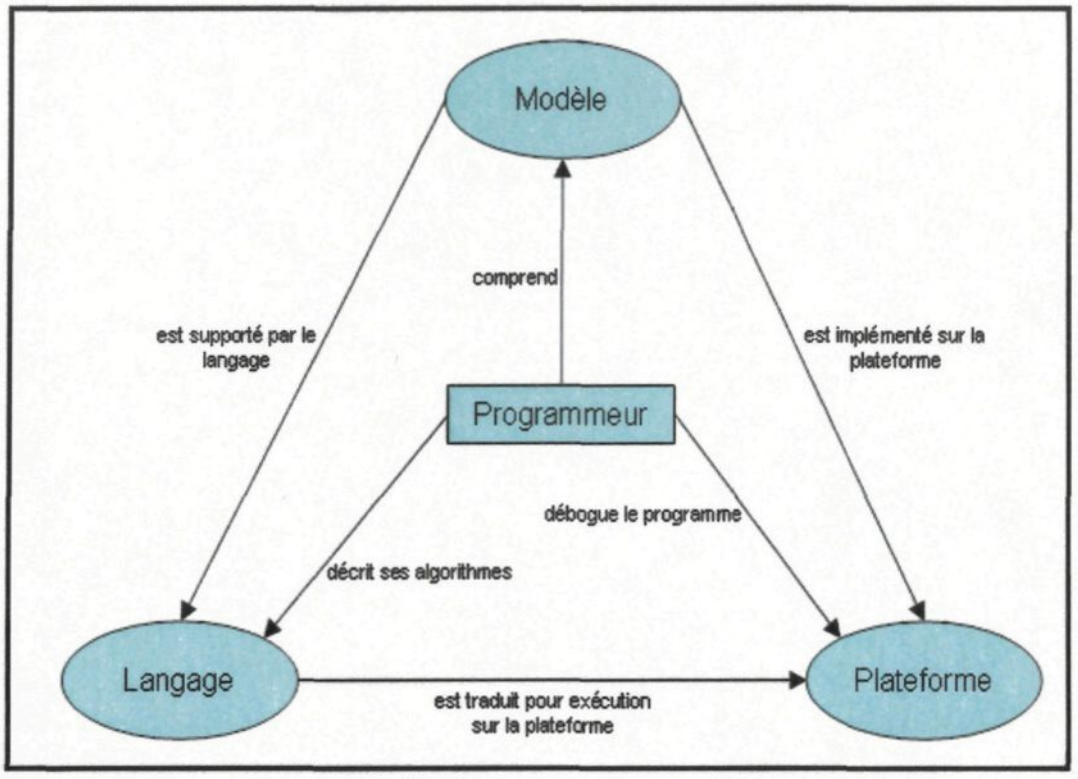

Figure 2 : Trilogie du programmeur [18]

La relation entre les concepts se résume en ces termes :

- ✓ Le langage supporte le modèle de programmation;
- ✓ La plate-forme implémente le modèle;
- ✓ Le programme est compilé pour exécution sur la plate-forme.

3.4.2. Paradigme PPOE

Il existe de nombreux paradigmes de programmation (par exemple, séquentiel ou orienté objet). Le paradigme PPOE les utilise et les intègre d'une manière différente :

- Le modèle d'exécution est orienté évènement, c'est-à-dire qu'il n'y a pas de séquence d'exécution automatique des instructions : chaque fonction est associée à un évènement et seulement l'occurrence de cet évènement initie l'exécution de la fonction. Ainsi, la séquence d'exécution des instructions est déterminée par l'occurrence des évènements, elle n'est donc pas prédéfinie. Dans un programme séquentiel, l'ordre d'exécution est défini à l'avance, c'est l'ordre dans lequel les instructions sont écrites dans le code source.
- Le paradigme est basé sur un modèle d'exécution parallèle. Ainsi, si deux évènements apparaissent simultanément, les fonctions correspondantes seront exécutées en même temps. Bien évidemment, sur une machine séquentielle, le parallélisme va être simulé. En fait, excepté le temps d'exécution, le programmeur ne verra aucune différence entre un programme PPOE exécuté sur une machine séquentielle ou sur un FPGA.
- Le paradigme utilise pleinement le modèle de composant. Les composants sont totalement indépendants, ils communiquent uniquement à travers des signaux et des évènements. Ainsi, il est aisé pour les programmeurs de développer et d'utiliser des librairies en utilisant des outils graphiques conviviaux.
- Le modèle de programmation de psC est assez simple, il est basé sur les déclencheurs (ou *triggers*). Un déclencheur est un évènement qui initie l'exécution d'une fonction ou d'une instruction; en programmation séquentielle, les déclencheurs sont implicites puisque la fin d'une instruction déclenche automatiquement l'exécution de la suivante.

La figure suivante montre le mode d'exécution d'un programme psC.

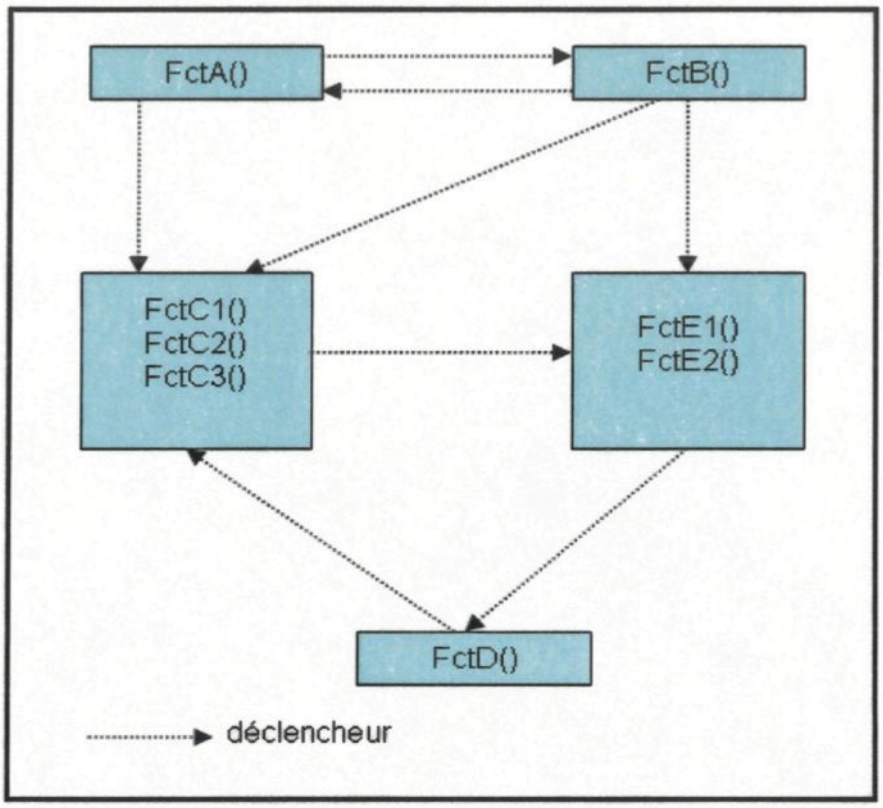

Figure 3 : Mode d'exécution d'un programme psC [18]

Dans un programme psC, chaque composant rassemble un certain nombre de fonctions, chacune associée à un déclencheur. Par exemple, le composant A envoie un déclencheur au composant B qui va initier l'exécution de la fonction FctB(). De la même manière, le composant C pourra recevoir trois déclencheurs associés à trois fonctions. Une fois enclenchée, l'exécution d'un programme est gouvernée par les évènements; les composants s'envoient des évènements pour initier l'exécution des fonctions.

En réalité, les signaux ne transportent pas seulement des déclencheurs. Les signaux sont aussi des variables partagées utilisées pour la communication entre les composants. Un signal peut être vu simplement comme une variable écrite par un port de sortie et lue par un port d'entrée. Pour le programmeur, l'écriture sur un port de sortie est assimilable à l'écriture sur une variable locale tandis que la lecture sur un port d'entrée est assimilable à la lecture du contenu d'une variable locale.

Un évènement devient un déclencheur lorsqu'on lui associe une fonction. Par exemple, dans la figure ci-dessous, les évènements sur le signal Y sont des déclencheurs. Lorsqu'un port d'entrée est associé à une fonction, il est dit *actif*, autrement il est dit *passif*.

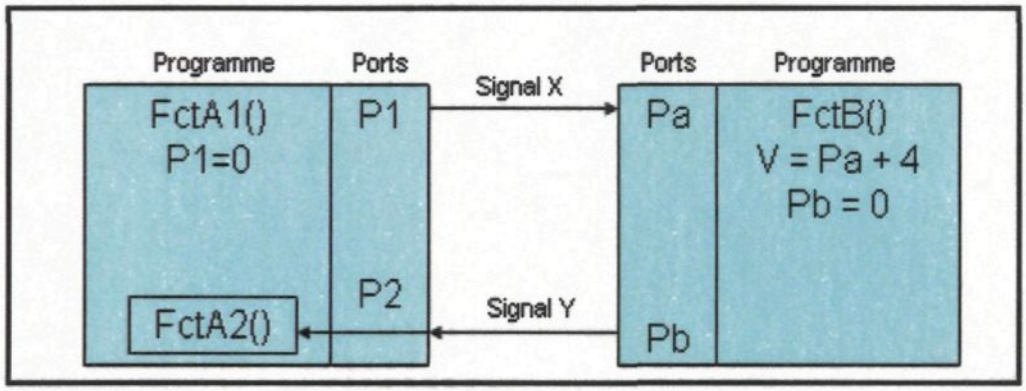

Figure 4 : Exemple de signal déclencheur

Un point important à noter est le fait qu'un évènement n'est pas un appel de fonction: lorsqu'un évènement est créé, il est placé dans une liste d'évènements et la fonction associée ne sera exécutée qu'au prochain pas de temps (l'exécution des programmes se fait par *saut* ou *pas de temps*, les calculs sont faits à des instants précis).

3.5. Modèle temporel de psC

Le langage psC supporte présentement un modèle temporel simple : tous les composants élémentaires (c'est-à-dire sans structure interne) possèdent un délai de calcul fixe uniforme. Tous les traitements se font de manière unicycle : le calcul est déclenché par un évènement d'entrée et produit un évènement (résultat) à la sortie après un délai correspondant à 1 cycle d'horloge. Cette contrainte stricte force toutes les opérations, effectuées à l'interne par le composant, à être très courtes. Cela signifie également qu'il y a une horloge unique dans tout le système (*single-phase clocking*). Pour des fonctions complexes, le programmeur se doit de faire une conception hiérarchique qui effectue le calcul sur plusieurs cycles avec l'aide de plusieurs sous-composants. Notons toutefois que plusieurs calculs peuvent se dérouler simultanément s'il y a plusieurs sorties, c'est-à-dire que le composant peut mettre en œuvre des fonctions multi-entrées/multi-sorties : il en résulte un *multithreading* implicite.

3.6. Autres éléments de psC

Le langage psC possède un environnement de développement intégré *(Novakod Studio)* permettant l'écriture et le débogage aisés des programmes. L'environnement de développement *Novakod Studio* fonctionne actuellement sur les PC utilisant Windows 9x/NT/2000/XP. Il comprend un éditeur, un compilateur pour traduire les programmes sous forme exécutable, un simulateur ou machine virtuelle (*psC Virtual Machine*) pour

l'exécution des programmes sur les machines à un processeur. Pour l'exécution des programmes sur FPGA, le compilateur génère du code VHDL.

La machine virtuelle est un programme qui gère les évènements transportés par les signaux et contrôle l'exécution des fonctions en fonction de l'occurrence des déclencheurs. Avec le programme compilé, la machine virtuelle forme une machine qui reçoit, traite et produit des évènements. Les évènements sont insérés dans une liste d'évènements et, lorsque l'instant de traitement d'un évènement est arrivé, la machine virtuelle exécute toutes les fonctions qu'il déclenche. L'exécution des fonctions peut générer d'autres évènements qui transporteront d'autres déclencheurs.

Le langage psC gère de nombreux types de données (bit, entier, booléen, réel, tableau, etc.), il comprend aussi des instructions séquentielles comme les branchements conditionnels. Les opérateurs arithmétiques et logiques traditionnels sont intégrés dans le langage.

CHAPITRE 4 :

QUELQUES ALGORITHMES DE TRAITEMENT D'IMAGES

4.1. Introduction

Afin de permettre le traitement élémentaire des images, des algorithmes spécifiques ont été choisis pour faire ressortir des détails plus ou moins visibles sur les images. Nous présentons dans ce chapitre les algorithmes qui ont été choisis pour être mis en œuvre. Il convient de définir quelques termes que nous utilisons dans les lignes suivantes du mémoire :

Image bitmap: image composée d'un ensemble (ou matrice) de points (les pixels) auxquels sont associées une position et une couleur. La matrice peut être représentée par un tableau bidimensionnel $n \times m$ composé de n lignes et m colonnes.

Pixel: c'est le plus petit élément qui compose une image bitmap affichée sur un écran ou imprimée. Le nombre de pixels exprime généralement la résolution d'une image ou d'un écran.

Histogramme des niveaux de gris: représentation graphique ayant en abscisses les valeurs de niveaux de gris, et en ordonnées le nombre de pixels associés à chaque valeur de niveau de gris.

Voisinage d'un pixel : pour un pixel donné pris comme repère, il s'agit des pixels d'une image qui lui sont les plus proches par rapport à une distance donnée.

4.2. Brillance

La brillance a pour effet de déplacer vers la droite ou la gauche l'histogramme des niveaux de gris de pixels (concrètement, les couleurs deviennent plus claires ou plus sombres).

Mathématiquement, les niveaux de gris des pixels de l'image de sortie se calculent par la formule :

$output[a][b] = input[a][b] + brillance \quad a \in [1, M], b \in [1, N]$ où *brillance* est le facteur de brillance utilisé.

Un facteur de brillance positif entraîne un déplacement de l'histogramme vers la droite tandis qu'un facteur négatif entraîne un déplacement de l'histogramme vers la gauche.

Le calcul de la brillance est une opération assez simple à réaliser, sa parallélisation nécessite simplement un parallélisme de données.

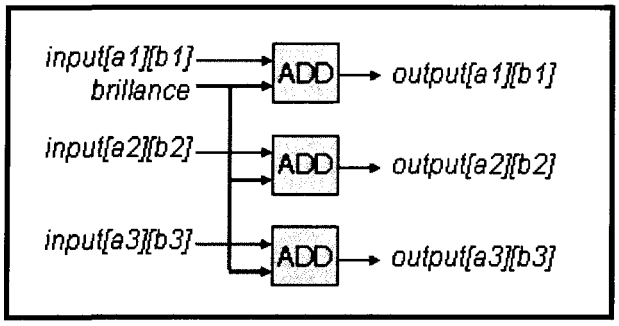

Figure 5 : Calcul de la brillance

brillance représente le facteur de brillance, *input* les données en entrée, *output* les données en sortie.

4.3. Contraste

Le contraste a pour but d'élargir ou d'amincir la distribution des valeurs de niveaux de gris des pixels sur l'histogramme d'une image (concrètement, en augmentant le contraste, la différence entre les couleurs claires et les couleurs sombres est plus accentuée; l'inverse est observé lorsqu'on diminue le contraste).

La formule de calcul des niveaux de gris de l'image de sortie est :

$output[a][b] = input[a][b] \times contraste \quad a \in [1, M], b \in [1, N]$, où *contraste* désigne le facteur de contraste utilisé.

Cette formule est assez simple et ne nécessite qu'un multiplicateur. Comme dans le cas de la brillance, sa parallélisation nécessite simplement un parallélisme de données.

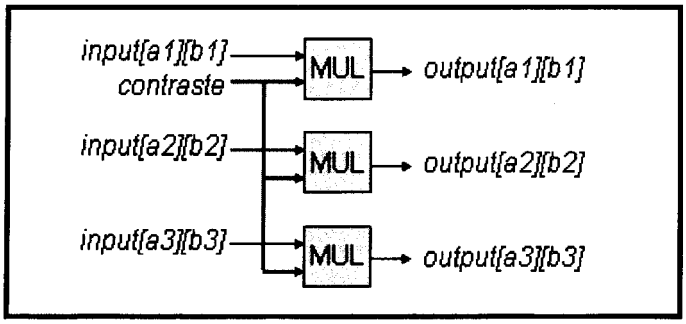

Figure 6 : Calcul du contraste

contraste représente le facteur de contraste, *input* les données en entrée, *output* les données en sortie.

4.4. Fenêtrage

Le fenêtrage a pour but de ramener la plage des valeurs de niveaux de gris de pixels dans un intervalle $[0,255]$ par rapport à une fenêtre spécifiée **[4]**. Les pixels d'une image sont parcourus un à un en ligne puis en colonne, un calcul différent est effectué en fonction des valeurs de niveau de gris.

La formule de calcul des niveaux de gris de pixels est donnée par :

if

$$\left(input[a][b] \le WinCenter - \frac{WinWidth}{2}\right) \quad output[a][b] = 0$$

else if

$$\left(input[a][b] > WinCenter + \frac{WinWidth}{2}\right) \quad output[a][b] = 255$$

else

$$output[a][b] = \frac{input[a][b] - \left(WinCenter - \frac{WinWidth}{2}\right)}{WinWidth} \times 255$$

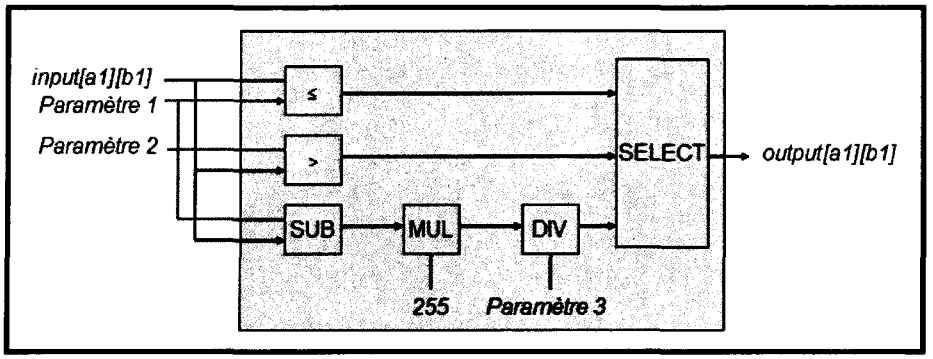

Figure 7 : Calcul du fenêtrage

WinCenter représente le centre de la fenêtre, *WinWidth* sa largeur, *input* les données en entrée, *output* les données en sortie.

$$Paramètre\,1 = WinCenter - \frac{WinWidth}{2}$$

$$Paramètre\,2 = WinCenter + \frac{WinWidth}{2}$$

$$Paramètre\,3 = WinWidth$$

4.5. Effet gamma

Il consiste à modifier non linéairement les valeurs de pixels affichés, c'est-à-dire qu'il permet de modifier l'affichage des teintes pâles ou foncées pour les rendre plus visibles et discernables à l'œil. Le paramètre *gamma* (γ) traduit la non linéarité de la représentation de l'intensité des couleurs sur les périphériques d'affichage.

La formule de calcul des niveaux de gris de pixels de l'image de sortie est :

$$output[a][b] = (input[a][b]/255)^{\gamma} \times 255 \quad a \in [1, M], b \in [1, N]$$

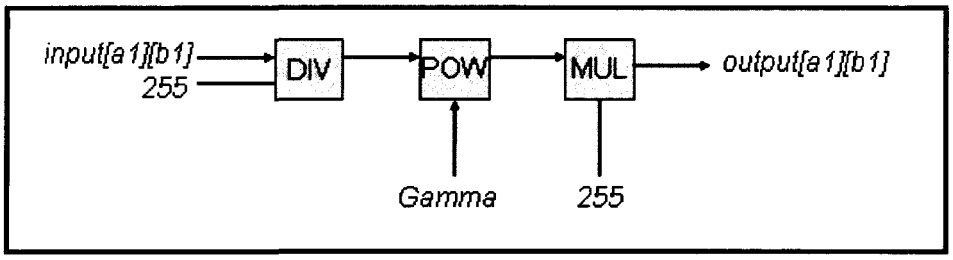

Figure 8 : Architecture parallèle pour le calcul de l'effet gamma

γ représente le facteur de l'effet, *input* les données en entrée, *output* les données en sortie. Les niveaux de gris en sortie sont toujours compris entre 0 et 255 pour les images bitmap 8 bits.

4.6. Zoom

Le zoom (ou *resizing*) est une opération qui a pour but d'augmenter ou de réduire la taille de tout ou partie d'une image suivant un facteur d'échelle.

En imagerie, le zoom est effectué à l'aide des formules d'interpolation (un pixel de l'image finale est une combinaison de pixels de l'image à traiter) **[17]**. On retrouve ainsi plusieurs méthodes de calcul du zoom parmi lesquelles l'interpolation par la valeur moyenne, l'interpolation bilinéaire, l'interpolation bicubique, etc.

Dans le cadre de notre projet, nous nous intéressons au zoom par interpolation bilinéaire : si on compare au zoom par interpolation bicubique, les résultats obtenus sont comparables **[2, 12]**. Nous avons écarté le zoom par l'interpolation bicubique en raison de sa complexité élevée par rapport au zoom bilinéaire; de plus, chaque pixel de l'image finale nécessite 16 voisins dans l'image originale pour effectuer les calculs, ce qui entraîne plus de lectures des données en mémoire que dans l'interpolation bilinéaire où on a besoin que de 4 voisins pour calculer chaque pixel de l'image de sortie.

Nous avons écarté les méthodes d'interpolation par le plus proche voisin et par la valeur moyenne pour les résultats pas très satisfaisants qu'ils fournissent. Les images résultantes connaissent beaucoup d'effets d'escaliers qui rendent les images imprécises et parfois floues.

4.6.1. Interpolation bilinéaire

Soit (x',y') les coordonnées d'un pixel dans l'image finale, soit (x,y) les coordonnées réelles correspondant à ce pixel dans l'image initiale (rappelons-nous qu'un zoom fait passer une image de taille $L \times C$ (nombre de lignes × nombre de colonnes) à une image de taille $L' \times C'$) :

$$x = x' \times \frac{C}{C'}, \quad y = y' \times \frac{L}{L'}$$

On veut calculer la valeur de niveau de gris associée au pixel en (x',y') dans l'image finale.

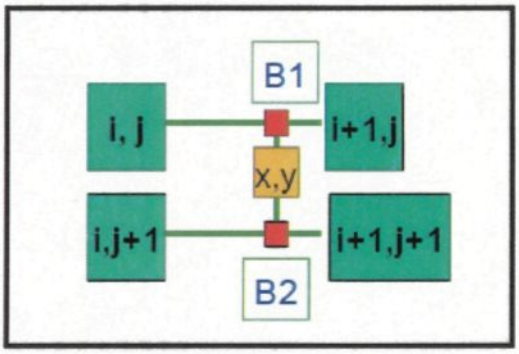

Figure 9 : Zoom bilinéaire

Si on désigne par $\lfloor x \rfloor$ la partie entière du nombre réel x, $I(x,y)$ le niveau de gris associé au pixel se trouvant en position (x,y) dans l'image originale et $I'(x,y)$ le niveau de gris associé au pixel se trouvant en position (x,y) dans l'image finale, alors on a les formules suivantes :

$$I'(x',y') = \lfloor (1-dy) \times B1 + dy \times B2 \rfloor$$
$$B1 = (1-dx) \times I(i,j) + dx \times I(i+1,j)$$
$$B2 = (1-dx) \times I(i,j+1) + dx \times I(i+1,j+1)$$
$$dx = x - i$$
$$dy = y - j$$
$$i = \lfloor x \rfloor$$
$$j = \lfloor y \rfloor$$

dx et *dy* sont des différences entre les valeurs réelles et entières des abscisses (respectivement des ordonnées), *B1* et *B2* sont des sommes pondérées de pixels.

4.7. Convolution : algorithmes systoliques

En traitement d'images en général, et dans le cas spécifique de l'imagerie médicale, la convolution constitue un algorithme de base permettant d'effectuer des opérations plus ou moins complexes sur une image; ce calcul est aussi un des plus gourmands aussi bien en temps de calcul qu'en espace mémoire.

Dans ce paragraphe, nous faisons l'analyse de quelques architectures systoliques pour le calcul des convolutions et étudions les caractéristiques algorithmiques des architectures pour des besoins de comparaison.

On suppose que nous travaillons avec une image de taille M×N (M lignes, N colonnes) et un masque de convolution (matrice de coefficients qui est utilisée pour le calcul de la somme pondérée de pixels dans une opération de convolution) de taille K×K (en pratique, K sera fixé à 3).

Formule de convolution: $y[i,j]=\sum_{s=1}^{K}\sum_{t=1}^{K} w[s,t]\times x[i+s-1,j+t-1]$, x est la matrice représentant l'image d'entrée, y est la matrice représentant l'image convoluée.

4.7.1. Architecture 1: diagramme 1D de Kung

4.7.1.1. Description

Nous présentons l'architecture proposée par Kung **[14]**. Ici, le problème de la convolution 2D est converti à un problème de calcul de la convolution 1D.

On suppose que le flot des pixels entre dans la matrice des cellules ligne par ligne et que les valeurs du masque de convolution sont fixées dans chaque cellule de la matrice, avec $N-K=N-3$ zéros entre deux lignes du masque.

L'architecture globale nécessite $(N-K)\times(K-1)+K\times K=N\times(K-1)+K$ cellules.

Pour fonctionner correctement, le flot des pixels de sortie doit circuler à une vitesse double de celle du flot des pixels en entrée.

Le graphique suivant illustre cette architecture dans le cas où $N=5$. On remarque la présence de $N-3=5-3=2$ zéros entre chaque ligne de la matrice des cellules.

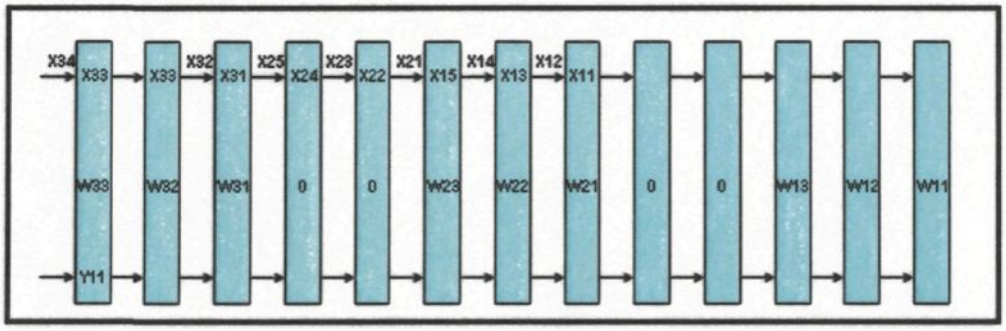

Figure 10 : Modèle 1D de Kung

Dans cette architecture, on fera entrer 3 lignes au départ avant de pouvoir faire sortir les premiers pixels. Par la suite, pendant que le flot des pixels en entrée continuera de progresser, les pixels en sortie pourront être recueillis au fur et à mesure.

