UNIVERSITÉ DE MONTRÉAL

CONCEPTION DE PROCESSEURS SPÉCIALISÉS POUR LE TRAITEMENT VIDÉO EN TEMPS RÉEL PAR FILTRE LOCAL

PHILIPPE AUBERTIN
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION DU DIPLÔME DE
MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
AOÛT 2010

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

CONCEPTION DE PROCESSEURS SPÉCIALISÉS POUR LE TRAITEMENT VIDÉO EN TEMPS RÉEL PAR FILTRE LOCAL

présenté par : AUBERTIN Philippe
en vue de l'obtention du diplôme de : Maîtrise ès Sciences Appliquées
a été dûment accepté par le jury constitué de :

M. DAVID Jean-Pierre, Ph.D., président.
M. SAVARIA Yvon, Ph.D., membre et directeur de recherche.
M. LANGLOIS Pierre J.M., Ph.D., membre et codirecteur de recherche.
M. BOYER François-Raymond, Ph.D., membre.

À Alan Turing

Remerciements

Je désire tout d'abord remercier le professeur Yvon Savaria pour avoir accepté de diriger ces travaux et m'avoir ainsi permis de bénéficier de son expérience. Ses suggestions et commentaires étaient toujours pertinents et constructifs.

Je désire aussi remercier le professeur Pierre Langlois, mon co-directeur de recherche, pour sa disponibilité constante, et pour son aide constante qui est allée bien au delà de ce qu'on doit s'attendre d'un co-directeur de recherche.

Je tiens à remercier tous mes collègues au labo pour avoir rendu bien plus agréables ces deux dernières années, et en particulier Gilbert Kowarzyk et Maria Mbaye pour leurs petits coups de pouce qui m'ont aidé à démarrer cette maîtrise du bon pied.

Je remercie aussi Réjean Lepage et Ghyslaine Éthier-Carrier pour leur support et leur assistance tout au long de ma maîtrise.

Finalement, je désire remercier Normand Bélanger pour les discussions fort intéressantes au cours desquels il m'a transmis un peu de son expérience professionnelle et de son expérience de vie.

Résumé

Ce mémoire décrit les travaux visant à explorer les possibilités qu'offrent les processeurs à jeu d'instructions spécialisé pour des applications de vidéo numérique. Spécifiquement une classe particulière d'algorithmes de traitement vidéo est considérée : les filtres locaux. Pour cette classe d'algorithmes, une exploration architecturale a permis d'identifier un ensemble de techniques formant une approche cohérente et systématique pour la conception de processeurs spécialisés performants adaptés au traitement vidéo en temps réel.

L'approche de conception proposée vise une utilisation efficace de la bande passante vers la mémoire, laquelle bande passante constitue le goulot d'étranglement de l'application du point de vue de la vitesse de traitement. Il est possible d'approcher la performance limite imposée par ce goulot par une stratégie appropriée de réutilisation des données et en exploitant le parallélisme des données inhérent à la classe d'algorithmes visée. L'approche comporte quatre étapes : tout d'abord, une instruction parallèle (SIMD) qui effectue le calcul de plusieurs pixels de sortie à la fois est créée. Puis, des registres à décalage permettant la réutilisation intra-ligne des pixels d'entrée sont ajoutés. Ensuite, un pipeline est créé par le découpage de l'instruction parallèle et l'ajout de registres pour les résultats intermédiaires. Finalement, les instructions spécialisées de chargement et de sauvegarde sont créées. Quelques-unes de ces étapes ouvrent la porte à des simplifications matérielles spécifiques pour certains algorithmes de la classe cible. La structure matérielle obtenue au final, alliée à la parallélisation des instructions par l'utilisation d'une architecture VLIW, se comporte d'une manière semblable à un réseau systolique pipeliné.

Afin de démontrer expérimentalement la validité de l'approche de conception proposée, sept processeurs spécialisés pour des algorithmes de la classe visée ont été conçus par extension du jeu d'instructions d'un processeur configurable à jeu d'instructions extensible.

Trois de ces processeurs spécialisés mettent en œuvre autant d'algorithmes de désentrelacement intra-trames, et quatre visent plutôt la convolution 2D, différant entre eux par la taille de la fenêtre de convolution.

Les résultats de performance obtenus sont prometteurs. Pour les algorithmes de désentrelacement intra-trames, les facteurs d'accélération varient entre 95 et 1330, alors que les facteurs d'amélioration du produit temps-surface varient entre 29 et 243, tout ceci par rapport à un processeur d'usage général de référence roulant une implémentation purement logicielle de l'algorithme. Dans le cas de la convolution 2D, les facteurs d'accélération varient entre 36 et 80, alors que les facteurs d'amélioration du produit temps-surface varient entre 12 et 22. Dans tous les cas, le traitement en temps réel de données vidéo à haute définition au format 1080i (désentrelacement) ou 1080p (convolution) est possible avec une fabrication dans une technologie de circuit intégré à 130 nm.

Abstract

This master thesis explores the possibilities offered by Application-Specific Instruction-Set Processors (ASIP) for digital video applications, more specifically for a particular algorithm class used for video processing: local neighbourhood functions. For this algorithm class, an architectural exploration lead to the identification of a set of design techniques which, together, form a coherent and systematic approach for the design of high performance ASIPs usable for real-time video processing.

The proposed design approach aims at an efficient utilization of available bandwidth to memory, which constitutes the main performance bottleneck of the application. It is possible to approach the processing speed limit imposed by this bottleneck through an appropriate data reuse strategy and by exploiting the data parallelism inherent to the target algorithm class. The design approach comprises four steps: first, a Single Instruction Multiple Data (SIMD) instruction which calculates more than one pixel in parallel is created. Then, shift registers, which are used for intra-line input pixel reuse, are added. Next, a processing pipeline is created by the addition of application-specific registers. Finally, the custom load/store instructions are created. Some of these steps lead to possible hardware simplifications for some algorithms of the target class. The hardware structure thus obtained, together with the instruction-level parallelism made possible through the use of a Very Long Instruction Word (VLIW) architecture, mimics a pipelined systolic array.

In order to demonstrate the validity of the proposed design approach experimentally, seven ASIPs have been designed by extending the instruction-set of a configurable and extensible processor. Three of the ASIPs implement intra-field deinterlacing algorithms, and four implement the 2D convolution with different kernel sizes.

The results show a significant improvement in performance. For the intra-field deinterlacing algorithms, speedup factors are between 95 and 1330, while the factors of im-

provement of the Area-Time (AT) product are between 29 and 243, all this compared to a pure software implementation running on a general-purpose processor. In the case of the two-dimensional convolution, speedup factors are between 36 and 80, while factors of improvement of the AT product are between 12 and 22. In all cases, real-time processing of high definition video in the 1080i (deinterlacing) or 1080p (convolution) format is possible given a 130 nm manufacturing process.

Table des matières

Liste des tableaux

Liste des figures

Liste des listages de code source

Liste des sigles et abréviations

ASIP	*Application-Specific Instruction-Set Processor*
AT	*Area-Time product*, produit temps-surface
DSP	*Digital Signal Processor*
ELA	*Edge-Based Line Averaging*
FPGA	*Field-Programmable Gate Array*
GRM	Groupe de recherche en microélectronique et microsystèmes
PBDI	*Pattern-Based Directional Interpolation*
SIMD	*Single Instruction Multiple Data*
TIE	*Tensilica Instruction Extension*
VHDL	*VHSIC Hardware Description Language*
VHSIC	*Very-High-Speed Integrated Circuits*
VLIW	*Very Long Instruction Word*

Chapitre 1

INTRODUCTION

Le nombre de transistors à coût minimal pouvant être placés sur une puce double tous les deux ans, tendance décrite par la loi de Moore [1]. Cette loi sert de guide sur lequel s'aligne toute l'industrie de la microélectronique. La puissance de calcul d'une puce étant fortement corrélée au nombre de transistors, celle-ci évolue aussi dans le temps selon une courbe exponentielle. Ceci se traduit non seulement par une augmentation continue de la puissance de calcul des processeurs, mais aussi par la succession des nouvelles applications rendues possibles.

Un domaine d'application ayant particulièrement bénéficié de cette évolution technologique est le traitement vidéo. On observe en effet sur le marché une explosion des applications de vidéo numérique, notamment du côté de la télévision haute définition et des lecteurs vidéo portatifs. Des produits inimaginables il y a à peine dix ans sont maintenant monnaie courante et abordables.

Par contre, tout ceci ne facilite pas la tâche des concepteurs de matériel électronique. L'évolution de la complexité des systèmes numériques sur puce suit celle du nombre de transistors, tendance que la productivité des ingénieurs chargés de concevoir ces systèmes peine à suivre. Il est sans cesse nécessaire de réinventer les méthodes de conception de manière à être à même de tirer pleinement avantage de la technologie actuelle. C'est de cette nécessité d'augmenter la productivité de la conception que sont issus les processeurs spécialisés et les outils d'aide à leur conception. Les processeurs spécialisés permettent la mise en œuvre de fonctions complexes avec une performance en mesure de rivaliser avec celle des circuits intégrés dédiés, mais avec un effort de conception plus près de celle des solutions logicielles s'exécutant sur un processeur d'usage général.

1.1 Aperçu et objectifs

Les travaux ci-décrits s'inscrivent dans le cadre des activités du groupe vidéo du Groupe de Recherche en Microélectronique et Microsystèmes (GRM) de l'École. Ils visent à explorer les possibilités qu'offrent les processeurs à jeu d'instructions spécialisé pour les applications vidéos. Spécifiquement, les travaux ont porté sur une classe particulière d'algorithmes de traitement vidéo : les filtres locaux. Pour cette classe d'algorithmes, une exploration architecturale a permis d'identifier un ensemble de techniques formant une approche cohérente et systématique pour la conception de processeurs spécialisés performants. Le suivi de cette approche permet la conception rapide de processeurs spécialisés adaptés au traitement vidéo en temps réel.

Ces travaux ont fait l'objet d'un article soumis à la revue *Transactions on Very Large Scale Integration (VLSI) Systems* de l'IEEE, ainsi que d'un article présenté à la conférence NEWCAS-TAISA 2009 [2] de l'IEEE. L'article de revue constitue le chapitre 4 de ce mémoire.

1.2 Méthodologie

L'approche de conception présentée dans ce mémoire a été proposée à l'issue d'une exploration architecturale. Sept processeurs spécialisés ont été conçus dans le but d'en démontrer expérimentalement la validité. Les trois premiers visent autant d'algorithmes de désentrelacement intra-trames, une sous-classe des filtres locaux. Le critère principal ayant mené à la sélection de ces trois algorithmes est qu'ils sont de complexités logicielles très différentes. Quatre processeurs spécialisés pour la convolution 2D ont aussi été conçus. Chacun de ces quatre processeurs supporte une taille distincte de fenêtre de convolution.

Le processeur configurable et extensible Xtensa LX2 et son environnement de conception Xtensa Xplorer de Tensilica ont été sélectionnés pour réaliser l'exploration architec-

turale. Quoique ce choix ait été influencé principalement par la disponibilité des outils, il s'agit néanmoins d'une solution intéressante pour ce genre de tâche, en particulier du point de vue de la facilité d'utilisation et de la durée des cycles de développement.

Les résultats de performance obtenus sont issus de simulations réalisées à l'aide de la suite d'outils de Tensilica, ainsi que d'estimations fournies par ces outils concernant la fréquence d'horloge et la taille du processeur. Nous nous sommes intéressés à certaines métriques de performance particulières. La première est le facteur d'accélération par rapport à une implémentation logicielle de l'algorithme roulant sur un processeur d'usage général. Le processeur en question choisi comme référence est le Xtensa LX2 sans extension du jeu d'instruction. Dans cette configuration, il s'agit d'un processeur RISC 32 bits semblable à bien d'autres.

La deuxième métrique sur laquelle s'est portée notre attention est le nombre de pixels produits par seconde. Cette métrique permet d'extrapoler les résultats à diverses situations de définition d'image et de taux d'image de manière à vérifier, par exemple, si la vitesse d'exécution est suffisante pour permettre le traitement en temps réel dans ces situations. Contrairement aux autres métriques de performance considérées, celle-ci a l'inconvénient de dépendre du procédé de fabrication utilisé. Dans tous les cas, les estimations de fréquence d'horloge nécessaires à déterminer la valeur de cette métrique ont été obtenues des outils en supposant un même procédé CMOS à 130 nm.

La troisième métrique considérée est la taille relative, soit le rapport entre la taille, en nombre de portes logiques, du processeur spécialisé, et celle du processeur d'usage général de référence. Cette métrique est une mesure du coût matériel des accélérations obtenues.

Finalement, le facteur d'amélioration (et donc de diminution) du produit temps-surface a été considéré. Celui-ci est obtenu en divisant le facteur d'accélération par la taille relative. Cette métrique donne une mesure de l'amélioration de l'efficacité d'utilisation de la surface de silicium. Par exemple, un facteur d'amélioration du produit temps-surface de 243 signifie qu'on obtient un traitement plus rapide par un facteur de 243 pour une surface de

silicium donnée. Il a d'ailleurs été suggéré que cette métrique représente de plus un estimateur fiable et pessimiste de l'amélioration obtenue en termes d'efficacité énergétique [3, 4].

1.3 Plan du mémoire

Le chapitre 2 de ce mémoire présente une revue de la littérature pertinente ainsi que le contexte technique de la recherche. Il est découpé de la manière suivante : la section 2.1 introduit les filtres locaux et discute de leurs caractéristiques. La section 2.2 donne un aperçu du désentrelacement, notamment concernant son rôle, ses implications et les différentes méthodes pouvant être utilisées pour le mettre en œuvre. La section 2.3 discute plus spécifiquement du désentrelacement intra-trame et présente les trois algorithmes de désentrelacement pour lesquels des processeurs spécialisés ont été conçus dans le cadre des travaux ci-décrits. Finalement, la section 2.4 présente les processeurs spécialisés.

Le chapitre 3 présente la démarche de l'ensemble du travail de recherche et situe l'article présenté au chapitre 4 dans le cadre de cette recherche. On trouve notamment dans ce chapitre le résumé en français de l'article.

Le chapitre 4 présente l'article *Real-Time Computation of Local Neighbourhood Functions in Application-Specific Instruction-Set Processors* soumis à la revue *Transactions on Very Large Scale Integration (VLSI) Systems* de l'IEEE. Cet article constitue le cœur du présent mémoire, et expose l'approche de conception proposée et les résultats de performance obtenus.

Finalement, le chapitre 5 contient une discussion générale concernant l'approche de conception proposée et la méthodologie utilisée, et le chapitre 6 conclut le mémoire.

Chapitre 2

CONTEXTE ET REVUE DE LITTÉRATURE

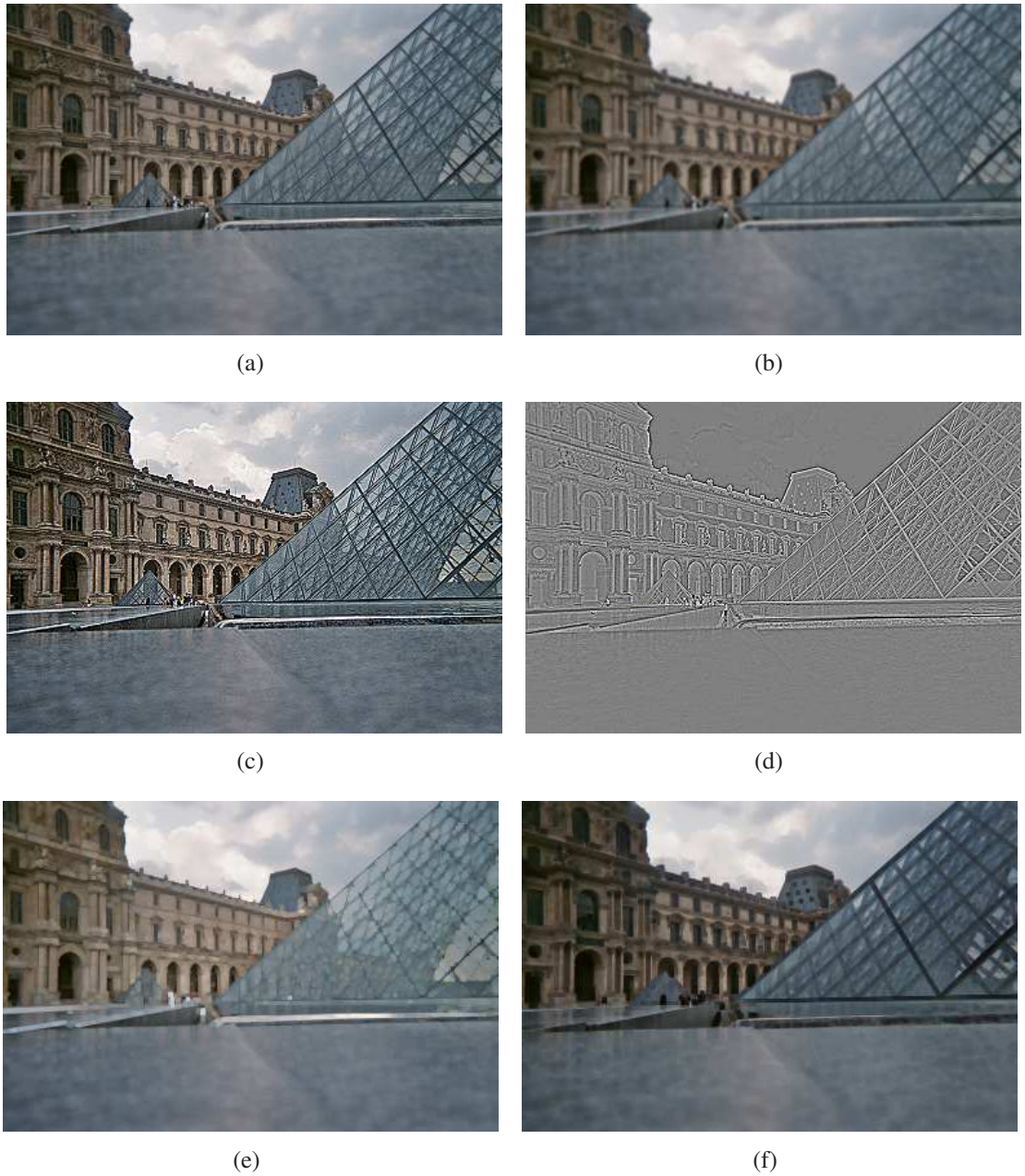
2.1 Filtres locaux

Les *filtres locaux*, ou *fonctions de filtrage local* (angl. *local neighbourhood functions*), forment une classe d'algorithmes couramment utilisés en traitement d'image. Un filtre local permet de transformer une image, c'est-à-dire de produire une image-résultat à partir d'une image originale. Lors d'une transformation par filtre local, chaque pixel de l'image-résultat est généré en appliquant une certaine fonction à un voisinage restreint de pixels de l'image originale : la *fenêtre*. Une définition formelle des filtres locaux est donnée à la section 4.3.

