THESE PRESENTEE POUR OBTENIR LE GRADE DE DOCTEUR EN SCIENCES DE L'UNIVERSITE PIERRE ET MARIE CURIE (PARIS 6)

- SPECIALITE : ROBOTIQUE -

PAR

Frantz LOHIER

Méthodologies de programmation et évaluation des processeurs de traitement de signal parallèles pour le traitement d'images en temps réel

SOUTENUE LE 4 FEVRIER 2000

devant le Jury composé de :

Mr	**D. Demigny**	: Rapporteur
Mr	**P. Garda**	: Directeur de la thèse
Mme	**C. Lambert-Nebout**	: Examinatrice
Mr	**F. Luthon**	: Rapporteur
Mr	**M. Paindavoine**	: Président
Mr	**J.-P. Ricaud**	: Examinateur

Méthodologies de programmation et évaluation des processeurs de traitement de signal parallèles pour le traitement d’images en temps réel

Par Frantz LOHIER

Je remercie le Groupe Cofidur représenté par Monsieur Jean-Philippe Ricaud pour m'avoir permis de mener à bien mes recherches.

Je remercie le Professeur Michel Paindavoine qui a bien voulu accepter la présidence du Jury. J'adresse mes vifs remerciements au Professeur Didier Demigny et à Monsieur Franck Luthon, Maître de Conférences, pour avoir accepté d'être rapporteurs de ce manuscrit. Je remercie également Madame Catherine Lambert-Nebout pour l'intérêt qu'elle a manifesté envers mes travaux.

Je tiens à remercier tout particulièrement Monsieur Patrick Garda, Professeur à l'université de Paris 6 et Directeur du LIS, pour avoir accepté l'encadrement scientifique de cette thèse au sein de son laboratoire. Je lui suis reconnaissant de la grande disponibilité qu'il m'a toujours accordée ainsi que de son soutien tout au long de mes recherches et pendant la rédaction de cette thèse.

Je remercie Lionel Lacassagne pour avoir favorisé une coopération constructive de nos thèses menées en parallèle tant au niveau des recherches que des publications.

Je remercie mes parents pour le soutien affectif et matériel qu'ils m'ont toujours apporté. Je leur suis très reconnaissant de la liberté et de la confiance qu'ils m'ont toujours manifestées.

Je dédie cette thèse à Anne pour son soutien de tous les instants et sa patience vis-à-vis de l'énorme investissement personnel que nécessite l'accomplissement d'une thèse.

A Anne ;

“A ton invitation au Voyage....”

Table des Matières

Introduction 15

1 Intérêt des DSP et état de l'art pour leur mise en œuvre 21

1.1. Les architectures DSP programmables 21
1.1.1. Un aperçu d'architectures DSP parallèles avancées 21
1.1.1.1. Le Trimedia de Philip 22
1.1.1.2. Le Sharc 21160 et Tiger-Sharc d'Analog Device 24
1.1.1.2.1. Le Sharc 21160 24
1.1.1.2.2. Le Tiger Sharc 28
1.1.1.3. Le TMS320C6X et TMS320C8X de Texas Instruments 30
1.1.1.3.1. Le TMS320C6X 30
1.1.1.3.2. Le TMS320C8X 32
1.1.2. Programmation des DSP 46
1.1.3. Intérêts des DSP 49
1.1.3.1. Concurrence des processeurs RISC 49
1.1.3.2. Adéquation des DSP au marché de l'embarqué 50
1.1.3.3. Intérêt du C80 53
1.1.4. Conclusions 54
1.2. Etat de l'art des outils et méthodes de programmation parallèles 59
1.2.1. Outils et librairies pour le TMS320C80 59
1.2.1.1. Les outils de développement Texas Instruments 59
1.2.1.2. Les librairies de traitement d'images 62
1.2.1.2.1. La librairie de l'université de Washington 62
1.2.1.2.2. Le système Genesis/MIL de Matrox 63
1.2.1.2.3. Conclusions 64
1.2.2. Les langages de programmation dédiés 65
1.2.2.1. Les langages parallèles 66
1.2.2.1.1. Quelques exemples et critères de classification 66
1.2.2.1.2. Le parallélisme de données 67
1.2.2.2. Les langages synchrones 70
1.2.3. Les plates-formes de développement intégrées 71
1.2.3.1. L'approche SynDEx 72
1.2.3.2. Le projet Ptolemy 73
1.2.3.3. Le système Grape II 75
1.2.3.4. Conclusions 76
1.3. Conclusions : Perspectives de contribution 77