Entre deux lignes de pixels en sortie, on doit observer un certain temps d'attente pour ne pas enchevêtrer les pixels en entrée nécessaires au calcul des pixels de sortie de la nouvelle ligne.

Les tests ont permis de constater qu'entre deux lignes de pixels de sortie, on doit ralentir le déplacement des pixels en sortie de 2 pas de temps ($\Delta_{line} = 2$, si on désigne par Δ_{line} la latence entraînée par la sortie des pixels d'une nouvelle ligne après la fin de la ligne précédente).

4.7.1.2. Analyse de l'algorithme

Chaque cellule de l'architecture accueillant les valeurs du masque de convolution est constitué d'un composant comportant un multiplicateur et un additionneur.

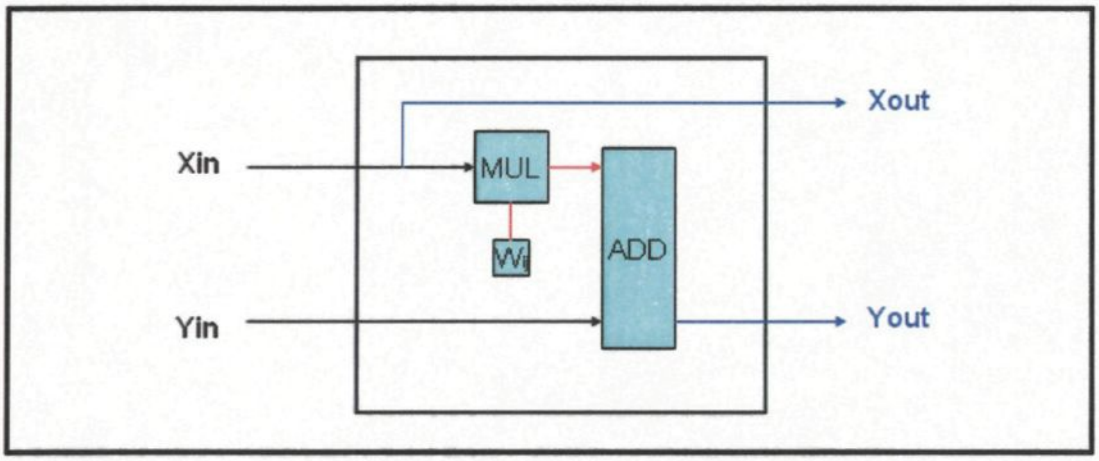

Figure 11 : Modèle 1D de Kung, vue interne d'un élément de calcul

Le calcul effectué par chacune de ces cellules peut être exprimé comme suit :

$\begin{cases} Xout = Xin \\ Yout = Yin + Xin \times W_{ij} \end{cases}$, on suppose que W_{ij} est la valeur du masque, contenue dans la cellule.

En termes de composants dans l'architecture globale, on aura ainsi 9 additionneurs et 9 multiplicateurs. À chaque pas de temps, les opérations suivantes sont effectuées : 9 additions et 9 multiplications. Les cellules ne comportant pas les valeurs du masque de convolution sont considérées comme des composants de transmission de valeur, leur rôle étant tout simplement d'envoyer sur la cellule voisine les valeurs reçues en entrée.

Pour estimer le nombre de pas de temps qu'il faut pour faire sortir le premier pixel (on désignera ce nombre par $\Delta_{startup}$), appelons Δ_{mul} le nombre de pas de temps nécessaire au calcul d'une multiplication, Δ_{add} celui nécessaire au calcul d'une addition, Δ_{mask} le temps nécessaire au passage d'un pixel à travers les cellules contenant les éléments du masque de convolution, $\Delta_{transfer}$ le temps nécessaire au passage d'un pixel à travers les cellules ne contenant pas les éléments du masque de convolution.

On a alors :

$\begin{cases} \Delta_{startup} = \Delta_{mask} + \Delta_{transfer} \\ \Delta_{mask} = 9 \times (\Delta_{mul} + \Delta_{add}) \\ \Delta_{transfer} = [N \times (K-1) + K] - 9 \end{cases}$, c'est-à-dire:

$\begin{cases} \Delta_{startup} = 9 \times (\Delta_{mul} + \Delta_{add}) + [N \times (K-1) + K - 9] \\ \Delta_{startup} = 9 \times (\Delta_{mul} + \Delta_{add}) + (2 \times N - 6) \end{cases}$

Après la sortie du premier pixel, les autres pixels seront calculés un à un à chaque pas de temps. Au passage à une nouvelle ligne, le retard $\Delta_{line} = 2$ sera observé.

Au bout du compte, si on désigne par le temps nécessaire à l'entrée de tous les pixels dans la matrice systolique, alors le temps nécessaire au calcul de l'image de sortie entière est :

$$\begin{cases} \Delta = \Delta_{startup} + \Delta_{in} \\ \Delta_{in} = M \times (N + \Delta_{line}) \end{cases}, \text{ c'est-à-dire :}$$

$$\begin{cases} \Delta = \Delta_{startup} + M \times (N + \Delta_{line}) \\ \Delta = 9 \times (\Delta_{mul} + \Delta_{add}) + (2 \times N - 6) + M \times (N + 2) \end{cases}$$

4.7.2. Architecture 2: diagramme 1D de Kung et Picard

4.7.2.1. Description

Cette architecture a été proposée par Kung et Picard **[16]**. Le problème de la convolution 2D est converti à un problème de calcul de la convolution 1D.

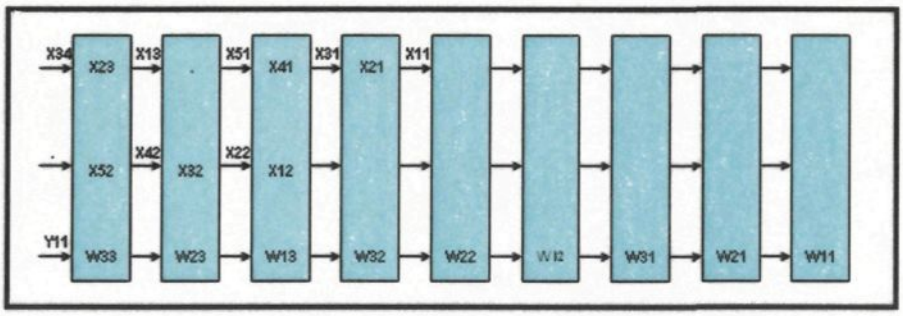

Figure 12 : Modèle 1D de Kung et Picard

Ici, il existe deux flots de données pour les pixels de l'image en entrée, et à chaque étape de calcul, les cellules doivent choisir un des deux pixels disponibles en entrée. Les pixels correspondant aux colonnes impaires dans la représentation matricielle de l'image se

déplacent dans le flot de données du haut, tandis que ceux correspondant aux colonnes paires utilisent le flot de données du bas.

Dans un flot de pixels en entrée, entre chaque colonne de pixels on introduit un retard.

Chaque cellule est supposée emmagasiner une valeur de pixel du masque de convolution. Chaque cellule utilise un flot de données pendant K cycles de calculs consécutifs avant de changer de flot de données.

Comme dans l'architecture précédente, le flot des pixels de sortie doit circuler à une vitesse double du flot des pixels en entrée.

Au départ, les pixels sont rentrés un à un, jusqu'à ce que le premier pixel de l'image résultante soit généré. Ensuite, à chaque pas de temps, les pixels suivants sont générés colonne après colonne (en fait, les pixels sont générés un à un suivant des blocs de taille 3×3).

4.7.2.2. Analyse de l'algorithme

Pour ce qui est du nombre de cellules requises dans l'architecture, elle correspond tout simplement à la taille du masque de convolution. Comme nous supposons que nous travaillons avec des masques de taille 3×3, on déduit que nous avons besoin de 9 cellules de calculs.

Chaque cellule peut être représentée comme un composant de haut niveau comportant un multiplicateur et un additionneur :

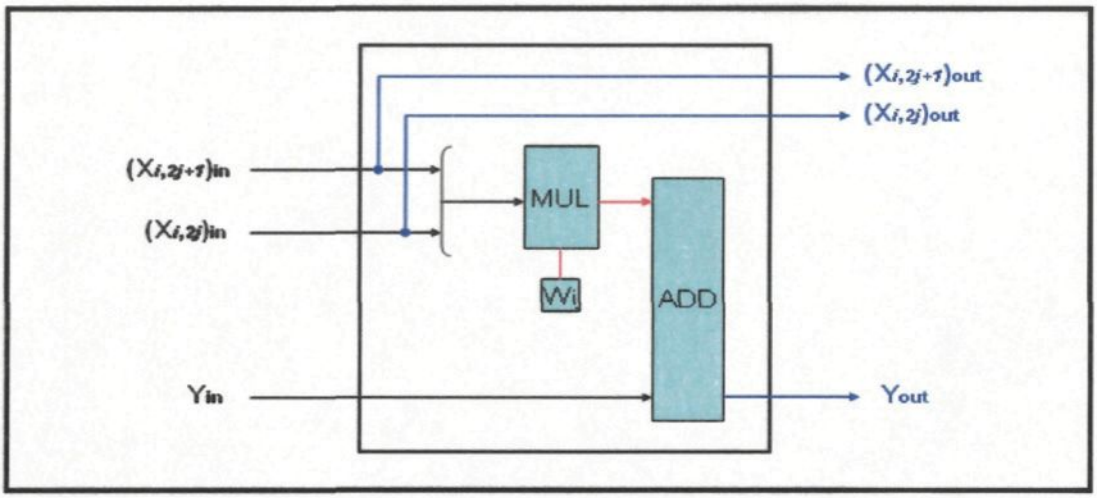

Figure 13 : Modèle 1D de Kung et Picard, vue interne d'un élément de calcul

Chaque cellule effectue l'un des deux ensembles de calculs suivants :

$$\begin{cases} \left(X_{i,2j+1}\right)_{out} = \left(X_{i,2j+1}\right)_{in} \\ \left(X_{i,2j}\right)_{out} = \left(X_{i,2j}\right)_{in} \\ Yout = Yin + \left(X_{i,2j+1}\right)_{in} \times W_{ij} \end{cases} \quad \text{ou} \quad \begin{cases} \left(X_{i,2j+1}\right)_{out} = \left(X_{i,2j+1}\right)_{in} \\ \left(X_{i,2j}\right)_{out} = \left(X_{i,2j}\right)_{in} \\ Yout = Yin + \left(X_{i,2j}\right)_{in} \times W_{ij} \end{cases}$$

, on suppose que W_{ij} est la valeur du masque contenue dans la cellule.

En termes de composants dans l'architecture globale, on aura ainsi 9 additionneurs et 9 multiplicateurs.

A chaque pas de temps, les opérations suivantes sont effectuées : 9 additions et 9 multiplications.

Pour estimer le nombre de pas de temps qu'il faut pour faire sortir le premier pixel (on désignera ce nombre par $\Delta_{startup}$), appelons Δ_{mul} le nombre de pas de temps nécessaire au calcul d'une multiplication, Δ_{add} celui nécessaire au calcul d'une addition.

On a alors :

$$\Delta_{startup} = K \times K \times (\Delta_{mul} + \Delta_{add})$$
$$\Delta_{startup} = K^2 \times (\Delta_{mul} + \Delta_{add})$$

Après la sortie du premier pixel, les autres pixels seront calculés un à un (par bloc de taille $K \times K$) à chaque pas de temps. Au passage à une nouvelle ligne de blocs, on observe une latence $\Delta_{line} = K$.

Au bout du compte, le temps nécessaire au calcul de l'image de sortie entière est estimé à:

$$\Delta = \Delta_{startup} + M \times (N + \Delta_{line})$$
$$\Delta = K^2 \times (\Delta_{mul} + \Delta_{add}) + M \times (N + 3)$$

4.7.3. Architecture 3: diagramme 2D de Sharma, Allen et Pargas

4.7.3.1. Description

Cette architecture a été proposée par Sharma, Allen et Pargas **[23]**. Le problème de la convolution 2D est résolu en utilisant un design 2D.

Dans cette architecture, les pixels du masque de convolution sont fixés dans les cellules de la manière suivante : le vecteur W_i (formé des pixels de la $i^{ème}$ ligne du masque) est fixé dans chaque cellule de la $i^{ème}$ colonne de la matrice systolique.

La figure ci-dessous correspond à un masque de taille 3×3 .

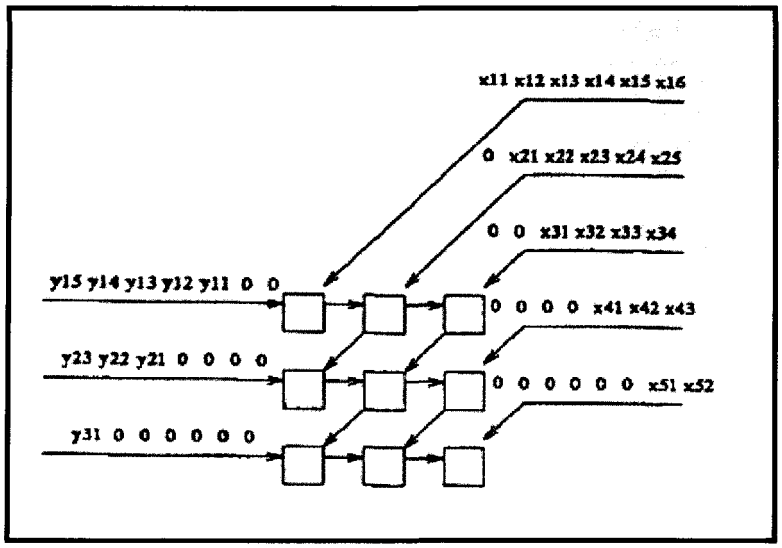

Figure 14 : Architecture systolique 2D de Sharma, Allen et Pargas

Le flot des pixels de sortie y_{ij}, initialement à 0, entre dans la matrice systolique par ordre croissant de ligne. Chaque valeur y_{ij} doit traverser une cellule suivant une direction de la gauche vers la droite. Le flot des pixels d'entrée x_{ij} pénètre dans la matrice systolique par ordre de ligne croissante. Les pixels parcourent le réseau systolique suivant une diagonale dans la direction du nord-est vers le sud-ouest. Afin de permettre à un certain nombre de pixels x_{ij} d'entrer dans le réseau avant le calcul des premiers pixels des lignes, le flot des pixels de sortie y_{ij} doit être retardé de $K-1$ pas par rapport au flot des pixels d'entrée. Au niveau de chaque cellule, l'entrée d'un pixel entraîne la sortie d'un autre pixel utilisé lors de la précédente étape.

4.7.3.2. Analyse de l'algorithme

Chaque cellule peut être représentée par le composant suivant :

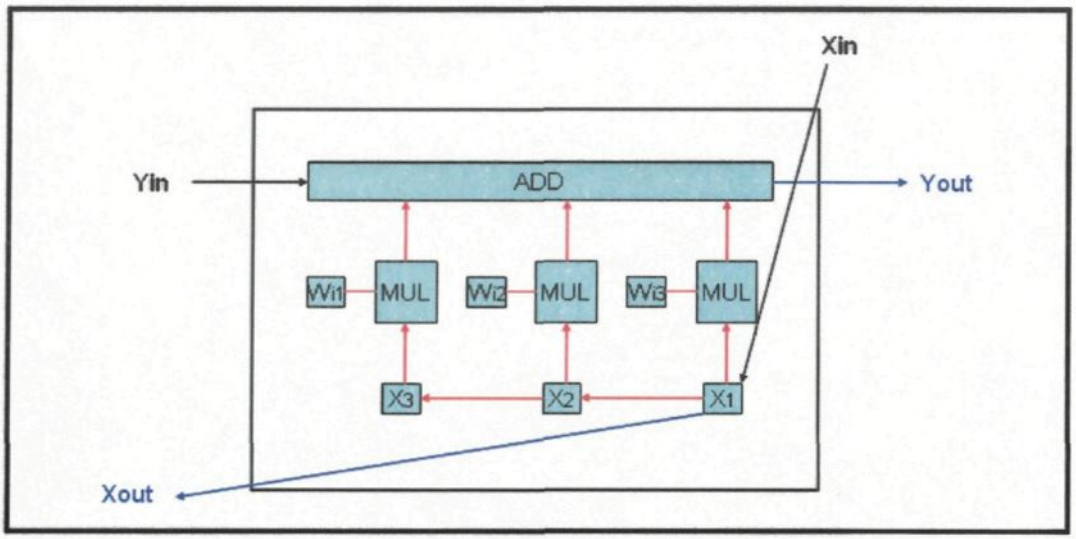

Figure 15 : Architecture systolique 2D de Sharma, Allen et Pargas, vue interne d'un élément de calcul

A l'intérieur de chaque cellule, les opérations suivantes sont effectuées : 3 multiplications (en parallèle), 1 addition à 4 opérandes:

Le calcul effectué par chacune de ces cellules peut être exprimé comme suit :

$$\begin{cases} Xout = X_1 \\ X_3 = X_2 \\ X_2 = X_1 \\ X_1 = Xin \\ Yout = Yin + X_1 \times W_{i1} + X_2 \times W_{i2} + X_3 \times W_{i3} \end{cases}$$

Dans l'architecture globale, le nombre de calculateurs dont on a besoin est: K^3 multiplicateurs, $3K^2$ additionneurs.

De plus, $\Delta_{line} = K - 1$

Pour estimer le nombre de pas de temps qu'il faut pour faire sortir le premier pixel (on désignera ce nombre par $\Delta_{startup}$), appelons Δ_{mul} le nombre de pas de temps nécessaire au calcul d'une multiplication et Δ_{add} celui nécessaire au calcul d'une addition.

On a alors, compte tenu du fait que dans la première cellule on attend $\Delta_{transfer} = K$ pas avant que les calculs commencent et que dans les *K-1* autres cellules il n'y a pas cette attente :

$$\Delta_{transfer} = K$$
$$\Delta_{startup} = \left(\Delta_{transfer} + \Delta_{mul} + \Delta_{add}\right) + \left(K-1\right) \times \Delta_{add}$$
$$\Delta_{startup} = \Delta_{mul} + K \times \Delta_{add} + K$$

Après la sortie du premier pixel, les autres pixels seront calculés, pour 3 lignes, un à un. Au passage à un autre groupe de 3 lignes, on observera une latence de *K-1* pas, on désignera celle-ci par Δ_{line}.

Au bout du compte, le temps nécessaire au calcul de l'image de sortie entière est estimé à :

$$\Delta = \Delta_{startup} + \frac{M}{3} \times \left(N + \Delta_{line}\right)$$
$$\Delta = \Delta_{mul} + K \times \Delta_{add} + K + \frac{M}{3} \times \left(N + K - 1\right)$$

Il est possible d'augmenter le nombre de lignes de l'image de sortie pouvant être calculées simultanément; pour ce faire, il suffit d'ajouter dans le réseau systolique une ligne de K cellules identiques aux précédentes. Pour L lignes $\left(L \geq 3\right)$ que l'on veut sortir simultanément :

Le nombre de calculateurs est : $L \times K^2$ multiplicateurs, $3 \times L \times K$ additionneurs.

Le temps de calcul nécessaire est estimé à :

$$\Delta = \Delta_{startup} + \frac{M}{L} \times (N + \Delta_{line})$$

$$\Delta = \Delta_{mul} + K \times \Delta_{add} + K + \frac{M}{L} \times (N + K - 1)$$

4.7.4. Architecture 4: autre diagramme 2D

4.7.4.1. Description

Cette architecture a été proposée en collaboration avec les directeurs de recherches, Luc Morin et Yves Chiricota.

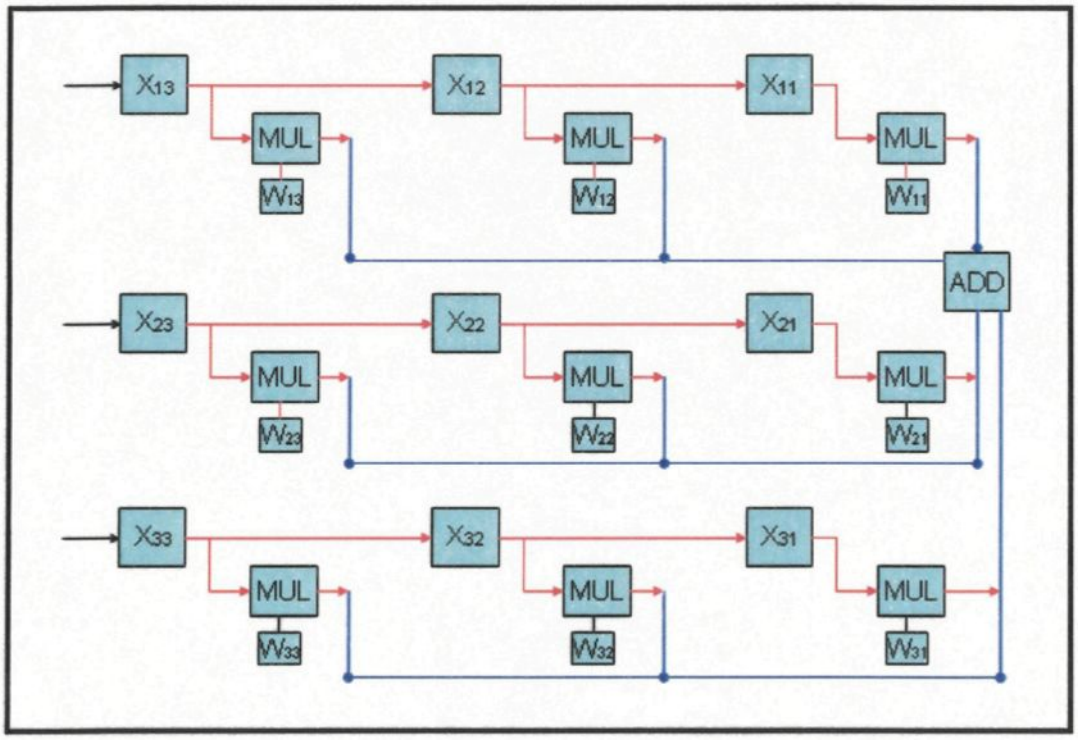

Figure 16 : Architecture systolique 2D proposée

Dans cette architecture, les pixels entrent 3 à la fois et se déplacent vers la droite dans les composants. Les pixels passent par des multiplicateurs avant d'atteindre un additionneur qui va permettre de calculer un pixel de sortie.

4.7.4.2. Analyse de l'algorithme

À chaque pas de temps on comptera $K \times K$ multiplications en parallèle et 1 addition à $K \times K$ opérandes.

Pour estimer le nombre de pas de temps qu'il faut pour faire sortir le premier pixel (on désignera ce nombre par $\Delta_{startup}$), appelons Δ_{mul} le nombre de pas de temps nécessaire au calcul d'une multiplication, Δ_{add} celui nécessaire au calcul d'une addition, $\Delta_{transfer}$ le temps que le met un pixel pour parcourir les composants Xij sur son chemin.

$$\begin{cases} \Delta_{startup} = \Delta_{mul} + \Delta_{add} + \Delta_{transfer} \\ \Delta_{transfer} = K \end{cases}$$

Après la sortie du premier pixel, les autres pixels seront calculés un à un. Au passage à une nouvelle ligne de pixels de sortie, on observera une latence $\Delta_{line} = K - 1$.

Au bout du compte, le temps nécessaire au calcul de l'image de sortie entière est estimé à :

$$\begin{cases} \Delta = \Delta_{startup} + M \times (N + \Delta_{line}) \\ \Delta_{startup} = \Delta_{mul} + \Delta_{add} + \Delta_{transfer} \\ \Delta_{line} = K - 1 \\ \Delta_{transfer} = K \end{cases}$$

Il est possible d'augmenter le nombre de lignes de l'image de sortie pouvant être calculées simultanément; pour ce faire, il suffit d'ajouter dans le réseau systolique, pour chaque nouvelle ligne supplémentaire, $K \times K$ multiplicateurs et 1 additionneur à $K \times K$ opérandes. Ainsi, pour L lignes $(L \geq 1)$ de sorties à générer simultanément, on aura besoin de $K \times K \times L$ multiplicateurs et L additionneurs à $K \times K$ opérandes.

Le nombre de pas de temps requis pour effectuer les calculs est estimé à :

$$\Delta = \Delta_{mul} + \Delta_{add} + K + \frac{M}{L} \times (N + K - 1)$$

4.7.5. Algorithme retenu

D'autres algorithmes de convolution ont été proposés par divers auteurs **[5, 13, 22]**, mais ils s'appliquent beaucoup plus à des convolutions avec des masques ayant certaines caractéristiques (symétrie horizontale, symétrie verticale, etc.). Dans notre cas, nous travaillons avec des masques génériques et l'algorithme retenu est le dernier proposé ci-dessus. La principale raison à ce choix est le meilleur temps d'exécution que l'on obtient en utilisant cet algorithme : par exemple, dans le cas d'un calcul de convolution sur une image de taille 2048×2048 pixels avec un masque de 3×3 et en utilisant des opérateurs parallèles d'addition et de multiplication dont le délai est de 1 pas, le modèle 1D de Kung ainsi que celui de Kung et Picard donnent des temps de calcul de l'ordre de 42 ms contre 14 ms pour le modèle 2D de Sharma et al.; le modèle à trois pixels en sortie que nous proposons donne aussi un temps de calcul de l'ordre de 14 ms, mais très légèrement inférieur à celui de Sharma et al. L'architecture 2D de Sharma et al. offre un temps d'exécution très proche de

celui de notre architecture, mais elle impose une gestion des points de bordure par la méthode des zéros, ce qui implique un certain nombre de conditions à prendre en compte dans un contexte réel d'accès en mémoire de FPGA.