Il existe plusieurs filtres locaux. Quelques uns sont présentés ici. Tout d'abord, parmi les plus utilisés se trouve la *convolution bidimensionnelle* ou *convolution 2D*. Une image est transformée par la convolution 2D en multipliant la valeur de chaque pixel de la fenêtre par un coefficient, ou poids. En supposant une taille de fenêtre de $h \times w$, ceci peut être exprimé ainsi

$$q_{i,j} = \sum_{k=0}^{h-1} \sum_{l=0}^{w-1} a_{k,l} \cdot p_{i-k,j-l} \tag{2.1}$$

où $q_{i,j}$ représente le pixel de position (i, j) dans l'image-résultat, $p_{i,j}$ représente le pixel de position (i, j) dans l'image originale et les $a_{k,l}$ représentent les coefficients. Les coefficients $a_{k,l}$ sont typiquement représentés dans une matrice, nommée *matrice de convolution*. En variant ceux-ci, il est possible d'obtenir divers effets. Quelques exemples sont donnés à la figure 2.1 (les matrices de convolution ont été prises de [5] pour ces exemples). La figure 2.1(b) montre une figure originale 2.1(a) transformée de manière à obtenir un effet de flou, alors que les figures 2.1(c) et 2.1(d) présentent respectivement une accentuation des



(a) (b)

(c) (d)

(e) (f)

FIGURE 2.1 Exemples de filtres locaux : (a) original, (b) flou, (c) amélioration de la netteté, (d) détection des contours, (e) dilatation et (f) érosion

détails et le résultat de l'application d'un filtre de détection des contours. Ces trois effets peuvent être obtenus par la convolution 2D avec une taille de fenêtre de 3×3, simplement en utilisant des matrices de convolution différentes.

Les filtres locaux comprennent aussi les morphologies mathématiques qui, contrairement à la convolution 2D, sont non-linéaires. Deux exemples de morphologies mathématiques sont données à la figure 2.1 : la dilatation (minimum sur l'élément structurant) à la figure 2.1(e) et l'érosion (maximum sur l'élément structurant) à la figure 2.1(f). Finalement, les filtres de désentrelacement intra-trame, décrits plus bas, sont aussi des filtres locaux.

Plusieurs caractéristiques des filtres locaux font en sorte qu'il soit possible d'en faire une mise en œuvre performante au sens de la vitesse de traitement. Premièrement, puisque chaque pixel de l'image-résultat est produit de manière indépendante, les algorithmes de cette classe présentent un fort parallélisme des données. Ceci rend possible le calcul de plusieurs pixels en parallèle. Deuxièmement, le fait que chaque pixel de sortie ne dépende que des pixels d'entrée situés dans une fenêtre restreinte se traduit par une très bonne localité des données. Ceci est avantageux lorsqu'on cherche à tirer avantage de caches ou de registres pour réutiliser les données, de manière à éviter d'avoir à les recharger. Finalement, le fait que les pixels de sortie soient générés de manière indépendante les uns des autres implique que l'ordre dans lequel ils sont générés n'importe pas. Il est donc possible de parcourir les pixels de l'image originale dans l'ordre le plus avantageux. Lorsque l'image est parcourue de manière régulière, c'est-à-dire ligne par ligne (ou colonne par colonne), dans l'ordre, la distance maximale de réutilisation des données est bornée et minimale. Dans cette situation, si on désire emmagasiner dans une mémoire (cache, registres ou autres) tous les pixels d'entrée de l'image originale, chacun durant la totalité de son temps de vie, la taille d'une telle mémoire ne sera que celle nécessaire à contenir les pixels de quelques lignes d'image.

Une topologie de circuit logique adaptée aux filtres locaux, et donc très utilisée pour leur mise en œuvre, est le *réseau systolique linéaire* [6]. Dans un tel réseau systolique,

à mesure que les pixels d'entrée sont lus suivant l'ordre de parcours régulier suggéré au paragraphe précédent, ils sont placés dans un long registre à décalage. Chaque fois qu'un nouveau pixel d'entrée est ajouté (à chaque cycle d'horloge), un nouveau pixel de sortie est produit. Pour ce faire, la fonction du filtre est appliquée aux pixels de la fenêtre, qui se trouvent à des positions définies et fixes dans le registre à décalage. L'ordre de parcours régulier fait en sorte que les pixels se retrouvent au bon endroit, au bon moment. Une telle topologie permet d'obtenir une vitesse de traitement élevée, puisque chaque pixel n'est lu qu'une seule fois. Il peut être possible d'augmenter encore le débit de pixels produits en pipelinant la logique de mise en œuvre de la fonction du filtre. Nous obtenons alors un *réseau systolique pipeliné*.

On retrouve dans la littérature récente une alternative intéressante pour la mise en œuvre des filtres locaux [7]. Il s'agit d'une architecture composée de cellules programmables. Chaque cellule possède une mémoire locale lui permettant de stocker une région de l'image dont la hauteur est de quelques lignes et la largeur est d'environ la largeur de l'image divisée par le nombre de cellules (il y a en fait un léger chevauchement entre les régions). De plus, plutôt que d'avoir un chemin des données, éventuellement pipeliné, permettant le calcul de la fonction complète du filtre local, chaque cellule possède plutôt un chemin des données plus simple et générique nécessitant, contrairement au cas typique d'un réseau systolique, plus d'un cycle d'horloge pour produire un pixel. Par contre, le fait que plusieurs cellules travaillent en parallèle, chacune sur sa région de l'image, permet d'atteindre un débit d'un pixel par cycle d'horloge. Par exemple, si une cellule nécessite n cycles d'horloge pour générer un pixel, n cellules seront utilisées, et l'image sera divisée en n bandes verticales, chacune attitrée à une cellule. Selon les auteurs, l'avantage principal de cette architecture est sa flexibilité. En effet, le chemin des données plus générique et la programmabilité des cellules font en sorte qu'il soit possible de modifier le filtre local mis en œuvre en modifiant le microcode des cellules.

2.2 Désentrelacement

Les systèmes de télévision analogique sont appelés à disparaître, remplacés progressivement par des systèmes numériques. Durant l'actuelle période de transition, il est nécessaire d'être en mesure de convertir les signaux de télévision analogiques en signaux conformes aux nouvelles normes numériques pour fins de compatibilité. Une étape importante de cette conversion est le désentrelacement.

2.2.1 Balayage entrelacé et désentrelacement

Le fonctionnement de la télévision analogique est celui-ci : une caméra balaie une scène de manière répétée, ligne par ligne, produisant des signaux analogiques représentant l'intensité lumineuse et la couleur. Ces signaux sont acheminés, via un canal de communication, vers un téléviseur, où ils sont utilisés, dans la technologie des téléviseurs à écran cathodique, pour moduler l'intensité de faisceaux d'électrons qui balaient les luminophores de l'écran, aussi ligne par ligne, reconstituant ainsi les images capturées. L'image a donc une résolution verticale fixe en nombre de lignes. Sur une ligne donnée, on ne peut pas réellement parler de résolution fixe en nombre de points ou de colonnes, puisque la ligne est décrite par des signaux analogiques variant de manière continue dans le temps. Par contre, il y a bien une limite au niveau détail possible, liée à la bande passante du canal.

La question de l'ordre dans lequel les lignes sont balayées et transmises se pose. Dans le cas le plus simple, toutes les lignes sont balayées dans l'ordre, de sorte que des images complètes sont transmises l'une à la suite de l'autre. Ce cas est nommé *balayage progressif*. En pratique, les normes de télévision analogique spécifient un ordre de balayage légèrement plus complexe nommé *balayage entrelacé*. Dans le cas du balayage entrelacé, seulement que les lignes paires ou impaires, en alternance, sont balayées et transmises. Il y a donc transmission d'une demi-image à la fois, constituée soit que de lignes paires, soit que de lignes impaires. Une telle demi-image est nommée *trame* ou encore *champ*.

Le balayage entrelacé permet d'augmenter le taux de rafraîchissement de l'écran sans nécessiter une plus grande bande passante. Par exemple, dans le cas de la norme NTSC, la bande passante du canal permet la transmission d'environ 30 images par seconde au maximum. Transmettre 60 trames plutôt que 30 images complètes chaque seconde nécessite la même bande passante, mais double le taux de rafraîchissement.

Les téléviseurs et équipements vidéo modernes effectuent, après conversion analogique-numérique des signaux analogiques reçus, divers traitements numériques des données vidéo. Le balayage entrelacé complexifie ces traitements, et il est donc nécessaire de convertir la vidéo à balayage entrelacé en vidéo à balayage progressif. Une telle conversion est nommée *désentrelacement*.

2.2.2 Critère de Nyquist

Lorsque des signaux vidéo sont reçus, on aimerait idéalement être en mesure de reconstituer parfaitement l'image de la scène originale à partir de ceux-ci. Or, le balayage d'une image ligne par ligne constitue un échantillonnage vertical. Le critère de Nyquist dicte donc les conditions sous lesquelles ceci est possible. Spécifiquement, pour qu'elle puisse être reconstituée sans perte d'information, une image parfaite de la scène ne doit pas comporter de composantes fréquentielles verticales de fréquence plus élevée ou égale à la moitié de la fréquence d'échantillonnage vertical. Plus intuitivement, ceci signifie que l'ordre de grandeur des détails les plus fins de l'image doit être telle que ceux-ci soient traversés par plusieurs lignes d'image. Puisque le nombre de lignes d'image des normes de télévision varie entre seulement quelques centaines de lignes pour la télé analogique régulière et environ un millier de lignes pour la télé haute définition, alors qu'une scène réelle contient typiquement des détails arbitrairement fins, il va sans dire que le critère de Nyquist n'est pratiquement jamais respecté.

Les conséquences d'une violation du critère de Nyquist sont plus graves qu'une simple perte d'information concernant les détails les plus fins. S'il y a bien des composantes fré-

FIGURE 2.2 Exemple de repliement spectral (chemise du présentateur)[1]

quentielles verticales de fréquence trop élevée, on peut observer des artefacts causés par le phénomène de repliement spectral. Au lieu de simplement disparaître, ces composantes se retrouvent dans l'image sous forme de composantes de fréquence spatiale plus faible. On trouve un exemple de repliement spectral à la figure 2.2, sur la chemise du présentateur de nouvelles. La manière habituelle d'éviter le repliement spectral en traitement du signal est d'introduire un filtre passe-bas – un filtre anti-repliement – entre la source du signal et l'endroit où se produit l'échantillonnage, de manière à éliminer les composantes fréquentielles dommageables. Cette manière de faire n'est cependant pas applicable dans le cas qui nous occupe, puisque la transformation par un capteur de la grandeur physique qui nous intéresse – la luminosité – en un signal électrique suit l'échantillonnage plutôt que de le précéder [8].

[1]Cette image est tirée d'un bulletin de nouvelles de mardi le 8 juin 2010 par la chaîne de nouvelles Al Jazeera English. Le bulletin de nouvelles complet est disponible à l'adresse `http://www.youtube.com/watch?v=Vh2ijY7X9KY`.

Une trame contient la moitié moins de lignes qu'une image complète, ce qui implique une plus grande quantité d'information perdue. Cette perte d'information supplémentaire fait en sorte qu'il ne soit pas possible, dans le cas général, de reconstituer une image complète à partir d'une trame, ce qui est précisément l'objectif du désentrelacement. Tout n'est pas perdu cependant. En effet, les images qui sont d'intérêt pour le téléspectateur ne sont pas constituées de variations arbitraires d'intensité et de couleur, mais représentent plutôt des objets continus, définis par des contours et présents d'une image à l'autre. Les pixels d'une image sont donc fortement corrélés à leurs voisins dans l'espace et dans le temps. Il est possible, par des méthodes heuristiques, de tirer avantage de ce fait afin d'obtenir une approximation acceptable d'une image complète à partir d'une trame.

2.2.3 Méthodes de désentrelacement

Un aperçu très complet des méthodes de désentrelacement est donné dans [8]. On peut tout d'abord distinguer trois types de méthodes de base : les méthodes intra-trames, inter-trames et à compensation du mouvement. Les méthodes intra-trames ne considèrent que l'information contenue dans la trame présente afin de retrouver l'image complète. Il s'agit donc d'une interpolation spatiale. Des méthodes simples de ce type, donnant des résultats de piètre qualité visuelle, consistent à simplement recopier les lignes existantes, ou encore à retrouver les pixels des lignes manquantes par simple interpolation linéaire des pixels voisins. De telles méthodes sont quelquefois utilisées dans les logiciels de lecture vidéo sur PC puisqu'elles nécessitent peu de temps de calcul. Il en existe d'autres, plus performantes et complexes, notamment celles basées sur les arêtes, décrites à la section suivante. Les méthodes intra-trames, peu importe leur complexité, ont l'avantage de ne nécessiter généralement que très peu de mémoire pour leur mise en œuvre.

Les méthodes inter-trames sont, quant à elles, des méthodes d'interpolation temporelle. Une méthode simple et très commune, souvent utilisée elle aussi dans les logiciels de lecture vidéo sur PC, consiste à recopier les lignes d'une trame dans la trame suivante. Cette

méthode donne de très bons résultats dans les régions statiques de l'image. Ce n'est pas le cas cependant des régions où il y a du mouvement. Le mouvement horizontal rapide, en particulier, engendre l'effet de peigne si caractéristique.

Finalement, les méthodes à compensation du mouvement, plus gourmandes en puissance de calcul, comportent deux étapes. Premièrement, l'estimation du mouvement consiste à trouver la position de mêmes objets dans différentes trames, et à utiliser cette information pour déterminer les vecteurs de déplacement de ces objets entre ces trames. Cette étape est généralement réalisée par une recherche des blocs de taille fixe à ressemblance maximale entre une trame et une autre. Les vecteurs de déplacement ainsi obtenus sont ensuite utilisés pour interpoler ou extrapoler la trajectoire des objets afin de les placer au bon endroit dans des images reconstituées à partir d'autres trames.

Il est possible de combiner ces types de méthodes de base. Par exemple, les méthodes à adaptation au mouvement consistent à détecter les régions statiques et à utiliser des méthodes inter-trames pour celles-ci, alors que des méthodes intra-trames sont utilisées dans les régions où il y a mouvement. Une autre possibilité consiste à utiliser une méthode à compensation du mouvement, complétée d'une méthode intra-trame pour les régions où il n'est pas possible d'obtenir des vecteurs de déplacement fiables.

2.3 Désentrelacement intra-trame

Les travaux ci-décrits portent sur des méthodes de désentrelacement intra-trames. Celles-ci ont pour avantage d'être généralement peu coûteuses en termes de puissance de calcul requise, et de ne nécessiter que peu de mémoire pour le stockage temporaire de pixels originaux. En effet, lors de la génération d'un pixel sur une ligne manquante, seuls les pixels voisins dans la même trame sont considérés. Les exigences en termes de mémoire ne sont donc que de quelques lignes d'image, par opposition à quelques trames ou images pour les méthodes inter-trames et à compensation du mouvement. Les méthodes intra-trames ont

donc leur pertinence dans les applications où la puissance de calcul disponible est limitée. Elles sont aussi utiles comme complément à d'autres méthodes afin d'obtenir des résultats de meilleure qualité visuelle dans les situations particulières où il est connu que ces dernières donnent des résultats de qualité sous-optimale.

Les trois algorithmes de désentrelacement ayant fait l'objet d'une mise en œuvre dans le cadre de ces travaux sont décrits ci-après. Ils font partie des méthodes basées sur les arêtes, un sous-ensemble des méthodes intra-trames qui vise principalement à éviter un type d'artefact particulier : le crénelage des arêtes. Pour ce faire, les arêtes sont détectées, et l'interpolation spatiale est réalisée parallèlement à celles-ci dans leur voisinage. La qualité d'une méthode intra-trame repose principalement sur la qualité de la détection des arêtes, et sur la résolution angulaire de la méthode.

2.3.1 Algorithme ELA

L'algorithme *Edge-Based Line Averaging* (ELA) [9] est très simple, et illustre bien le principe de fonctionnement des méthodes basées sur les arêtes. Chaque pixel d'une ligne manquante est généré de manière indépendante en considérant les trois pixels juste au-dessus et les trois juste au-dessous, comme c'est illustré à la figure 2.3. Tout d'abord, les pixels originaux sont comparés deux à deux dans chacune des trois directions notées a, b et c à la figure 2.3. Chaque comparaison consiste à calculer la valeur absolue de la différence de luminance entre les pixels, soit

$$\delta_{ELA} = |p_1 - p_2| \tag{2.2}$$

où p_1 et p_2 représentent les luminances des pixels comparés. Ensuite, une interpolation linéaire (moyenne arithmétique) est réalisée, de manière indépendante pour chaque composante de couleur, entre les deux pixels pour lesquels cette valeur est minimale.

Cette méthode est donc basée sur l'hypothèse qu'en présence d'une arête, la valeur

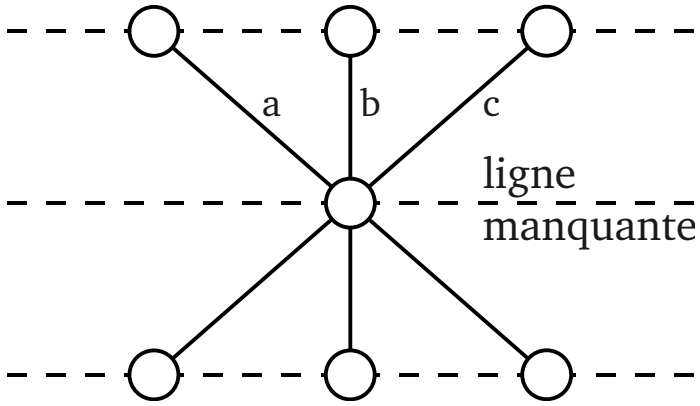


FIGURE 2.3 Comparaisons de pixels et directions d'interpolation pour ELA

absolue de la différence de luminance est minimale dans la direction parallèle à celle-ci. Une limitation importante de l'algorithme ELA est sa faible résolution angulaire. En effet, seules trois directions d'interpolation sont possibles.