2 Méthodologies de développement 83

2.1. Techniques d'optimisation pour la programmation des opérateurs 84
2.1.1. Intérêt de l'optimisation de la granularité des nœuds 85
2.1.2. Le déroulage de boucle 86
2.1.3. Utilisation des capacités SIMD 88
2.1.4. Le pipeline logiciel 89
2.1.5. Utilisation de table de transcodage (LUT) 95
2.2. Une approche logicielle originale pour l'optimisation des flots de données 96
2.2.1. Une approche générique 96
2.2.1.1. Définition des objets élémentaire 97
2.2.1.1.1. Les buffers image 97
2.2.1.1.2. Nœud et chaîne de traitement 99
2.2.1.1.3. Le patron de traitement 99
2.2.1.1.4. La notion centrale des patrons de données 99
2.2.1.2. Cas des patrons de données se recouvrant 103
2.2.1.3. Balayage des données internes/externes 108
2.2.1.4. Conclusions 110
2.2.2. Gestion performante des bancs de mémoire internes 110
2.2.2.1. Intérêt du triple buffering 110
2.2.2.2. Gestion des bancs mémoire avec plusieurs flux 116
2.2.2.3. Portabilité de l'approche : le cas du C6X 116
2.2.3. Partitionnement SPMD flexible des donnée 119
2.2.3.1. Modélisation du coût des transferts selon notre méthodologie 119
2.2.3.2. Partitionnement performant des données 126
2.2.3.2.1. Découpage d'un buffer 126
2.2.3.2.2. Ordre de parcours des données du maillage 132
2.2.3.2.3. Le cas multi-buffers 134
2.2.3.3. Cas des chaînes multi-nœuds 136
2.2.3.3.1. Calcul du patron virtuel 137
2.2.3.3.2. Découpage multi-buffers et patrons virtuels 143
2.2.3.4. Conclusions 145
2.3. Estimation des performances sur C80 146
2.3.1. Un modèle simple 146
2.3.2. Evaluation de la performance des nœuds 149
2.3.3. Les sources de divergence de l'estimation globale 152
2.3.4. Vers une meilleure performance des chaînes de traitement 153
2.4. Conclusions 155

3 Validation des méthodologies de développement 157

3.1. Librairie C80 de traitement d'images bas niveau 158
3.2. Etude des performances de chaînes élémentaires 163
3.2.1. Contexte 163
3.2.2. Résultats 164
3.2.3. Conclusions 168
3.3. Implantation d'un algorithme de détection optimale de contours 169

3.3.1. Etat de l'art des méthodes pour la détection de contours 170
3.3.2. Implantations matérielles sur DS 175
3.3.2.1. Implantation sur le C80 175
3.3.2.1.1. Codage des nœuds 176
3.3.2.1.2. Partitionnement et gestion des flux 181
3.3.2.1.3. Conclusions 187
3.3.2.2. Implantation sur C62 189
3.3.2.2.1. Implantation du nœud gradient 189
3.3.2.2.2. Implantation du lisseur 198
3.3.2.2.3. Perspectives de performances avec le C62 202
3.3.3. Conclusions 203
3.4. Conclusions 204

4 Implantation sur C80 d'une technique de compression vidéo originale basée sur la détection de mouvement 207

4.1. Etat de l'art des techniques pour la détection/compression d'images en mouvement 207
4.2. La détection de mouvement par modélisation Markovienne 209
4.2.1. Champs de Markov 210
4.2.2. Champs de Markov en traitement d'images 211
4.2.3. Un Modèle de fonction potentiel pour la détection de mouvement 213
4.2.4. Conclusions 216
4.3. Implantation de la détection markovienne sur C80 218
4.3.1. Le pré-traitement 218
4.3.2. L'ICM 219
4.3.3. Conclusions 223
4.4. Amélioration des performances d'une librairie de compression M-JPEG sur C80 225
4.4.1. Intérêt de l'encodage JPEG 225
4.4.2. Du format M-JPEG à la structure d'un encodeur original 226
4.4.3. Une autre approche à base de seuillage spatio-temporel fréquentiel 229
4.4.4. Implantation du codeur M-JPEG expérimental 230
4.5. Performances du système de compression proposé 236
4.5.1. Cadence du traitement 236
4.5.2. Comparaisons quantitatives et qualitatives des deux algorithmes 238
4.5.2.1. Analyses quantitatives comparées 242
4.5.2.2. .Analyses qualitatives comparées 247
4.6. Conclusions 254

Conclusions 255
5.1. L'apport scientifique 255
5.2. L'apport des application 257
5.3. Les perspectives de nos recherches 258