CHAPITRE 5 :

MISE EN ŒUVRE DES ALGORITHMES SUR FPGA

5.1. Introduction

Ce chapitre présente les différentes architectures qui ont été mises en œuvre sur FPGA pour réaliser les traitements d'images. Nous commençons par décrire l'architecture générale du système de traitement d'images ainsi que le processus de communication entre les principaux composants intervenant dans le traitement global d'une image (PC, FPGA, mémoire associée au FPGA). Nous présentons par la suite les critères utilisés pour le choix final de la carte FPGA utilisée pour les traitements d'images. Le paragraphe suivant du chapitre traite de l'API Novakod utilisée pour la communication et le transfert de données entre un ordinateur et une carte FPGA. Le dernier paragraphe présente les différents groupes de traitements d'images (brillance, contraste, fenêtrage, gamma, convolution, zoom) mis en œuvre sur FPGA avec une description des composants utilisés.

5.2. Architecture du système de traitement des images sur FPGA

Le schéma global descriptif de l'ordre d'enchaînement des différentes opérations nécessaires aux traitements des images est présenté dans la figure suivante :

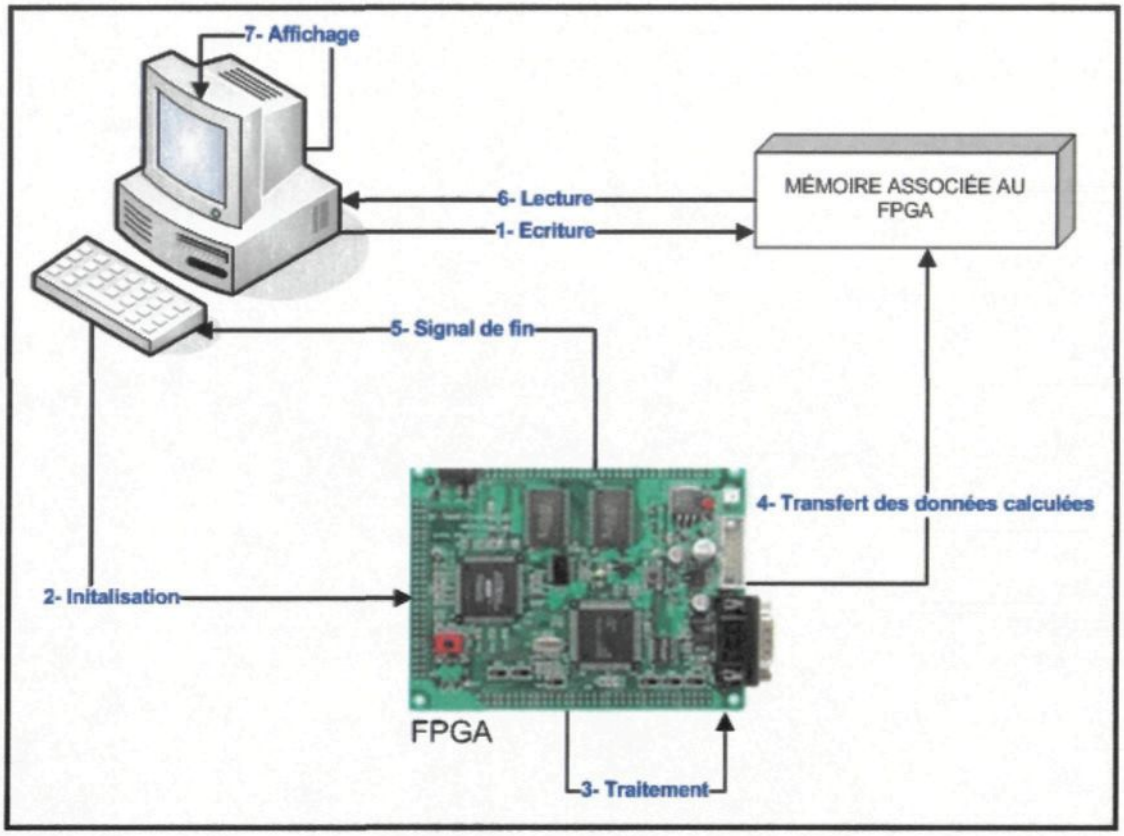

Figure 17 : Architecture générale du cycle de traitement des images

On retrouve dans ce schéma trois principaux composants :

1. L'ordinateur de type PC dans lequel les images sont stockées;
2. Le circuit FPGA qui effectue les traitements proprement dits;
3. La mémoire associée au FPGA; elle permettra de stocker les données de l'image originale et de l'image finale après traitement.

Les opérations permettant le traitement d'images sur FPGA sont enchaînées dans l'ordre suivant :

a. L'image stockée sur le disque dur du PC est transférée dans la mémoire de la carte FPGA à travers un canal PCI. Avec une carte PCI express 133 MHz à 32 bits par

exemple, c'est-à-dire ayant un débit de 532 Mo/s, une image en niveau de gris de 512×512 pixels de 8 bits (taille : 0,25 Mo) sera transférée en environ 0,47 ms.

b. Le PC initialise les opérations de calculs en envoyant un signal de lancement ainsi que les valeurs des paramètres nécessaires à l'exécution des traitements désirés;
c. Le circuit FPGA exécute les algorithmes avec comme données en entrée l'image stockée dans sa mémoire associée ainsi que les paramètres transmis par le PC lors de la phase d'initialisation;
d. Les données calculées par le FPGA sont enregistrées dans sa mémoire associée;
e. Le FPGA envoie au PC un signal indiquant que les traitements sont achevés et que les données de l'image finale sont disponibles en mémoire;
f. Le PC se charge d'aller lire les données calculées par le FPGA et qui sont stockées dans la mémoire qui lui est associée;
g. Finalement, l'image calculée est affichée à l'écran de l'ordinateur de type PC.

5.3. Choix de la carte FPGA

De nos jours, il existe une grande variété de cartes FPGA qui différent les unes des autres par de nombreuses caractéristiques telles que le coût, la densité d'intégration des circuits logiques, la quantité de mémoire disponible, la fréquence d'horloge maximale, le débit de transfert des données entre les ports d'entrée/sortie, le nombre de ports d'entrée/sortie, la disponibilité d'une API de communication entre le FPGA et un ordinateur de type PC, etc.

Dans le cadre de notre projet de recherche, il était nécessaire de choisir une carte ayant un bon rapport qualité/prix et qui permettrait de mettre en œuvre nos algorithmes de traitement d'images. Il convient de noter que la priorité a été mise sur une carte permettant

de faire des tests pour étudier la faisabilité d'exécution des algorithmes de traitement d'images sur FPGA. Le travail d'études des cartes FPGA pouvant répondre à nos besoins a été réalisé par l'entreprise Novakod Technologies **[8]**.

5.3.1. Critères d'évaluation

Six critères ont été pris en considération pour parvenir à classer les 31 plates-formes FPGA retenues pour analyse (les cartes FPGA recensées sont présentées à l'annexe B). Ces dernières ont été sélectionnées du fait qu'elles semblaient à priori, correspondre aux exigences minimales d'une application de traitement d'images. Les critères d'évaluation sont les suivants :

- ✓ Le type de FPGA;
- ✓ Les fonctionnalités;
- ✓ Le prix;
- ✓ Un facteur de confiance (réputation, renommée, expérience) du fabriquant;
- ✓ La vitesse de l'interface PCI;
- ✓ Le support aux usagers offert par les fabricants.

5.3.2. Pondération des critères

Une pondération est établie pour chacun de ces critères selon l'importance relative de ces derniers. Une note finale est ensuite attribuée pour la carte analysée. Il est à noter qu'il est possible de simplement modifier la pondération dépendamment de l'importance

accordée à chacun des critères, ce qui peut évidemment changer le classement d'une situation à l'autre. La pondération adoptée est la suivante :

Critère	Pondération
Prix	4
Performance du FPGA	3
Fonctionnalités	3
Facteur de confiance	2
Support	2
Vitesse de transfert PCI	1

Tableau 1 : Pondération des critères d'évaluation des cartes FPGA

5.3.3. Attribution des notes

Plusieurs facteurs entrent en ligne de compte pour l'attribution d'une cote. Pour chaque critère, une brève explication de la méthode utilisée est donnée dans la présente section.

5.3.3.1. Performance du FPGA

Pour l'évaluation de ce critère, l'échelle est déterminée non seulement par rapport à la puissance du FPGA mais aussi à la pertinence par rapport à l'application à implanter. Les meilleures notes sont attribuées aux cartes qui contiennent des multiplicateurs intégrés ainsi qu'un bon nombre de portes logiques programmables.

5.3.3.2. Les fonctionnalités

Certaines fonctionnalités ont été considérées comme étant essentielles à l'implantation de l'application sur la plate-forme choisie. Une note est attribuée selon le fait que la plate-forme offre de la mémoire RAM, des contrôleurs pour la mémoire RAM et pour le port PCI, de la mémoire cache avec son contrôleur.

Fonctionnalité	Points alloués
RAM disponible	2
Cache disponible	1
Cores PCI	1,5
Autres cores	0,5

Tableau 2 : Points alloués aux cartes FPGA par fonctionnalité

5.3.3.3. Le prix

Pour le critère du prix de la plate-forme, l'échelle est construite par tranches.

Prix	Points alloués
0-2499 USD	5
2500-4999 USD	4
5000-7499 USD	3
7500-9999 USD	2
10000-14999 USD	1
15000 et + USD	0

Tableau 3 : Points alloués aux cartes suivant le prix

5.3.3.4. Le facteur de confiance

Il s'agit en fait d'un amalgame de l'expérience de l'entreprise et de sa position dans le marché en plus d'un facteur d'appréciation basé sur la qualité du site Web et des personnes ressources contactées.

Facteur de confiance	Points alloués
Moins de 5 ans d'expérience	0
Entre 5 et 10 ans d'expérience	1
Plus de 10 ans d'expérience	2
Appréciation	2

Tableau 4 : Points alloués aux cartes FPGA en fonction du facteur de confiance

5.3.3.5. L'interface

La vitesse des entrées et sorties constitue souvent un goulot d'étranglement dans le développement des systèmes reconfigurables. C'est pourquoi la vitesse de transfert des données sur le port PCI doit être prise en compte dans l'attribution d'une note pour la plate-forme étudiée.

Vitesse du bus PCI	Points alloués
1 GB/s	5
768 MB/s	4
528 MB/s	3
266 MB/s	2
132 MB/s	1
Moins de 132 Mb/s	0

Tableau 5 : Points alloués aux cartes FPGA par fonctionnalité

5.3.3.5. Le support

Une note de 1 à 5 est attribuée selon les plans de supports disponibles et la disponibilité de références techniques chez le fabricant.

5.3.4. Conclusion

Au total, 31 cartes FPGA ont été étudiées afin d'identifier une ou plusieurs qui permettraient de faire du traitement d'images avec des contraintes de mise en œuvre souples et un prix raisonnable. Finalement, à partir du système de pondération défini plus haut, le choix s'est porté sur la carte Gidel PROCSpark : elle possède une mémoire RAM avec un contrôleur, l'interface PCI est fournie, le prix est relativement bas, elle a assez de portes logiques pour mettre en œuvre chacun de nos algorithmes de traitement d'images, et il existe un support technique actif.

5.4. API Novakod pour la carte Gidel PROCSpark

Il s'agit d'un ensemble de fonctions C++ ayant pour but de permettre les communications et transferts de données entre une application s'exécutant sur un ordinateur PC et un programme psC s'exécutant sur FPGA; les échanges de données ou informations peuvent porter par exemple sur les opérations de lecture et écriture en mémoire du FPGA, le signalement au PC de la fin de calcul d'un algorithme par le FPGA, le transfert des paramètres pour le traitement d'images depuis le PC à destination du FPGA.

Quelques fonctions couramment utilisées lors des opérations de traitement d'images avec la carte Gidel PROCSpark sont présentées à l'annexe C.

5.5. Groupes de traitements

L'analyse des algorithmes de traitement d'images présentés au chapitre IV a permis de tirer les conclusions suivantes par rapport à une mise en œuvre sur la carte Gidel PROCSpark:

- Les algorithmes, pris individuellement, ne sont pas assez gourmands en ressources matérielles pour nécessiter la quasi-totalité des circuits logiques disponibles sur la carte Gidel;
- La carte Gidel ne possède pas suffisamment de blocs logiques pour supporter simultanément tous les algorithmes de traitement d'images sélectionnés.

Fort de ce constat, nous avons choisi de regrouper autant que possible les algorithmes entre-eux afin d'utiliser au maximum la quantité de blocs logiques programmables disponibles sur la carte Gidel PROCSpark. Les regroupements vont permettre d'effectuer des traitements individuellement ou en combinaison. Trois grands groupes de traitement ont été retenus et ils sont présentés dans les paragraphes suivants.

5.5.1. Combinaison « brillance, contraste, gamma »

Cette combinaison permet d'effectuer les traitements suivants : brillance, contraste, gamma, brillance suivie de contraste, brillance suivie de gamma, contraste suivi de gamma et brillance suivie de contraste puis gamma. Elle nécessite 3 960 éléments logiques sur les 4 160 disponibles sur la carte Gidel PROCSpark (95% d'utilisation des éléments logiques de la carte).

Mathématiquement parlant, le calcul effectué par cette combinaison peut être représenté comme suit :

$$output[a][b] = \left(\left(input[a][b] + brillance \right) \times contraste / 255 \right)^{\gamma} \times 255$$

γ est le facteur utilise pour le traitement gamma.

Le tableau suivant présente les valeurs à affecter spécifiquement à un ou plusieurs paramètres afin de réaliser les différents traitements ou combinaisons de traitements qui ont été indiquées précédemment.

Traitement	Paramètres à modifier
Brillance	$contraste = 1, \quad \gamma = 1$
Contraste	$brillance = 0, \quad \gamma = 1$
Gamma	$brillance = 0, \quad contraste = 1$
Brillance + Contraste	$\gamma = 1$
Brillance + Gamma	$contraste = 1$
Contraste + Gamma	$brillance = 0$
Brillance + Contraste + Gamma	

Tableau 6 : Paramètres pour la chaîne de traitements « brillance, contraste, gamma »

Lorsque les traitements sont effectués en combinaison, les calculs se font en pipeline d'après les schémas de la page suivante qui représentent l'architecture finale utilisée pour les calculs de la combinaison brillance, contraste et gamma.

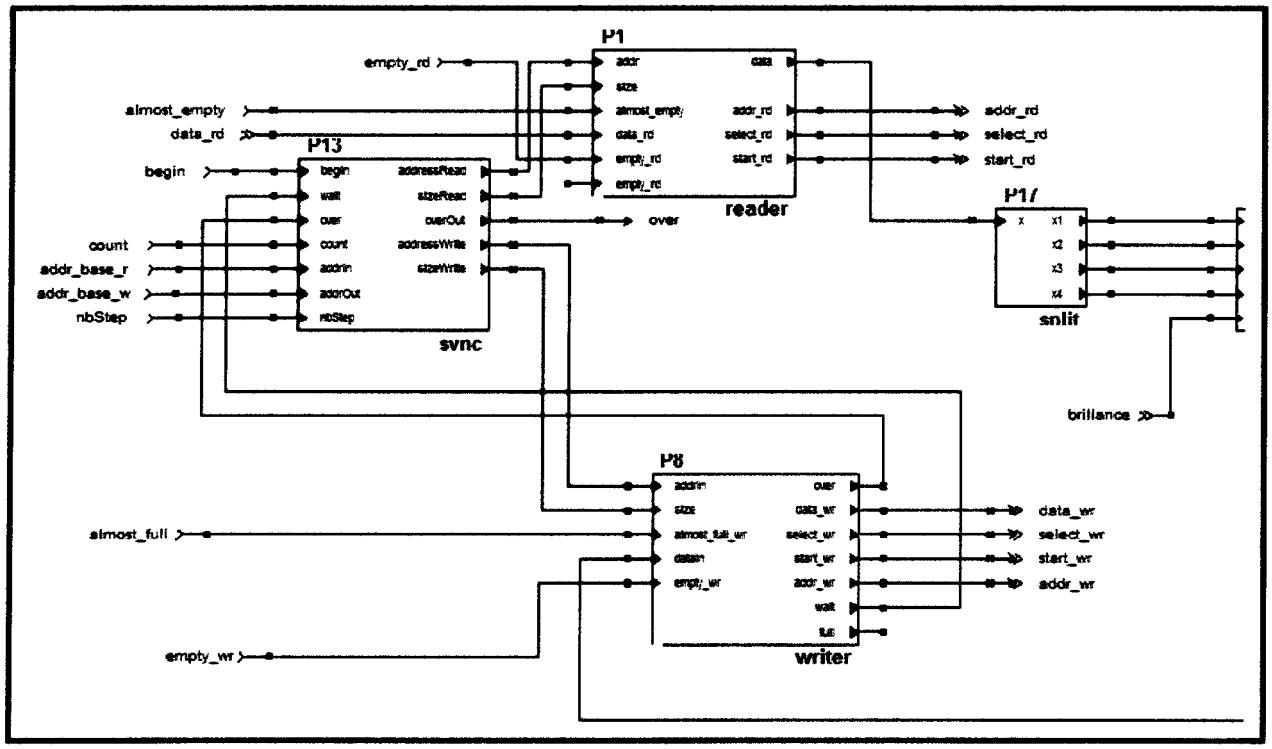

Figure 18 : Brillance, contraste, gamma

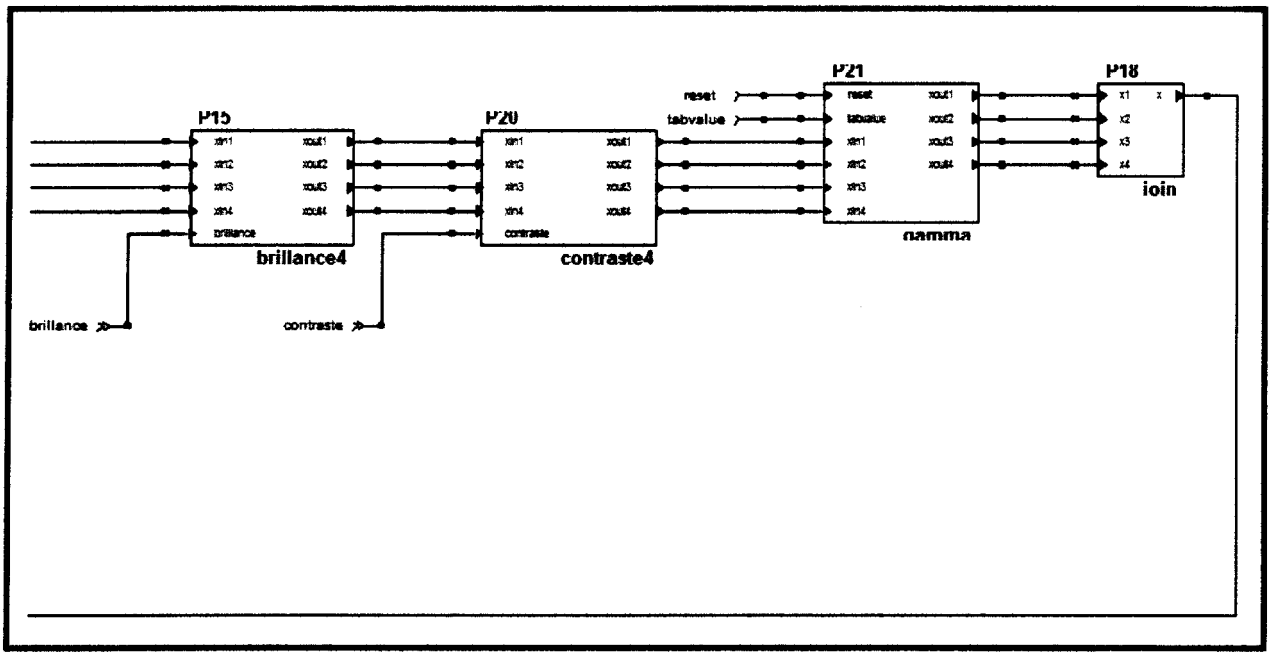

Figure 19 : Brillance, contraste, gamma (suite)

5.5.1.1. Description des composants de l'architecture de calcul

5.5.1.1.1. Composant « sync »

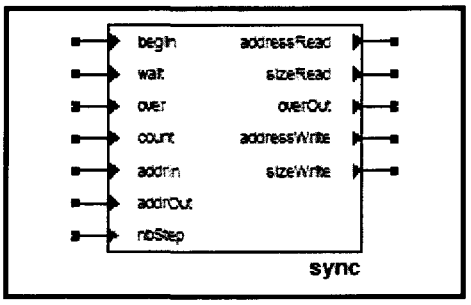

Figure 20 : Composant « sync »

Ce composant s'occupe principalement de gérer la synchronisation entre les accès en lecture/écriture de la mémoire de la carte Gidel. Cette synchronisation est nécessaire en raison du fait que les données lues ou écrites en mémoire se font de manière séquentielle et, dans ce mode, le contrôleur multiports fourni avec la carte Gidel utilise deux tampons mémoires (un pour la lecture et un pour l'écriture) qui ne peuvent accéder à la mémoire que de façon exclusive. Le composant possède aussi des ports chargés d'initier le lancement des opérations de traitements et d'envoyer les signaux de fin d'exécution des calculs. Ci-dessous sont présentés quelques ports du composant « sync ».

begin : associé au port d'entrée « begin », ce port déclenche l'exécution des calculs lorsqu'il reçoit un évènement. L'évènement en question est envoyé par l'ordinateur et transféré à travers le port d'entrée « begin ».

wait : drapeau permettant d'attendre que les tampons de lecture/écriture se vident ou se remplissent pour assurer le bon traitement des données en mémoire.

count : taille totale des données à lire/écrire en mémoire; ce sera en général la taille 32 bits de l'image d'entrée. Ce port reçoit sa donnée à travers le port d'entrée « count » qui est initialisé par l'application de traitement d'images s'exécutant sur ordinateur PC.

addrIn : associé au port d'entrée « addr_base_r », ce port contient l'adresse où commencent les données correspondant aux pixels de l'image d'entrée en mémoire FPGA.

addrOut : associé au port d'entrée « addr_base_w », ce port contient l'adresse à partir de laquelle les données correspondant aux pixels de l'image de sortie vont être écrites en mémoire FPGA.

overOut : ce port est mis à 1 lorsque les traitements sont terminés, ce qui permet au port de sortie « over » de recevoir un évènement qui sera redirigé ensuite vers l'ordinateur.

addressRead : adresse de lecture d'une donnée en mémoire.

addressWrite : adresse où une donnée calculée va être écrite en mémoire.

sizeRead: nombre de blocs de données 32 bits lues simultanément en mémoire à destination des tampons du contrôleur multiports.

sizeWrite : nombre de blocs de données 32 bits écrites simultanément en mémoire à destination des tampons du contrôleur multiports.

5.5.1.1.2. Composant « reader »

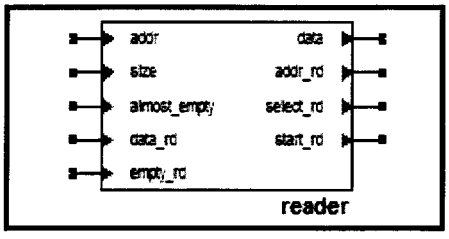

Figure 21 : Composant « reader »

Ce composant communique avec le contrôleur multiports pour effectuer des opérations de lecture de données en mémoire FPGA. La description des ports de ce composant est présentée ci-dessous :

addr : adresse de lecture d'une donnée en mémoire, est fournie par le composant « sync ».

size : taille des données à lire en mémoire.

data_rd : donnée lue en mémoire et fournie par le contrôleur multiports.

data: donnée lue en mémoire et fournie par le port « data_rd »

addr_rd : adresse de lecture d'une donnée en mémoire, elle est fournie par le port « addr » et est associée au port de sortie « addr_rd ».

almost_empty, *empty_rd*, *select_rd*, *start_rd*: drapeaux utilisés par le contrôleur multiports pour gérer les instants de lecture/écriture en mémoire.

5.5.1.1.3. Composant « writer »

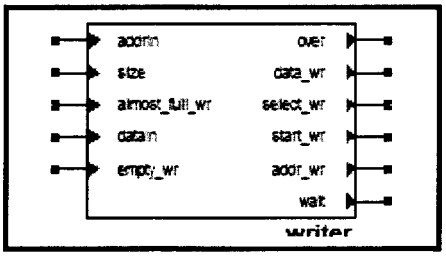

Figure 22 : Composant « writer »

Ce composant communique avec le contrôleur multiports pour effectuer des opérations d'écriture de données en mémoire FPGA. La description des ports de ce composant est présentée ci-dessous :

addrIn : adresse d'écriture d'une donnée en mémoire, est fournie par le composant « sync ».

size : taille des données à écrire en mémoire.

dataIn : donnée calculée à écrire en mémoire.

data_wr : donnée à écrire en mémoire par le contrôleur multiports et fournie par le port « dataIn ». Il est associé au port de sortie du même nom « data_wr » géré par le contrôleur multiports.

addr_wr : adresse où écrire une donnée en mémoire. Ce port est associé au port de sortie du même nom « addr_wr » géré par le contrôleur multiports.

almost_full_wr, *empty_wr*, *select_wr*, *start_wr*: drapeaux utilisés par le contrôleur multiports pour gérer les instants de lecture/écriture en mémoire.

5.5.1.1.4. Composant « split »

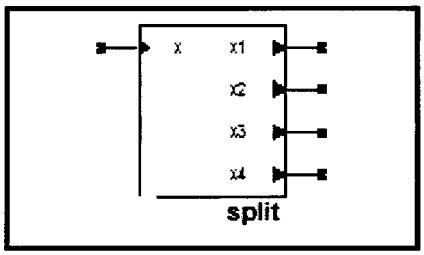

Figure 23 : Composant « split »

Il permet de dissocier une donnée 32 bits en quatre données de 8 bits. La mémoire de la carte Gidel est 32 bits, ce qui signifie qu'on ne peut lire et écrire en mémoire que des données de taille 32 bits; cependant, les pixels sur lesquels nous effectuons nos calculs ont une taille de 8 bits, d'où la nécessité d'avoir un composant qui, à partir d'une donnée de 32 bits lues en mémoire, va permettre d'extraire les valeurs de niveau de gris sur 8 bits des quatre pixels regroupés dans les 32 bits.

Les ports du composant sont décrits comme suit :

x : donnée sur 32 bits qui doit être éclatée en quatre données de 8 bits chacune.

x1, *x2*, *x3*, *x4* : 1ère, 2ième, 3ième et 4ième données extraites de 8 bits.