2.3.2 Algorithme Enhanced ELA

L'algorithme Enhanced ELA [10] est en fait l'algorithme nommé Modified ELA [11] auquel une étape de filtrage médian est ajoutée. Comme le montre la figure 2.4, cinq directions sont considérées plutôt que trois, ce qui constitue une amélioration par rapport à ELA au niveau de la résolution angulaire. Une méthode de détection des arêtes plus fiable, mais plus complexe, est aussi utilisée.

La première étape consiste à calculer la valeur absolue de la différence de luminance dans la direction verticale, soit celle notée c à la figure 2.4. Cette valeur est comparée à un seuil fixe. Si elle est sous le seuil, il est déterminé qu'aucune arête ne traverse ce pixel, et l'interpolation linéaire est alors réalisée dans la direction verticale. Autrement, les quatre autres comparaisons sont effectuées. Si la différence absolue est minimale dans la direction verticale, c'est suivant cette direction qu'est effectuée l'interpolation.

Dans le cas d'une arête oblique, il est nécessaire de vérifier que celle-ci soit dominante.

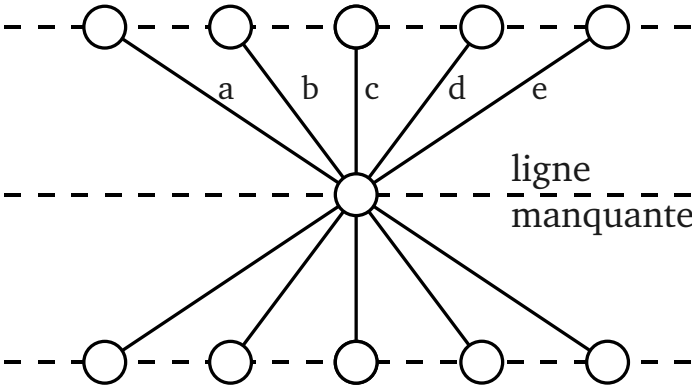


FIGURE 2.4 Comparaisons de pixels et directions d'interpolation pour Enhanced ELA

Pour ce faire, la valeur absolue de la différence de luminance pour les deux directions perpendiculaire à l'arête sont comparées à un second seuil. Par exemple, si la différence absolue dans la direction b est minimale, celles des directions d et e sont comparées au seuil. Les deux valeurs doivent être au-dessus du seuil pour que l'arête soit déclarée dominante. Dans ce cas, l'interpolation est réalisée dans la direction de l'arête, alors qu'elle est effectuée dans la direction verticale dans le cas contraire.

Finalement, si une arête dominante a été détectée, suite à l'interpolation, un filtre médian est appliqué des manière à réduire les effets de pixels éclatants. Le pixel choisi comme pixel final est celui de luminance médiane entre 1) le résultat de l'interpolation, 2) le pixel original juste au-dessus et 3) celui juste au-dessous.

2.3.3 Algorithme PBDI

L'algorithme nommé Pattern-Based Directional Interpolation (PBDI) a été développé au GRM par Hossein Mahvash Mohammadi, étudiant au doctorat. La description de cet algorithme se trouve dans sa thèse de doctorat [12]. Le processeur spécialisé conçu pour cet algorithme est décrit dans [2], où on retrouve aussi une description sommaire de l'algorithme.

La particularité de PBDI par rapport aux autres algorithmes décrits plus haut est que ce ne sont pas les pixels individuels qui sont comparés deux à deux, mais bien des patrons de pixels, ce qui a pour effet de rendre la détection des arêtes plus robuste. Les patrons en question sont formés de pixels situés sur une même ligne et horizontalement adjacents. Dans [12], il est fait mention que diverses tailles de patrons sont possibles, mais une taille de trois pixels est supposée. Nous effectuons la même supposition pour le reste de notre description de l'algorithme, et c'est cette taille qui a été utilisée lors de la conception du processeur spécialisé pour cet algorithme.

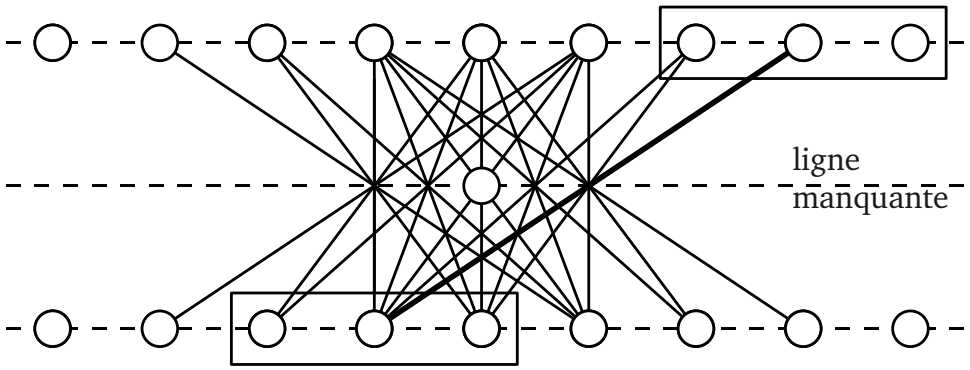


FIGURE 2.5 Comparaisons de patrons pour PBDI

Une fenêtre de recherche de neuf pixels est définie sur les lignes au-dessus et au-dessous de la ligne manquante, comme c'est illustré à la figure 2.5. Dans une telle fenêtre de neuf pixels se trouvent sept patrons de trois pixels. Il existe donc $7 \times 7 = 49$ couples de patrons tels que l'un des patrons est pris dans la fenêtre de recherche du haut et l'autre dans celle du bas. Cependant, ce ne sont pas tous ces couples qui représentent des comparaisons pertinentes. Premièrement, pour plusieurs couples, le segment de droite reliant les pixels centraux des patrons ne passe pas suffisamment près du pixel qu'on cherche à générer. D'autres couples doivent aussi être éliminés puisque le faire permet à l'algorithme d'être moins susceptible aux arêtes trompeuses. Au final, 21 comparaisons sont effectuées, soit les 21 montrées à la figure 2.5.

La fonction utilisée pour la comparaison est plus complexe que celle utilisée dans le cas

des autres algorithmes décrits plus haut. Elle permet d'obtenir une détection plus fiable des arêtes en présence de dégradés de luminosité sur celles-ci. Il s'agit en fait de la combinaison linéaire de deux fonctions de comparaison : la somme des différences absolues (SAD) et la différence de dégradé (GBD) :

$$\delta_{PBDI} = \alpha \cdot \delta_{SAD} + (1 - \alpha) \cdot \delta_{GBD}. \tag{2.3}$$

La SAD est l'extension logique pour des patrons de pixels de la fonction de comparaison utilisée dans le cas des deux autres algorithmes. Elle est obtenue en calculant la valeur absolue de la différence de luminance entre les pixels de droite de chaque patron du couple, en effectuant ce même calcul pour les pixels centraux et pour ceux de gauche, et en additionnant les trois valeurs ainsi obtenues, soit

$$\delta_{SAD} = |p_{1g} - p_{2g}| + |p_{1c} - p_{2c}| + |p_{1d} - p_{2d}| \tag{2.4}$$

où p_{1g}, p_{1c} et p_{1d} représentent respectivement la luminance du pixel de gauche, du centre et de droite du premier patron, et p_{2g}, p_{2c} et p_{2d} représentent les pixels de même position, mais pour le deuxième patron. La SAD quantifie la similarité des patrons en termes de luminosité. La GBD quantifie quant à elle la similarité en termes de texture. C'est ce qui permet de détecter correctement les arêtes même en présence d'un dégradé de luminosité. Elle est calculée de la manière suivante : premièrement, pour chacun des deux patrons, une différence de gauche et une différence de droite sont calculées. La différence de gauche est définie comme la différence de luminance entre le pixel de gauche et le pixel central du patron. La différence de droite est quant à elle définie comme la différence de luminance entre le pixel central et le pixel de droite du patron. Lorsque ces deux valeurs ont été calculées pour chaque patron, on obtient la GBD en additionnant 1) la valeur absolue de la différence entre la différence de gauche du premier et du second patron ; et 2) la valeur absolue de la différence entre la différence de droite du premier patron et du second patron,

soit

$$\delta_{GBD} = |(p_{1g} - p_{1c}) - (p_{2g} - p_{2c})| + |(p_{1c} - p_{1d}) - (p_{2c} - p_{2d})| \tag{2.5}$$

On peut remarquer à la figure 2.5 que plusieurs comparaisons sont dans des directions parallèles entre elles. La figure 2.6 montre les neuf directions distinctes présentes. Il s'agit aussi des neuf directions d'interpolation. L'interpolation, dans le cas de cet algorithme, n'est pas forcément effectué entre les pixels centraux du couple de patrons pour lesquels la valeur de la fonction de comparaison est minimale. Par contre, ce couple indique la direction de l'arête, qui est l'une des neuf directions de la figure 2.6. C'est parallèlement à cette direction qu'est effectuée l'interpolation. La possibilité de détecter neuf directions d'arêtes distinctes constitue une amélioration de la résolution angulaire par rapport aux algorithmes ELA et Enhanced ELA.

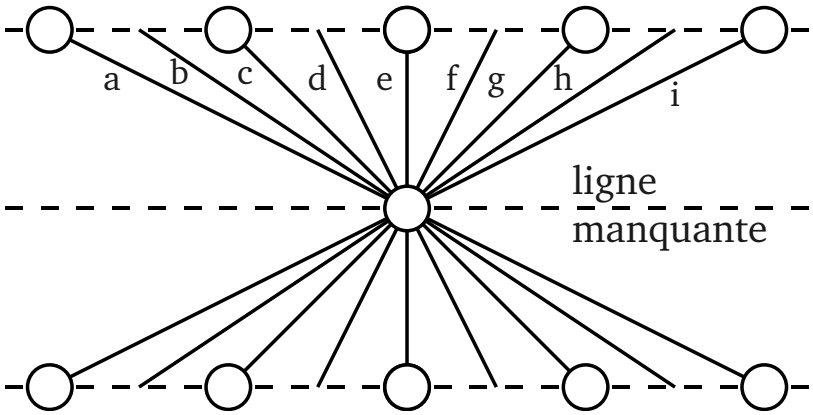


FIGURE 2.6 Directions d'interpolation pour PBDI

Si les comparaisons de patrons indiquent que la direction de l'arête est l'une de celles identifiées a, c, e, g et i à la figure 2.6, alors une interpolation linéaire entre les deux pixels reliés par le segment de droite correspondant à cette direction est effectuée. Par contre, pour les quatre directions restantes, soit b, d, f et h, un problème se pose. En effet, le segment de droite représentant chacune de ces directions n'intercepte aucun pixel dans les fenêtres de recherche supérieure et inférieure. Dans ce cas, l'interpolation est effectuée entre quatre pixels plutôt que deux. Par exemple, s'il a été déterminé que l'arête est orientée suivant la

direction d, l'interpolation linéaire est effectuée entre les pixels constituant les sommets du parallélogramme de diagonales c et e. Après que le pixel interpolé ait été obtenu, un filtre médian est appliqué comme dans le cas de l'algorithme Enhanced ELA.

2.4 Processeurs spécialisés et outils de conception

Un processeur à jeu d'instructions spécialisé, ou simplement processeur spécialisé (angl. *Application-Specific Instruction-set Processor* – ASIP) est un processeur dont le jeu d'instruction est adapté à une application ou à un domaine d'application précis. Cette adaptation permet généralement une exécution plus rapide et/ou avec une meilleure efficacité énergétique. D'autres objectifs peuvent aussi être recherchés, par exemple un processeur de plus petite taille.

L'utilisation de processeurs spécialisés pour divers domaines d'application a fait l'objet de plusieurs publications récentes. Pour en citer quelques uns, une architecture de processeur spécialisé pour la transformée de Fourier rapide visant à améliorer la performance par une réduction du nombre d'accès à la mémoire est proposée dans [13] ; un processeur spécialisé décrit à l'aide du langage LISA couplé à un coprocesseur et destiné à être utilisé dans le récepteur d'un système de géolocalisation est présenté dans [14] ; un processeur spécialisé reconfigurable permettant le décodage de divers types de codes correcteurs est présenté dans [15] ; il est proposé dans [16] que les choix effectués dans le cadre d'une exploration algorithmique doivent tenir compte de ce que ces choix impliquent quant aux possibilités d'optimisation par les outils automatiques d'identification d'instructions spécialisées, ce qui est supporté par une étude de cas concernant la cryptographie ; une architecture de processeur spécialisé composée d'un processeur de signaux numériques (DSP) et d'un accélérateur matériel et visant le décodage de données vidéo encodées au format H.264/MPEG-4 AVC est proposée dans[17] ; une architecture de processeur spécialisé pour le traitement vidéo en temps réel par un filtre de correction des couleurs Retinex est présentée dans [18].

Il existe présentement sur le marché quelques solutions commerciales permettant la conception rapide de processeurs spécialisés. Trois de ces solutions sont décrites ci-après : l'une proposée par CoWare, une autre par Tensilica et la dernière par Altera. Le problème est approché de manière très différente par chacune des trois solutions décrites.

2.4.1 Processor Designer de CoWare et langage LISA

L'approche privilégiée par CoWare (entreprise récemment acquise par Synopsys) est l'utilisation d'un langage de description architecturale permettant de décrire complètement le fonctionnement du processeur. Le langage en question se nomme LISA. Avec le langage LISA, le concepteur a un contrôle complet sur le jeu d'instruction du processeur et sur presque tous les détails de son architecture.

Le logiciel Processor Designer permet de générer automatiquement un assembleur et un simulateur, notamment, à partir du modèle LISA du processeur. Il est aussi possible de créer une version du compilateur C CoSy ciblant ce processeur en définissant un ensemble de règles permettant de faire correspondre la représentation interne du code qu'utilise le compilateur et le jeu d'instructions du processeur. Finalement, Processor Designer permet de générer une description synthétisable du processeur en langage VHDL ou Verilog. Le concepteur peut alors utiliser l'outil de synthèse de son choix, qu'il cible un FPGA ou un circuit intégré dédié.

Le listage 2.1 présente un extrait d'un modèle LISA de manière à donner un aperçu de la syntaxe. Dans la section `RESOURCE` sont entre autres décrits :

- un compteur de programme de 32 bits ;
- un registre, un fichier de registres et un drapeau ; et
- une mémoire de programme et une mémoire de données, avec chacune sa taille et sa plage d'adresse.

Une section `OPERATION` suit la description des ressources. Celle-ci décrit le comportement et une partie de l'encodage d'une instruction d'addition. Dans un modèle LISA, le

jeu d'instruction est décrit par une hiérarchie d'opérations, où chaque niveau de la hiérarchie contribue aux instructions un élément de comportement, d'encodage et de syntaxe du langage assembleur.

```
RESOURCE {
  PROGRAM_COUNTER uint32 pc;

  REGISTER unsigned bit[48] ir;
  REGISTER uint32 g_regfile[0..31];
  REGISTER unsigned bit[1] zf;

  RAM unsigned bit[48] prog_mem {
    SIZE(PROG_SIZE);
    BLOCKSIZE(48);
    FLAGS(R|X);
  };

  RAM uint32 data_mem {...};

  MEMORY_MAP {
    RANGE(PROG_START, PROG_END) -> prog_mem[(21..0)];
    RANGE(DATA_START, DATA_END) -> data_mem[(21..0)];
  }

  uint32 oper1, oper2;
}

OPERATION add {
  SYNTAX   {"add"}
  CODING   {0b0000}
  BEHAVIOR {result = oper1 + oper2;}
}
```

LISTAGE 2.1 Exemple de syntaxe LISA

2.4.2 Processeur Xtensa de Tensilica

Le Xtensa LX2 de Tensilica est un processeur configurable et extensible. Il existe un jeu d'instructions de base reconnu par tous les processeurs Xtensa LX2. Le fait qu'il s'agisse d'un processeur extensible veut dire que son jeu d'instruction peut être étendu par l'ad-

dition d'instructions propres à l'application. Ces instructions doivent être décrites à l'aide d'un langage propriétaire nommé Tensilica Instruction Extension (TIE). Il s'agit aussi d'un processeur configurable, ce qui veut dire qu'il est possible d'ajouter ou non certaines fonctionnalités et régler certains paramètres. Il est possible, par exemple :

- d'ajouter ou non une unité de calcul à virgule flottante et/ou un multiplieur ;
- de régler la taille des bus et des caches ;
- de choisir un fichier de registres d'usage général de 32 ou 64 registres ; et
- de définir le nombre d'interruptions disponibles et leur priorité relative.

L'Xtensa est un processeur RISC 32-bit à architecture à chargement et sauvegarde, comme le MIPS, le SPARC, le NIOS II et bien d'autres. Ses instructions sont typiquement encodées sur 24 bits, ce qui permet une bonne densité de code. Quelques instructions plus courantes peuvent être encodées sur 16 bits. Les mots d'instruction FLIX (pour *Flexible Length Instruction Xtensions*) de 32 ou 64 bits, qui sont en fait des mots d'instruction VLIW, permettent l'exécution simultanée de plusieurs instructions. Une particularité intéressante de ce processeur est qu'il est possible de mélanger dans un même programme les instructions 24 bits, 16 bits et FLIX.

La conception est effectuée dans l'environnement de développement intégré Xtensa Xplorer, basé sur Éclipse. À partir des options de configuration sélectionnées et des instructions spécialisées décrites en langage TIE, un simulateur et un compilateur C/C++ basé sur GCC sont automatiquement générés. Un fichier d'en-tête fournissant les déclarations permettant des appels directs aux instructions spécialisées dans le code en langage C est aussi généré. Il est alors possible de créer des projets logiciels qui peuvent être compilés et profilés.