Bibliographie par Chapitre 263

Liste des Figures

1-1 L'architecture du TM1000/TM3000 ou CPU32 22
1-2 L'architecture du Sharc 21160 25
1-3 L'architecture TigerSharc 28
1-4 Blocs fonctionnels du TMS320C6X 30
1-5 Blocs fonctionnels du TMS320C80 32
1-6 L'ALU des PP 35
1-7 Niveaux de priorité du DMA C8X 39
1-8 Liste chaînée des requêtes de transfert (PT) 40
1-9 Structure d'un P 40
1-10 Paramétrage des 3 dimensions 41
1-11 Transferts variables 44
1-12 Transferts variables et géométrie variable des blocs 45
2-1 Principe du déroulage de boucle (loop unrolling) 87
2-2 Principe du pipeline logiciel (software pipelining) 89
2-3 Principe du repliement du graphe de dépendance 90
2-4 Pipeline logiciel et définition de l'IMI 92
2-5 Minimisation de la longueur du prologue et de l'épilogue 93
2-6 Formatage des éléments structurant : les patrons synchrones d'entrée/sortie 100
2-7 Intégration des contraintes de taille dans les patrons 102
2-8 Effet de bord et rechargement des données 104
2-9 Rechargement des données pour les filtres FIR et IIR (cas vertical) 105
2-10 Cas général du rechargement des filtres IIR pour la direction verticale 106
2-11 Récapitulatif des paramètres géométriques des patrons : un exemple 109
2-12 Décalage des résultats face au triple buffering 112
2-13 Optimisation de la gestion des bancs pour le cas de n œuds diadiques 116
2-14 Entrelacement des bancs de mémoire interne pour le C6201 et C6701 117
2-15 Double buffering sur le C6201 et C6701 118
2-16 Définitions des paramètres N_W et N_R 122
2-17 Partitionnement "horizontal" des données 132
2-18 Découpage vertical des données 133
2-19 Découpage multi-buffers et exemple de nœud diadique 135
2-20 Propagation des contraintes de synchronisation 140
2-21 Un exemple de chaîne de traitement 144
2-22 Allure caractéristique des courbes de traitement/transfert sur C8X 149
3-1 Synopsis de l'architecture de la librairie de traitement d'images sur C80 159
3-2 Performances du nœud d'inversion logique 164
3-3 Performances du nœud de convolution 3x3 optimisé 165
3-4 Performances SPMD de la chaîne statistique 166
3-5 Filtrage de Deriche en 2D 172
3-6 Filtres cascades récursifs du 1 er ordre de FGL 174
3-7 Multiplication SIMD 16 bits des PP 176
3-8 Chaînes de détection de contours sur le C80 182
3-9 $N_W > 1$ et multiples itérations du lisseur de FGL 182
3-10 Application verticale du lisseur de FGL 183
3-11 Copie d'écran du résultat des 2 chaînes FGL sur C80 188
3-12 Graphe de dépendance du gradient sur C6X 191

3-13 Résolution des conflits d'unités du graphe de dépendance 193
3-14 Première étape du repliement du graphe de dépendance "Gradient" sur C62 195
3-15 Deuxième étape du repliement du graphe "Gradient" sur C62 197
3-16 Graphe de dépendance du lisseur de 2ème ordre sur C62 199
3-17 Déroulage de boucle et pipeline logiciel du lisseur sur C6X 200
3-18 Estimation des performances brutes sur le C6202 202
4-1 Voisinage d'ordre 2 et cliques associées 210
4-2 Modèle de voisinage et cliques spatio temporelles 215
4-3 Approche de Caplier et al. pour la détection markovienne du mouvement 217
4-4 Regroupement des données au sein des transferts du nœud ICM 219
4-5 Organisation externe des buffer 220
4-6 Principe d'un encodeur M-JPEG original 228
4-7 Structure du prototype d'encodeur 232
4-8 Aperçu de l'organisation des buffers externes 233
4-9 Copié d'écran de l'organisation des buffers 235
4-10 Durée de l'encodage/décodage pour différents facteurs de qualité JPEG 237
4-11 Courbe d'évolution de la taille du flux 238
4-12 Performance des procédés de compression 243
4-13 Synthèse des performances de compression avec différentes valeurs de T et R 245
4-14 Exemple de reconstruction : pourcentage de zone en mouvement identique 248
4-15 Différences seuillées des zones de dégradation 249
4-16 Qualité de reconstruction à taille de flux égale : séquence "ping-pong" 250
4-17 Qualité de reconstruction à taille de flux égale : séquence "Salesman" 251
4-18 Gros plan sur quelques défauts de reconstruction 252