5.5.1.1.5. Composant « brillance4 »

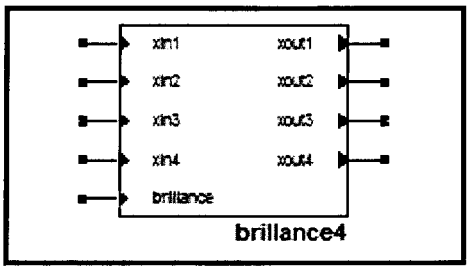

Figure 24 : Composant « brillance4 »

Il permet d'effectuer un calcul de brillance conformément à la formule présentée au chapitre IV. Il agit sur 4 données en parallèle : ainsi, lorsque 4 valeurs de pixels parviennent à ce composant, il effectue le calcul de brillance simultanément sur ces 4 valeurs et retourne 4 autres données en sortie.

Les ports du composant sont les suivants :

brillance : facteur de brillance à appliquer au traitement.

xin1, *xin2*, *xin3*, *xin4* : valeurs de pixels en entrée.

xout1, *xout2*, *xout3*, *xout4* : valeurs calculées des pixels en sortie.

5.5.1.1.6. Composant « contraste4 »

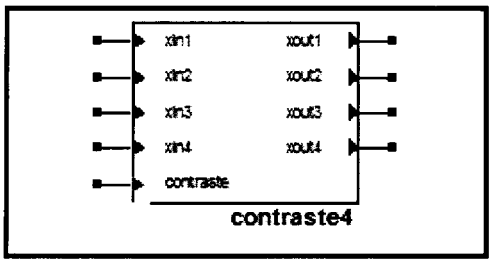

Figure 25 : Composant « contraste4 »

Il permet d'effectuer un calcul de contraste conformément à la formule présentée au chapitre IV. A l'instar du composant « brillance4 », il agit sur 4 données en parallèle : ainsi, lorsque 4 valeurs de pixels parviennent à ce composant, il effectue le calcul de contraste simultanément sur ces 4 valeurs et retourne 4 autres données en sortie.

Les ports du composant sont les suivants :

contraste : facteur de contraste à appliquer au traitement.

xin1, *xin2*, *xin3*, *xin4* : valeurs de pixels en entrée.

xout1, *xout2*, *xout3*, *xout4* : valeurs calculées des pixels en sortie.

5.5.1.1.7. Composant « gamma »

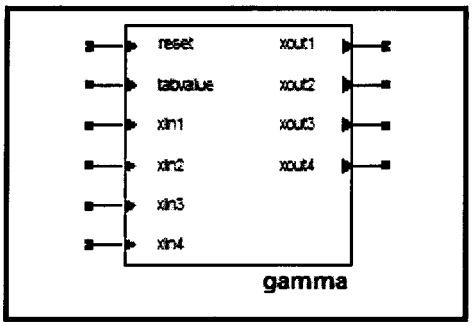

Figure 26 : Composant « gamma »

Il permet d'effectuer un calcul de gamma. Cependant, au lieu d'utiliser la formule présentée au chapitre IV, nous utilisons plutôt une table de valeurs. La raison de ce choix est liée au fait que l'opérateur de calcul de puissance n'est pas encore supporté par le langage psC, ce qui nous empêche d'y faire appel dans nos programmes. Pour remédier au problème, on se charge d'effectuer sur ordinateur tous les calculs de gamma pour les $2^8 = 256$ valeurs de niveau de gris de pixels, puis les valeurs calculées sont transférées dans un tableau de valeurs que nous pouvons lire à partir de psC. Dès lors, pour chaque valeur *xin* de pixel en entrée du composant, celui-ci génère en sortie la donnée du tableau se trouvant à l'index *xin*. Le composant gamma, comme les deux composants précédents, traite quatre données simultanément.

Les ports du composant sont les suivants :

gamma : facteur gamma à appliquer au traitement.

xin1, *xin2*, *xin3*, *xin4* : valeurs de pixels en entrée.

xout1, *xout2*, *xout3*, *xout4* : valeurs calculées de pixels en sortie qui seront écrites ultérieurement en mémoire.

reset : signale qu'un nouveau tableau de valeurs va être rempli pour le traitement gamma.

tabvalue : chaque nouvelle donnée du tableau de valeurs calculées sur PC est transmise à travers ce port avant de pouvoir être stockée dans un tableau en mémoire FPGA.

5.5.1.1.8. Composant « join »

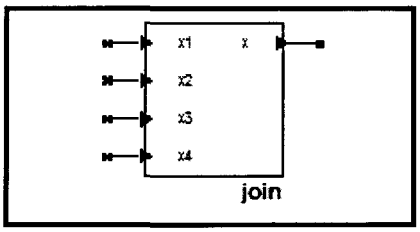

Figure 27 : Composant « join »

A l'opposée du composant « split », il permet de rassembler quatre données de 8 bits chacune en une donnée 32 bits. Les composants de calcul (brillance, contraste, gamma) travaillent sur des données de 8 bits, tandis que la mémoire de la carte Gidel est 32 bits; ce composant permet ainsi de regrouper les données calculées en 8 bits pour qu'elles se présentent dans un format 32 bits nécessaire pour être écrites en mémoire.

Les ports du composant sont décritscomme suit :

x1, *x2*, *x3*, *x4* : 1ère, 2ième, 3ième et 4ième données de 8 bits qui doivent être fusionnées en 32 bits.

x : donnée fusionnée sur 32 bits.

5.5.1.2. Étapes d'exécution des opérations de la combinaison des traitements

Pour réaliser le traitement complet d'une image (du chargement de l'image en mémoire du FPGA à l'affichage sur l'écran d'un ordinateur de l'image résultante), plusieurs étapes doivent être franchies :

i. L'image est transférée dans la mémoire de la carte PROCSpark de Gidel par DMA (*Direct Memory Access* / accès direct à la mémoire) à l'aide des fonctions de l'API Novakod;

ii. Les paramètres de la taille de l'image ainsi que l'adresse de base de l'image originale et de celle traitée sont envoyés par la partie logicielle;

iii. Les paramètres de brillance, contraste et gamma sont également assignés sur les ports par la partie logicielle (API sur ordinateur PC) ; certains paramètres auront des valeurs prédéfinies suivant le traitement ou la série de traitements à effectuer;

iv. Le signal de départ correspondant au port « begin » est donné;

v. La partie matérielle communique, à travers les composants « reader » et « writer », avec le contrôleur « multiports » de lecture/écriture fourni par la carte PROCSpark pour lire les données de la mémoire. Elles sont lues par paquet de 100 données de 32 bits, un paquet contient quatre pixels de 8 bits;

vi. Les quatre pixels sont ensuite traités en parallèle par les modules de brillance, contraste et gamma. Ceux-ci ont la capacité de traiter 4 pixels à chaque pas;

vii. Les pixels traités sont ensuite écrits dans la mémoire par le module « writer » à l'aide du contrôleur « multiports »;

viii. Les étapes v à vii sont répétées jusqu'à ce que les données lues atteignent la taille de l'image;

ix. Un signal est transmis par la carte pour signifier que les traitements sont terminés : cela est fait par l'intermédiaire du port de sortie « over »;

x. La partie logicielle lit la nouvelle image dans la mémoire de la carte PROCSpark de Gidel par DMA à l'aide des fonctions de l'API Novakod;

xi. L'image lue est finalement affichée à l'écran.

5.5.2. Combinaison « brillance, fenêtrage, gamma »

Cette combinaison permet d'effectuer les traitements suivants : brillance, fenêtrage, gamma, brillance suivie de fenêtrage, brillance suivie de gamma, fenêtrage suivi de gamma et brillance suivie de fenêtrage puis gamma. Elle nécessite 3 842 éléments logiques sur les 4 160 disponibles sur la carte Gidel PROCSpark (92 % d'utilisation des éléments logiques de la carte).

Dans les figures 28 et 29 (page 68) illustrant l'architecture de fonctionnement de la combinaison des traitements, en ce qui concerne les ports associés au calcul du fenêtrage, on retrouve les variables suivantes calculées à partir des paramètres décrits au chapitre IV où l'opération du fenêtrage est présentée :

$$low\_limit = WinCenter - \frac{WinWidth}{2}$$

$$high\_limit = WinCenter + \frac{WinWidth}{2}$$

$$factor = \frac{255}{WinWidth}$$

WinCenter est le centre du fenêtrage et *WinWidth* est sa largeur.

Afin de bien effectuer le calcul individuel (mais aussi chaîné) des différentes opérations de la combinaison à l'aide de la même architecture systolique, nous avons eu à modifier le

codage du fenêtrage pour qu'il retourne en sortie le pixel en entrée lorsque *low_limit = 0* et *high_limit = 0*. Ceci résulte du fait qu'il n'existe pas de valeurs prédéfinies du centre et de la largeur du fenêtrage pour lesquelles chaque donnée en entrée produit la même donnée en sortie.

Le tableau suivant présente les valeurs à affecter spécifiquement à un ou plusieurs paramètres afin de réaliser les différents traitements ou combinaisons de traitements qui ont été indiquées précédemment.

Traitement	Paramètres à modifier
Brillance	*low_limit = 0, high_limit = 0, γ = 1*
Fenêtrage	*brilliance = 0, γ = 1*
Gamma	*brilliance = 0, low_limit = 0, high_limit = 0*
Brillance + Fenêtrage	*γ = 1*
Brillance + Gamma	*low_limit = 0, high_limit = 0*
Fenêtrage + Gamma	*brillance* = 0
Brillance + Fenêtrage + Gamma	

Tableau 7 : Paramètres pour la chaîne de traitements « brillance, contraste, gamma »

Lorsque les traitements sont effectués en combinaison, les calculs se font en pipeline d'après les schémas de la page suivante qui représentent l'architecture finale utilisée pour les calculs de la combinaison brillance, fenêtrage et gamma.

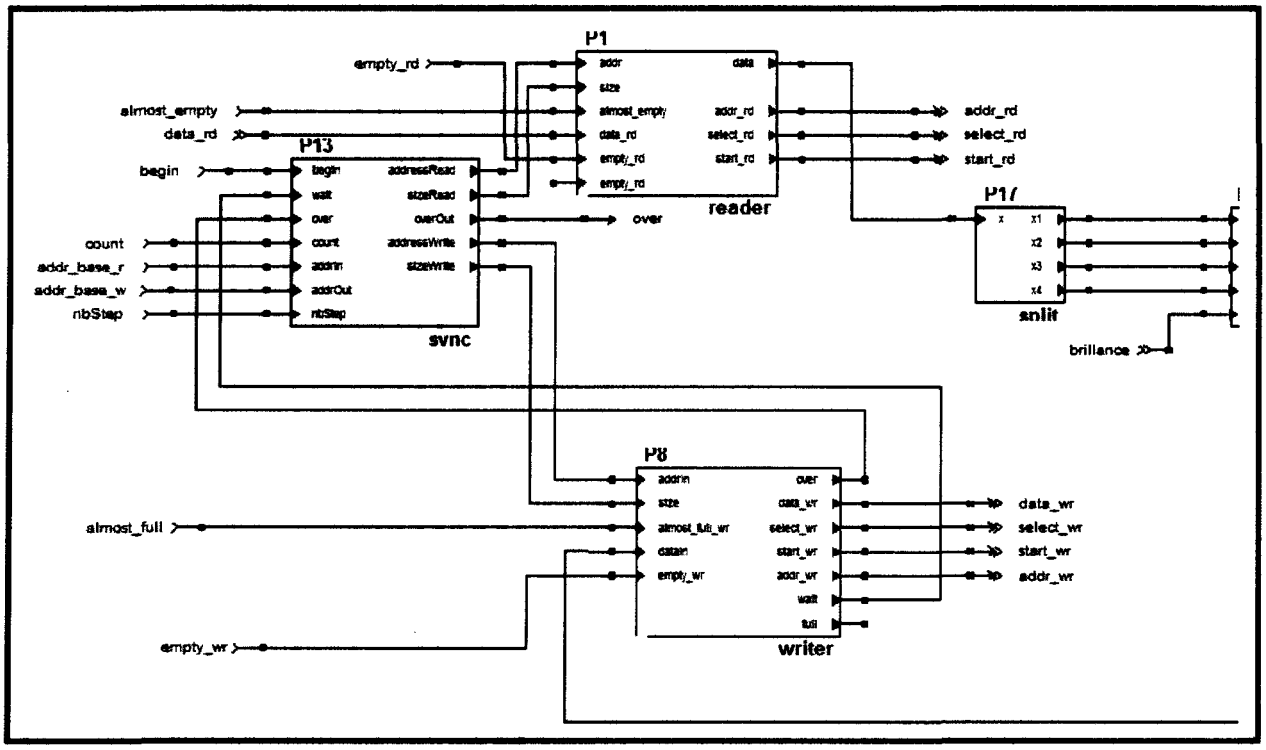

Figure 28 : Brillance, fenêtrage, gamma

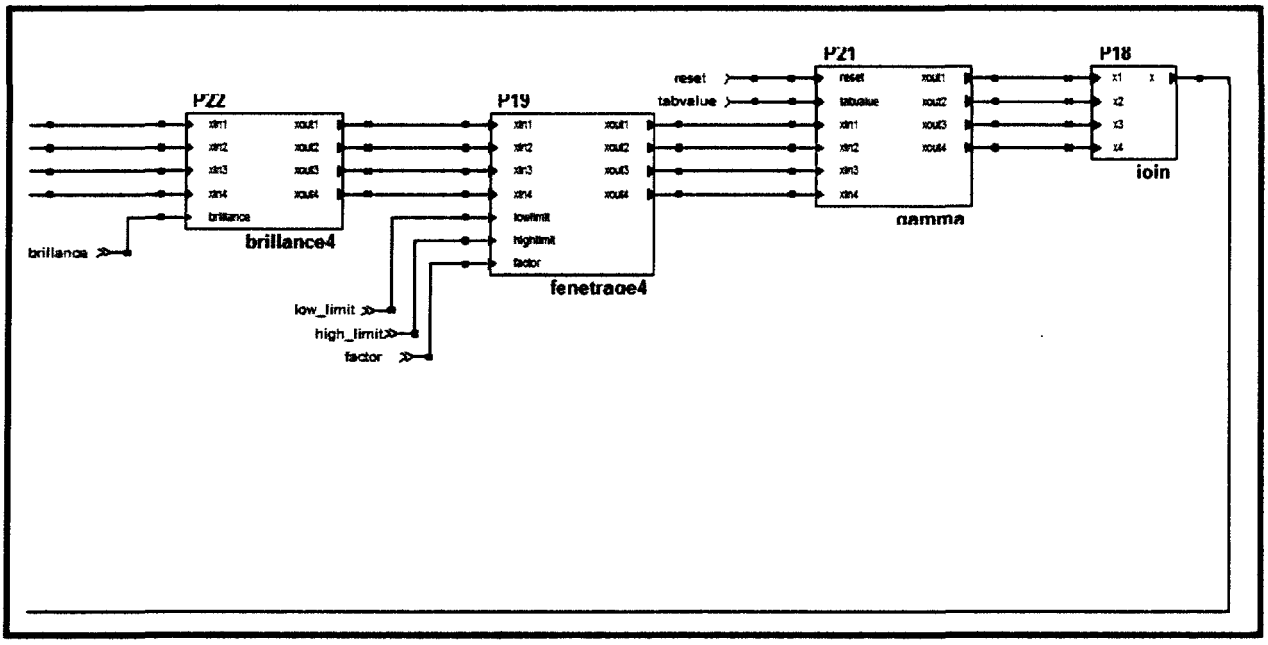

Figure 29 : Brillance, fenêtrage, gamma (suite)

5.5.2.1. Description des composants de l'architecture de calcul

5.5.2.1.1. Composants « sync », « reader », « writer », « split », « join », « brillance4 », « gamma »

Ces composants sont exactement les mêmes que ceux présentés précédemment dans le cas de l'architecture de calcul de la combinaison des traitements « brillance, contraste, gamma ».

5.5.2.1.2. Composant « fenêtrage4 »

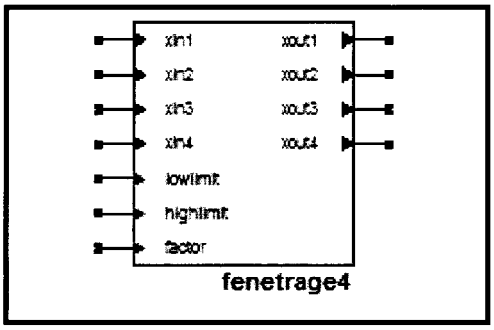

Figure 30 : Composant « fenêtrage4 »

Il permet d'effectuer un calcul de fenêtrage conformément à la formule présentée au chapitre IV. Il agit sur 4 données en parallèle : ainsi, lorsque 4 valeurs de pixels parviennent à ce composant, il effectue le calcul de fenêtrage simultanément sur ces 4 valeurs et retourne 4 autres données en sortie.

Les ports du composant sont les suivants :

factor : en l'absence du support de l'opérateur de division pour FPGA dans psC, ce paramètre a été défini pour stocker la valeur $\frac{255}{WinWidth}$ apparaissant dans l'algorithme du fenêtrage.

low_limit : borne inférieure de l'intervalle (la fenêtre) des valeurs de niveau de gris de pixels à considérer. Cette borne est égale à $WinCenter - \frac{WinWidth}{2}$, avec WinCenter représentant le centre de la fenêtre et WinWidth sa largeur (voir le chapitre IV pour les détails de la formule de calcul du fenêtrage).

high_limit : borne supérieure de l'intervalle (la fenêtre) des valeurs de niveau de gris de pixels à considérer. Elle est égale à $WinCenter + \frac{WinWidth}{2}$

xin1, *xin2*, *xin3*, *xin4* : valeurs de pixels en entrée.

xout1, *xout2*, *xout3*, *xout4* : valeurs calculées des pixels en sortie.

5.5.2.2. Étapes d'exécution des opérations de la combinaison des traitements

Pour réaliser le traitement complet d'une image (du chargement de l'image en mémoire du FPGA à l'affichage sur l'écran d'un ordinateur de l'image résultante), plusieurs étapes doivent être franchies :

i. L'image est transférée dans la mémoire de la carte PROCSpark de Gidel par DMA (accès direct à la mémoire) à l'aide des fonctions de l'API Novakod;
ii. Les paramètres de la taille de l'image ainsi que l'adresse de base de l'image originale et de celle traitée sont envoyés par la partie logicielle;

iii. Les paramètres de brillance, fenêtrage et gamma sont également assignés sur les ports par la partie logicielle (API sur ordinateur PC) ; certains paramètres auront des valeurs prédéfinies suivant le traitement ou la série de traitements à effectuer;

iv. Le signal de départ correspondant au port « begin » est donné;

v. La partie matérielle communique, à travers les composants « reader » et « writer », avec le contrôleur « multiports » de lecture/écriture fourni par la carte PROCSpark pour lire les données de la mémoire. Elles sont lues par paquet de 100 données de 32 bits, un paquet contient quatre pixels de 8 bits;

vi. Les quatre pixels sont ensuite traités en parallèle par les modules de brillance, fenêtrage et gamma. Ceux-ci ont la capacité de traiter 4 pixels à chaque pas;

vii. Les pixels traités sont ensuite écrits dans la mémoire par le module « writer » à l'aide du contrôleur « multiports »;

viii. Les étapes v à vii sont répétées jusqu'à ce que les données lues atteignent la taille de l'image;

ix. Un signal est transmis par la carte pour signifier que les traitements sont terminés : cela est fait par l'intermédiaire du port de sortie « over »;

x. La partie logicielle lit la nouvelle image dans la mémoire de la carte PROCSpark de Gidel par DMA à l'aide des fonctions de l'API Novakod;

xi. L'image lue est finalement affichée à l'écran.

5.5.3. Combinaison « brillance, convolution »

Cette combinaison permet d'effectuer les traitements suivants : brillance, convolution, brillance suivie de convolution. Elle nécessite 3 418 éléments logiques sur les 4 160 disponibles sur la carte Gidel PROCSpark (82 % d'utilisation des éléments logiques de la carte).

Mathématiquement parlant, le calcul effectué par cette combinaison peut être représenté comme suit :

$$output[i,j] = \sum_{s=1}^{3}\sum_{t=1}^{3} w[s,t] \times (brillance + input[i+s-1, j+t-1])$$

Le tableau suivant présente les valeurs à affecter spécifiquement à un ou plusieurs paramètres afin de réaliser les différents traitements ou combinaisons de traitements.

Traitement	Paramètres à modifier
Brillance	$w[1,1]=0,\ w[1,2]=0,\ w[1,3]=0$ $w[2,1]=0,\ w[2,2]=1,\ w[2,3]=0$ $w[3,1]=0,\ w[3,2]=0,\ w[3,3]=0$
Convolution	$brillance = 0$
Brillance + Convolution	

Tableau 8 : Paramètres pour la chaîne de traitements « brillance, convolution »

Lorsque les traitements sont effectués en combinaison, les calculs se font en pipeline d'après les schémas de la page suivante qui représentent l'architecture finale utilisée pour les calculs de la combinaison brillance, contraste et gamma.

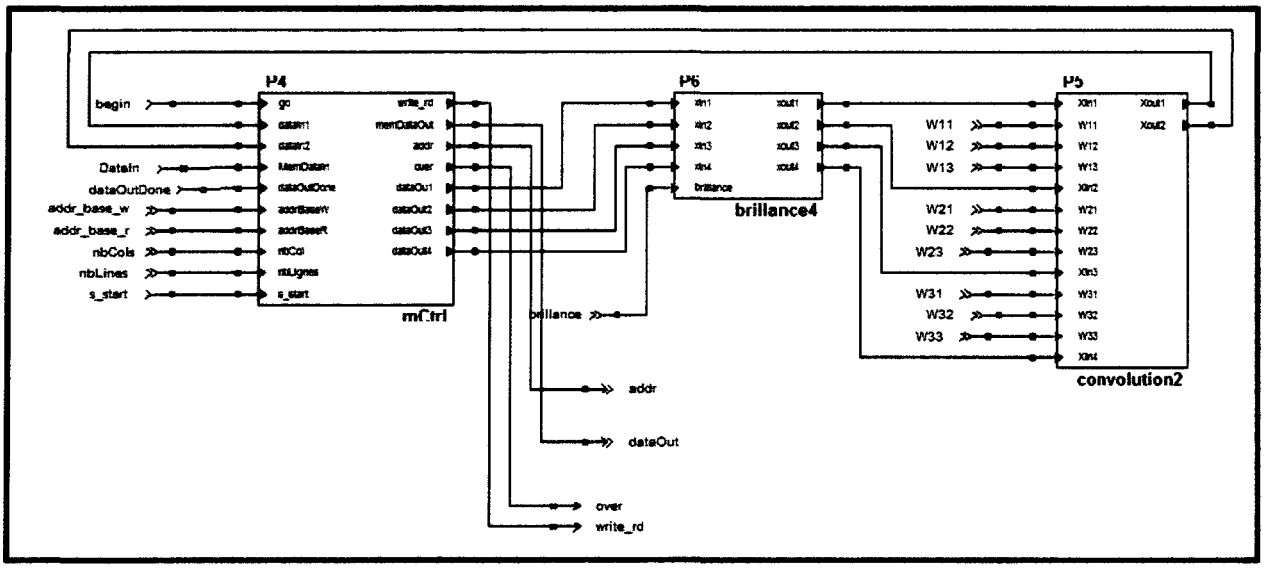

Figure 31 : Brillance, convolution

5.5.3.1. Description des composants de l'architecture de calcul

5.5.3.1.1. Composants « mCtrl »

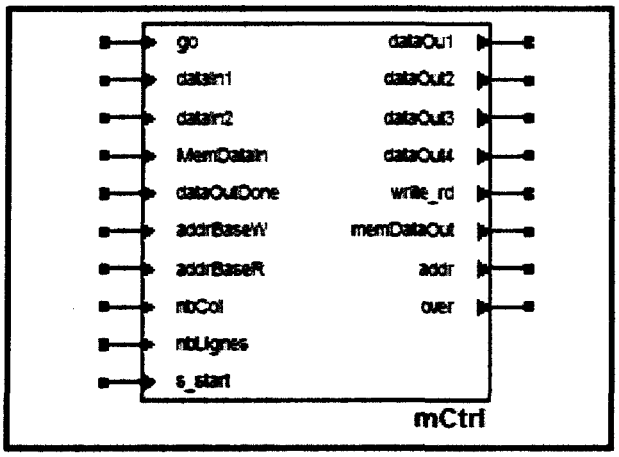

Figure 32 : Composant « contrôleur multiports »

Il s'agit du composant d'interface avec le contrôleur multiports. Il se charge des opérations de lecture/écriture en mémoire. Les principaux ports de ce composant sont les suivants :

go : port chargé de lancer les opérations de traitement d'images par convolution.

dataIn1 : donnée de 8 bits calculée par convolution et à écrire en mémoire.

dataIn2 : autre donnée de 8 bits.

MemDataIn : donnée de 32 bits lue en mémoire : ce sont les pixels de l'image d'entrée.

MemDataOut : donnée de 32 bits écrite en mémoire : ce sont les pixels de l'image de sortie.

addrBaseW : adresse mémoire où sera écrite l'image de sortie.

addrBaseR : adresse où est stockée l'image d'entrée en mémoire.

nbCol : nombre de colonnes de l'image d'entrée.

nbLignes : nombre de lignes de l'image d'entrée.

dataOut1, *dataOut2*, *dataOut3*, *dataOut4* : les 4 pixels extraits de la donnée MemDataIn lue en mémoire.

addr : adresse pour la lecture/écriture en mémoire.

over : port utilisé pour signifier que les calculs sont terminés.

5.5.3.1.2. Composant « brillance4 »

Ce composant est identique au composant « brillance4 » décrit dans le cadre de la combinaison des traitements « brillance, contraste, gamma ».

5.5.3.1.3. Composant « convolution »

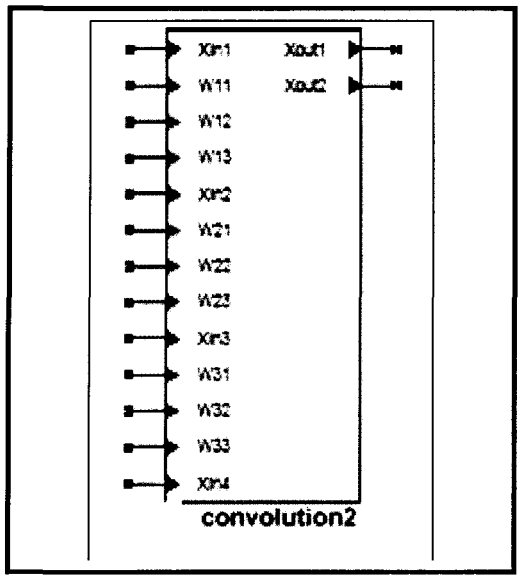

Figure 33 : Composant « convolution2 »

Il permet d'effectuer un calcul de convolution conformément à la formule présentée au chapitre IV. Il agit sur 4 données en entrée et génère 2 données en sortie.