Le listage 2.2 donne un aperçu de la syntaxe du langage TIE. Dans cet extrait de code, deux registres et une table de constantes sont tout d'abord déclarés. Ensuite, une section `operation` définie l'instruction spécialisée `myop`. Celle-ci échange la valeur des deux registres spécialisés et retourne le contenu de l'un d'eux dans un registre d'usage général du processeur. La sélection du registre en question est encodée dans le mot d'instruction.

```
state reg_Y0 32 add_read_write
state reg_Y1 32 add_read_write

table FSb  8 24 {
  8'h63, 8'h7C, 8'h77, 8'h7B, 8'hF2, 8'h6B, 8'h6F, 8'hC5,
  8'h30, 8'h01, 8'h67, 8'h2B, 8'hFE, 8'hD7, 8'hAB, 8'h76,
  8'hCA, 8'h82, 8'hC9, 8'h7D, 8'hFA, 8'h59, 8'h47, 8'hF0
}

operation myop {out AR result} {inout reg_Y0,
  inout reg_Y1} {

  assign reg_Y0 = reg_Y1;
  assign reg_Y1 = reg_Y0;
  assign result = reg_Y1;
}
```

LISTAGE 2.2 Exemple de syntaxe TIE

2.4.3 Processeur NIOS II d'Altera

Le NIOS II est un processeur RISC 32 bits ciblant les FPGA d'Altera. Il est possible de lui ajouter des instructions spécialisées en greffant à son unité arithmétique et logique des modules VHDL ou Verilog dont le nombre et le type des ports est compatible à une interface standardisée. Le listage 2.3 donne un exemple d'entité VHDL appropriée.

```
entity foo is
port(
  n:in         std_logic_vector( 2 downto 0);
  dataa:in     std_logic_vector(31 downto 0);
  datab:in     std_logic_vector(31 downto 0);
  result:out   std_logic_vector(31 downto 0);
end entity;
```

LISTAGE 2.3 Module VHDL à interface standardisée pour le NIOS II

Dans l'exemple du listage 2.3, `dataa` et `datab` sont des valeurs lues de registres d'usage général, alors que le résultat de l'instruction doit être placé sur `result` de manière à ce qu'il soit écrit dans le registre de destination. Le port `n` est un numéro de fonction ; il

permet que plusieurs instructions puissent être réalisées à l'aide d'un seul module VHDL. Certains de ces signaux auraient pu être omis, par exemple si une seule instruction était implémentée par le module (`n` omis) ou si ou si les instructions implémentées ne nécessitaient qu'un seul opérande (`datab` omis). D'autres ports optionnels existent, notamment pour permettre la réalisation d'instructions multicycles.

Contrairement à ce qui est le cas pour l'Xtensa ou un processeur décrit par un modèle LISA, le NIOS II possède un jeu d'instructions immuable, et ne necessite donc pas d'assembleur et de compilateur reciblables. Le jeu d'instructions comporte une instruction réservée aux fonctionnalités ajoutées et dont la syntaxe assembleur est : `custom N, xC, xA, xB`, où N est un numéro de fonction compris entre 0 et 255, et xC, xA et xB sont respectivement le registre de destination et les deux registres d'opérande. Du côté du compilateur, cette instruction est encapsulée par des fonctions intrinsèques qui ne diffèrent entre elles que par le type des opérandes et de la valeur de retour. Quelques exemples de déclarations sont données au listage 2.4.

```
int  __builtin_custom_inii (int n, int dataa, int datab);
void __builtin_custom_np (int n, void *dataa);
...
```

LISTAGE 2.4 Fonctions intrinsèques encapsulant l'instruction `custom`

L'outil SoPC Builder d'Altera, intégré à Quartus, permet de créer un système constitué d'un ou plusieurs processeurs NIOS II, spécialisés ou non, et de périphériques reliés via le bus Avalon. C'est dans cet outil que sont sélectionnés, à l'aide d'une interface graphique, les modules VHDL ou Verilog devant servir à étendre la fonctionnalité du processeur. C'est aussi dans cet outil qu'il est possible d'assigner les numéros de fonction ou les plages de numéros de fonction, selon le cas, aux modules greffés. Lorsque la configuration est terminée, l'outil génère une description du système pouvant être synthétisée par Quartus, ainsi qu'un fichier d'entêtes qui définit des macros encapsulant les fonctions intrinsèques du compilateur décrites plus haut. Il est alors possible d'utiliser l'environnement Quartus pour programmer le système dans un FPGA, compiler des projets logiciels, les charger dans le système et les faire s'exécuter.

Chapitre 3

DÉMARCHE DE LA RECHERCHE ET COHÉRENCE DE L'ARTICLE PAR RAPPORT AUX OBJECTIFS

3.1 Démarche de la recherche

Les travaux décrits dans ce mémoire visent à explorer les possibilités qu'offrent les processeurs à jeu d'instructions spécialisé pour les applications vidéos. Spécifiquement, ils portent sur une classe particulière d'algorithmes de traitement vidéo : les filtres locaux. Ces travaux s'inscrivent le cadre des activités d'une équipe de recherche du GRM qui s'intéresse au traitement vidéo.

La poursuite des objectifs de recherche a consisté en trois tâches principales. Tout d'abord, une étude de la classe d'algorithmes visée a été réalisée, dans le but d'identifier les caractéristiques de ces algorithmes susceptibles d'offrir des opportunités d'accélération. En effet, la performance et l'efficacité des processeurs à jeu d'instructions spécialisé repose sur les hypothèses qui sont posées quant aux applications devant être exécutées. L'étude comprend, d'une part, la revue de la littérature pertinente et, d'autre part, la comparaison de quelques algorithmes de la classe afin d'en faire ressortir les points communs. Les algorithmes choisis pour cette comparaison sont les trois algorithmes de désentrelacement intra-trames présentés à la section 2.3, et la convolution 2D. Les trois algorithmes de désentrelacement ont été sélectionnés parce qu'ils sont de complexités logicielles très différentes et visent une application concrète. La convolution 2D a été choisie, quant à elle, parce qu'il s'agit d'un filtre local polyvalent très utilisé en traitement d'image, et parce qu'il a été jugé judicieux de sélectionner un filtre local qui ne soit pas un algorithme de

désentrelacement intra-trame, de manière à obtenir un échantillon plus varié, et donc plus représentatif, de la classe d'algorithmes visée.

Le deuxième objectif a été l'exploration architecturale. Cette exploration s'est effectuée en deux étapes réalisées de manière itérative. Tout d'abord, sur la base des caractéristiques identifiées lors de l'étude de la classe d'algorithmes, des techniques permettant potentiellement d'obtenir une accélération ont été proposées. Ensuite, ces techniques ont été mises en application de manière à en vérifier expérimentalement la validité. Par mise en application, nous entendons la création d'une extension au jeu d'instructions d'un processeur Xtensa LX2 à l'aide du langage propriétaire TIE. La vérification expérimentale a consisté, quant à elle, à réaliser une simulation à l'aide de l'environnement de conception Xtensa Xplorer et à recueillir les métriques de performance pertinentes issues de cette simulation.

De manière à accélérer les travaux, un seul algorithme a été considéré lors de l'exploration architecturale : l'algorithme PBDI, conçu par un étudiant au doctorat du groupe de recherche. À l'issue des multiples itérations de l'exploration architecturale, un processeur spécialisé permettant d'exécuter cet algorithme à une vitesse permettant le traitement en temps réel a été obtenu. Ce processeur a fait l'objet d'un article [2] présenté à la conférence NEWCAS-TAISA 2009 de l'IEEE.

Finalement, une approche de conception systématique et cohérente a été proposée. Cette approche comprend l'ensemble des techniques retenues à l'issue de l'exploration architecturale, ainsi qu'un ordre logique dans lequel elles peuvent être appliquées efficacement. Afin d'évaluer expérimentalement le bien fondé de cette approche, des processeurs spécialisés supplémentaires ont été conçus : un pour l'algorithme ELA, un autre pour l'algorithme Enhanced ELA, et quatre pour la convolution 2D. Chacun des processeurs pour la convolution 2D supporte une taille distincte de fenêtre de convolution. La mise en œuvre d'un même algorithme avec quatre tailles distinctes de fenêtre a permis de tirer des conclusions concernant l'influence de cette taille sur la performance finale.

3.2 Présentation de l'article et cohérence en rapport aux objectifs

Tel que mentionné à la section précédente, une approche systématique de conception de processeurs spécialisés pour les filtres locaux constitue l'aboutissement des travaux faisant l'objet de ce mémoire. Cette approche est décrite dans l'article *Real-Time Computation of Local Neighbourhood Functions in Application-Specific Instruction-Set Processors*, soumis à la revue *Transactions on Very Large Scale Integration (VLSI) Systems* de l'IEEE. Le chapitre 4 du mémoire présente cet article.

En plus d'exposer dans les détails l'approche de conception, l'article décrit les filtres locaux, avec une emphase sur les caractéristiques desquelles il est possible de tirer avantage afin d'obtenir une vitesse de traitement élevée. On y retrouve aussi les résultats de performance supportant la validité de l'approche, ainsi qu'une brève discussion de ces résultats. Les résultats en question concernent les sept processeurs ayant été conçus pour vérifier expérimentalement la pertinence de l'approche et dont il a été fait mention à la section précédente.

3.3 Résumé de l'article

L'article présente une approche systématique pour la conception de processeurs spécialisés (ASIP) pour le désentrelacement intra-trame et les filtre locaux. Le type d'application visé est le traitement en temps réel de vidéo à haute définition. L'approche vise une utilisation efficace de la bande passante vers la mémoire, laquelle bande passante constitue le goulot d'étranglement. Il est possible d'approcher la performance limite imposée par ce goulot par une stratégie appropriée de réutilisation des données et en exploitant le parallélisme des données inhérent à la classe d'algorithmes visée.

L'approche de conception comprend quatre étapes principales. Quelques-unes de ces

étapes ouvrent la porte à des simplifications matérielles spécifiques pour certains algorithmes de la classe cible. Dans l'article, une fonction de filtrage local simple est utilisée comme exemple illustratif. Des figures présentent en effet chaque étape et chaque simplification telles qu'appliquées à cet exemple.

La première étape consiste à créer une instruction parallèle (SIMD) qui effectue le calcul de plusieurs pixels de sortie à la fois. Le nombre de pixels de sortie qu'il est avantageux de calculer en parallèle peut être déterminé à partir de la taille des pixels et de la taille du bus de données du processeur. Lorsque l'instruction parallèle est dérivée en copiant en plusieurs exemplaires l'instruction permettant de calculer un seul pixel de sortie, certaines opérations peuvent s'avérer redondantes. L'élimination de ces opérations redondantes constitue une première simplification permettant une économie matérielle.

La deuxième étape consiste à créer des registres à décalage permettant la réutilisation intra-ligne des pixels d'entrée. Ces registres à décalage, en tirant avantage du parcours régulier de l'image, permettent l'élimination d'opérations de chargement, ce qui mène à une amélioration de la performance. Leur création ne nécessite pas l'ajout de nouveaux registres, simplement une modification du parcours des données entre les registres existants.

La troisième étape consiste à pipeliner les instructions en ajoutant des registres intermédiaires. Cette étape a pour but l'augmentation de la fréquence d'horloge, et donc du débit de pixels produits. Deux simplifications matérielles peuvent être rendues possibles par cette étape. Premièrement, il se peut dans certaines conditions que des registres intermédiaires ajoutés lors du pipelinage soient redondants avec des registres formant le registre à décalage servant à la réutilisation intra-ligne des pixels d'entrée. De tels registres peuvent être éliminés. Une seconde simplification matérielle consiste à réutiliser des résultats intermédiaires déjà calculés plutôt que de les calculer à nouveau. Cette simplification, qui permet donc d'éliminer des opérations de calcul, est analogue à la méthode utilisée pour la réutilisation intra-ligne des pixels d'entrée : les résultats intermédiaires entre deux étages de pipeline sont réutilisés en les redirigeant dans un registre à décalage.

L'étape finale consiste à créer les instructions de chargement et de sauvegarde. Un mécanisme de mise en tampon des données permet d'adapter l'infrastructure de chargement et de sauvegarde au pipeline de traitement de manière à ce que les données soient fournies à une cadence appropriée. La réutilisation inter-ligne des pixels d'entrée est effectuée via la cache des données du processeur ; cette réutilisation est donc transparente dans la perspective de la conception des instructions spécialisées.

Des résultats de performance sont fournis pour trois algorithmes de désentrelacement intra-trames (ELA, Enhanced ELA et PBDI – pixels couleur) et pour quatre tailles de voisinage de la convolution 2D (3×3, 5×5, 3×5 5×3 – pixels monochromes). Pour le désentrelacement, les facteurs d'accélération obtenus varient entre 95 et 1330, alors que les facteurs d'amélioration du produit AT varient entre 29 et 243, tout ceci par rapport à un processeur d'usage général de référence roulant une implémentation purement logicielle de l'algorithme. Dans le cas de la convolution 2D, les facteurs d'accélération varient entre 36 et 80, alors que les facteurs d'amélioration du produit AT varient entre 12 et 22. Dans tous les cas, le traitement en temps réel de données vidéo à haute définition au format 1080i (désentrelacement) ou 1080p (convolution) est possible avec une fabrication dans une technologie de circuit intégré CMOS à 130 nm.

Chapitre 4

ARTICLE I : REAL-TIME COMPUTATION OF LOCAL NEIGHBOURHOOD FUNCTIONS IN APPLICATION-SPECIFIC INSTRUCTION-SET PROCESSORS

Philippe Aubertin *Student Member*, *IEEE*, J.M. Pierre Langlois, *Member*, *IEEE* and Yvon Savaria, *Fellow*, *IEEE*

4.1 Abstract

This paper presents a systematic approach to the design of Application-Specific Instruction-set Processors for high speed computation of local neighbourhood functions and intra-field deinterlacing. The intended application is real-time processing of high definition video. The approach aims at an efficient utilization of the available memory bandwidth by fully exploiting the data parallelism inherent to the target algorithm class. An appropriate choice of custom instructions and application-specific registers is used together with a Very Long Instruction Word architecture in order to mimic a pipelined systolic array. This leads to a processing speed close to the limit imposed by memory bandwidth constraints. For three intra-field deinterlacing algorithms and 2D convolution with four kernel sizes, we report speedup factors between 36 and 1330 using this design approach compared to a pure software implementation, with factors of improvement of the Area-Time product between 12 and 243.

4.2 Introduction

Moore's law [1] is both a blessing and a curse. On one hand, the continuing scaling of semiconductor processes allows for faster systems and opens the door to applications which would not have been possible or affordable otherwise. For example, we have seen in recent years an explosion in offerings of high definition and portable video applications reaching the consumer market. However, this continuing scaling also brings forward the problem of the ever-increasing design complexity of digital systems. To mitigate this problem and make sure time-to-market objectives are not jeopardized by increasing design and verification efforts, new tools and design methodologies must be developed in parallel with the improvement of semiconductor processes. These tools and methodologies typically improve designers' productivity by leveraging design automation and higher levels of abstraction [19].

A possible solution is to implement complex digital functions using Application-Specific Instruction-Set Processors (ASIPs) [20]. This approach maintains some of the flexibility, ease of design and productivity typically associated with general-purpose processors and software solutions while approaching the efficiency and performance of dedicated hardware.

The use of ASIPs to implement complex functions in digital systems is an active field of research. Work has been published in recent years about the use of ASIPs in areas as diverse as calculation of fast Fourier transforms (FFTs) [13], processing of global positioning data [14], error correcting codes decoding [15, 21], cryptography [16], video compression [17] and video processing [18]. Also, a number of design automation solution vendors have introduced to the market EDA tool suites for ASIP design [22, 23]. These tools take a high-level description of the instruction-set as input and automatically generate a synthesizable description of the processor, as well as related development tools, such as an assembler, a compiler and an instruction-set simulator.

In this paper, we propose a systematic approach to the design of ASIPs for high speed

computation of local neighbourhood functions and intra-field deinterlacing. An appropriate choice of custom instructions and application-specific registers is used together with a Very Long Instruction Word (VLIW) architecture in order to mimic a pipelined systolic array. This leads to processing speeds close to the limits imposed by memory bandwidth constraints. Control dependencies are first removed by transforming them into data dependencies. Then, data processing Single Instruction-Multiple Data (SIMD) instructions and custom registers are chosen so as to form the structure of a pipeline. Finally, an appropriate choice of custom load/store instructions makes it possible to find a scheduling of the instructions inside the inner loop of the algorithm where one load or store instruction is executed every single instruction cycle. Data processing occurs concurrently with load and store operations.

The best known local neighbourhood function is the two-dimensional convolution. Many fast and area-efficient architectures for implementing the two-dimensional convolution have been proposed in the litterature, with recent work focussing on implementation in FPGAs [24, 25]. Architectures based on a linear systolic array are commonplace for the fast implementation of local neighbourhood functions. Examples of this are the two-dimensional convolver described in [26] and [27] which presents the implementation of another class of local neighbourhood functions: morphological functions. As an alternative to the linear systolic array, a novel cell-based architecture for fast local neighbourhood image processing is proposed in [7].

This paper focusses on higher productivity in the design of local neighbourhood video processing implementations through the use of a systematic ASIP design approach and high-level EDA tools. It is organized as follows: section II presents the application for better understanding of subsequent sections. In this section, terminology and notation used in the remainder of this paper are introduced. Section III highlights the characteristics of software code for the target class of algorithm and suggests some code transformation which aim at removing control dependencies. In section IV, the steps involved in the design of custom instructions are presented. Finally, results are presented in section V, followed

by a conclusion summarizing our main results in section VI.

4.3 Local Neighbourhood Functions

Local neighbourhood functions form a class of algorithms of common use in image and video processing. Each pixel in the output image is calculated independently and is a function of a small subset of the input image. This subset – the window – contains the input pixel having the same coordinates as the output pixel, as well as some of its close neighbours. As pixels of differing coordinates are produced, the window “slides along". Hence, local neighbourhood functions are also called *sliding window functions* [6].

An image of width W and height H may be represented by the $H \times W$ matrix

$$P = \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,W} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,W} \\ \vdots & \vdots & \ddots & \vdots \\ p_{H,1} & p_{H,2} & \cdots & p_{H,W} \end{pmatrix} \tag{4.1}$$

where each element $p_{i,j}$ is a pixel. The representation of each such pixel is application-dependent. It is usually either a scalar representing a luminance in the case of grayscale images, or a vector representing the colour of the pixel in some colour space in the case of colour images.

The output image, represented by matrix Q, is produced by applying the local neighbourhood function F, as follows:

$$q_{i,j} = F(\, X(i,j)\,) \tag{4.2}$$

where $X(i,j)$, the window, is the $h \times w$ matrix

$$X(i,j) = \begin{pmatrix} x_{1,1} & \cdots & x_{1,w} \\ \vdots & \ddots & \vdots \\ x_{h,1} & \cdots & x_{h,w} \end{pmatrix} \tag{4.3}$$

where

$$\begin{gathered} x_{k,l} = p_{i+k-1,j+l-1} \\ \forall i \in [1, H-h+1], j \in [1, W-w+1] \end{gathered} \tag{4.4}$$

meaning $x_{1,1}$ is $p_{i,j}$, and other elements of $X(i,j)$ are the neighbouring pixels below and/or right of $p_{i,j}$. Formally, it is implied by (4.4) that the output image is smaller than the input image by $h-1$ lines and $w-1$ columns. This is because these are the only coordinates for which all the pixels in the window are defined. For example, in $X(i,W)$, $x_{1,2}$ is pixel $p_{i,W+1}$, which is outside the image. In practical implementations, the size of the output image is often extended to match that of the input image. Various means of doing this exist. One option consists in generating boundary pixels by using a separate function which depends on a smaller window. Another option is to conceptually enlarge the input image by copying existing pixels or by giving the non-existent pixels outside the input image a fixed value.

Whole images are scanned following a regular path. This leads to a relationship where some pixels in the window used to produce one pixel are also present in the window used to generate the next pixel. Without loss of generality, let us assume that each image line is scanned from left to right. In that case, if the N$^{\text{th}}$ pixel produced is $q_{i,j}$, the N+1$^{\text{th}}$ pixel produced is $q_{i,j+1}$, assuming $p_{i,j}$ is not a boundary pixel. This means that, when producing the N$^{\text{th}}$ pixel, $x_{1,2}$ is $p_{i,j+1}$, and that same pixel becomes $x_{1,1}$ when producing the N+1$^{\text{th}}$ pixel. More generally, as the window slides from one pixel to the next, $x_{k,l}$ for the N$^{\text{th}}$ pixel becomes $x_{k,l-1}$ for the N+1$^{\text{th}}$ pixel for $k > 1$. This property of the local neighbourhood functions offers opportunities for data reuse.

The most common local neighbourhood function is the two-dimensional convolution. It consists in individually weighting each pixel of the window and summing the weighted values, or

$$q_{i,j} = \sum_{k=1}^{h} \sum_{l=1}^{w} a_{k,l} \cdot x_{k,l} \tag{4.5}$$

where the $a_{k,l}$ are the weights. The two-dimensional convolution is useful to apply various types of filters to an image in order to sharpen it, blur it, enhance features such as edges, etc. Other simple local neighbourhood functions include dilation and erosion.

A notable class of complex local neighbourhood functions are the intra-field deinterlacing algorithms. Deinterlacing is the process of converting an interlaced video sequence into a progressive one. Various classes of methods for performing this conversion exist [8]. Intra-field methods work by generating each complete image from the pixel data of a single field. Missing pixels are generated by interpolating neighbouring available pixels. Thus, these methods may be classified as local neighbourhood functions.

Simple intra-field deinterlacing methods consist in generating missing pixels by copying the available pixel just above or below (a method commonly called BOB) or by linear interpolation between the pixel right above and the one right below. These methods yield poor quality results. Edge-based methods aim at improving quality by tackling a specific artifact: jagged edges. They work by detecting edges in video fields using various methods, and then interpolating in a direction parallel to these edges. The vanilla edge-based method is Edge-Based Line Averaging (ELA) [9]. Other, more complex methods aim at further improving image quality through higher angular resolution and more reliable edge detection. They include Enhanced ELA (EELA) [10], Modified ELA (MELA) [11], Pattern-Based Directional Interpolation (PBDI) [12] and others.

4.4 Code Analysis and Transformations

It is possible to decompose the calculation of a local neighbourhood function into the successive application of simpler operations, and the data dependencies between the operations in such a decomposition may be represented by a data-flow graph. A data-flow graph is a directed acyclic graph $D = (V, E)$, where V is the set of nodes and $E \subset V \times V$ is the set of edges. Each node $v \in V$ is either an operation, a primary input or a primary output, while each edge $(v_i, v_j) \in E$ has the meaning "v_j depends on data produced by v_i". Each node v_j has a dependency set $dep(v_j) \subset V$ such that

$$v_i \in dep(v_j) \iff (v_i, v_j) \in E \tag{4.6}$$

In a data-flow graph derived from a local neighbourhood function, the primary inputs are the elements $x_{k,l}$ of the window X and the primary output is the pixel $q_{i,j}$. In the context of a digital system which obtains data from and stores data to an external system component such as a memory, the primary input nodes actually represent load operations which produce the needed data, while the primary output node represents a store operation. Thus, every node $v_i \in V$ represents an operation. Let us define $inputs(V)$ as the subset of V which represents the primary inputs.

There is an annotation $op : V \rightarrow O$ of all nodes in V where O is a set of implementable operations. In addition to the load and store operations already mentioned, O may contain any *primitive operation*, which we define as the common operations readily implementable in hardware and commonly found as built-in operators in programming languages. Examples of primitive operations are arithmetic and logic operations, bit shifts and the data selection operator (multiplexer). It is also often convenient to collapse subgraphs made from primitive operations into single nodes in order to define higher-level operations, which are then also in O.

Local neighbourhood functions are typically prototyped as software implementations.

It is these implementations which are used as a starting point for the creation of ASIP implementations in the proposed design approach. Fig. 4.1 shows the code structure of the software implementation of a typical local neighbourhood function. Image scanning is performed by a certain number of nested fixed-length loops – two for single image processing and three for video processing. Inside the innermost loop, a sequence of statements express the calculation of one output pixel as a function of input pixels from the window, i.e. $F(\,X(i,j)\,)$. In this sequence of statements, all dependencies may be expressed by a data-flow graph. Control dependencies may be eliminated or converted into data dependencies through simple code transformations. This eases the creation of the hardware structure.

```
for(f = 0; f < Nframes; ++f)
    for(i = 0; i < H; ++i)
        for(j = 0; j < W; ++j) {
            Sequential statement 1;
            Sequential statement 2;
            ...
            Sequential statement N;
        }
```

FIGURE 4.1 Code structure of a typical local neighbourhood function

In the software implementation of a given local neighbourhood function, most statements inside the innermost loop commonly consist in the assignment of the value of expressions to temporary variables or as the final output pixel value. Inputs to these expressions are input pixels taken from the window $X(i,j)$, intermediate results or constants. The expressions themselves are assembled from built-in operators or pure function calls. The pure function calls may be treated in two ways: either a single node in the data-flow graph represents the function call, or function inlining is performed. Function inlining in that case gives a finer granularity, but also adds more complexity to the task of choosing pipeline stages.

Conditional statements may be replaced by equivalent data selection operations. This has the effect of transforming control dependencies into data dependencies. Indeed, instead

of conditionally executing one branch or the other based on a condition, both branches are always executed, but the result of one or the other is selected based on the condition. Multiple-case statements may be treated in a similar manner.

Finally, a given sequential statement may be a loop statement, which is a control statement. However, in the software implementation of local neighbourhood functions, loops are usually fixed in length. In this case, these loops may be unrolled completely. In the case where the length of a loop is not fixed, it will necessarily have an upper bound in real-time applications. This case, which is uncommon in image filtering applications, may be handled by unrolling the maximum number of iterations for the loop, and then selecting the result from the proper iteration based on the run-time loop bound.

4.5 Design of Custom Instructions

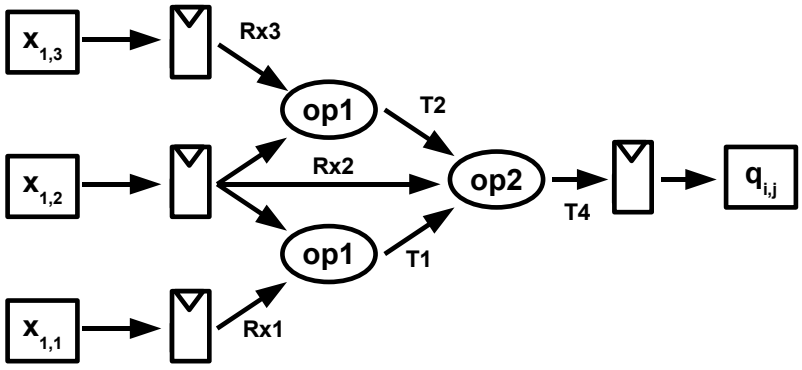
In this section, we present the steps involved in the choice of custom instructions and application-specific registers. These steps are as follows: first, SIMD instructions which produce multiple output pixels in parallel are created. These instructions make use of a specific intra-line data reuse scheme. Then, each SIMD instruction is split into multiple instructions through pipelining in order to increase throughput. Some hardware simplifications may be possible at that point. Finally, the load and store instructions are created.

Fig. 4.2(a) shows the C language code of a very simple local neighbourhood function. This example will be used to illustrate the various steps involved in the design of custom instructions and was contrived to exhibit all the proposed simplifications for illustrative purposes. A practical algorithm will likely allow for some of the possible simplifications but not all them. In the code of Fig. 4.2(a), `op1` and `op2` are arbitrary pure functions.

Fig. 4.2(b) shows a representation of a single custom instruction which could implement the local neighbourhood function of Fig. 4.2(a). In Fig. 4.2(b), labelled squares indicate input and output pixels. These pixels are typically stored in memory locations. Thus, the

```
for(i = 0; i < H; ++i)
   for(j = 0; j < W-2; ++j) {
      a  = op1(p[i][j],   p[i][j+1]);
      b  = op1(p[i][j+1], p[i][j+2]);
      q[i][j] = op2(a, p[i][j+1], b);
   }
```

(a)




```
Rx1 := (Z-1 x1,1)        T1  := (op1 Rx1 Rx2)
Rx2 := (Z-1 x1,2)        T2  := (op1 Rx2 Rx3)
Rx3 := (Z-1 x1,3)        T4  := (op2 T1 Rx2 T2)
                         qi,j := (Z-1 T4)
```

(b)

FIGURE 4.2 Simple local neighbourhood function example with (a) C language code and (b) equivalent single custom instruction

squares are indicative of load and store operations. Labelled ellipses represent processing operations, and edges represent intermediate results. These edges are indicative of data dependencies. Finally, the rectangles represent registers. In order to simplify data reuse and instruction scheduling, a load-store architecture is used. Consequently, there are registers for the input pixels between the load operations and the processing operations, and also for the output pixels between the processing operations and the store operations. These registers are noted in the textual representation by the Z^{-1} time delay operator, with $(Z^{-1}a)$ having the semantic "a delayed by one execution cycle".

4.5.1 Custom SIMD Instructions

The creation of Single-Instruction Multiple-Data (SIMD) instructions which calculate more than one pixel in parallel is discussed in this section. Calculating many pixels in parallel leads to an increase in silicon area requirements, and only leads to a performance improvement if the datapath is not starved by an inability to feed data and retrieve results at a sufficient rate. For this reason, the number of pixels m calculated in parallel must be balanced with the size of the bus used to access the data in memory. The link between m and the bus size is addressed in section 4.5.6.

It is useful to conceptualize the parallel calculation of pixels as a new local neighbourhood function which yields a vector and which depends on an extended window. Let $\Omega(i, j)$ be the output vector, which is the row vector

$$\Omega(i, j) = [\omega_1 \ldots \omega_m] \tag{4.7}$$

where

$$\omega_k = q_{i,k+j-1} = F(\, X(i, j + k - 1)\,). \tag{4.8}$$

This may be expressed as a new function

$$\Omega(i, j) = G(\, Y(i, j)\,) \tag{4.9}$$

where Y, the extended window, is defined as

$$Y(i, j) = \bigcup_{k=1}^{m} X(i, j + k - 1). \tag{4.10}$$

$Y(i, j)$ is a subset of P with height h and width

$$w' = w + m - 1. \tag{4.11}$$

4.5.2 Redundant Operations in SIMD Instructions

Let $D_k = (V_k, E_k)$ be the data-flow graph derived from $F(\, X(i, j+k-1)\,)$ for $k \in [1, m]$. The data-flow graph $D_G = (V_G, E_G)$ derived from $G(\, Y(i, j)\,)$ may be obtained by

$$D_G = \bigcup_{k=1}^{m} D_k \tag{4.12}$$

which implies

$$V_G = \bigcup_{k=1}^{m} V_k. \tag{4.13}$$

Depending on the local neighbourhood function F, some operations in V_G may be redundant. This property of some local neighbourhood functions to exhibit redundant operations when multiple pixels are computed in parallel can be exploited advantageously, because the elimination of redundant operations leads to a lowering of hardware costs. It was found that when trying to identify redundant operations, the data-flow graph alone is not sufficient since it does not express the fact that some operations are not commutative. For example, $a - b$ is not the same as $b - a$, thus it is not sufficient for operations to be redundant that they share the same dependency set and operator; the order matters. Let us define an operand tuple for an operation as a tuple containing the operands for that operation. In a data-flow graph $D = (V, E)$, for a node $v_i \in V$, if $op(v_i)$ is an operation which requires n operands, an operand tuple for v_i is an n-tuple where each element is in $dep(v_i)$. Let us further define $ots(v_i)$, the operand tuple set of v_i, as the set of the operand tuples for v_i which lead to the correct value in the calculation of the local neighbourhood function. For a non-commutative operation, this set will contain only one operand tuple, while it will contain more than one in the case of a commutative operation.

Two operations $v_i, v_j \in V_G$ are redundant if

$$op(v_i) = op(v_j) \tag{4.14}$$

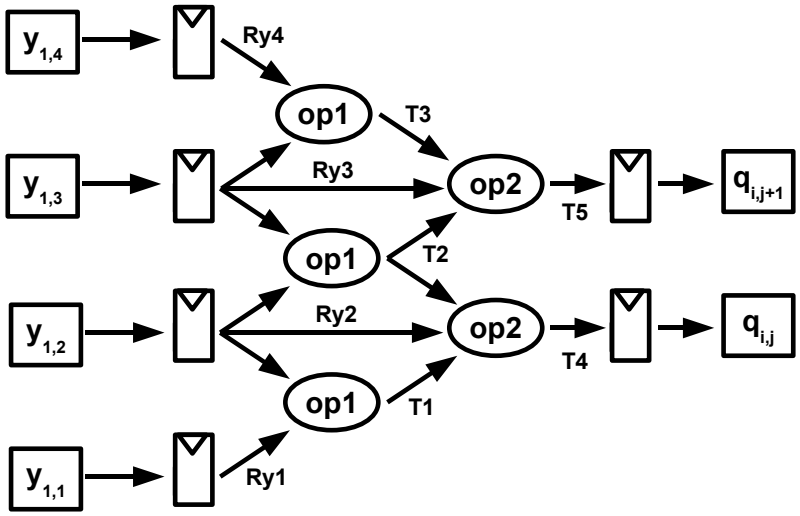
and if

$$ots(v_i) \cap ots(v_j) \neq \emptyset \tag{4.15}$$

in which case, v_i can be removed from V_G if each edge $(v_i, v_k) \in E_G$ is replaced by an edge (v_j, v_k). Only a subset of V_G has to be examined in search for redundant operations. Indeed, since each pixel in the output image is calculated independently,

$$V_k \cap V_l \subset inputs(V_G), \qquad \text{for } k, l \in [1, m], k \neq l. \tag{4.16}$$

For this reason, if two operations $v_i, v_j \in V_G$ are redundant, $dep(v_i) = dep(v_j)$ may only contain either primary inputs, or operations of which a corresponding redundant operation was previously removed.




```
Ry1 := (Z-1 y1,1)
Ry2 := (Z-1 y1,2)
Ry3 := (Z-1 y1,3)
Ry4 := (Z-1 y1,4)
T1  := (op1 Ry1 Ry2)
T2  := (op1 Ry2 Ry3)
T3  := (op1 Ry3 Ry4)
T4  := (op2 T1 Ry2 T2)
T5  := (op2 T2 Ry3 T3)
qi,j := (Z-1 T4)
qi,j+1 := (Z-1 T5)
```

FIGURE 4.3 Single SIMD custom instruction

The creation of a SIMD instruction and the elimination of redundant operations which it sometimes renders possible are illustrated in Fig. 4.3 This figure shows a SIMD version of the instruction from Fig. 4.2(b) that calculates two output pixels in parallel. This SIMD instruction is obtained by creating a copy of the instruction from Fig. 4.2(b) and translating all data references in the copy by one pixel compared to the original. In this example, it is possible to eliminate one instance of `op1` leaving three instances instead of four. Indeed, when one instance of `op1` in the copy is translated by one pixel, it becomes redundant with another instance from the original. Specifically, when the statement `T1 := (op1 Ry1 Ry2)` is translated, it becomes `T1' := (op1 Ry2 Ry3)`, which is equivalent to `T2`.

The creation of SIMD instructions in the way suggested here, followed by redundant operation elimination, is equivalent to partial loop unrolling with common sub-expression elimination, which are common compiler optimizations.

4.5.3 Intra-Line Reuse of Input Pixel Data

We mentioned in section 4.3 that the regular way in which pixels are accessed leads to data reuse opportunities. In general, only one block of pixels per line of the window (i.e. h blocks of m pixels) needs to be loaded in order to produce one block of output pixels. These pixels may then be reused as many times as required through shift registers.