Les ports du composant sont les suivants :

W11, *W12*, *W13*, *W21*, *W22*, *W23*, *W31*, *W32*, *W33* : les ports représentant la matrice du masque de convolution.

Xin1, *Xin2*, *Xin3*, *Xin4* : les 4 pixels en entrée qui vont permettre de produire 2 pixels en sortie. Ces pixels d'entrée proviennent de l'image d'entrée stockée en mémoire.

Xout1, *Xout2* : les pixels calculés qui seront écrits plus tard en mémoire et affichés à l'écran.

5.5.3.2. Étapes d'exécution des opérations de la combinaison des traitements

Les étapes de fonctionnement sont différentes de celles rencontrées dans les autres combinaisons de traitements décrites précédemment. Le module de convolution traite les pixels en colonne car, pour chaque pixel de l'image de sortie, il est nécessaire d'avoir les pixels de son voisinage dans l'image d'entrée pour le calcul de la valeur de niveau de gris. La lecture séquentielle des pixels en mémoire devient impossible puisqu'il faut passer d'une colonne à une autre en modifiant l'adresse de lecture à chaque fois. L'accès direct à la mémoire est donc utilisé pour l'écriture en passant par le contrôleur « multiports » offert par la carte PROCSpark. La performance est diminuée de beaucoup puisque ce type d'accès utilise un grand nombre de cycles d'horloge dans le FPGA.

Voici les étapes d'exécution :

i. L'image est transférée dans la mémoire de la carte PROCSpark de Gidel par DMA (accès direct à la mémoire);
ii. Les paramètres de la taille de l'image ainsi que l'adresse de base de l'image originale et de celle traitée sont envoyés par la partie logicielle;
iii. Les paramètres de brillance et de convolution sont également assignés sur les ports par la partie logicielle; pour obtenir uniquement le traitement de la convolution, il suffit de mettre le paramètre de brillance à 0 (de cette façon, les niveaux de gris des pixels générés en sortie par la composante *« brillance4 »* sont exactement les mêmes que ceux reçus en entrée, tout se passe alors comme si la combinaison « brillance, convolution » se ramenait à une simple opération de convolution).
iv. Le signal de lancement des calculs, correspondant au port « begin », est donné;

v. La partie matérielle communique avec le contrôleur « multiports » de lecture/écriture fourni par la carte PROCSpark pour lire les données en mode d'accès direct dans la mémoire. Les données sont lues en paquets de 32 bits correspondant à quatre pixels (chaque pixel est représenté sur 8 bits);
vi. Trois pixels sont simultanément pris en compte par la convolution, ce qui permet de générer en sortie un pixel. Notons que nous ne considérons que trois pixels en raison du fait que la carte FPGA utilisée ne possède pas assez de blocs logiques pour supporter une convolution qui génère plus d'un pixel en sortie.
vii. Les pixels traités sont ensuite écrits à la mémoire par le mode d'accès direct;
viii. Les étapes 5 à 7 sont répétées jusqu'à ce que les données lues atteignent la taille de l'image;
ix. Un signal est transmis par la carte pour signifier que les traitements sont terminés ;
x. La partie logicielle lit la nouvelle image dans la mémoire de la carte PROCSpark de Gidel par DMA.
xi. L'image est affichée à l'écran après deux rotations horizontale et verticale.

5.5.4. Traitement du zoom

Contrairement aux autres traitements où on parcourt l'image d'entrée pour calculer l'image de sortie, le zoom procède autrement : on parcourt les pixels de l'image de sortie par ligne et par colonne; pour chaque pixel *xout*, on détermine les pixels de l'image d'entrée qui serviront de voisinage au calcul du niveau de gris associé audit pixel *xout*. À

cause de ce mode de calcul particulier, le traitement par le zoom n'a pas pu être associé aux autres types de traitement.

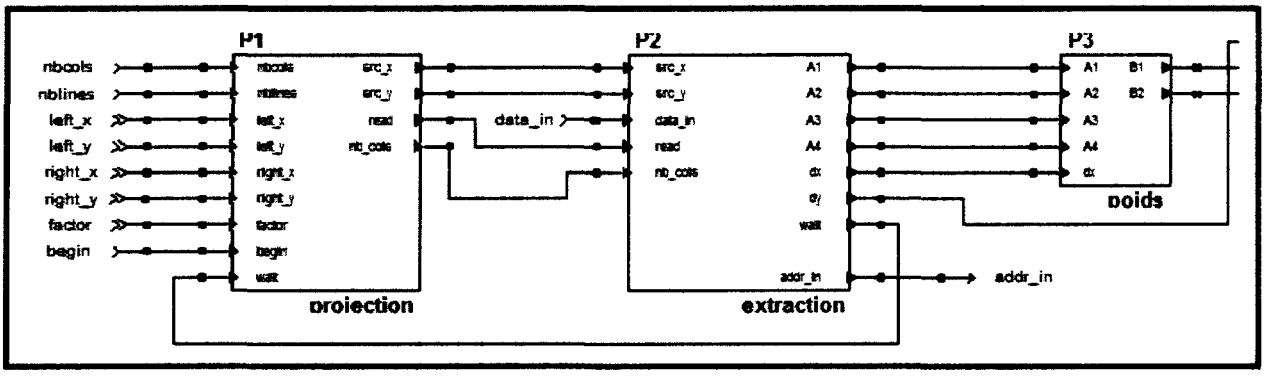

Figure 34 : Zoom

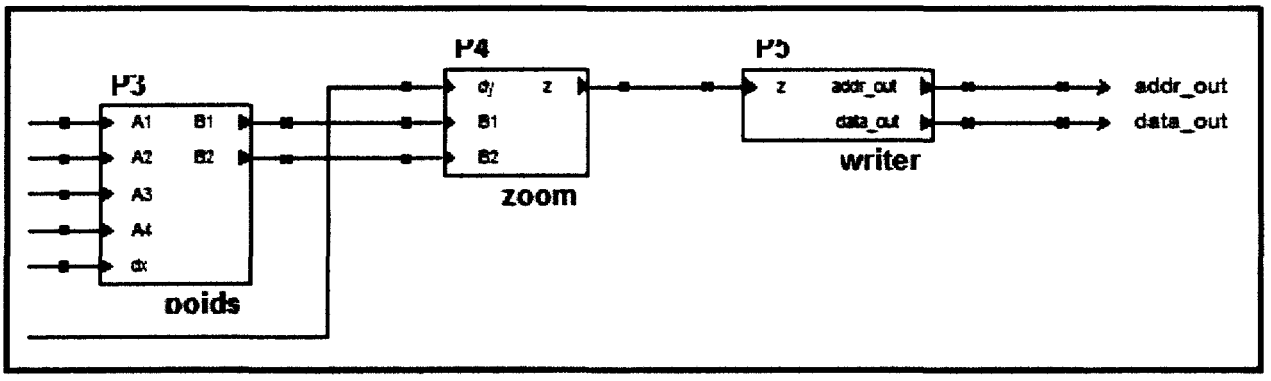

Figure 35 : Zoom (suite)

La configuration de traitement par le zoom a été testée en simulation avec une mémoire ne nécessitant qu'un cycle pour la lecture/écriture. La configuration n'a jamais été testée sur un circuit FPGA réel en raison du fait que nous ne disposions pas d'une mémoire physique à accès direct monocycle : le zoom procède aussi par voisinage comme la convolution et les difficultés rencontrées dans la mise en oeuvre du zoom nous ont amené à prendre conscience de la nécessité d'avoir une mémoire très rapide en accès direct.

5.5.4.1. Description des composants de l'architecture de calcul

5.5.4.1.1. Composants « projection »

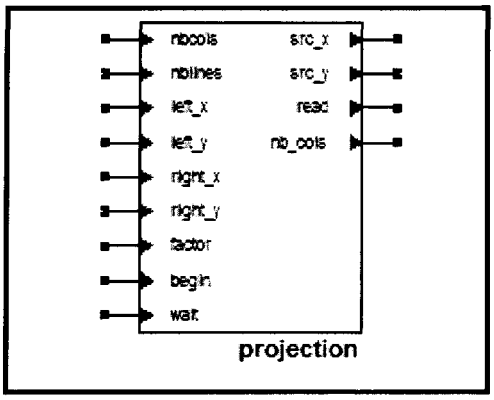

Figure 36 : Composant « projection »

Ce composant effectue le calcul de projection (il ne s'agit pas de la projection au sens mathématique, mais d'une opération de correspondance entre des images de tailles différentes) qui permet de déterminer le correspondant d'un pixel de l'image de sortie dans l'image d'entrée : étant donné que les images d'entrée et de sortie peuvent avoir des tailles différentes, la projection est calculée en utilisant une « règle de trois »; ainsi, si x' et y' sont les coordonnées (colonne et ligne) d'un pixel dans l'image de sortie de taille L'×C' (nombre de lignes × nombre de colonnes), si x et y sont les coordonnées correspondantes dans l'image d'entrée de taille L ×C, alors x et y se calculent par les formules :

$$x = x' \times \frac{C}{C'} \quad et \quad y = y' \times \frac{L}{L'}$$

Les principaux ports de ce composant sont les suivants :

begin : port chargé de lancer les opérations de traitement d'images par zoom.

nbCols : nombre de colonnes de l'image d'entrée.

nbLines : nombre de lignes de l'image d'entrée.

left_x : abscisse du coin supérieur gauche délimitant la zone à zoomer dans l'image d'entrée.

left_y : ordonnée du coin supérieur gauche délimitant la zone à zoomer dans l'image d'entrée.

right_x : abscisse du coin inférieur droit délimitant la zone à zoomer dans l'image d'entrée.

right_y : ordonnée du coin inférieur droit délimitant la zone à zoomer dans l'image d'entrée.

factor : inverse du facteur de zoom : ainsi, si on veut faire un zoom par 2 d'une portion de l'image d'entrée, factor prendra la valeur 1/2=0,5. Nous procédons de la sorte à cause d'un problème rencontré lors de la programmation et qui était dû à l'indisponibilité de l'opération de division dans psC.

wait : signal d'attente permettant de ralentir la cadence de lecture/écriture des données en mémoire.

src_x : valeur réelle de l'abscisse correspondant à la projection d'un pixel de l'image de sortie dans l'image d'entrée. Par rapport à la formule présentée à la page 29, *src_x* représente la donnée *x*.

src_y : valeur réelle de l'ordonnée correspondant à la projection d'un pixel de l'image de sortie dans l'image d'entrée. Par rapport à la formule présentée à la page 29, *src_y* représente la donnée *y*.

read : signal permettant de ne plus lire en mémoire des pixels issus d'un voisinage qui aurait déjà été parcouru.

nb_cols : nombre de colonnes de l'image d'entrée que l'on divise par 4 (les données sont lues et écrites en blocs de $4 \times 8 = 32$ bits).

5.5.4.1.2. Composant « extraction »

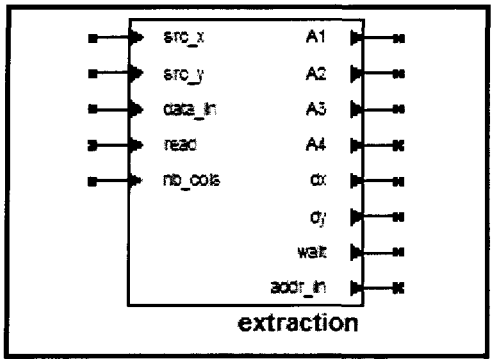

Figure 37 : Composant « extraction »

Ce composant est chargé d'aller lire en mémoire les pixels de l'image d'entrée nécessaires au calcul des pixels de l'image de sortie. Les principaux ports de ce composant sont les suivants :

data_in : donnée 32 bits lue en mémoire et correspondant à quatre pixels de l'image d'entrée.

A1, *A2*, *A3*, *A4* : 1er, 2ième, 3ième et 4ième pixels du voisinage nécessaire au calcul d'un pixel de l'image de sortie. Par rapport aux formules de la page 29, on a les correspondances suivantes :

$$A1 = I(i, j)$$
$$A2 = I(i+1, j)$$
$$A3 = I(i, j+1)$$
$$A4 = I(i+1, j+1)$$

dx : partie décimale de l'abscisse issue de la projection d'un pixel de l'image de sortie. Son calcul est présenté à la page 29.

dy : partie décimale de l'ordonnée issue de la projection d'un pixel de l'image de sortie. Son calcul est présenté à la page 29.

addr_in : adresse de lecture d'une donnée 32 bits (quatre pixels) en mémoire.

5.5.4.1.3. Composant « poids »

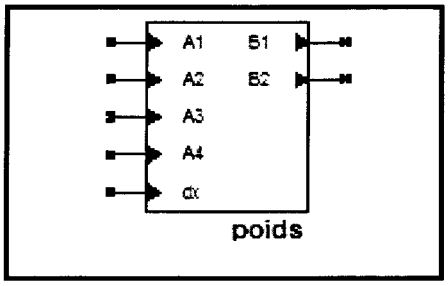

Figure 38 : Composant « poids »

Il permet d'effectuer un calcul de poids conformément à la formule présentée au chapitre IV. Les principaux ports du composant sont les suivants :

A1, *A2*, *A3*, *A4* : 1er, 2ième, 3ième et 4ième pixels du voisinage nécessaire au calcul d'un pixel de l'image de sortie.

dx : partie décimale de l'abscisse issue de la projection d'un pixel de l'image de sortie.

B1, *B2* : données pondérées calculées à partir de A1, A2, A3, A4 et dx. Les formules de la page 29 se réécrivent comme suit :

$$B1 = (1 - dx) \times A1 + dx \times A2$$
$$B2 = (1 - dx) \times A3 + dx \times A4$$

5.5.4.1.4. Composant « zoom »

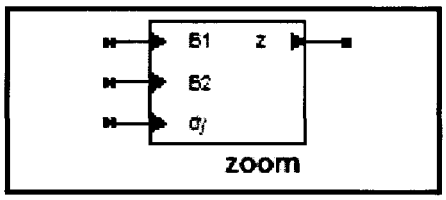

Figure 39 : Composant « zoom »

Il permet d'effectuer un calcul de poids conformément à la formule présentée au chapitre IV. Les principaux ports du composant sont les suivants :

B1, *B2* : données calculées par le composant « poids ».

dy : partie décimale de l'ordonnée issue de la projection d'un pixel de l'image de sortie.

z : valeur finale du zoom calculée par pondération à partir de B1, B2 et dy.

5.5.4.1.5. Composant « writer »

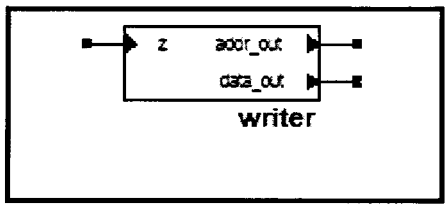

Figure 40 : Composant « writer »

Il permet d'écrire en mémoire les données calculées par l'algorithme du zoom. Les données sont écrites en 32 bits, c'est la raison pour laquelle on écrit les données du zoom par bloc de quatre. Les principaux ports du composant sont les suivants :

z : donnée 8 bits représentant un pixel de l'image de sortie.

addr_out : adresse où écrire en mémoire une donnée 32 bits de l'image de sortie.

data_out : donnée 32 bits à écrire en mémoire : cette donnée est constituée en regroupant à chaque fois quatre données représentant les pixels de l'image de sortie.

5.5.4.2. Étapes d'exécution du zoom

Le module de zoom traite les pixels en colonne car le calcul de chaque valeur de niveau de gris d'un pixel nécessite de connaître les voisins dudit pixel. La lecture séquentielle des pixels en mémoire devient impossible puisqu'il faut passer d'une colonne à une autre en modifiant l'adresse de lecture à chaque fois. L'accès direct à la mémoire est donc utilisé pour l'écriture en passant par le contrôleur « multiports » offert par la carte PROCSpark. La performance est diminuée de beaucoup puisque ce type d'accès utilise beaucoup de cycle dans le FPGA.

Les étapes de fonctionnement sont décrites ainsi qu'il suit :

i. On suppose que l'image est déjà stockée en mémoire
ii. Les paramètres sur la taille de l'image ainsi que les zones à zoomer sont transférés au programme sur FPGA;
iii. Le signal de lancement des calculs, correspondant au port « begin », est donné;
iv. L'image de sortie est parcourue pixel par pixel : pour chaque pixel *xout*, on détermine le pixel correspondant par projection dans l'image d'entrée (il ne s'agit pas de la projection au sens mathématique, mais d'une opération de correspondance entre des images de tailles différentes) : ce pixel permet de déterminer le voisinage des pixels de l'image d'entrée (il y a 4 pixels dans le voisinage) qui sera utilisé pour

déterminer le niveau de gris associé au pixel *xout* . Les pixels du voisinage sont lus par accès direct dans la mémoire.

v. Les pixels traités sont ensuite écrits, quatre à la fois, dans la mémoire par le mode d'accès séquentiel (on rappelle qu'on parcourt les pixels de l'image de sortie les uns à la suite des autres par ligne puis colonne);

vi. Les étapes iv à v sont répétées jusqu'à ce que les données lues atteignent la taille de l'image de sortie;

vii. L'image de sortie est ensuite écrite en mémoire puis affichée.

5.6. Conclusion

Dans ce chapitre, nous avons présenté les différentes architectures mises en œuvre sur les cartes FPGA Gidel pour effectuer les opérations de traitement d'images portant sur la brillance, le contraste, le fenêtrage, le gamma, la convolution et le zoom. En raison du nombre pas très élevé de ressources logiques dans les cartes utilisées, les différents algorithmes n'ont pu tous être mis ensemble dans la même architecture ou configuration; afin de pouvoir effectuer tous les traitements, il a été nécessaire de scinder les opérations de traitement en groupes de manière à utiliser dans chaque cas le maximum de ressources logiques disponibles. Toutefois, compte tenu de la scalabilité des FPGAs, il serait possible d'avoir tous les traitements dans un même programme ou diagramme si l'on disposait d'une carte FPGA dotée d'un nombre plus important de blocs logiques que ceux fournis par la carte Gidel utilisée.

CHAPITRE 6 :

RÉSULTATS : ANALYSE ET INTERPRÉTATION

6.1. Introduction

Ce chapitre présente l'ensemble des résultats des temps de traitements d'images réalisés à partir des cartes FPGAs et d'ordinateurs PC. Une série d'histogrammes sont représentés pour exposer nos données de tests et permettre des comparaisons entre les différents résultats obtenus sur les différentes plates-formes. Nous rappelons que nous utilisons des images bitmap en niveau de gris de profondeur 8 bits. Le chapitre se termine par une large analyse et interprétation des résultats dans laquelle de nombreux commentaires sont apportés sur les performances obtenues dans les traitements d'images.

6.2. Formules des temps de calculs

Les résultats des traitements d'images que nous avons effectués sont représentés dans des histogrammes. Pour chaque traitement, cinq types de résultat apparaissent dans les histogrammes :

- Le calcul effectué sur PC : pour la carte Gidel PROCSpark, il s'agit des résultats obtenus en exécutant les traitements sur PC à l'aide de la célèbre librairie *opensource* de traitement d'images VTK (Visualization ToolKit). La version de VTK utilisée est la 4.2 et les traitements sont exécutés sur un ordinateur doté d'un processeur Pentium 4 cadencé à 2,4 GHz et disposant de 512 Mo de mémoire vive. En ce qui concerne les histogrammes correspondant à la carte Gidel PROCSpark-H (elle possède plus de blocs logiques que la carte Gidel PROCSpark et elle dispose de deux bancs de mémoire pour la lecture et l'écriture des données; cette carte est

employée pour étudier les gains possibles par rapport aux performances de la carte Gidel PROCSpark et du PC), les résultats sur PC sont obtenus à partir de la librairie VTK dans laquelle l'accès aux pixels ainsi que l'exécution des algorithmes ont été très simplifiés (moins de généricité) pour accélérer davantage les calculs; l'ordinateur utilisé est doté d'un processeur Pentium 4 cadencé à 2,8 GHz et il dispose d'une mémoire vive de 1 Go.

- Le calcul effectué sur FPGA avec prise en compte du temps de transfert de l'image de la carte FPGA vers le PC pour affichage: il correspond au temps global d'exécution des traitements avec affichage de l'image.
- Le calcul effectué sur FPGA sans prise en compte du temps de temps de transfert de l'image de la carte FPGA vers le PC pour affichage: il correspond au temps d'exécution des traitements sur le FPGA.
- Le calcul dit « OPTIMISÉ 1 » (uniquement pour la carte Gidel PROCSpark) : il correspond à une configuration de la carte FPGA dans laquelle on dispose de deux bus pour l'accès en mémoire (un bus pour les opérations de lecture et un autre pour les opérations d'écriture). Cette optimisation permet de doubler les temps de traitement des données sur FPGA : en effet, dans la configuration actuelle, la lecture et l'écriture en mémoire sont des opérations mutuellement exclusives (on ne peut donc pas lire et écrire en même temps), ce qui diminue les performances de nos architectures pipelines. Cette optimisation n'est pas valable pour la carte Gidel PROCSpark puisque celle-ci possède déjà deux bus distincts pour les opérations de lecture et d'écriture en mémoire.

- Le calcul dit « OPTIMISÉ 2 » : il correspond à la supposition que tous les accès en mémoire se font en 1 cycle d'horloge. Il s'agit d'une optimisation extrême difficilement réalisable, mais qui fixe quelque peu les limites vers lesquelles on peut tendre dans nos gains de performance.
- Le calcul dit « OPTIMISÉ 3 » (uniquement pour la carte Gidel PROCSpark-H): il correspond à une configuration de la carte FPGA dans laquelle on dispose d'un canal PCI ayant une vitesse de transfert des données de 1 Go/s. Ici, on tient compte du temps de transfert des images de la carte FPGA vers le PC. Au cours de nos tests, nous avons relevé que la vitesse de transfert des données entre le PC et la carte FPGA pouvait avoir une influence sur le temps global d'exécution de nos traitements. Par exemple, une image de 4 Mo qui est transférée en 30 ms, dans la configuration actuelle de 132 Mo/s, sera transférée en 4 ms dans la configuration à 1 Go/s, ce qui représente un gain de plus de 26 ms non négligeable dans une problématique de temps réel où toutes les millisecondes sont importantes.

Dans les lignes suivantes, nous décrivons comment sont obtenus les résultats correspondant aux différents types de calcul sur FPGA. Nous utilisons pour cela les conventions suivantes :

Δ_{calcul_fpga} : le temps mesuré d'exécution d'un algorithme de traitement d'image sur la carte Gidel;

$\Delta_{transfert}$: le temps de transfert des données entre la carte FPGA et le PC: on suppose que la cadence est de 132 Mo/s;

Δ_{FPGA} : le temps global nécessaire au traitement d'une image sur FPGA;

$\Delta_{OPTIMISÉ1}$: le temps nécessaire à l'exécution d'un traitement d'image sur FPGA : la carte FPGA est supposée se trouver dans l'état « OPTIMISÉ 1 » décrite plus haut;

$\Delta_{OPTIMISÉ2}$: le temps nécessaire à l'exécution d'un traitement d'image sur FPGA : la carte FPGA est supposée se trouver dans l'état « OPTIMISÉ 2 » décrite plus haut;

$\Delta_{OPTIMISÉ3}$: le temps global nécessaire au traitement d'une image sur FPGA : la carte FPGA est supposée se trouver dans l'état « OPTIMISÉ 3 » décrite plus haut.

6.2.1. Calcul du temps de traitement sur FPGA avec transfert des données

Ici, on suppose que l'image d'entrée est présente dans la mémoire du FPGA; cela est acceptable dans la mesure où on suppose qu'on se trouve en traitements interactifs d'une image avec des variations de paramètres des algorithmes. Dès lors, le temps global de traitement d'une image sur FPGA inclut le temps d'exécution de l'algorithme sur FPGA et le temps de transfert des données du FPGA au PC. Nous obtenons alors la formule suivante :

$$\Delta_{FPGA} = \Delta_{calcul\_fpga} + \Delta_{transfert}$$

Le temps de transfert d'une image du PC vers le FPGA et vice-versa dépend de la taille de l'image, de la fréquence et de la largeur du bus PCI. Si T désigne la taille de l'image en mégaoctets, si L représente la largeur en octets du bus PCI et si F représente la fréquence en MHz du bus PCI, nous avons la formule suivante :

$$\Delta_{transfert} = \frac{T}{F \times L} \text{ (en secondes)}$$

Ainsi, avec un bus PCI 32 bits cadencé à 33 MHz comme celui de la carte Gidel PROCSpark, nous obtenons les temps de transfert suivants :

Taille de l'image (en octets)	Temps de transfert (en millisecondes)
512×512	1,89
1024×1024	7,57
2048×2048	30,30

Tableau 9 : Taille maximale des images pouvant être traitées en 50 ms

6.2.2. Calcul du temps de traitement sur FPGA sans transfert des données

Ici, on se contente uniquement du temps d'exécution des algorithmes de traitement d'images sur FPGA, cela permet de mieux apprécier la vitesse d'exécution du FPGA et de juger l'influence des temps de transfert des données (entre le PC et le FPGA) sur les résultats finaux.

Avec cette hypothèse, on a la formule :

$$\Delta_{FPGA} = \Delta_{calcul\_fpga}$$

6.2.3. Calcul du temps de traitement sur FPGA à l'état « OPTIMISÉ 1 »

On se place dans les hypothèses où l'image d'entrée est déjà stockée en mémoire du FPGA. En rajoutant la condition de l'existence de deux bus distincts pour la lecture et l'écriture en mémoire du FPGA, il s'en suit que le temps d'exécution des algorithmes de traitement d'images est divisé par 2 puisque la lecture et l'écriture en mémoire peuvent se

faire simultanément. Par conséquent, la formule de calcul du temps de traitement d'une image sans prise en compte du temps de transfert de l'image résultante du FPGA vers le PC est :

$$\Delta_{OPTIMISÉ1} = \frac{\Delta_{calcul\_fpga}}{2}$$

6.2.4. Calcul du temps de traitement sur FPGA à l'état « OPTIMISÉ 2 »

On se place dans les hypothèses d'une carte FPGA avec des temps d'accès en mémoire d'un cycle d'horloge. Pour déterminer la formule de calcul du temps de traitement, nous utilisons les désignations suivantes :

L : nombre de lignes de l'image à traiter;

C : nombre de colonnes de l'image à traiter;

$\Delta_{algorithme}$ le délai d'un composant chargé de l'exécution d'un algorithme de traitement d'images (dans notre cas, ce délai correspond à 1 cycle d'horloge);

f : la fréquence d'horloge.