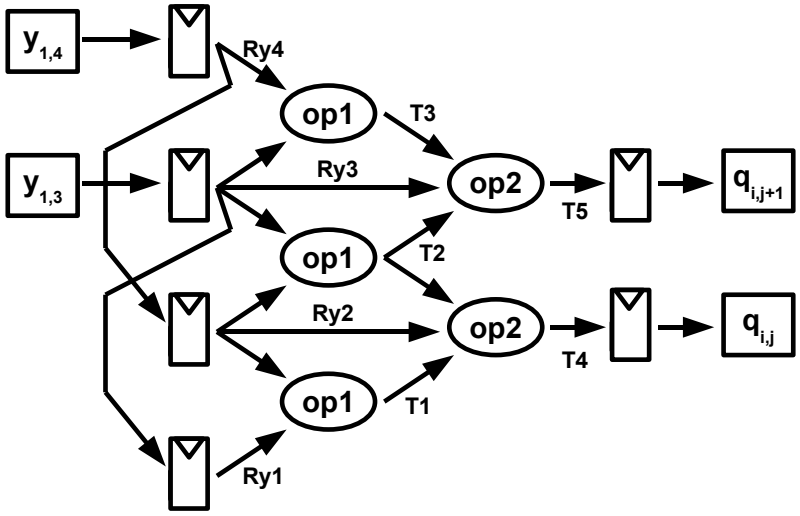
Formally, in order to calculate one block of m pixels, the $h \cdot m$ rightmost pixels in $Y(i,j)$ are new pixels which must be loaded either from the external source, or from the data cache as will be discussed in section 4.5.5. Remaining pixels in $Y(i,j)$ may be reused as follows:

$$\mathtt{y_{k,l}} = (\mathtt{Z^{-1}}\ \mathtt{y_{k,l+m}}) \qquad (4.17)$$
$$\text{for } k \in [1,h], l \in [1, w' - m + 1]$$

where, once again, $(\mathtt{Z}^{-1}\ \mathtt{y_{k,l+m}})$ is a mnemonic notation for $y_{k,l+m}$ delayed by one cycle. The number of load operations eliminated by intra-line reuse is

$$(w' - m) \cdot h = (w - 1) \cdot h. \tag{4.18}$$

This number becomes more significant as the window width increases.




```
Ry3 := (Z-1 y1,3)
Ry4 := (Z-1 y1,4)
Ry1 := (Z-1 Ry3)
Ry2 := (Z-1 Ry4)
T1  := (op1 Ry1 Ry2)
T2  := (op1 Ry2 Ry3)
```

```
T3  := (op1 Ry3 Ry4)
T4  := (op2 T1 Ry2 T2)
T5  := (op2 T2 Ry3 T3)
qi,j := (Z-1 T4)
qi,j+1 := (Z-1 T5)
```

FIGURE 4.4 Load operations elimination through intra-line data reuse (changes over previous version are highlighted in bold)

Fig. 4.4 illustrates this for the example introduced previously. In our example, pixels are produced in blocks of two. This means that the window slides by two pixels each execution cycle. This can be expressed as

$$\mathtt{y_{1,1}} = (\mathtt{Z}^{-1}\mathtt{y_{1,3}}). \tag{4.19}$$

Thus,

$$\begin{aligned} \mathtt{Ry1} &= (\mathtt{Z}^{-1}\, \mathtt{y}_{1,1}) \\ &= (\mathtt{Z}^{-1}\, (\mathtt{Z}^{-1}\, \mathtt{y}_{1,3})) \\ &= (\mathtt{Z}^{-1}\, \mathtt{Ry3}) \end{aligned} \tag{4.20}$$

and, in the same way,

$$\mathtt{Ry2} = (\mathtt{Z}^{-1}\mathtt{Ry4}). \tag{4.21}$$

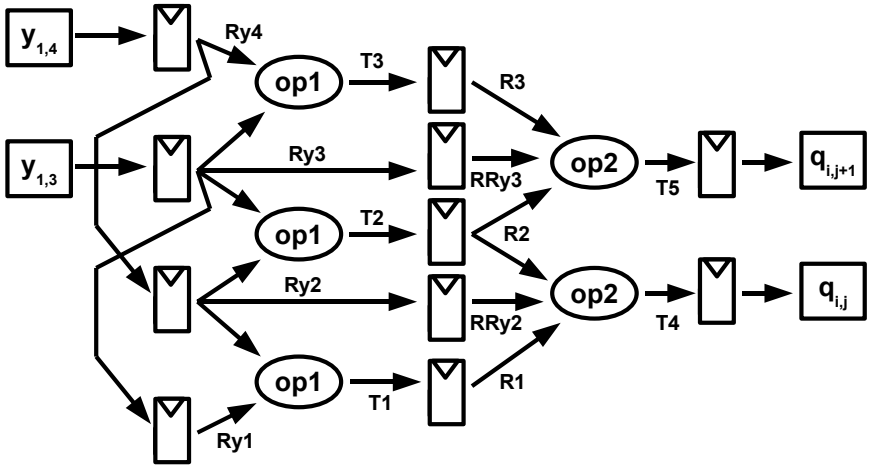
This leads to the elimination of two load operations. If the window were wider, it would also be possible to eliminate the additional load operations in this way, and longer shift registers would be formed.

4.5.4 Instruction Pipelining

It is possible to use pipelining in order to improve throughput. To do so, a single processing instruction is split into multiple instructions with intermediate results stored in application-specific registers. The multiple instructions may then be scheduled concurrently as in a pipeline through the use of a VLIW architecture. In the usual way, pipeline stages are chosen so as to distribute latency as evenly as possible. In order to maximize throughput, the cycle time of each custom instruction should be at most the cycle time of the processor without custom instructions. However, a better compromise between performance and processor size may be obtained by reducing the number of pipeline stages, which leads to lower hardware costs through a reduction of the number of application-specific registers between stages.

Pipelining leads in some cases to opportunities for two types of simplification: redundant register elimination and redundant operation elimination. In order to illustrate both types, Fig. 4.5 is used as a starting point. It is in fact the SIMD instruction from Fig. 4.4

which has been split into two pipeline stages.




```
Ry3 := (Z-1 y1,3)
Ry4 := (Z-1 y1,4)
Ry1 := (Z-1 Ry3)
Ry2 := (Z-1 Ry4)
T1  := (op1 Ry1 Ry2)
T2  := (op1 Ry2 Ry3)
T3  := (op1 Ry3 Ry4)
```

```
R1  := (Z-1 T1)
R2  := (Z-1 T2)
R3  := (Z-1 T3)
RRy2 := (Z-1 Ry2)
RRy3 := (Z-1 Ry3)
T4  := (op2 R1 RRy2 R2)
T5  := (op2 R2 RRy3 R3)
qi,j := (Z-1 T4)
qi,j+1 := (Z-1 T5)
```

FIGURE 4.5 Custom instructions pipelining through the use of application-specific registers

In the pipelined structure, input pixels move towards smaller column indices in the shift registers intended for intra-line reuse, and possibly also towards later pipeline stages if they are needed by such stages. This leads to redundant register elimination opportunities. As an example, in Fig. 4.5, RRy3 and Ry1 are both a delayed version of Ry3, RRy3 for proper pipelining and Ry1 for intra-line reuse. Thus, one register may be eliminated, as shown in Fig. 4.6, where Ry1 is used in both instances.

Generally, let N_p be the total number of pipeline stages. Furthermore, let each pipeline stage be identified by an integer $n \in [0, N_p - 1]$ such that the operations in pipeline stage 0 directly read the input pixels in the shift registers intended for intra-line reuse, and pipeline

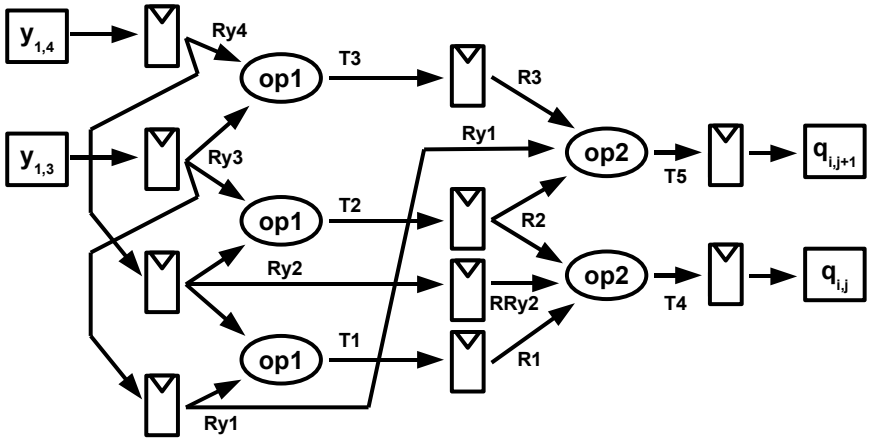
stage $n \in [1, N_p - 1]$ is the pipeline stage which follows pipeline stage $n - 1$. If a pixel $y_{k,l}$ is needed by operations in pipeline stage $n \in [1, N_p - 1]$, it is usually necessary to delay it by n cycles by adding registers for that purpose. However, instead of doing so, if the condition

$$l - n \cdot m \geq 1 \tag{4.22}$$

is satisfied, then the contents of the register holding $y_{k,l-n\cdot m}$ may be used directly instead. Otherwise, if there is an integer $c < n$ such that

$$l - c \cdot m \geq 1 \tag{4.23}$$

then the contents of the register holding $y_{k,l-c\cdot m}$ delayed by $n - c$ cycles may be used, which still reduces the number of registers required.




```
Ry3 := (Z-1 y1,3)
Ry4 := (Z-1 y1,4)
Ry1 := (Z-1 Ry3)
Ry2 := (Z-1 Ry4)
T1  := (op1 Ry1 Ry2)
T2  := (op1 Ry2 Ry3)
T3  := (op1 Ry3 Ry4)
```

```
R1   := (Z-1 T1)
R2   := (Z-1 T2)
R3   := (Z-1 T3)
RRy2 := (Z-1 Ry2)
T4   := (op2 R1 RRy2 R2)
T5   := (op2 R2 Ry1  R3)
qi,j := (Z-1 T4)
qi,j+1 := (Z-1 T5)
```

FIGURE 4.6 Redundant register elimination

Opportunities for operation elimination by reusing already computed values are offered by the fact that input pixels move through shift registers. This is done through the application of retiming [28], which may be expressed as

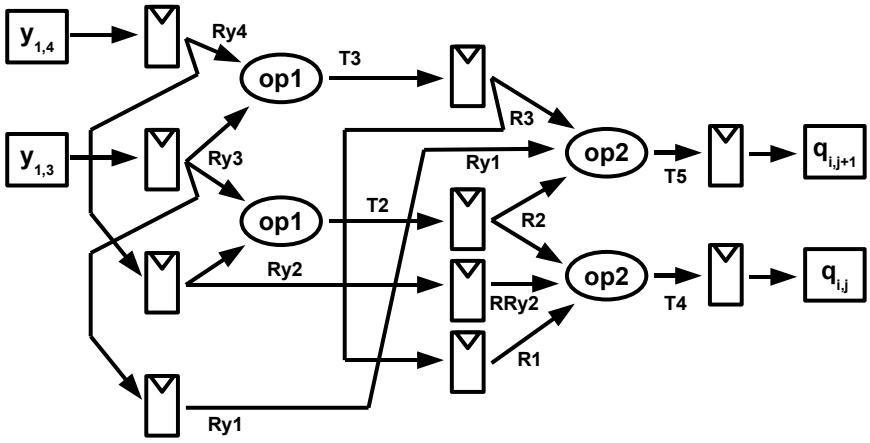
$$\left(\mathtt{OPx}\ \left(\mathtt{Z}^{-1}\ \mathtt{a}_1\right)\ \ldots\ \left(\mathtt{Z}^{-1}\ \mathtt{a}_n\right)\right) = \left(\mathtt{Z}^{-1}\ \left(\mathtt{OPx}\ \mathtt{a}_1\ \ldots\ \mathtt{a}_n\right)\right). \tag{4.24}$$

To illustrate, Fig. 4.7 shows the elimination of an operation by reusing an intermediate result which has already been computed through the application of retiming. Indeed,

$$\begin{aligned} \mathtt{T1} &= \left(\mathtt{op1}\ \mathtt{Ry1}\ \mathtt{Ry2}\right) \\ &= \left(\mathtt{op1}\ \left(\mathtt{Z}^{-1}\ \mathtt{Ry3}\right)\ \left(\mathtt{Z}^{-1}\ \mathtt{Ry4}\right)\right) \\ &= \left(\mathtt{Z}^{-1}\ \left(\mathtt{op1}\ \mathtt{Ry3}\ \mathtt{Ry4}\right)\right) \\ &= \left(\mathtt{Z}^{-1}\ \mathtt{T3}\right) \\ &= \mathtt{R3}. \end{aligned} \tag{4.25}$$

Operation elimination is first applied to pipeline stage 0. This has the effect of creating a shift register for intermediate results, which may enable further opportunities for register or operation elimination in subsequent pipeline stages.

On the software side, pipelining leads to an increase in complexity in the form of initialization and finalization code before and after the innermost loop. Indeed, the pipeline has to be filled in order to reach a steady-state before entering the innermost loop, and it must be emptied at the end of the loop execution. Operation elimination also requires some additional initialization code and is equivalent to unrolling the first iteration of the inner processing loop. An example of this is shown in Fig. 4.8, which is the code from Fig. 4.2(a) with the first iteration moved outside the loop. Note that the value of `a` is no longer calculated inside the loop, but that the value calculated for `b` in the previous iteration is instead reused. The increase in software complexity does not lead to significant performance degradation, especially for wide images, since it is outside the innermost loop.




```
Ry3 := (Z-1 y1,3)          R3   := (Z-1 T3)
Ry4 := (Z-1 y1,4)          R1   := (Z-1 R3)
Ry1 := (Z-1 Ry3)           RRy2 := (Z-1 Ry2)
Ry2 := (Z-1 Ry4)           T4   := (op2 R1 RRy2 R2)
T2  := (op1 Ry2 Ry3)       T5   := (op2 R2 Ry1  R3)
T3  := (op1 Ry3 Ry4)       qi,j := (Z-1 T4)
R2  := (Z-1 T2)            qi,j+1 := (Z-1 T5)
```

FIGURE 4.7 Redundant operation elimination

4.5.5 Load/Store Infrastructure

The load/store infrastructure proposed in this work is based on a linear systolic array [6]. The structure of such an array is shown in Fig. 4.9(a). In this figure, each square represents a register which contains one input pixel, while arrows represent data flow. Data reuse is accomplished through the use of a single long shift register. Pixel data flows from one pixel to the next in a given line. When pixels reach $y_{k,1}$, the end of a line, they are then delayed by $W - w'$ clock cycles before they reach $y_{k+1,w'}$. This structure exhibits optimal input pixel reuse, since each input pixel is read from the external source only once and is then kept inside the processor until no longer needed. Obvious similarities between this structure and the structure described in section 4.5.3 are that the entire pixel window is available at once and that *intra-line* reuse is performed through shift registers.

Fig. 4.9(b) shows a variation on this structure which is convenient in the context of an

```
for(i = 0; i < H; ++i) {
   a    = op1(p[i][0], p[i][1]);
   b    = op1(p[i][1], p[i][2]);
   q[i][j] = op2(a, p[i][1], b);

   for(j = 1; j < W-2; ++j ) {
      a = b;
      b = op1(p[i][j+1], p[i][j+2]);
      q[i][j] = op2(a, p[i][j+1], b);
   }
}
```

FIGURE 4.8 Redundant operation elimination in C code

ASIP. In this figure, the labelled ellipses represent load operations. Data is obtained by a load operation the first time it is needed in each line of the kernel. However, given a data cache of sufficient size – that is, of the size of a few image lines – the pixels are loaded from the external source only the first time they are used, leading to a reasonably low decrease in performance when compared to the optimal reuse case of a linear systolic array. When used in subsequent lines, the data is loaded from the data cache. This approach has two main advantages: it works for any image width as long as the data cache size is sufficient, and inter-line reuse is completely transparent in the perspective of custom instructions design, which reduces design complexity.

The complete data-loading infrastructure proposed in this work is presented in Fig. 4.9(c). In this figure, the squares which represent input pixel registers are grouped by two. This reflects the fact that, in accordance with the intra-line reuse scheme described in section 4.5.3, the constituting elements of the shift registers used for intra-line reuse are not individual pixels, but rather blocks of m pixels (here, the particular case $m = 2$ is shown). A mechanism for decoupling data loading and data processing is used. This decoupling serves two objectives: to present atomically as input to the processing operations the pixels loaded from various lines of the kernel, leading to more flexibility and simplicity in the scheduling of instructions, and to handle the common case where data words handled by the load

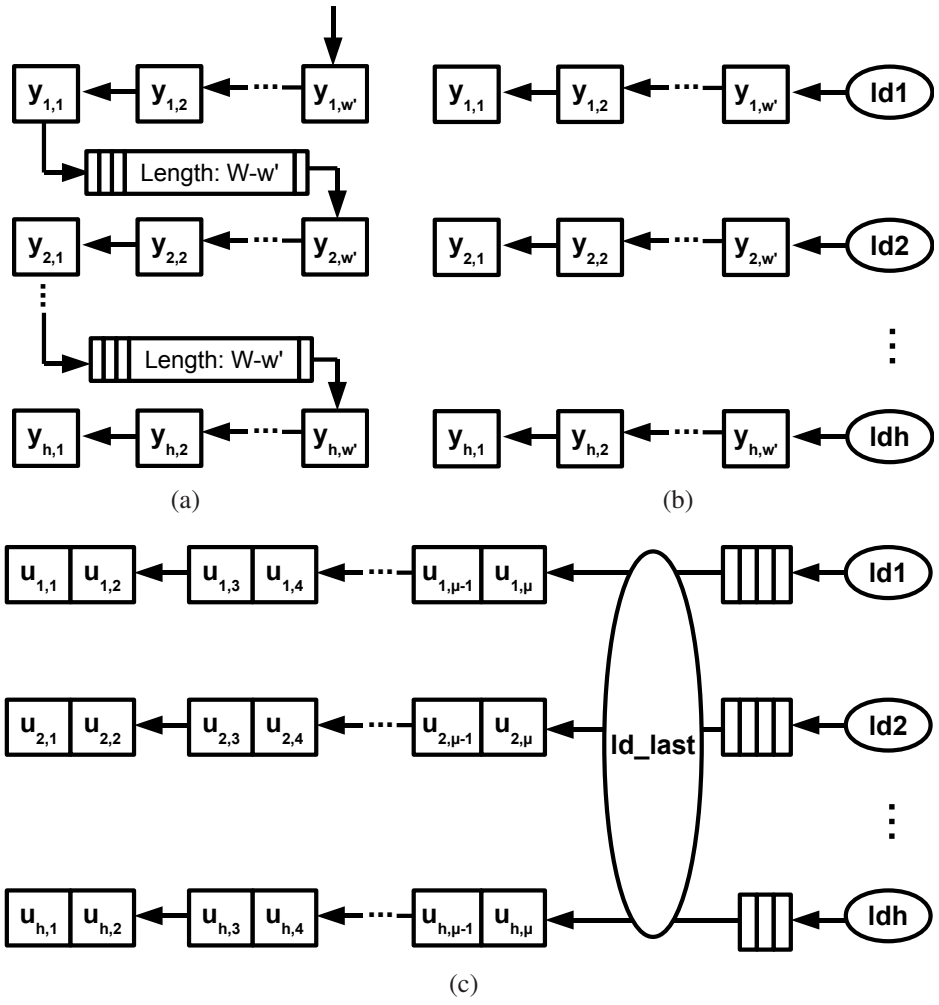


FIGURE 4.9 Data loading infrastructure

instructions do not contain an integer number of pixels.

The decoupling mechanism for data loading works as follows: for each kernel line, there is one load instruction which makes use of a dedicated load buffer. The size in bits of each buffer is a multiple of both the size of a pixel block and the native size of data words loaded by load instructions. When a load instruction loads one data word, it places it in its buffer and shifts the data words already there as in a shift register. Then, when all buffers are filled with new data, a custom instruction copies atomically all their content to the shift

registers which serve as primary inputs to the processing operations. This mechanism may be further improved by making one buffer shorter than the others by one data word and creating a custom instruction which combines the loading of the last data word with the copying of the data. This is the situation shown in Fig. 4.9(c) where the buffer for `ldh` is one data word shorter than the others, and where the instruction `ld_last` both loads the last word from line h and copies the content of the buffers to the shift registers in the left part of the figure.

A similar mechanism is used to decouple the production of output pixels by the processing operations and the storing of these pixels. When output pixels are produced by the processing pipeline, they accumulate in a shift register. Then, the pixels are copied by a custom instruction to a dedicated store buffer when there are enough pixels to fill this buffer. Finally, a custom store instruction reads one word from the buffer, stores it, and shifts the contents of the buffer in preparation for the next execution of the instruction.

Each custom load and store instruction has a dedicated address register which contains the address for the load or store, and which is automatically incremented by the instruction.

4.5.6 Implementation Parameters

In this section, the number m of pixels computed in parallel by SIMD instructions, as well as the size of the load and store buffers introduced in section 4.5.5, are addressed.

Let us define s_p as the size of a pixel in bits and s_w the size of a data word loaded by a load instruction or stored by a store instruction. In order to reduce the gate count of the processor, the size of load buffers is chosen as

$$s_b = LCM(s_w, s_p) \tag{4.26}$$

where LCM stands for the lowest common multiple. Load buffers contain an integer

number N_w of data words, given by

$$N_w = \frac{s_b}{s_w}. \tag{4.27}$$

It takes $h \cdot N_w$ instruction cycles to complete all load steps and N_w to complete all store steps. Thus, the number of instruction cycles required by all load and store steps is

$$N_w \cdot (h + 1). \tag{4.28}$$

Load buffers contain an integer number N_m of pixel blocks given by

$$N_m = \frac{s_b}{m \cdot s_p} \tag{4.29}$$

which means N_m instructions cycles are required to process all this data. Since we want data processing to be fully parallelized with loads and stores, the condition

$$N_m \leq (h + 1) \cdot N_w \tag{4.30}$$

must be satisfied. However, N_m must be chosen as high as possible in order to lower m, which implies a narrower, less costly datapath. The value of N_m is chosen as the largest possible value such that (4.30) is satisfied and that the size of a block of m pixels evenly divides the size of a load buffer. If N_m is strictly smaller in (4.30), that is if $N_m < (h + 1) \cdot N_w$, no processing step occur during some instruction cycles.

Since the number of pixel blocks per line copied by the load instructions as input to the processing operations is $N_m \geq 1$, N_m processing steps must occur to process all this data. To allow for this, the shift registers which serve as input to the processing operations must be enlarged. This is reflected in Fig. 4.9(c) by the fact that the elements of the shift registers are labelled with the letter u instead of y as in figs. 4.9(a) and 4.9(b). $U(i, j)$ is

the $h \times \mu$ subset of image P where

$$\mu = w' + (N_m - 1) \cdot m \tag{4.31}$$

such that

$$u_{1,1} = y_{1,1} \tag{4.32}$$

which implies

$$Y(i,j) \subseteq U(i,j) \tag{4.33}$$

that is, $U(i,j)$ is either $Y(i,j)$ if $N_m = 1$ or $Y(i,j)$ extended to the right by $N_m - 1$ blocks of m pixels if $N_m > 1$.

4.6 Results

This section presents the results obtained for the implementation of various local neighbourhood functions using the approach described in previous sections. Three intra-field deinterlacing algorithms were implemented, as well as two-dimensional convolution with four different kernel sizes. Results for one of the deinterlacing algorithms were previously published in [2]. All implementations are based on the Xtensa LX2 configurable and extensible processor [22, 29]. In the configuration which is used for the ASIPs, it has a 128-bit memory bus and load-store unit which may be used by custom load-store instructions.

The results presented here were obtained from cycle-accurate simulations performed using the Tensilica tools. For comparison purposes, in each case, simulation results are presented for the ASIP, but also for a pure software implementation running on a bare Xtensa LX2 without instruction-set extension. In this reference configuration, the Xtensa LX2 is similar to other common general-purpose 32-bit RISC processors. The reference configuration uses a 32-bit memory bus. Indeed, increasing the bus width to 128 bits, which leads to an increase of 29% in processor area, does not improve execution speed, because

the marginal improvement in terms of clock cycles is more than made up for by the slight decrease in clock frequency caused by the presence of the 128-bit load/store unit.

4.6.1 Intra-Field Deinterlacing

Three ASIP implementations of intra-field deinterlacing algorithms were created: one of the ELA algorithm [9], one of Enhanced ELA [10] and one of the PBDI algorithm [12]. These are three edge-based algorithms of differing complexity.

Each implementation is intended to process 24-bit colour pixel data (eight bits per colour component). Sixteen pixels ($N_m = 16$) are generated in nine instruction cycles ($N_w \cdot (h + 1) = 9$), and the data path has sufficient width to process four pixels in parallel ($m = 4$). The architectures of the three implementations are very similar. The architecture of the PBDI implementation is described in detail in [2].

Simulations were performed with a data set of four images having each a resolution of 288 $\times$ 352 pixels (144 original lines). A summary of the profiling results is given in table 4.1. The clock frequency and processor area are estimates given by the Tensilica tools and assume a 130 nm low voltage process.

TABLE 4.1 Simulation results for processing four images of 352 $\times$ 288 pixels

Description	Exec. time (Mcycles)	Area (k gates)	Freq. (MHz)	Speedup factor
ASIP ELA	0.23	162	300	95
Software ELA	24.02	50	325	1
ASIP Enh. ELA	0.24	171	300	117
Software Enh. ELA	29.99	50	325	1
ASIP PBDI	0.23	273	300	1330
Software PBDI	337.40	50	325	1

The slight decrease in clock frequency of the ASIPs when compared to the reference configuration is due to the presence of the 128-bit load/store unit. The custom instructions were designed and pipelined to leave the clock frequency unchanged.

The three algorithms possess very different computational complexities. The execution time of the software implementation of the most complex algorithm, PBDI, is 14 times that of the fastest algorithm, ELA. However, the results show that the processing speed is roughly the same for all three ASIP implementations. This is to be expected since the pattern of access to data is the same: in all three cases, the kernel has a height of two pixels, which means that two blocks of input pixels are needed to produce one output block.

Interpolated pixels are produced at a rate of 0.86 pixels per clock cycle, which is 260 Mpixels/s. This exceeds the requirements to process high-definition video in real-time. For example, it is four times the processing speed required to process 1080i high-definition video in real-time at the standard rate of 60 fields per second [2]. However, these numbers do not take into account the latency involved in fetching the input pixels and storing the results off-chip.

Table 4.2 summarizes, for each algorithm, the comparison between the ASIP implementation and the reference software implementation. The first column of data shows the speedup factor, which is repeated from table 4.1. The second column shows the size of the ASIP relative to that of the reference processor. Finally, the third column shows the factor of improvement of the AT product, which may be obtained by dividing the speedup factor by the relative processor area.

TABLE 4.2 Comparison of the improvement of the AT product for the three algorithms

Algorithm	Speedup factor	Relative area	Relative AT product
ELA	95	3.2	29
Enhanced ELA	117	3.4	34
PBDI	1330	5.5	243

Relative areas vary between 3.2 and 5.5, which constitutes significant increases. However, even given the area increase, the AT product is still improved by a factor between 29 and 243. This metric is a measure of the relative performance per unit of area. As relatively large overheads are invested to obtain a performance faster than real-time, it would be worth

trimmimg back the bandwidth to obtain designs that meet just the real-time. The excess bandwidth clearly opens possible trade-offs between complexity and speed as a function of application specific performance requirements.

4.6.2 Convolution

Implementations of the two-dimensional convolution with four different kernel sizes were created. Each implementation is intended to process grayscale pixel data with eight bits of luminance per pixel. Simulations were again performed with a data set of four images having each a resolution of 288 $\times$ 352 pixels. A summary of the profiling results is given in table 4.3.

TABLE 4.3 Simulation results for processing four images of 352 $\times$ 288 pixels

Description	Exec. time (Mcycles)	Area (k gates)	Freq. (MHz)	Accel.
ASIP 3×3	0.19	154	300	36
Software 3×3	7.28	50	325	1
ASIP 3×5	0.19	168	300	74
Software 3×5	15.00	50	325	1
ASIP 5×3	0.24	191	300	59
Software 5×3	15.38	50	325	1
ASIP 5×5	0.24	207	300	80
Software 5×5	20.85	50	325	1

Speedup factors are between 36 and 80. Once again, performance of the ASIPs is the same for kernels with the same number of lines: 0.19 Mcycles for kernel sizes 3×3 and 3×5, and 0.24 Mcycles for kernel sizes 5×3 and 5×5. Relative areas vary between 3.1 and 4.1, with factors of improvement of the AT product between 12 and 22, as shown in table 4.4.

TABLE 4.4 Comparison of the improvement of the AT product for the four kernel sizes

Kernel size	Speedup factor	Relative area	Relative AT product
3×3	36	3.1	12
3×5	74	3.4	22
5×3	59	3.8	15
5×5	80	4.1	19

4.7 Conclusion

A systematic approach to the design of ASIPs for high speed computation of local neighbourhood functions and intra-field deinterlacing was proposed. This approach aims at an efficient utilization of the available memory bandwidth by fully exploiting the data parallelism inherent to the target algorithm class. An appropriate choice of custom instructions and application-specific registers is used together with a VLIW architecture in order to mimic a pipelined systolic array.

Results for the implementation of three intra-field deinterlacing algorithms and for four kernel sizes of two-dimensional convolution were presented. On the basis of these results, it was shown that a significant improvement of both processing speed and the AT product is possible using this approach.

4.8 Acknowledgments

The authors would like to thank Mathieu Desnoyers for his work on the Enhanced ELA implementation. Also, they would like to acknowledge the products and services provided by CMC Microsystems (www.cmc.ca) that facilitated this research, and the financial support of the Canadian Natural Sciences and Engineering Research Council (NSERC) and the Canada Research Chair program.

Chapitre 5

DISCUSSION GÉNÉRALE

5.1 Exemple d'application de l'approche

L'approche de conception présentée au chapitre 4 comprend quatre étapes. Quelques-unes de ces étapes ouvrent la porte à des simplifications matérielles permettant une économie de surface. Dans la présente section est discutée la manière dont les différentes étapes et simplifications s'appliquent à PBDI, l'algorithme ayant été l'objet de l'exploration architecturale. Cet exemple d'application de l'approche de conception est fourni dans le but de la clarifier. La discussion de cette section recoupe en partie le contenu de l'article présenté à la conférence NEWCAS-TAISA 2009 [2].

La première étape de l'approche, qui fait l'objet de la section 4.5.1, consiste à concevoir une instruction parallèle de type SIMD. Dans le cas du processeur spécialisé conçu pour PBDI, un bloc de quatre pixels est calculé par l'instruction en question. Ce choix est discuté plus bas.

Cette première étape ouvre la voie à des simplifications matérielles par élimination d'opérations redondantes décrites à la section 4.5.2. La figure 5.1 illustre la situation : la figure 5.1(a) présente les pixels d'entrée et les comparaisons de patrons impliqués dans le calcul d'un pixel de sortie, alors que la figure 5.1(b) montre la même chose pour un bloc de quatre pixels. Dans ces figures, chaque cercle représente un pixel, et chaque segment de droite représente une comparaison de patrons de trois pixels à l'aide de la fonction de comparaison décrite à la section 2.3.3. Pour une comparaison donnée, le segment de droite relie les pixels centraux des patrons comparés.

En observant côte-à-côte les figures 5.1(a) et (b), on remarque deux choses d'intérêt. Premièrement, alors que le calcul d'un pixel de sortie dépend de dix-huit pixels d'entrée, le calcul d'un bloc n'en nécessite que 24, soit bien moins que $4 \times 18 = 72$. En calculant quatre pixels en parallèle, il est donc possible d'éliminer 48 opérations de chargement par bloc, ce qui se traduit par une amélioration de la vitesse de traitement, puisque l'accès aux données constitue le goulot d'étranglement principal de l'application.

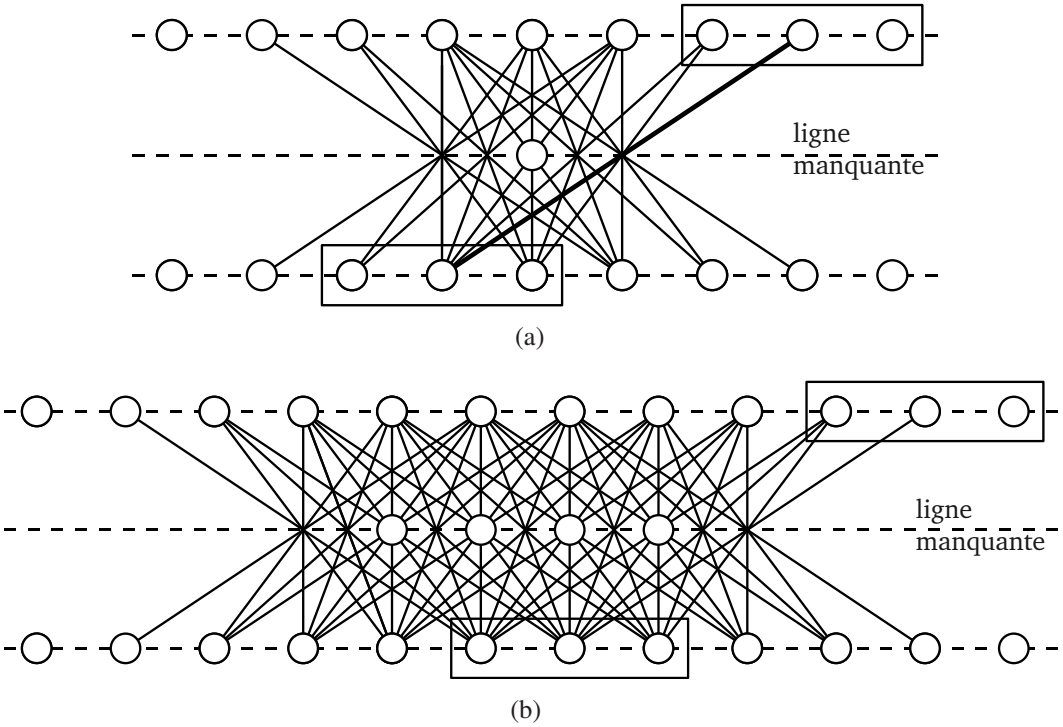


FIGURE 5.1 Comparaisons de patrons pour le calcul (a) d'un seul et (b) de quatre pixels-résultats

Il est facile de se convaincre que ces opportunités d'élimination d'opérations de chargement s'appliquent à tous les filtres locaux ayant une largeur de fenêtre supérieure à un pixel. Dans tous les cas, le calcul d'un pixel de sortie supplémentaire nécessite seulement le chargement d'un nombre de pixels d'entrée supplémentaires égal au nombre de lignes de la fenêtre de l'algorithme.

La seconde observation issue de la comparaison des figures 5.1(a) et (b) est que des

comparaisons de patrons sont aussi éliminées. En effet, bien que 21 comparaisons soient nécessaires au calcul d'un pixel de sortie, seuls 50 sont nécessaires au calcul d'un bloc de quatre pixels, soit bien moins que $4 \times 21 = 84$. Parmi les algorithmes sélectionnés pour ces travaux, une telle élimination d'opérations redondantes au-delà des opérations de chargement n'a été possible que pour PBDI.

L'élimination des comparaisons de patrons peut être visualisée en construisant la figure 5.1(b) par une superposition de quatre copies de la figure 5.1(a), chacune translatée d'un pixel vers la droite par rapport à la précédente. Dans une telle superposition, des segments de droite d'une copie donnée représentant des comparaisons de pixels chevaucheront ceux d'autres copies. Ces chevauchements représentent les comparaisons redondantes.

La deuxième étape de l'approche de conception, décrite à la section 4.5.3, consiste à créer les registres à décalage permettant la réutilisation des pixels d'entrée d'un bloc de quatre pixels à l'autre. Cette étape permet de réduire à huit le nombre d'opérations de chargement, conformément à l'équation (4.18). Alors qu'à l'étape précédente, les redondances interne à un bloc de pixels étaient considérés, à cette seconde étape, on tire plutôt avantage des redondances inter-bloc. L'élimination d'opérations de chargement que permet cette étape est applicable à tous les filtres locaux.

La troisième étape, décrite à la section 4.5.4 consiste en la création d'un pipeline. Pour ce faire, l'instruction parallèle est découpée en plusieurs nouvelles instructions communicant via des registres propres à l'application prévus à cet effet. Le pipeline du processeur spécialisé pour PBDI comprend quatre étages :

- `lum` effectue le calcul de la luminance de chaque pixel d'entrée ;
- `err1` réalise une première partie des comparaisons ;
- `err2` complète les comparaisons ; et
- `dir` effectue l'interpolation.

Deux types de simplifications matérielles sont proposées à la section 4.5.4, soit une élimination d'opérations et une élimination de registres. L'élimination d'opérations, a pour

effet la création de registres à décalage dans lesquels transitent des résultats intermédiaires en vue de leur réutilisation. Il a été possible de l'appliquer de manière à éviter d'avoir à recalculer 1) certaines valeurs de luminance, dans le cas des trois algorithmes de désentrelacement ; et 2) certaines valeurs de la fonction de comparaison, dans le cas de l'algorithme PBDI. Des 50 comparaisons impliquées dans le calcul d'un bloc de pixels de sortie à l'aide de PBDI, seuls 36 représentent des nouvelles valeurs devant être calculées. Les autres peuvent simplement être réutilisées. Pour ce qui est des calcul de luminance, ceux-ci sont réalisés de manière identique pour les trois algorithmes de désentrelacement. Chaque valeur de luminance est calculée en ne prenant pour opérande qu'un pixel d'entrée. Il est donc peu surprenant qu'il soit possible d'éliminer ces calculs pour les mêmes pixels que ceux pour lesquels il avait été possible d'éliminer les opérations de chargement à la deuxième étape de l'approche. Dit autrement, la valeur de luminance est réutilisée, plutôt que recalculée, pour les pixels d'entrée qui sont réutilisés, plutôt que rechargés. Ainsi, pour obtenir un bloc de quatre pixels-résultats, seules huit valeurs de luminances doivent être calculées, plutôt que 12 pour ELA, 14 pour Enhanced ELA et 18 pour PBDI.