Étant donné que nous traitons en général quatre pixels simultanément, on a la formule suivante :

$$\Delta_{calcul\_fpga} = \frac{L \times C \times \Delta_{algorithme}}{4} \times \frac{1}{f} = \frac{L \times C \times 1}{4} \times \frac{1}{f} = \frac{L \times C}{4} \times \frac{1}{f}$$

Ainsi, le temps de traitement d'une image dans l'état nommé « OPTIMISÉ 2 » (sans prise en compte du temps de transfert de l'image résultante vers le PC) s'exprime comme suit :

$$\Delta_{OPTIMISÉ2} = \Delta_{calcul\_fpga} = \frac{L \times C \times \Delta_{algorithme}}{4} \times \frac{1}{f}$$

Dans le cas de la convolution sur la carte Gidel PROCSpark, étant donné que nous ne produisons en sortie que deux pixels simultanément, le temps de traitement global s'exprimera comme suit :

$$\Delta_{OPTIMISÉ2\_Convolution\_PROCSpark} = \Delta_{calcul\_fpga} = \frac{L \times C \times \Delta_{algorithme}}{2} \times \frac{1}{f}$$

Pour la carte Gidel PROCSpark-H, la convolution génère quatre pixels en sortie (cette carte possède plus de blocs logiques que la carte Gidel PROCSpark, ce qui permet de mettre en œuvre de plus grosses opérations) et on a la formule :

$$\Delta_{OPTIMISÉ2\_Convolution\_PROCSpark-H} = \Delta_{calcul\_fpga} = \frac{L \times C \times \Delta_{algorithme}}{4} \times \frac{1}{f}$$

6.2.5. Calcul du temps de traitement sur FPGA à l'état « OPTIMISÉ 3 »

On se place dans les hypothèses où l'image d'entrée est déjà stockée en mémoire de la carte Gidel PROCSpark-H. En rajoutant la condition que les transferts de données entre la carte FPGA et le PC sont multipliés par 8 (cadence de 1 Go/s), le temps de transfert initial se retrouve divisé par 8. Par conséquent, le temps de traitement global du FPGA dans l'état « OPTIMISÉ 3 » est :

$$\Delta_{OPTIMISÉ3} = \Delta_{calcul\_fpga} + \frac{\Delta_{transfert}}{8}$$

6.3. Graphes des résultats pour la carte Gidel PROCSpark

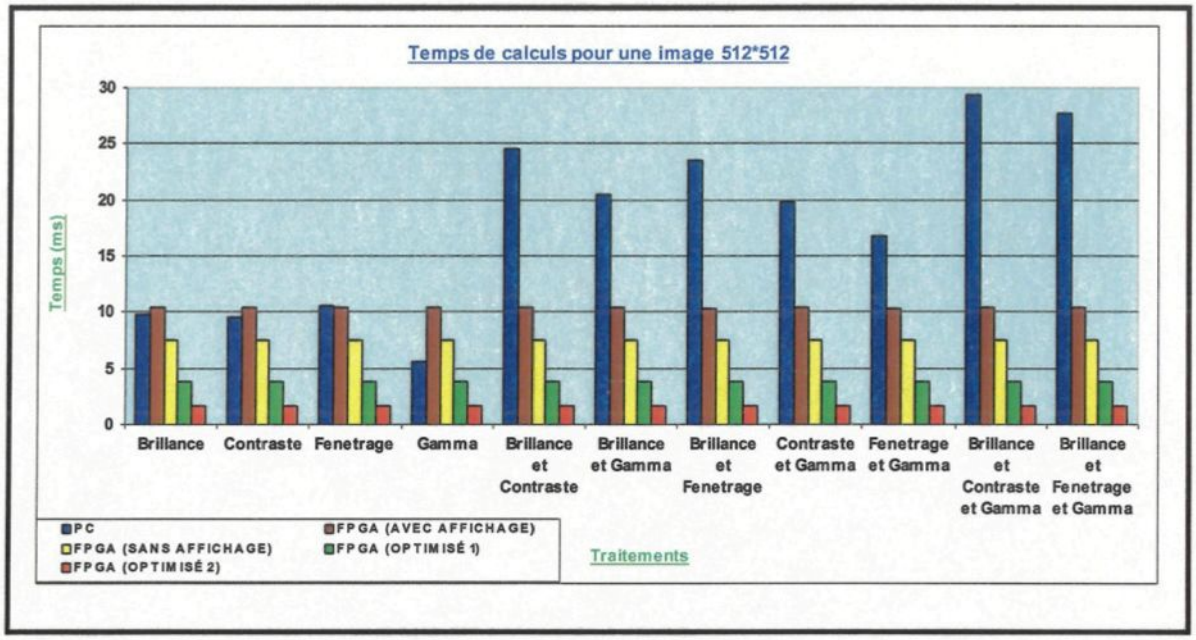

Figure 41 : Temps de calculs pour une image 512*512 pixels

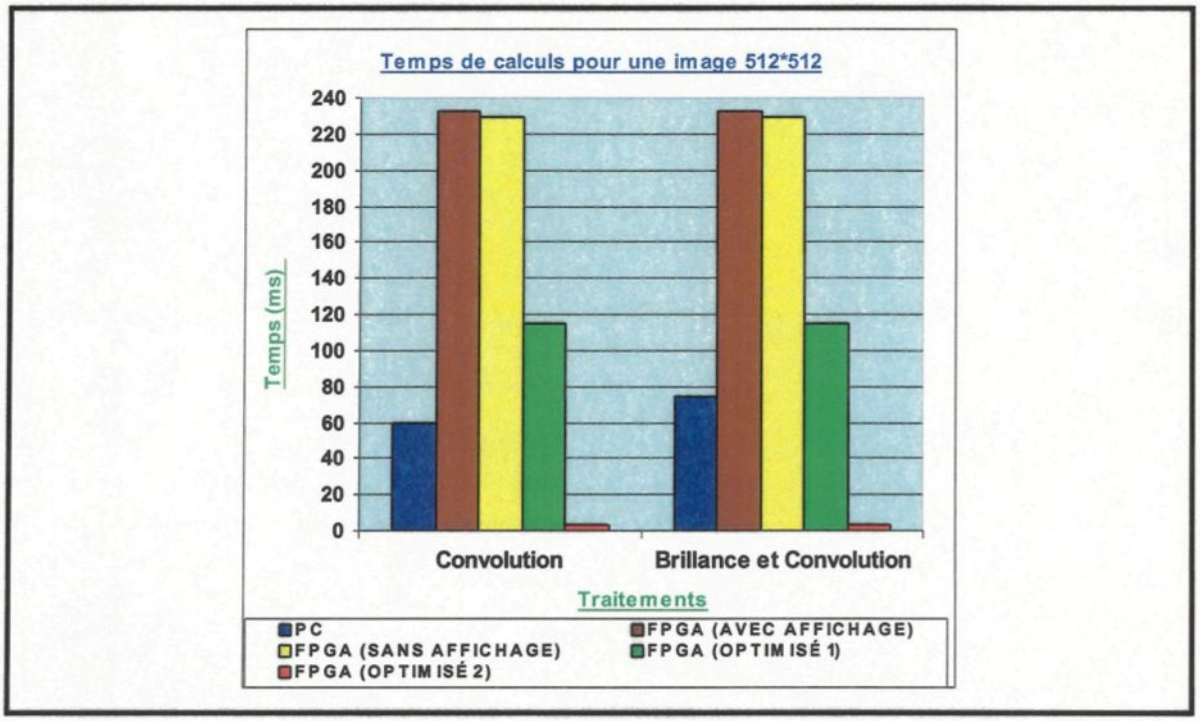

Figure 42 : Temps de calculs pour une image 512*512 pixels (suite)

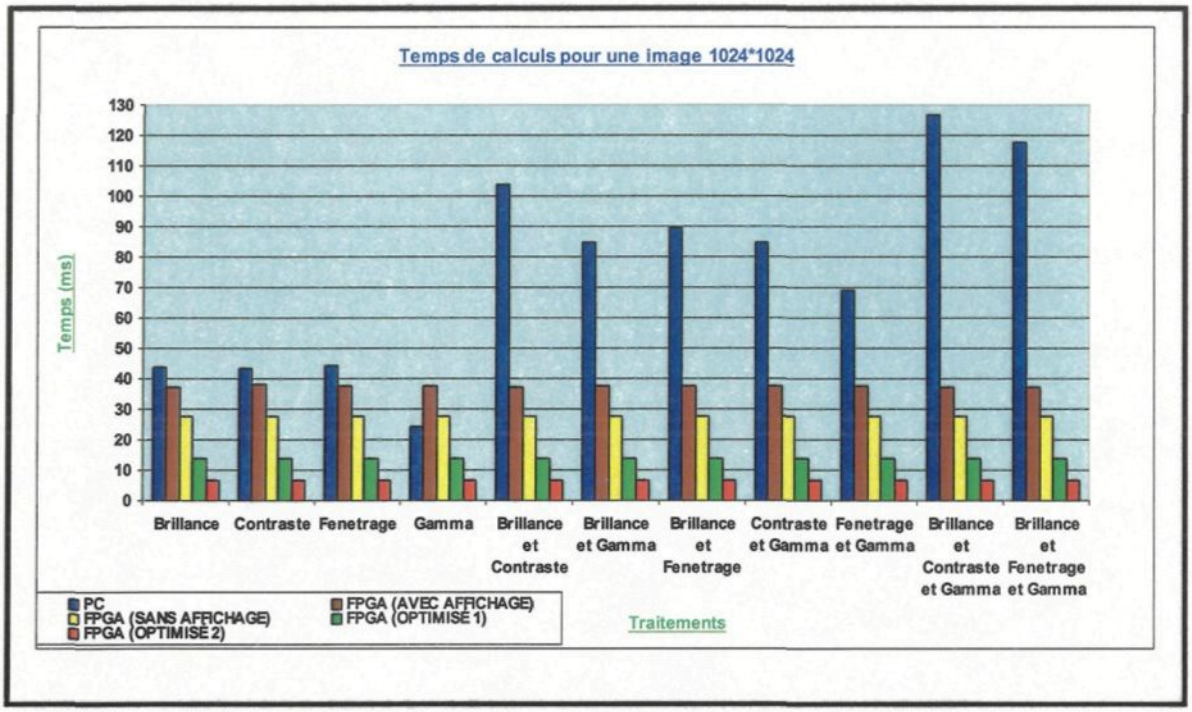

Figure 43 : Temps de calculs pour une image 1024*1024 pixels

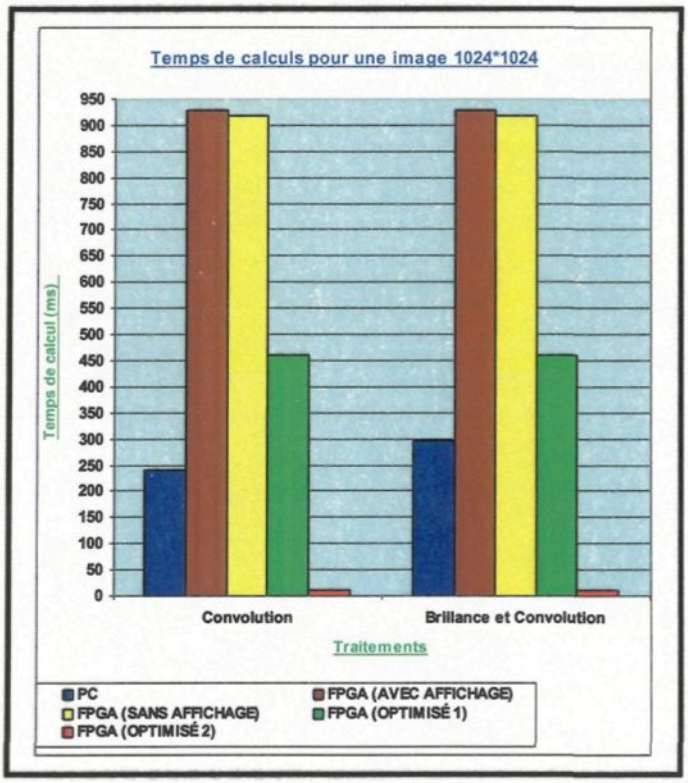

Figure 44 : Temps de calculs pour une image 1024*1024 pixels (suite)

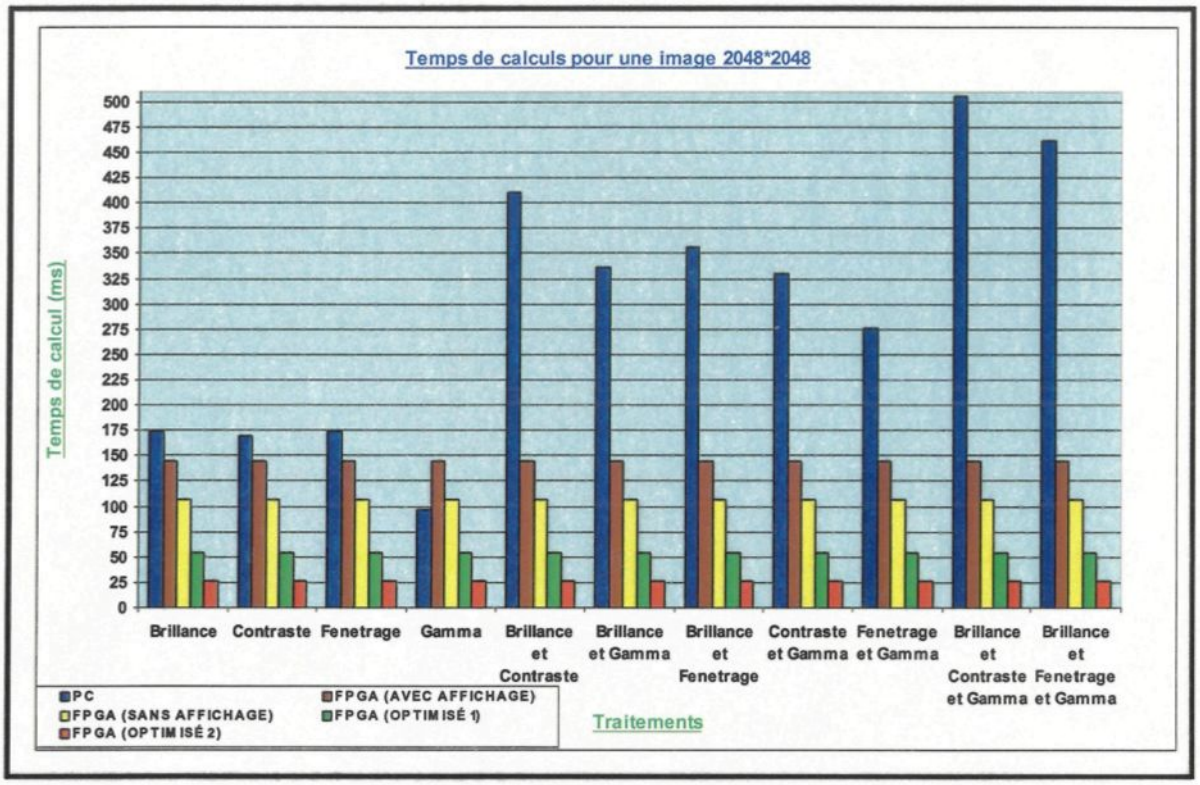

Figure 45 : Temps de calculs pour une image 2048*2048 pixels

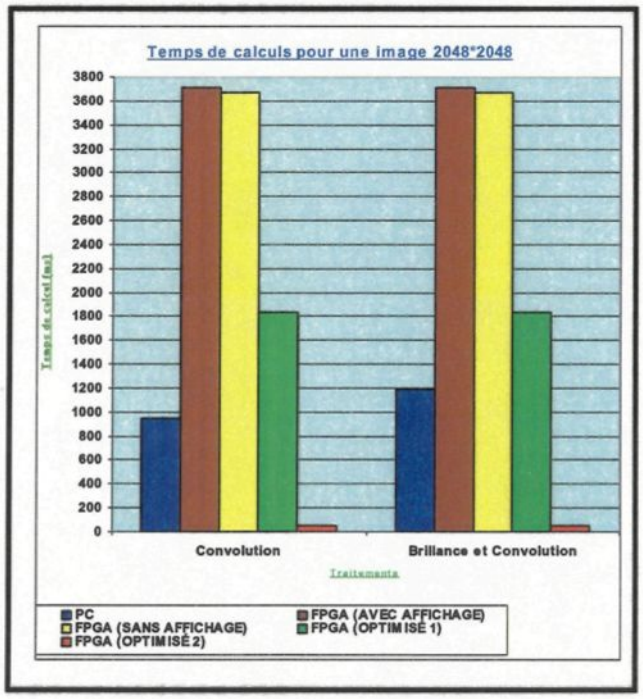

Figure 46 : Temps de calculs pour une image 2048*2048 pixels (suite)

6.4. Graphes des résultats pour la carte Gidel PROCSpark-H

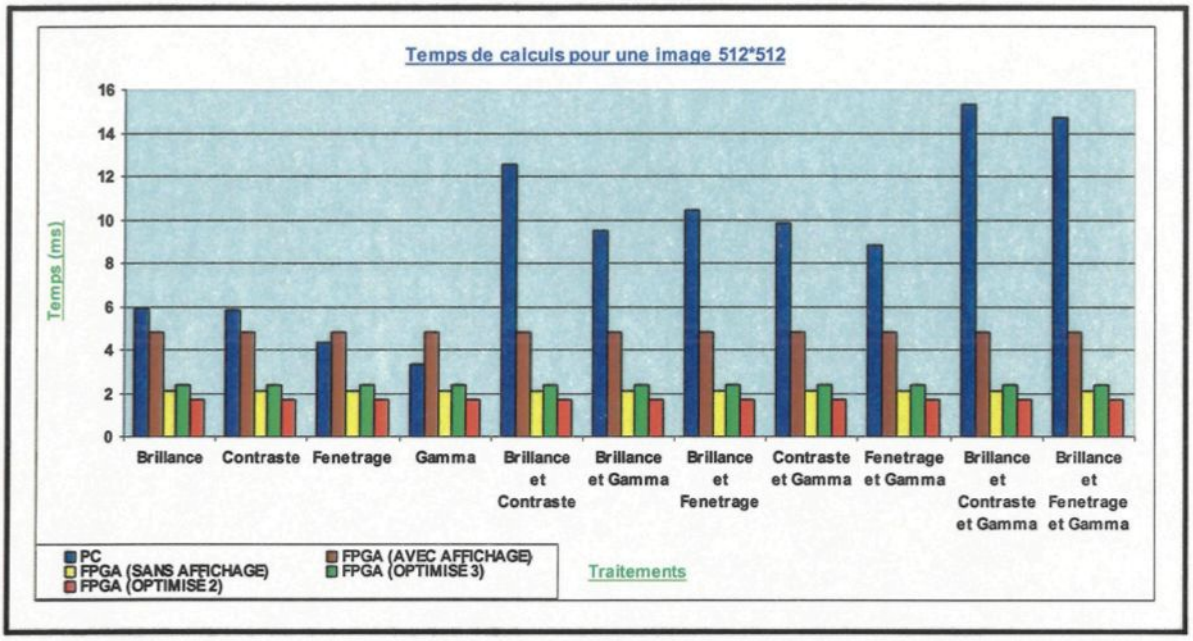

Figure 47 : Temps de calculs pour une image 512*512 pixels

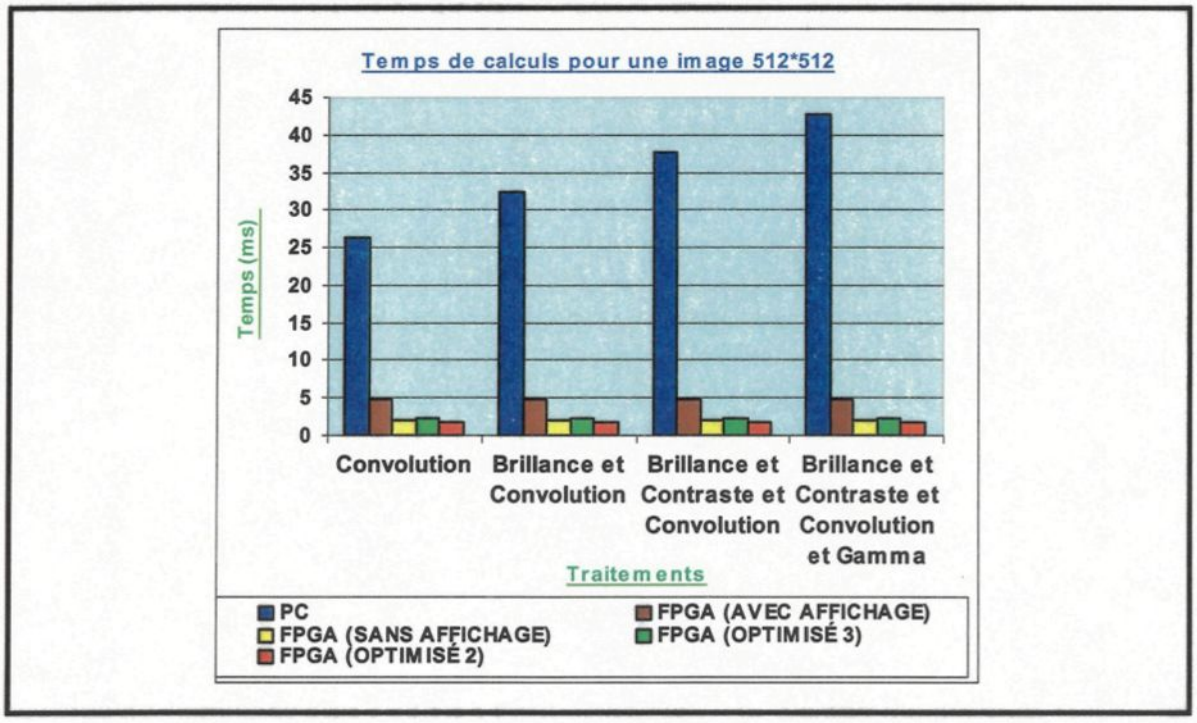

Figure 48 : Temps de calculs pour une image 512*512 pixels (suite)

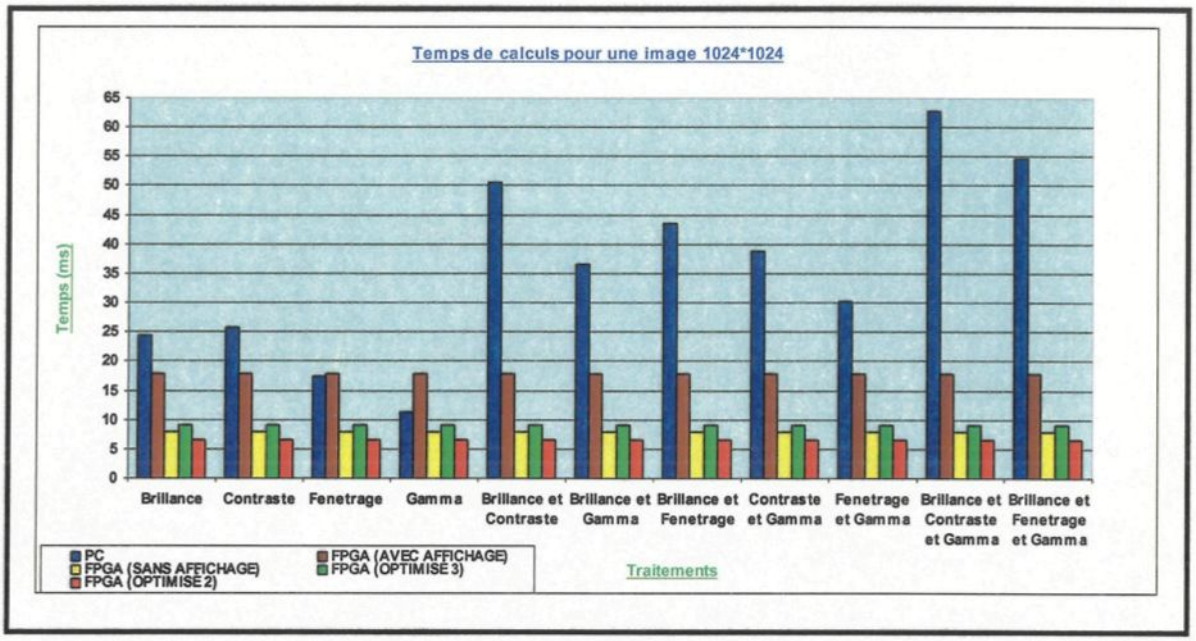

Figure 49 : Temps de calculs pour une image 1024*1024 pixels

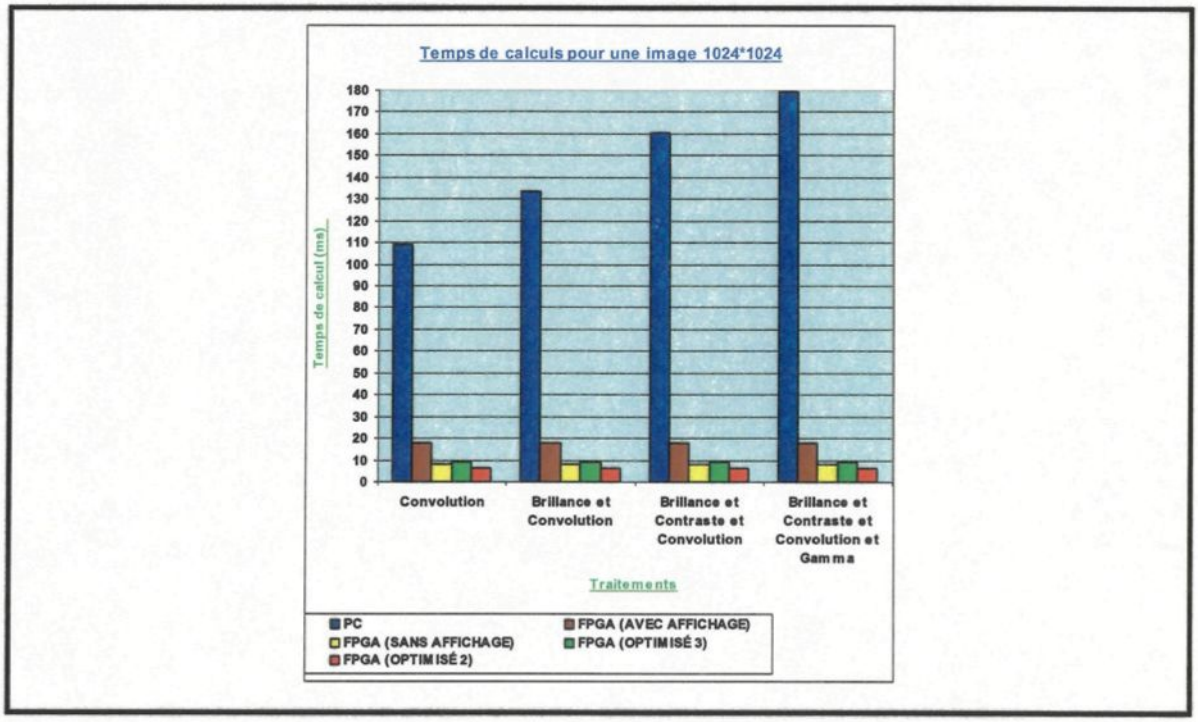

Figure 50 : Temps de calculs pour une image 1024*1024 pixels (suite)

6.5. Analyse et interprétation des résultats

Les graphiques des résultats présentés dans les pages précédentes nous permettent de tirer les conclusions suivantes :

En ce qui concerne la carte FPGA Gidel PROCSpark,

- ✓ Les temps de calculs sur FPGA dépendent de la taille de l'image traitée. Cela est visible à travers les temps de calculs croissants que nécessite chaque traitement lorsqu'on passe d'une image de 0,25 Mo (512×512 pixels) à une image de 4 Mo (2048×2048 pixels).
- ✓ La brillance n'apporte pas des gains significatifs sur FPGA par rapport au PC. Il s'avère même que pour des images de petite taille (cas d'une image de 0,25 Mo, figure 41), les temps de calculs sont meilleurs sur PC. En moyenne, la carte Gidel est environ 15% plus rapide que le PC (figures 43 et 45). Il convient de noter cependant que les temps de calculs sur la carte tiennent compte à la fois du temps de traitement des pixels ainsi que du temps de transfert des données de la mémoire de la carte Gidel vers la mémoire du PC; si l'on ne considère que le temps de traitement des données, la carte Gidel a des gains légèrement supérieurs à ceux décrits plus haut.
- ✓ En ce qui concerne le contraste, on tire les mêmes conclusions de performance que celles relatives à la brillance (figures 41, 43, 45).
- ✓ Le traitement gamma apparaît toujours moins rapide sur la carte Gidel que sur PC (figures 41, 43, 45). Cela peut s'expliquer par le mode d'exécution retenu pour ce

type de traitement : un tableau de correspondances est utilisé au lieu de faire des calculs de gamma pour chaque pixel de l'image en entrée; ainsi, sur PC, la première étape consiste à calculer la formule gamma pour chacune des 256 valeurs de niveau de gris de pixels; par la suite, l'image en entrée est lue pixel par pixel et, pour chaque valeur de niveau de gris de pixel *xin*, on affecte au pixel correspondant de l'image de sortie la valeur de niveau de gris se trouvant à l'index *xin* du tableau de correspondances. Au final, le traitement sur PC se résume à une lecture de tableau, ce qui s'est avéré avoir de meilleures performances que la même stratégie utilisée sur FPGA. Notons enfin qu'au fur et à mesure que la taille de l'image croît, la différence entre les temps de calculs sur PC et FPGA croît aussi.