L'élimination de registres, quant à elle, vise une situation particulière, c'est-à-dire la situation où :

1. des données doivent être acheminées d'un premier étage du pipeline vers un second étage qui ne suit pas immédiatement le premier ; et
2. ces mêmes données transitent aussi dans un registre à décalage ayant été formé lors de l'application de l'élimination d'opérations.

Cette situation se présente à un seul endroit dans le processeur spécialisé pour l'algorithme PBDI : des pixels et leur information de luminance doivent être acheminés de l'étage `err1` à l'étage `dir`. Il a donc été possible d'éliminer des registres à cet endroit. Parmi les algorithmes considérés par ces travaux, PBDI est le seul pour lequel ce type de simplification a été possible.

L'étape finale de l'approche, décrite à la section 4.5.5, consiste à réaliser l'infrastructure

de chargement et de sauvegarde. À cette étape, aucune simplification matérielle particulière n'est proposée. Le mécanisme de mise en tampon décrit à la section 4.5.5 est utile pour PBDI et les autres algorithmes de désentrelacement. En effet, les pixels ont une taille de 24 bits (trois composantes de couleur 8 bits chacun) alors que le bus de données à une taille de 128 bits. La taille du bus n'est donc pas divisible par la taille d'un pixel.

Quoique l'approche de conception proposée soit applicable à tous les filtres locaux, sujette aux contraintes concernant la surface de silicium disponible, on retient de la discussion de cette section que tous les algorithmes ne sont pas égaux du point de vue des simplifications matérielles applicables. Il peut être avantageux de garder ce fait en tête lors du choix ou de la conception d'un algorithme : certains ajouts de complexité peuvent être moins coûteux qu'il n'y paraît au premier abord, si les simplifications matérielles décrites peuvent s'y appliquer.

Nous avons mentionné plus haut que le processeur spécialisé pour PBDI possède un chemin des données qui permet le calcul de quatre pixels en parallèle. C'est aussi le cas des processeurs pour ELA et Enhanced ELA. Par contre, dans tous ces cas, il aurait été possible d'atteindre la même performance en ne calculant que deux pixels à la fois, comme l'indique la section 4.5.6. En effet, dans le cas qui nous occupe, $s_w = 128$ (taille du bus de données) et $s_p = 24$ (taille d'un pixel). Donc, de (4.26)

$$s_b = LCM(128, 24) = 384 \tag{5.1}$$

et de (4.27)

$$N_w = \frac{s_b}{s_w} = \frac{384}{128} = 3 \tag{5.2}$$

et donc, si nous posons $m = 2$, nous avons, de (4.29)

$$N_m = \frac{s_b}{m \cdot s_p} = \frac{384}{2 \cdot 24} = 8 \tag{5.3}$$

ce qui implique que la condition donnée en (4.30) soit respectée, puisque

$$(h+1) \cdot N_w = (2+1) \cdot 3 = 9. \tag{5.4}$$

La raison qui explique le surdimensionnement du chemin des données est que l'infrastructure de chargement et sauvegarde a été améliorée après la conception de ces processeurs spécialisés. Par contre, les processeurs spécialisés pour la convolution 2D ont été conçus avec une taille minimale du chemin des données. Si deux pixels avait été calculés en parallèle plutôt que quatre pour les algorithmes de désentrelacement, la performance serait restée la même, mais la surface utilisée aurait vraisemblablement été réduite, ce qui aurait mené à une augmentation du facteur d'amélioration du produit temps-surface.

5.2 Détails techniques et limites de la méthodologie

L'environnement de conception Xtensa Xplorer de Tensilica, dont il est fait mention à la section 2.4.2, a été utilisé pour l'exploration architecturale, la conception des processeurs spécialisés et la réalisation des simulations dont sont issus les résultats de performance. Dans cette section sont discutées les étapes qu'impliquent la conception d'un processeur spécialisé pour un algorithme donné et l'obtention des résultats de performance dans cet environnement.

Tout d'abord, une mise en œuvre purement logicielle de l'algorithme est réalisée sous la forme d'un code source en langage C. La vérification de la fonctionnalité est effectuée par l'exécution du programme avec un ensemble d'images de test en entrée, suivi d'une inspection visuelle des images-résultats. Cette vérification est effectué hors d'Xtensa Xplorer, puisqu'une exécution native sur PC est plus rapide qu'une simulation. Les images de l'ensemble de test sont choisies en fonction de deux critères : elles doivent représenter des scènes variées, et elles doivent contenir des arêtes diagonales d'orientations diverses.

Lorsque la fonctionnalité est jugée correcte, une empreinte des images-résultats est obtenue à l'aide la fonction de hachage cryptographique MD5. Ceci permet de vérifier rapidement et de manière automatique que la fonctionnalité n'ait pas été modifiée après chacune des étapes subséquentes, sans qu'une inspection visuelle ne soit nécessaire.

Un projet logiciel contenant le code source correct est ensuite créé dans l'environnement Xtensa Xplorer. Une simulation de l'exécution de ce code source est réalisée, et le nombre de cycles d'horloge total de l'exécution est recueilli. Ce nombre de cycles d'horloge servira de référence lors du calcul du facteur d'accélération. À cette étape, la simulation cible un processeur Xtensa LX2 de référence sans extension du jeu d'instruction, et de configuration typique d'un processeur RISC 32 bits d'usage général. Le même processeur de référence est utilisé pour tous les algorithmes. Sa fréquence d'horloge et sa taille en nombre de portes sont connues, puisque Xtensa Xplorer en donne un estimé.

L'étape suivante consiste à décrire les instructions spécialisées et les registres propres à l'application à l'aide du langage propriétaire TIE. Une nouvelle configuration de processeur est réalisée en attachant le code TIE à un processeur de base identique au processeur de référence à un détail près : un bus de données de 128 bits est utilisé, plutôt que le bus de 32 bits utilisé par le processeur de référence. L'extension du jeu d'instruction en langage TIE est réalisée de manière incrémentale, avec vérification de la fonctionnalité après chaque modification. À cette étape, une simulation fonctionnelle est réalisée, plutôt qu'une simulation précise au cycle d'horloge, afin de réduire le temps nécessaire à la simulation.

Lorsque l'extension du jeu d'instruction est complète, une simulation précise au cycle d'horloge est réalisée, et le nombre de cycles d'exécution est recueilli en vue du calcul du facteur d'accélération. La taille du processeur spécialisé, fournie par l'environnement de conception, est aussi prise en note en vue du calcul de la taille relative et du facteur d'amélioration du produit temps-surface.

Une limite importante de la méthodologie utilisée vient du fait que l'estimé de la fréquence d'horloge donné par Xtensa Xplorer ne tient pas compte de l'extension au jeu

d'instructions codé en langage TIE. La manière correcte et recommandée d'obtenir cette fréquence d'horloge serait de générer le processeur prêt à être synthétiser pour ensuite en effectuer la synthèse, ce que ne permettent pas les licences disponibles à l'École. Nous avons plutôt procédé comme suit : la liste des opérations primitives utilisées (additions, multiplications, multiplexeurs, etc.), avec leur taille respective, a été dressée. Des modules équivalents ont été codées en langage VHDL. L'effort que ceci représente est faible, puisqu'il existe en VHDL un opérateur pour chacune de ces opérations primitives. Ensuite, une synthèse logique de chaque module a été réalisée, ce qui a permis d'obtenir un estimé du délai attribuable à chaque opération. Cette information a par la suite été utilisée lors du découpage en un pipeline de l'instruction parallèle afin d'évaluer le délai de chaque étage de pipeline. À cette étape, nous avons cherché à laisser inchangée la fréquence d'horloge du processeur, comme proposé à la section 4.5.4. Puisque l'évaluation du délai du chemin critique est basé sur une estimation du délai des opérations individuelle plutôt que sur les résultats de la synthèse du processeur complet, il est possible que la fréquence d'horloge ait été légèrement surévaluée, mais il est peu probable qu'elle l'ait été de manière importante.

Chapitre 6

CONCLUSION

Dans ce mémoire ont été présentés les travaux de recherche visant à explorer les possibilités qu'offrent les processeurs à jeu d'instructions spécialisés pour une classe d'algorithmes utilisée en traitement vidéo : les filtres locaux. Suite à une étude de cette classe d'algorithmes et à une exploration architecturale, une approche systématique pour la conception de processeurs spécialisés a été proposée. Sept processeurs spécialisés ont été conçus afin de démontrer expérimentalement la pertinence de cette approche.

L'approche de conception proposée comprend quatre étapes. La première consiste à créer une instruction parallèle (SIMD) qui effectue le calcul de plusieurs pixels de sortie à la fois. Le nombre de pixels de sortie qu'il est avantageux de calculer en parallèle peut être déterminé à partir de la taille des pixels et de la taille du bus de données du processeur. La deuxième étape consiste en la création de registres à décalage permettant la réutilisation intra-ligne des pixels d'entrée. Une telle réutilisation permet une utilisation plus efficace de la bande passante vers la mémoire, laquelle bande passante constitue le goulot d'étranglement de l'application. La troisième étape consiste à créer un pipeline en découpant en étages l'instruction parallèle et en ajoutant des registres pour les résultats intermédiaires. Ceci permet d'augmenter le débit de pixels produits. L'étape finale consiste à créer l'infrastructure de chargement et de sauvegarde. Cette infrastructure gère les incompatibilités de taille entre les pixels et le bus de données de manière à ce que des données soient fournies de manière ininterrompue au pipeline de traitement. De plus, une mise à jour automatique des pointeurs permet d'obtenir une accélération supplémentaire. Pour plusieurs des étapes décrites, des simplifications matérielles sont proposées. L'approche proposée mène à la création d'une structure matérielle où il est possible d'exécuter une instruction de charge-

ment ou de sauvegarde à chaque cycle d'instructions dans la boucle interne l'algorithme. Le traitement des données peut être complètement parallélisé avec les chargements et les sauvegardes, ce qui est réalisé grâce à une architecture VLIW.

Afin de valider expérimentalement l'approche de conception proposée, sept processeurs spécialisés ont été conçus. Trois d'entre eux visent chacun un algorithme de désentrelacement intra-trame. Les quatre autres visent la convolution 2D avec chacun une taille distincte de fenêtre de convolution. Les résultats de performance confirment la pertinence de l'approche. Pour les algorithmes de désentrelacement intra-trames, les facteurs d'accélération varient entre 95 et 1330, alors que les facteurs d'amélioration du produit temps-surface varient entre 29 et 243, tout ceci par rapport à un processeur d'usage général de référence roulant une implémentation purement logicielle de l'algorithme. Dans le cas de la convolution 2D, les facteurs d'accélération varient entre 36 et 80, alors que les facteurs d'amélioration du produit temps-surface varient entre 12 et 22. Dans tous les cas, le traitement en temps réel de données vidéo à haute définition au format 1080i (désentrelacement) ou 1080p (convolution) est possible avec une fabrication dans une technologie de circuit intégré CMOS à 130 nm.

6.1 Travaux futurs

L'approche de conception proposée pourrait être automatisée sous la forme d'un outil de synthèse haut-niveau. Deux manières de faire sont envisageables. Premièrement, il serait possible de créer un outil prenant en entrée le code source de l'algorithme, par exemple codé en langage C. Des outils de synthèse haut-niveau prenant un code source en langage C en entrée existent déjà [30], mais un outil ayant une connaissance préalable de la classe d'application visée serait plus simple qu'un outil plus général, tout en permettant d'atteindre de meilleures performances. Une autre manière de faire consisterait à utiliser plutôt un langage propre à l'application. Un langage fonctionnel serait tout indiqué étant

donné la classe d'algorithmes visée par ces travaux. Cette seconde manière de faire semble avantageuse, puisqu'il est plus simple de transformer la description de l'algorithme en une description matérielle synthétisable fonctionnelle s'il existe une relation directe entre ce qu'il est possible d'exprimer avec le langage et ce que l'outil est en mesure de générer comme description synthétisable.

RÉFÉRENCES

[1] E. Mollick, “Establishing Moore’s law,” *IEEE Annals of the History of Computing*, vol. 28, no. 3, p. 62–75, 2006.

[2] P. Aubertin, H. M. Mohammadi, Y. Savaria, et J. M. P. Langlois, “High performance ASIP implementation of PBDI – a new intra-field deinterlacing method,” dans *Proceedings of the Joint IEEE NEWCAS-TAISA Conference*, 2009, p. 372–375.

[3] N. Beucher, N. Bélanger, Y. Savaria, et G. Bois, “A methodology to evaluate the energy efficiency of application specific processors,” dans *Proceedings of the 14th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, 2007, p. 983–986.

[4] N. Beucher, “Conception et mise en oeuvre de processeurs configurables pour la conversion de taux de trames vidéos avec compensation de mouvement,” mémoire de maîtrise, École Polytechnique de Montréal, 2007.

[5] J. Hardelin et L’équipe de documentation de GIMP, “Matrices de convolution,” dans *GNU Image Manipulation Program – Guide utilisateur*, 2010. [En ligne]. Disponible : http://docs.gimp.org/fr/plug-in-convmatrix.html

[6] R. B. Porter, “Image processing,” dans *Reconfigurable Computing–Accelerating Computation with Field-Programmable Gate Arrays*, M. B. Gokhale et P. S. Graham, Eds. Springer, 2006, p. 119–139.

[7] R. B. Porter, J. R. Frigo, M. Gokhale, C. Wolinski, F. Charot, et C. Wagner, “A run-time re-configurable parametric architecture for local neighborhood image processing,” dans *Proceedings of the 9th EUROMICRO Conference on Digital System Design*. IEEE Computer Society, 2006, p. 107–115.

[8] G. De Haan et E. B. Bellers, “Deinterlacing–an overview,” *Proceedings of the IEEE*, vol. 86, no. 9, p. 1839–1857, 1998.

[9] T. Doyle, “Interlaced to sequential conversion for EDTV applications,” dans *Proceedings of the Second International Workshop on Signal Processing of HDTV*, 1988, p. 412–430.

[10] S.-F. Lin, Y.-L. Chang, et L.-G. Chen, “Motion adaptive interpolation with horizontal motion detection for deinterlacing,” *IEEE Transactions on Consumer Electronics*, vol. 49, no. 4, p. 1256–1265, 2003.

[11] H. Y. Lee, J. W. Park, S. U. Choi, T. M. Bae, et Y. H. Ha, “Adaptive scan rate up-conversion system based on human visual characteristics,” *IEEE Transactions on Consumer Electronics*, vol. 46, no. 4, p. 999–1006, 2000.

[12] H. M. Mohammadi, “An effective hybrid video deinterlacing algorithm,” thèse de doctorat, École Polytechnique de Montréal, 2009.

[13] X. Guan, H. Lin, et Y. Fei, “Design of an application-specific instruction set processor for high-throughput and scalable FFT,” dans *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, 2009, p. 1302–1307.

[14] G. Kappen, S. Bahri, O. Priebe, et T. Noll, “Evaluation of a tightly coupled ASIP / co-processor architecture used in GNSS receivers,” dans *Proceedings of the International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2007, p. 1302–1307.

[15] M. Alles, T. Vogt, et N. Wehn, “Flexichap : A reconfigurable ASIP for convolutional, turbo, and LDPC code decoding,” dans *Proceedings of the 5th International Symposium on Turbo Codes and Related Topics*, 2008, p. 84–89.

[16] J. Groszschadl, P. Ienne, L. Pozzi, S. Tillich, et A. K. Verna, “Combining algorithm exploration with instruction set design : A case study in elliptic curve cryptography,” dans *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, 2006, p. 218–223.

[17] S. D. Kim, J. H. Lee, C. J. Hyun, et M. H. Sunwoo, “ASIP approach for implementation of H.264/AVC,” dans *Proceedings of the conference on Asia South Pacific design automation*, 2006, p. 758–764.

[18] S. Saponara, L. Fanucci, S. Marsi, G. Ramponi, D. Kammler, et E. Witte, “Application-specific instruction-set processor for retinex-like image and video processing,” *IEEE Transactions on Circuits and Systems II : Express Briefs*, vol. 54, no. 7, p. 596–600, 2007.

[19] A. Gerstlauer et D. D. Gajski, “System-level abstraction semantics,” dans *Proceedings of the 15th International Symposium on System Synthesis (ISSS)*, 2002, p. 231–236.

[20] M. K. Jain, M. Balakrishnan, et A. Kumar, “ASIP design methodologies : survey and issues,” dans *Proceedings of the Fourteenth International Conference on VLSI Design*, 2001, p. 76–81.

[21] O. Muller, A. Baghdadi, et M. Jézéquel, “From parallelism levels to a multi-ASIP architecture for turbo decoding,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 1, p. 92–102, 2009.

[22] Tensilica Inc., “Tensilica : Customizable Processor Cores for the Dataplane,” 2009. [En ligne]. Disponible : http://www.tensilica.com

[23] CoWare Inc., “System-Level Design,” 2009. [En ligne]. Disponible : http://www.coware.com

[24] F. Cardells-Tormo et P.-L. Molinet, “Area-efficient 2-D shift-variant convolvers for FPGA-based digital image processing,” *IEEE Transactions on Circuits and Systems II : Express Briefs*, vol. 53, no. 2, p. 105–109, 2006.

[25] D. Yadav, A. Gupta, et A. Mishra, “A fast and area efficient 2-D convolver for real time image processing,” dans *Proceedings of the IEEE Region 10 Conference (TENCON)*, 2008, p. 1–6.

[26] B. Bosi, Y. Savaria, et G. Bois, “Reconfigurable pipelined 2-D convolvers for fast digital signal processing,” *IEEE Transactions on Very Large Scale Integrated (VLSI) Systems*, vol. 7, no. 3, p. 299–308, 1999.

[27] K. I. Diamantaras et S. Y. Kung, “A linear systolic array for real-time morphological image processing,” *Journal of VLSI Signal Processing*, vol. 17, no. 1, p. 43–55, 1997.

[28] C. E. Leiserson et J. B. Saxe, “Optimizing synchronous systems,” dans *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*, 1981, p. 23–36.

[29] R. Gonzalez, “Xtensa : a configurable and extensible processor,” *IEEE Micro*, vol. 20, no. 2, p. 60–70, 2000.

[30] S. Gupta, R. Gupta, N. Dutt, et A. Nicolau, *SPARK : A parallelizing Approach to the High-Level Synthesis of Digital Circuits*. Springer-Verlag New York, LLC, 2004.