- ✓ Le fenêtrage apparaît toujours plus rapide sur la carte Gidel que sur PC (figures 41, 43, 45), mais les gains ne sont pas très significatifs (environ 25% meilleurs). En comparaison des traitements comme la brillance et le contraste, le fenêtrage apparaît plus rapide; cela vient du fait que tous les pixels de l'image en entrée ne sont pas forcément utilisés dans les opérations arithmétiques du fenêtrage puisque cet algorithme fait intervenir un intervalle où les valeurs de niveau de gris doivent se situer pour être impliquées dans des calculs; cette sélection des pixels peut réduire les temps d'exécution du traitement général et fournir par conséquent des résultats plus rapidement que les traitements comme la brillance ou le contraste.
- ✓ La convolution donne de mauvais temps d'exécution sur la carte Gidel PROCSpark (figures 42, 44, 46). Cela est principalement la résultante de l'inefficacité de l'accès direct en mémoire par l'intermédiaire du contrôleur multiports fourni avec la carte

Gidel. Les tests et simulation ont permis de constater qu'une lecture/écriture en accès direct pouvait prendre jusqu'à 15 cycles d'horloge, ce qui est assez lent. D'un autre côté, nous ne pouvions utiliser l'accès séquentiel (plus rapide) en raison du fait qu'il entraînerait plus de perte dans les temps de calculs à cause même de la nature de l'algorithme de convolution : en effet, la convolution est fondée sur la notion de voisinage entre les pixels, et par conséquent, les données nécessaires au calcul d'un pixel de sortie ne se trouvent pas toujours sur la même ligne, d'où la nécessite d'un accès direct.

- ✓ Toutes les combinaisons de deux traitements (excepté celle de la brillance avec la convolution) donnent des gains sur la carte Gidel par rapport au PC. Les gains sont un peu plus du double et croissent avec la taille de l'image à traiter. Ce gain s'explique par le fait que la carte Gidel effectue la série de traitements en pipeline, ce qui ne change presque pas les temps de calculs par rapport aux traitements individuels (uniquement un seul traitement); par contre, sur PC, une combinaison de traitements s'exécute comme si chaque traitement individuel est effectué séparément avant que le traitement suivant prenne le relais : dans ce contexte, le temps total de traitement est presque équivalent à la somme des temps d'exécution de chaque algorithme impliqué pris individuellement.
- ✓ En ce qui concerne la combinaison « brillance – convolution », les temps de calculs sont médiocres sur FPGA (figures 42, 44, 46). Ici, les mêmes causes produisent les mêmes effets : le manque d'efficacité de la lecture/écriture mémoire par accès direct

est à l'origine du manque de performance observé dans les traitements comme expliqué plus haut dans le cas de la convolution.

- ✓ Toutes les combinaisons de trois traitements donnent des gains sur la carte Gidel par rapport au PC. Les gains croissent avec la taille de l'image à traiter et atteignent un rapport d'environ 4 avec une image de taille 4 Mo.

Plusieurs voies d'optimisation sont possibles et apportent quelques améliorations aux résultats déjà obtenus. La première optimisation consiste à utiliser une carte FPGA avec deux canaux pour l'accès en mémoire (un pour la lecture et un pour l'écriture) et à partir de laquelle les accès en mémoire se font en 1 cycle d'horloge. Dès lors, il est possible de lire et d'écrire en mémoire simultanément et à une grande vitesse. Pour ce faire, la carte Gidel PROCSpark-H a été utilisée. Avec cette optimisation, on note des gains notables en performance d'exécution, et les graphiques précédents permettent de tirer les conclusions suivantes :

- ➢ Les gains croissent avec la taille de l'image : si l'on ne tient compte que du temps d'exécution des algorithmes, il ressort que pour une image de 512 × 512 pixels (figures 47et 48), les algorithmes pris individuellement sont 1,5 à 13 fois plus rapides sur FPGA.

  En ce qui concerne les combinaisons de deux traitements en chaîne, les temps d'exécution sur FPGA sont 4 à 16 fois plus rapides.

  Les combinaisons de trois traitements en chaîne sont 7 à 19 fois plus rapides sur FPGA.

La combinaison convolution-brillance-contraste-gamma est 21 fois plus rapide sur FPGA.

- Pour une image de 1024 × 1024 pixels (figures 49 et 50), les gains sont plus élevés : les algorithmes pris individuellement sont 1,5 à 13 fois plus rapides sur FPGA.

  En ce qui concerne les combinaisons de deux traitements en chaîne, les temps d'exécution sur FPGA sont 4 à 17 fois plus rapides.

  Les combinaisons de trois traitements en chaîne sont 7 à 20 fois plus rapides sur FPGA.

  La combinaison convolution-brillance-contraste-gamma est 23 fois plus rapide sur FPGA.

- Les résultats des traitements sont meilleurs que ceux obtenus par Venkateshwar Rao Daggu et Muthukumar Venkatesan **[26]** : dans le calcul de la convolution avec le langage Handle-C et une carte FPGA Xilinx Virtex-E XCV2000E contenant 43 200 cellules logiques, ils ont obtenu des temps de calcul de 1,31 ms pour une image de 256×256 pixels et de 5,32 ms pour une image 512×512 pixels : nos traitements sont donc deux fois plus rapides que les leurs et nous disposons de beaucoup moins d'éléments logiques (8 320).

- Les gains observés sur FPGA sont plus élevés lorsqu'on fait des traitements en chaîne. Cela s'explique par le fait que les algorithmes sur FPGA se font en pipeline (les temps d'exécution sont quasiment identiques pour un traitement ou pour une

série de traitements en chaîne), ce qui n'est pas le cas sur PC (les traitements ici se font en séquence sans pipeline).

- Si nous tenons compte aussi du temps de transfert des images de la mémoire de la carte FPGA vers le PC pour affichage, alors les gains en performance présentés ci-dessus sont presque réduits de moitié (figures 47 à 50). En utilisant un canal PCI cadencé à 1 Go/s (état nommé « OPTIMISÉ 3 ») par exemple (c'est-à-dire près de 8 fois plus rapide que la cadence actuelle qui est de l'ordre de 132 Mo/s), nous obtenons des gains dans la phase de transfert des données de pixels de la mémoire du FPGA vers le PC; ces gains deviennent importants au fur et à mesure que la taille de l'image croît. Ainsi, une image de taille 512×512 (0,25 Mo) est transférée en 0,25 ms au lieu de 1,89 ms avec un canal PCI à 132 Mo/s; une image de taille 2048×2048 (4 Mo), qui normalement est transférée en environ 30 ms, sera transférée en seulement 4 ms avec un canal PCI cadencé à 1 Go/s. Avec ces considérations, les nouveaux gains se rapprochent de ceux obtenus sans prise en compte du temps de transfert des données (le gain maximal atteint pour le traitement d'une image de 2048×2048 pixels de 8 bits est de 20, ce gain était de 23 sans prise en compte du temps de transfert des images de la carte FPGA vers le PC).
- Les résultats obtenus sans la prise en compte du temps de transfert des données de la carte FPGA vers le PC sont très proches des résultats optimaux auxquels on peut s'attendre (figures 47 à 50) : ces résultats optimaux sont obtenus en supposant que tous les accès en mémoire se font en 1 cycle d'horloge. Cela s'explique par le fait que les accès en mémoire ont été considérablement optimisés sur la carte Gidel

PROCSpark-H pour s'assurer qu'il n'y ait presque pas de cycles d'attente dans l'accès aux données en mémoire. Le contrôleur multiports fourni avec la première carte Gidel utilisée est très flexible (permet de prendre en compte de nombreux paramètres de configurations de la carte FPGA, notamment les accès en mode séquentiel et les accès en mode direct), mais cette flexibilité est en même temps un goulot d'étranglement puisqu'il faut gérer et synchroniser plusieurs signaux (lecture en mémoire, écriture en mémoire, tampons pleins, accès séquentiel en mémoire, accès direct en mémoire). Avec la carte Gidel PROCSpark-H, le contrôleur d'accès en mémoire a été minimisé : c'est ainsi par exemple que seul l'accès direct en mémoire a été considéré. Par ailleurs, il existe un contrôleur pour les lectures en mémoire et un autre pour l'écriture en mémoire. Enfin, notons que la disponibilité de deux bancs de mémoire sur la carte Gidel PROCSpark-H permet de lire et d'écrire simultanément en mémoire, par conséquent il n'y a presque pas de cycles d'attente durant le traitement des données (les attentes sont en général observées lorsque les données lues/écrites se trouvent sur des lignes différentes dans l'organisation de la mémoire, ces attentes sont de l'ordre de 2 à 3 cycles d'horloge).

Le tableau suivant présente la taille maximale des images qui sont traitées en environ 50 ms dans les différentes configurations FPGA de la carte Gidel PROCSpark-H. Nous supposons ici que les temps de traitement des données sur FPGA sont proportionnels à la taille de l'image en entrée (en théorie, cette supposition est valide puisque nous travaillons sur des architectures parallèles systoliques) :

Plate-forme de traitement	Taille maximale d'une image 8 bits traitée en 50 ms
FPGA AVEC AFFICHAGE	2,8 Mo
FPGA SANS AFFICHAGE	6,25 Mo
FPGA « OPTIMISÉ 2 »	7,8 Mo
FPGA « OPTIMISÉ 3 »	5,6 Mo

Tableau 10 : Taille maximale des images pouvant être traitées en 50 ms

D'après le tableau précédent, on peut effectuer des séries de traitements qui provoquent la persistance rétinienne pour des images de taille atteignant 2,8 Mo (voire jusqu'à 6,25 Mo si on ne tient pas compte du temps de transfert des données de la carte FPGA vers le PC); après cette taille, il n'est plus possible d'effectuer des traitements d'images donnant l'illusion à l'œil humain que tout se déroule comme dans un film (le « temps réel » est rompu). Les optimisations proposées pour la carte FPGA repoussent les limites de la taille maximale des images traitées en 50 ms, ce qui laisse entrevoir des possibilités de traitement de gros volumes de données d'imagerie en des temps très courts (on atteindrait une taille de 5,6 Mo si on suppose que les images sont transférées du FPGA vers le PC par l'intermédiaire d'une carte PCI cadencée à 1 Go/s).

- ✓ En ce qui concerne le traitement par le zoom, les simulations donnent des résultats assez intéressants. L'algorithme de zoom utilisant l'interpolation bilinéaire a été écrit de manière à s'exécuter plus rapidement lorsque le facteur (échelle) de zoom est élevé. Ainsi, un zoom par 4 d'une image de 256×256 pixels est plus rapide qu'un zoom par 2 sur une image de 512×512 pixels (de même, entre un zoom par 8 sur une image

256×256 et un zoom par 4 sur une image 512×512) : cela s'explique par le fait que plus le facteur de zoom est élevé, plus il y a des pixels dans l'image de sortie qui utiliseront le même voisinage de pixels dans l'image d'entrée pour le calcul de leur niveau de gris, ce qui réduit le nombre d'accès en mémoire pour chercher les données et, par conséquent, diminue les temps de traitement. Dans l'ensemble, le zoom ne semble pas être une opération que l'on devrait effectuer sur de grosses images ou portions d'images si l'on veut que les traitements s'exécutent très rapidement en 50 ms ou moins; l'algorithme bilinéaire utilisé ne donne des résultats intéressants que pour des petites images (tout dépendant encore du facteur de zoom voulu).

	Zoom 2x	Zoom 4x	Zoom 8x
Image 256×256	18,35 ms	47,18 ms	136,31 ms
Image 512×512	73,40 ms	188,72 ms	545,25 ms

Tableau 11 : Temps de traitement du zoom

CHAPITRE 7 :

CONCLUSION ET RECOMMANDATIONS

Notre étude avait pour but de mettre en œuvre sur FPGA des algorithmes permettant des traitements d'images en des temps relativement courts et d'étudier les possibilités d'une mise en œuvre provoquant la persistance rétinienne (les images doivent être traitées en série de manière à ce que l'œil ne distingue pas nettement deux images successives, tout doit se passer comme dans un film; cela est réalisé à des vitesses de traitement variant entre 50 ms et 60 ms par image, c'est-à-dire 16 à 20 images par seconde).

La revue de la littérature ainsi que des études approfondies nous ont permis de mettre en œuvre sur une carte FPGA Gidel PROCSpark plusieurs algorithmes de traitement d'images. Les paramètres de traitement et les images sont issus d'une application informatique s'exécutant sur un ordinateur de type PC, le FPGA se charge uniquement de l'exécution des algorithmes. Au total, quatorze traitements ont été mis en œuvre : la brillance, le contraste, le fenêtrage, le gamma, la convolution, la combinaison brillance et contraste, la combinaison brillance et fenêtrage, la combinaison brillance et gamma, la combinaison brillance et convolution, la combinaison contraste et gamma, la combinaison fenêtrage et gamma, la combinaison brillance et contraste et gamma, la combinaison brillance et fenêtrage et gamma, le zoom.

Les résultats obtenus avec la carte FPGA sont encourageants. Ils ont été comparés à ceux obtenus avec une mise en œuvre sur PC (processeur Pentium 4 cadencé à 2,4 GHz, mémoire vive de 512 Mo) à l'aide de la célèbre librairie graphique VTK. Les gains obtenus sont dans un rapport variant entre 1,25 et 4 pour la plupart des traitements. Les gains croissent avec la taille de l'image à traiter et ils sont plus significatifs lorsqu'on utilise des combinaisons de traitements. La convolution n'a pas donné des résultats satisfaisants, les

temps observés sur FPGA sont très élevés par rapport à ceux relevés sur PC : ceci est la résultante de l'inefficacité des accès directs en mémoire par le contrôleur multiports fourni avec la carte Gidel. Il faut noter que, en raison de la caractéristique spéciale de la convolution faisant intervenir une notion de voisinage dans les calculs, le type d'accès-mémoire adéquat pour effectuer ce traitement est l'accès direct (à l'opposé de l'accès séquentiel utilisé dans les autres traitements), mais cet accès requiert beaucoup de cycles d'horloges sur la carte Gidel, ce qui entraîne une augmentation des temps de calculs.

La carte Gidel PROCSpark ne possède qu'un seul canal d'accès en mémoire, les opérations de lecture et d'écriture sont par conséquent mutuellement exclusives (on ne peut pas simultanément lire et écrire en mémoire) et ne nous permettent pas de profiter au maximum des performances de nos architectures pipelines. Afin d'améliorer les résultats obtenus, la carte Gidel PROCSpark-H (comportant deux bancs de mémoire, un pour la lecture en mémoire et un autre pour l'écriture en mémoire) a été utilisée. Les gains ont été considérablement améliorés et atteignent la valeur 23 dans l'exécution d'une chaîne de traitements composée de quatre opérations (brillance, contraste, gamma, convolution). Il faut relever que les accès en mémoire ne se font qu'en mode direct et ne requièrent en général qu'un seul cycle d'horloge. Les vitesses d'exécution relevées sont deux fois plus rapides que celles obtenues par Venkateshwar Rao Daggu et Muthukumar Venkatesan **[26]** en 2004 dans les calculs de convolution avec le langage Handle-C et une carte FPGA Xilinx Virtex-E XCV2000E contenant 43 200 cellules logiques (contre 8 320 pour la carte Gidel ProcSpark-H). Lorsqu'on tient compte du temps de transfert des données de la carte

FPGA vers le PC, on constate que le débit du canal PCI actuel (132 Mo/s) ralentit les performances globales même si des gains maxima de l'ordre de 11 sont atteints.

Pour améliorer les gains, une carte FPGA supportant des transferts par canal PCI à des vitesses de l'ordre de 1 Go/s ou plus est souhaitable (une image de 4 Mo est transférée en 4 ms avec un canal cadencé à 1 Go/s contre 30 ms avec un canal cadencé à 132 Mo/s, soit un gain de 26 ms qui est non négligeable dans une optique de traitement en temps réel). Dans cette configuration, les gains maxima qu'on obtient par rapport au PC se situent autour de 20 et la taille maximale des images 8 bits traitées en 50 ms est estimée à 5,6 Mo (cette taille est de 2,6 Mo avec le taux de transfert actuel du canal PCI qui est de 132 Mo/s).

D'autres avenues pour accroître les performances obtenues jusque là sont possibles : on peut citer l'utilisation d'une mémoire SDRAM *(Synchronous Dynamic Random Access Memory)* cadencée à 100 MHz (la mémoire actuelle est cadencée à 50 MHz). Dans cette considération, les accès en mémoire se font deux fois plus vite, ce qui permet de réduire de moitié les temps d'exécution des traitements d'images et de doubler les gains par rapport au PC (on atteindrait un gain maximal de 46 sans considération du transfert des images du FPGA vers le PC).

On peut aussi améliorer les temps de traitement grâce à l'utilisation de bus PCI Express (8x, 12x, 16x, 32x) qui peuvent atteindre des taux de transfert de 8 Go/s, ce qui permettrait de réduire considérablement les temps de transfert d'images entre le PC et la carte FPGA.

En ce qui concerne l'utilisation des ressources physiques utilisées pour la mise en œuvre des opérations de traitement d'images sur FPGA, des optimisations sont possibles et envisageables à travers notamment l'utilisation d'opérateurs arithmétiques (additions,

soustractions, etc.) nécessitant moins de cellules ou portes logiques sur FPGA que ceux utilisés afin de minimiser au maximum les temps de traitement d'images. Cependant, il faudra faire face au problème récurrent de gestion du compromis espace/temps : en effet, les composants les plus gourmands en ressources physiques sont en général les plus rapides. Des études plus approfondies devront être menées afin de faire ressortir les temps de traitements et le nombre de ressources physiques nécessaires pour les différents types d'opérateurs arithmétiques et logiques que fournit la littérature (bit-sériel, parallèle, parallèle séquentiel, sériel/parallèle, etc.).

D'un autre côté, il convient de noter qu'avec la densité de certains FPGAs actuels (par exemple, la carte Gidel PROCStar II avec 720 000 éléments logiques), il est possible de mettre en œuvre dans le même programme ou architecture toutes les opérations de traitement d'images que nous avons recensées, ce qui permettrait d'éliminer la reprogrammation du FPGA lors du changement du type de traitement dans une application pratique.

De manière générale, les mises en œuvre des algorithmes de traitement d'images sur FPGA à l'aide du langage psC ont permis de montrer qu'il est possible d'effectuer des traitements en temps réel (dans notre contexte, cette notion signifie des calculs en 50 ms ou moins) pour de nombreux algorithmes. Cependant, il y a des limitations qui existent sur la taille des images pouvant être traitées et les différentes optimisations possibles ne tendent qu'à repousser ces limites en réduisant les temps de calcul des différents algorithmes. D'autres opérations simples (seuillage par exemple) ou complexes (segmentation,

reconnaissance des formes, calculs en trois dimensions, etc.) sont envisageables pour une mise en œuvre sur FPGA à l'aide du langage psC.

Sur un tout autre plan, nous devons souligner les performances du langage psC utilisé pour la mise en œuvre sur FPGA des algorithmes de traitement d'images. Quelques difficultés, liées à des composants non disponibles (opérateurs de division, multiplicateurs en nombres fixes par exemple) ont été rencontrées durant le développement, mais nous avons réussi à les contourner à travers d'autres fonctions disponibles dans le langage. Notre étude a permis de mieux éprouver et tester le langage psC, les défauts relevés permettent d'améliorer sa convivialité et simplicité d'utilisation. En somme, notre étude a, dans une certaine mesure, permis de montrer que le langage psC est assez mature pour permettre de développer sans trop de difficultés des applications d'intérêt pour la science et l'industrie : plus spécifiquement, l'application de traitement d'images, qui a été développée pour faire des tests à travers la carte Gidel, peut constituer un modèle à améliorer et à enrichir pour faciliter le travail des radiologues dans leurs tâches de traitement et analyse d'images.

LISTE DES RÉFÉRENCES

[1] Académie de Lyon; *"L'imagerie médicale – fiche de présentation d'oral.* [En ligne]. *<http://www2.ac-lyon.fr/etab/lycees/lyc-42/jmonnet/physique/imagemedicale/oral.html>* consulté le 28 août 2004.

[2] BATTIATO, S.; GALLO, G.; STANCO, F.; (Septembre 2000) *"A new edge-adaptative zooming algorithm for digital images"*. Proceedings of IASTED Signal Processing and Communications SPC 2000, pp. 144-149.

[3] BENITEZ, Domingo; (Septembre 2003) *"Performance of reconfigurable architectures for image-processing applications"*. Journal of Systems Architecture: 30(4-6), pp. 193-210.

[4] Dicom; *documents officiels*. [En ligne] *<ftp://medical.nema.org/medical/dicom/2003/printed>* consulté le 06 décembre 2004.

[5] DOSHI, Kshitij; VARMAN, Peter; (1987) *"A modular systolic architecture for image convolutions"*. Proceedings of the 14th Annual International Symposium on Computer Architecture, pp. 56-63.

[6] DRAPER, Bruce A; BEVERIDGE, J. Ross; BÖHM, A.P. Willem; ROSS, Charles; CHAWATHE, Monica; (Août 2002) *"Implementing image applications on FPGAs"*. International Conference on Pattern Recognition, Quebec City.

[7] DUTRIEUX, L.; DEMIGNY, D.; (1997) *"Logique programmable"*. Paris: Editions.

[8] FORTIN, Jean-Michel; Novakod Technologies Inc; (Février 2004) *"Résultats de la recherche des différentes plates-formes FPGA pour port PCI disponibles sur le marché"*.

[9] FOUNTAIN, T.J.; (1994) *"Parallel computing: principles and practices"*. Cambridge: University Press.

[10] GENGLER, Marc; UBÉDA, Stéphane; DESPREZ, Frédéric; (1996) *"Initiation au parallélisme, concepts, architectures et algorithmes"*. Paris: Editions Masson.

[11] GONZALEZ, R.; WINTZ, P.; (1987) *"Digital Image Processing"*, Addison-Wesley.

[12] GOSHTASBY, Ardy; Wright State University and Image Fusion Systems Research; (Juin 2004) *"Image resampling"*.

[13] JAMRO, Ernest; (2001) *"Parameterised automated generation of convolvers implemented in FPGAs"*. Thèse de Doctorat. Krakow: University of mining and metallurgy, 143 p.

[14] KUNG, H.T.; (1984) *"Systolic algorithms for the CMU Warp Processor"*. Proceedings of the Seventh International Conference on Pattern Recognition, International Association for Pattern Recognition, pp. 334-343.

[15] KUNG, H.T.; (Janvier 1982) *"Why systolic architectures?"*. Computer: 15(1), pp. 37-46.

[16] KUNG, H.T.; PICARD, R.L.; (Octobre 1983) *"One-dimensional systolic arrays for multi-dimensional convolution and resampling"*. Technical Report, Department of Computer Science, Carnegie-Mellon University.

[17] LEHMANN, Thomas M.; GÖNNER, Claudia; SPITZER, Klaus; (Novembre 1999) *"Survey: interpolation methods in medical image processing"*. IEEE transactions on medical imaging: 18(11).

[18] MORIN, Luc; Novakod Technologies Inc; (2004) *"Rodin version 0.3, programmer's reference manual"*.

[19] MORIN, Luc; LAMONTAGNE, Marc; LATOUR, Patrick; Novakod Technologies Inc; (Juillet 2004) *"Rapport d'avancement jalon #2"*.

[20] MORSE, H. Stephen, (1994) *"Practical parallel computing"*. Boston, Toronto: Academic Press.

[21] PARGAS, R.P.; ALLEN, K.R.; (Mai 1986) *"SAGE: a systolic array generating engine"*. Proceedings of the ARO Workshop on the Future Directions in Computer Architecture and Software, Seabrook Island, SC, pp. 301-303.

[22] RATHA, Nalini K.; JAIN, Anil K.; ROVER, Diane T.; (Avril 1995) *"Convolution on Splash 2"*. Proceedings of the IEEE Conference, FCCM.

[23] SHARMA, Anup B.; ALLEN, Keith R.; PARGAS, Roy P.; (1988) "*Some new systolic designs for two-dimensional convolution*". Proceedings of the 1988 ACM sixteenth annual conference on Computer, pp. 350-356.

[24] TISSERAND, Arnaud; (Décembre 2003) *"Introduction aux circuits FPGA"*. INRIA LIP Arénaire, Séminaire MIM.

[25] TURNEY, Robert D.; DICK, Chris H.; Core Solutions Group, Xilinx, Inc. (2000) *"Real Time Image Rotation and Resizing, Algorithms and Implementations"*.

[26] VENKATESHWAR, Rao Daggu; MUTHUKUMAR Venkatesan; (Septembre 2004) *"Design and Implementation of an Efficient Reconfigurable Architecture for Image Processing Algorithms using Handel-C"*. Euromicro Symposium on Digital Systems Design, pp. 218-226.

[27] Xilinx; *"Programmable Logic Design Quick Start Handbook"*. [En ligne]. <*http://www.xilinx.com/univ/ beginnersbookjune2003ver2.pdf*> consulté le 28 août 2004.

[28] ZHAO, Y.; SERE, K.; (Novembre 1995) *"Derivation of systolic convolution arrays"*. Proceedings of the 7th Nordic Workshop on Programming Theory.

LES ANNEXES

A. Exemple de code psC et C/C++ pour l'addition de deux nombres

A.1. Code psC

```
//Composant hiérarchique « main »
Component main (in int ExtA, in int ExtB, out int ExtC)

{

        //composant simple additionneur

        Component adder (in active int A, in active int B, out active int C)

        {

                //action exécutée lorsqu'un évènement survient sur les ports A ou B

                addInputs(0) on A, B

                {

                        C := A + B ;

                }

        } ;

        //déclaration des processus

        adder P1 ;

        //déclaration des signaux ou chaînes de communication

        int signal_1 = {ExtA, P1.A} ;

        int signal_2 = {ExtB, P1.B} ;

        int signal_3 = {ExtC, P1.C} ;

} ;
```

L'architecture globale du programme est le suivant :

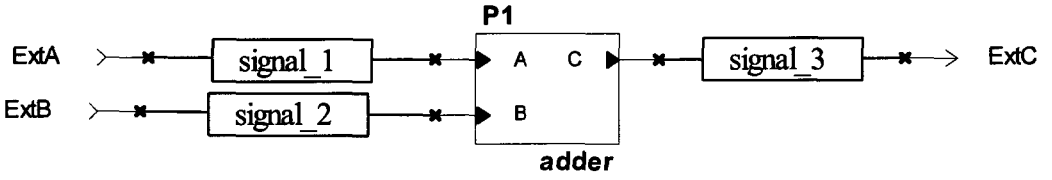

Figure 51 : Additionneur psC

A.2. Code C/C++

```
//fonction pour additionner 2 nombres à partir du FPGA
int adder_fpga (int ExtA, int ExtB, CFpgaApi* api)
//ExtA et ExtB sont les données en entrée, api est un pointeur vers l'API Novakod
{
  int ExtC //valeur de retour
  bool hasRcvEvent /*drapeau utilisé pour déterminer si le FPGA a retourné la valeur du
  calcul*/
  //Ports FPGA
  int port_ExtA;
  int port_ExtB;
  int port_ExtC;
  // Chargement du programme exécutable synthétisé pour FPGA à partir du code psC
  api -> loadBinaryProg("additionneur.rbf", 1);
  // On obtient le nom des ports de la mise en œuvre sur FPGA
  api -> GetPortNumberByName( "ExtA", port_ExtA );
```

```
  api -> GetPortNumberByName( "ExtB", port_ExtB );
  api -> GetPortNumberByName( " ExtC ", port_ExtC );
  //Insertion des arguments de l'additionneur dans des évènements à transmettre au FPGA
  api -> InsertEvent(port_ExtA, ExtA);
  api -> InsertEvent(port_ExtB, ExtB);
  //Transmission des évènements au FPGA
  api -> SendEvents( );
  // Le program attend un événement du FPGA.
  api -> WaitForEvents( );
  // Réception du résultat dans la variable ExtC
  GetPortEvent(port_ExtC, ExtC, hasRecvEvent);
  return ExtC ;
}
```

B. Cartes FPGA analysées pour la mise en œuvre des algorithmes de traitement d'images

Carte FPGA	Caractéristiques
PCI-X Sys Board (PLDA)	Stratix jusqu'à 30K LE SDRam Ctrl, PCI-X Core environ 4000 US Leader dans les IP Cores PCI-X 1Gb/s Support Technique Illimité
PCI Sys Board (PLDA)	FLEX jusqu'à 1M Gates Pas de RAM, PCI core environ 3000 USD Leader dans les IP Cores 528 MB/s Support en français
PROCSpark (GIDEL)	APEX 200K (bon petit board pour tests) RAM, PCI core 1900 USD 10 ans, bons clients 132 MB/s Expert en imagerie médicale
Prroc20KE (GIDEL)	APEX jusqu'à 7,5 M Gates (pas de multiplication) Pas de Core, RAM optionnelle N/A 10 ans, bons clients 132 MB/s Expert en imagerie médicale
ProcStar (GIDEL)	Stratix jusqu'à 30K LE Pas de contrôleur RAM, PCI core inclus 2000 USD 10 ans, bons clients 528 MB/s Expert en imagerie médicale

ProcSuperStar (GIDEL)	Stratix 80K LE PCI bridge, 8000 USD 10 ans, bons clients Expert en imagerie médicale
PCI High Speed Dev Kit (ALTERA)	Stratix 60K LE Quartus Inclus, Controlleurs RAM (licence 60 jours), Core PCI-X de PLDA 5000 USD 800 Mb/s
S25(ALTERA)	Stratix 25K LE Core PCI-X de PLDA 2000 USD
A10C(ALTERA)	APEX 1M Gates Contrôleur Ram, Peu de RAM (32 MB), PCI megaCore sans licence de développement. 4000 USD 528 MB/s
FireBird (ANNAPOLIS MICROSYSTEMS)	Virtex E 1M-2M Pas de RAM onboard, contôleur PCI Plusieurs générations de produits 528MB/s Support technique possible (semble gratuit)
WildStar II (ANNAPOLIS MICROSYSTEMS)	Virtex II 6M et 8M Pas de contrôleur RAM environ 17000 US 528 MB/s Support technique possible (semble gratuit)
BallyNuey 2 (NALLATECH)	Virtex 200K-800K Pas de Ram, extensions coûteuses 5-6000 USD 132 MB/s Service complet, Update et forums

BallyNuey 3 (NALLATECH)	Virtex II 1M a 3M PCI Ctrl sur Spartan, extensions pour RAM et autres IO 5-6K US 132 MB/s Service complet, Update et forums
BenNuey (NALLATECH)	Virtex II 1M a 8M Pas de RAM onboard, PCI Ctrl 10k-20k USD 266 MB/s Service complet, Update et forums
DN2000K10 (DINIGROUP)	Virtex 300K-1.8M Pas de RAM ni de core PCI 10K USD 15 ans dans le domaine 266 MB/s Support par mail
DN3000K10A (DINIGROUP)	Virtex II 4M a 8M PCI core, Ram Ctrl, Ram on-board 15K USD 15 ans dans le domaine PCI-X Support par mail
DN3000K10S (DINIGROUP)	Virtex II PCI core, Ram Ctrl, Ram on-board 20K USD 15 ans dans le domaine PCI-X Support par mail
DN5000K10 (DINIGROUP)	Stratix PCI core > 20K US 15 ans dans le domaine PCI-X Support par mail
DN5000K10S (DINIGROUP)	Virtex II Pro PCI core, Ram Ctrl, Ram on-board > 20K US 15 ans dans le domaine PCI-X Support par mail

DN6000K10S (DINIGROUP)	Virtex II Pro PCI core, Ram Ctrl, Ram on-board > 20K US 15 ans dans le domaine PCI-X Support par mail
DN6000K10SC (DINIGROUP)	Virtex II Pro PCI core, Ram Ctrl, Ram on-board > 20K US 15 ans dans le domaine PCI-X Support par mail
Vision Speedster (ISYTEC)	Virtex 400K-2,5M Peu de RAM, contrôleurs inclus 1,8-6,6K USD 20 Ans 132 MB/s Hotline
Vision Speedster II (ISYTEC)	Virtex II 4M a 8M Contrôleurs inclus, RAM en masse 8-23K USD 20 ans dans le domaine 132 MB/s HotLine
ChipIt Power (ISYTEC)	Virtex 400K-2,5M Peu de RAM, contrôleurs inclus 20 ans dans le domaine 132 MB/s HotLine
Osiris (CORETECH)	Virtex II - 1M to 8M RAM, Flash, Cache, PCI bridge 8000 USD 15 ans dans le domaine militaire 528MB/s
Virtex E Dev Kit (AVNET)	Virtex E (pas de Multiplication) 1M Video/Audio, USB, peu de RAM, pas de contrôleur PCI ni RAM 1500 USD Vendus par Xilinx 266 MB/s

SoC RDSO1 (AVNET)	Virtex E 1M Board spécial, non recommandé 3000 USD Vendus par Xilinx 266 MB/s
Virtex II Dev Kit (AVNET)	4M Gates MicroBlaze, Core USB2 4000 USD Xilinx l'a sur son site, 150 employés, PCI-X Environ 150 experts
Virtex II Pro Dev Kit (AVNET)	Virtex 2 Pro PCI Bridge, pas de RAM ctrl 4000 USD Xilinx l'a sur son site, 150 employés, PCI-X Environ 150 experts
ADC-RC1000 (ALPHA DATA)	Virtex 1M Pas de RAM, PCI bridge + interface on-board, 10-15K USD 132MB/s Contact par mail et téléphone, support payant disponible
Tsunami (AVVIDA)	Stratix 20K-128K LE Bcp RAM, PCI interface, Spécialistes en images > 7500 USD 15 ans d'expérience max. 360 MB/s Peu d'informations, support par téléphone et mail
Q5 (XIPHOS)	1,5 M Gates RAM, Pas de core PCI, Interface PCI via DDK environ 5000 USD, 132 MB Expérience domaine spatial Support par téléphone et mail

Tableau 12 : Liste des cartes FPGA analysées pour la mise en œuvre des algorithmes de traitement d'images

C. Quelques fonctions de l'API C++ Novakod

Fonctions de l'API : définition et paramètres
int LoadBinaryProg(const char* FileName, int chipNumber) Cette fonction permet de charger un programme FPGA synthétisé sur la carte. *FileName* : le chemin et le nom du fichier du programme à charger sur le FPGA. Le fichier peut être de type « .bit » (Xilinx) ou « .rbf » (Altera). *chipNumber* : numéro de la puce FPGA à programmer sur la carte (1 pour la première puce).
int InsertValue(int portNumber, int value) Cette fonction permet d'insérer une valeur sur le port spécifié par « portNumber ». Les valeurs « value » peuvent être de type 32 bits ou 64 bits. *portNumber* : numéro du port sur lequel une valeur doit être insérée. *value* : valeur à insérer sur le port.
int InsertEvent(int portNumber, int value) Cette fonction permet d'insérer une valeur et un événement sur le port spécifié par « portNumber ». Les valeurs « value » peuvent être de type 32 bits ou 64 bits. *portNumber* : numéro du port sur lequel une valeur et un événement doivent être insérés. *value* : valeur à insérer sur le port.
void SendEvents() Cette fonction permet d'envoyer tous les événements simultanément vers le FPGA. Il faut que cette fonction soit appelée pour envoyer des événements au FPGA.
void WaitForEvents( ) Cette fonction permet d'attendre la réception d'événements provenant du FPGA. Elle met le « thread » courant dans un état d'attente jusqu'à l'instant où un événement survient. Si aucun événement n'est reçu de la part du FPGA, la fonction attendra indéfiniment.
int GetPortValue(int portNumber, int& value)

Cette fonction permet de lire une valeur sur le port spécifié par « portNumber ». Les valeurs « value » peuvent être de type 32 bits ou 64 bits. *portNumber* : numéro du port sur lequel une valeur doit être insérée. *value* : valeur lue sur le port, retournée par référence par la fonction.
int GetPortEvent(int portNumber, int& value, bool& hasRecvEvent) Cette fonction permet de lire une valeur ainsi que de vérifier si un événement est survenu sur le port spécifié par « portNumber ». Le paramètre « hasRecvEvent » vaut « true » lorsqu'un événement est survenu sur le port, autrement il vaut « false ». *portNumber* : numéro du port sur lequel une valeur doit être insérée. *value* : valeur lue sur le port, retournée par référence par la fonction. *hasRecvEvent* : drapeau indiquant si un événement est survenu, il est retourné par référence par la fonction.
int InitDMAParams(int sizeSend, int sizeRecv, unsigned char*& sendBuffer, unsigned char*& recvBuffer) Cette fonction permet de créer les tampons permettant d'envoyer et recevoir des données dans la mémoire de la carte FPGA. *sizeSend* : ce paramètre permet de préciser la taille maximale du tampon contenant les données à envoyer. *sizeRecv* : ce paramètre permet de préciser la taille maximale du tampon contenant les données à recevoir. *sendBuffer* : ce paramètre contient un pointeur vers le tampon d'envoi alloué par l'API. L'utilisateur écrit les données à envoyer par l'intermédiaire de ce tampon. Il doit vérifier qu'il ne dépassera pas la taille maximale du tampon. *recvBuffer* : ce paramètre contient un pointeur vers le tampon de réception alloué par l'API. L'utilisateur lit les données reçues par l'intermédiaire de ce tampon.
int SendData(DWORD address, DWORD length) Cette fonction permet d'envoyer un nombre de données à une adresse de la mémoire sur la carte FPGA. Les données doivent être écrites dans le tampon «

sendBuffer » provenant de la fonction *InitDMAParams* avant l'appel de la fonction *SendData*. *address* : adresse où les données doivent être écrites. *length* : quantité en octets de données à envoyer.
int RecvData(DWORD address, DWORD length) Cette fonction permet de recevoir un nombre de données à une adresse de la mémoire sur le FPGA. Les données doivent se retrouver dans le tampon « recvBuffer » provenant de la fonction *InitDMAParams*. *address* : adresse ou les données doivent être lues. *length* : quantité en octet de données à lire.
int GetPortNumberByName(const char* portName, int& portNumber) Cette fonction retourne le numéro du port correspondant à « portName » dans le programme psC. *portName* : nom du port dont l'utilisateur désire connaître le numéro. *portNumber* : numéro du port, retourné par référence par la fonction.

Tableau 13 : Fonctions C++ de l'API Novakod

D. Captures d'écran de l'application informatique de traitement d'images s'exécutant sur PC

D.1. Description des éléments de la fenêtre « Traitements indépendants »

La fenêtre traitements indépendants comporte plusieurs boutons de déplacement permettant à l'utilisateur d'effectuer les traitements. La présente section décrit la fonction de chaque élément de la fenêtre.

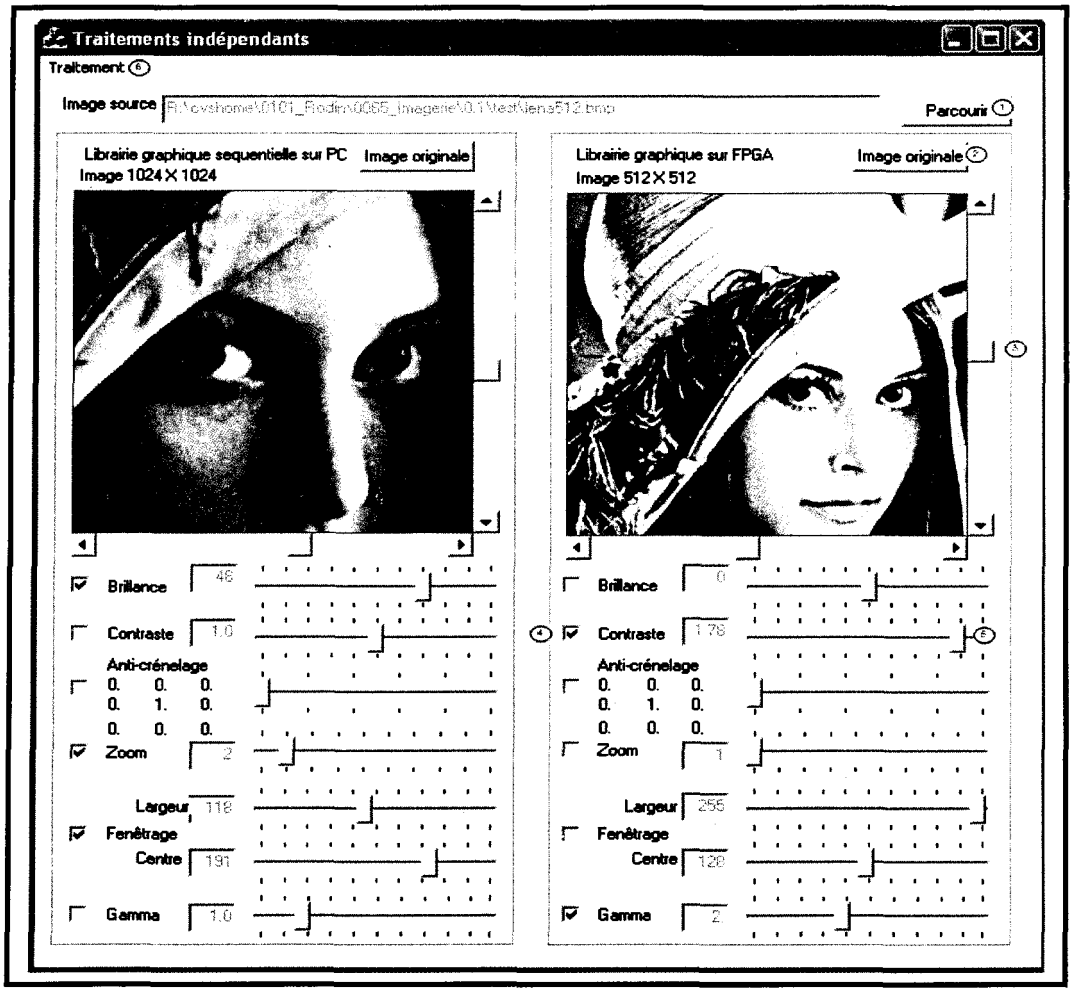

Figure 52 : Application de traitement d'images, fenêtre « traitements indépendants »

1- Bouton « Parcourir » : ouvre un fichier de type bitmap. Les images traitées doivent être en format de 8 bits avec 256 niveaux de gris.

2- Bouton « Image originale » : affiche l'image originale.

3- Boutons de défilement horizontal et vertical : déplace l'image dans la fenêtre.

4- Case à cocher : sélectionne un traitement. Pour les traitements sur PC toutes les combinaisons sont permises.

5- Boutons de déplacement : ces boutons modifient les paramètres d'entrée des traitements. Chaque déplacement correspond à un nouveau calcul des traitements sélectionnés par les cases à cocher.

6- Menu Traitement : permet de passer à la fenêtre « Traitements prédéfinis ».

D.2. Description des éléments de la fenêtre « Traitements prédéfinis »

La fenêtre traitements prédéfinis comporte un menu déroulant permettant à l'utilisateur de choisir les traitements prédéfinis dont la vitesse de traitement sera évaluée. La présente section décrit la fonction de chaque élément de la fenêtre.

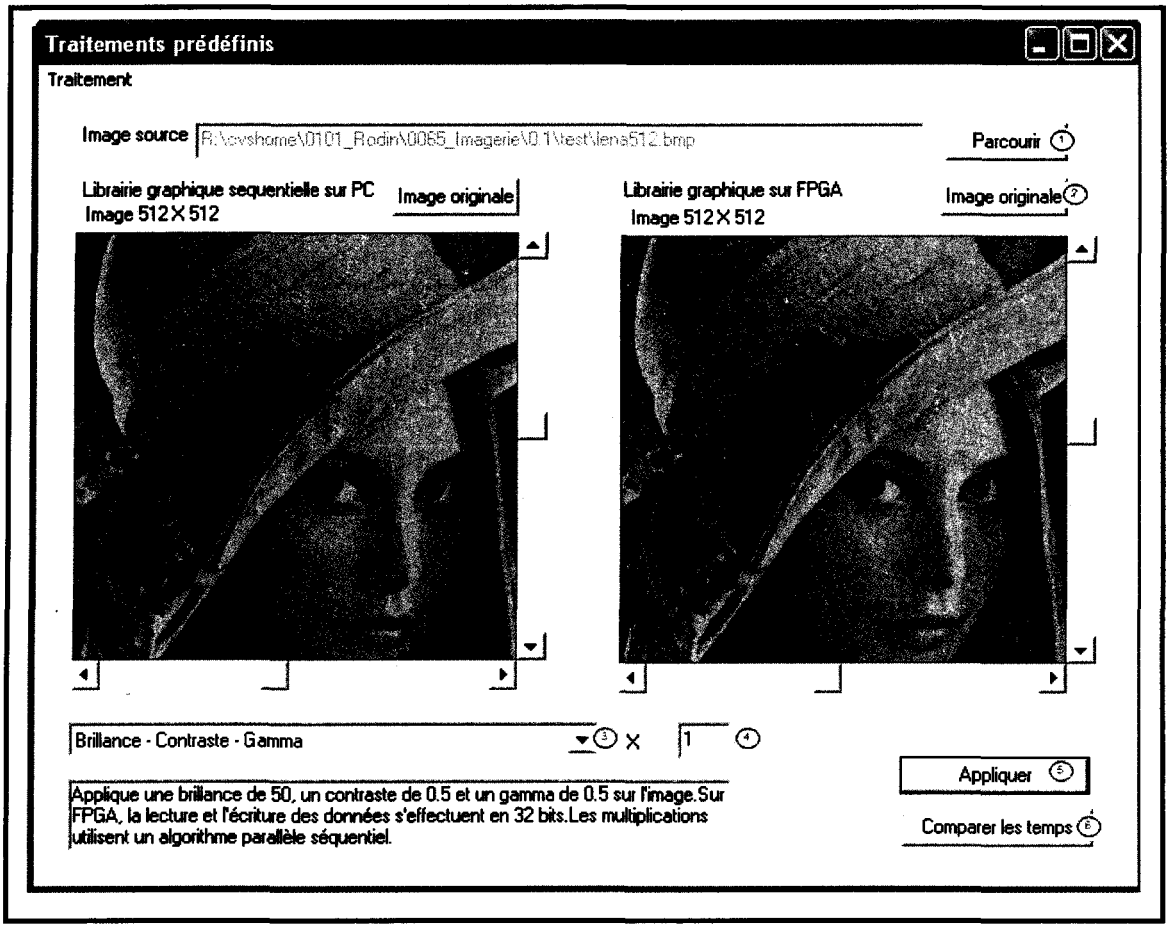

Figure 53 : Application de traitement d'images, fenêtre « traitements prédéfinis »

1- Bouton « Parcourir » : ouvre un fichier de type bitmap. Les images traitées doivent être en format de 8 bits avec 256 niveaux de gris.

2- Bouton « Image originale » : affiche l'image originale.

3- Menu déroulant sur les choix de traitements : affiche les traitements prédéfinis.

4- Nombre d'exécution : correspond au nombre de fois que l'exécution des traitements sera répétée.

5- Bouton « Appliquer » : applique les traitements prédéfinis en temps réel.

6- Bouton « Comparer les temps » : exécute le traitement prédéfini sélectionné sur PC et sur FPGA. Les temps d'exécutions sur les deux plateformes sont comparés et affichés dans une fenêtre texte.

E. Le glossaire

Anti-crénelage: fait de réduire les effets d'escalier d'un dessin en trichant avec notre perception visuelle et en ajoutant des couleurs intermédiaires dans les pixels adjacents (ce qui améliore la perception de l'ensemble grâce à une sorte de dégradé).

API (Application Programming Interface) : une interface de programmation définit la manière dont un composant informatique peut communiquer avec un autre. Dans le cas typique d'une bibliothèque, il s'agit généralement d'une liste de fonctions considérées comme utiles pour d'autres composants.

Architecture systolique: schéma de parallélisme utilisant une matrice d'éléments de calculs. Les données circulent dans la matrice de manière régulière; à chaque étage, des opérations identiques sont effectuées sur des données différentes.

FPGA (Field-Programmable Gate Array): matrice de blocs logiques programmables pouvant être reprogrammés à volonté pour effectuer différents types de calculs.

Histogramme des niveaux de gris: représentation graphique ayant en abscisses les valeurs de niveaux de gris, et en ordonnées le nombre de pixels associés à chaque valeur de niveau de gris.

Image bitmap: image composée d'un ensemble (ou matrice) de points (les pixels) auxquels sont associées une position et une couleur.

LUT (Look Up Table): réalise des fonctions combinatoires dans un FPGA.

Mode simulation psC: la lecture et l'écriture des données sur les ports physiques réels sont simulées par voie logicielle.

Pixel: c'est le plus petit élément qui compose une image bitmap affichée sur un écran ou imprimée. Le nombre de pixels exprime généralement la résolution d'une image ou d'un écran.