Liste des Tables

1-1 Caractéristiques des coeurs de traitement C80 33
1-2 Bande passante du C80 34
1-3 Ports de l'ALU triadique des PP 37
1-4 Performances des processeurs DSP et RISC en 1999 49
1-5 Consommation, encombrement et prix des processeurs DSP et RISC en 1999 51
1-6 Améliorations de la gestion des entrées/sorties : objectifs et perspectives 80
2-1 Récapitulatif des paramètres d'un buffer 98
2-2 Exemples de patrons internes pour divers nœuds 110
2-3 Rôle des bancs internes lors du triple buffering 111
2-4 Tour de rôle des buffers pour le triple buffering 111
2-5 Temporisation de la présence des données avec le triple buffering 113
2-6 Exemple simple d'une fiche descriptive de nœuds 151
3-1 Nœuds bas-niveau de la librairie de traitement d'images sur C80 161
3-2 Performances de la carte SDB face aux performances maximales du C80 163
3-3 Complexité comparée des filtres de Deriche et de FGL 175
3-4 Performance théorique sans l'impact DMA des traitements sur C80 181
3-5 Patron de données des nœuds de la chaîne FGL 181
3-6 Géométrie des patrons du lisseur vertical 184
3-7 Calcul du patron virtuel de référence pour la deuxième passe de FGL 184
3-8 Paramètres N_W des deux chaînes de FGL 185
3-9 Paramètres beta et gamma des deux chaînes de FGL 186
3-10 Nombre de requêtes DMA des deux chaînes de FGL 186
3-11 Résumé des performances des chaînes FGL pour une image 512^2 à 40 Mhz 187
3-12 Principales opérations C62 : latence et unité fonctionnelle d'exécution 189
3-13 Première étape du repliement du calcul du Gradient sur C62 196
3-14 Deuxième étape du repliement du calcul du Gradient sur C62 197
3-15 Projection du graphe sur les ressources processeurs 201
4-1 Notations pour la détection de mouvement par modélisation markovienne 209
4-2 Les fonctions potentiel du voisinage spatio-temporel 215
4-3 Paramétrage de l'algorithme de détection markovienne du mouvement 216
4-4 Géométrie et complexité des patrons pour la phase de pré-traitement 218
4-5 Géométrie et complexité des patrons pour la phase ICM 220
4-6 Performances de la détection de mouvement MRF sur quelques architectures 224
4-7 Performances de l'encodeur Markov-M-JPEG C80 236
5-1 Les architectures de type VLIW en 2000 259

Introduction

Les systèmes de vision et de perception doivent faire face à un environnement fortement évolutif : conditions d'éclairement variables, bruit aléatoire engendré par les capteurs ou présence de mouvement dans les scènes. Par conséquent, ils nécessitent des algorithmes avancés, robustes face à ces variations. Ils imposent également de sévères contraintes sur la durée des traitements.

De plus, la complexité calculatoire des algorithmes s'amplifie au fil des années alors que parallèlement, les systèmes de vision trouvent des débouchés de plus en plus nombreux grâce à la miniaturisation remarquable des circuits qui accroît leur capacité pour satisfaire le marché des applications embarquées aujourd'hui en pleine expansion.

Devant la demande affichée pour ce type de systèmes et pour faire face à l'évolution de la charge algorithmique, les processeurs utilisés intègrent des ressources de plus en plus nombreuses pour le calcul et la gestion des entrées/sorties. La mise en œuvre de ces composants nécessite la parallélisation des algorithmes afin d'exploiter au mieux les capacités matérielles offertes. Les performances crêtes des processeurs actuels s'évaluent à quelques milliards d'opérations par seconde (<10 GOPS) et offrent ainsi le potentiel permettant de satisfaire les contraintes d'implantation des applications vidéo. Cependant, le revers de la montée en puissance qu'affichent les étonnantes architectures actuelles réside dans la complexité croissante de l'étape de parallélisation. Nous pouvons corroborer cette remarque en citant quelques aspects architecturaux qui expliquent cette difficulté :

- le contrôle MIMD (*Multiple Instructions, Multiple Data*) ou VLIW (*Very Long Instruction Word*)
- l'hétérogénéité des processeurs (flottants/entiers)
- les hiérarchies mémoire (caches, mémoires internes et externes)
- les moyens de communication (entrées-sorties)

La complexité architecturale s'accompagne d'une difficulté accrue de la programmation et de la mise en œuvre des étapes traditionnelles du génie logiciel avec les aspects d'environnement de programmation, de chaînes de compilation, d'outils de debogage et l'analyse des performances. Cette constatation a plusieurs conséquences très importantes pour l'exploitation de ces processeurs :

1. La programmation s'effectue à bas niveau parce qu'elle doit prendre en compte les particularités de l'architecture que ne sont pas en mesure d'exploiter les compilateurs et les optimiseurs de code actuels,
2. Les durées effectives du traitement sont difficilement prévisibles,
3. Les performances effectives sont très inférieures aux performances crêtes.

La difficulté de programmation est renforcée par des contraintes de rendement toujours plus fortes et une durée de vie des processeurs qui diminue sous l'effet de la concurrence.

Dans ce contexte, le cahier des charges de cette thèse CIFRE stipule tout d'abord, en guise d'objectif scientifique, de proposer des méthodes de programmation et d'évaluation des performances qui permettent de dépasser les limitations actuelles de l'exploitation des processeurs de traitement de signal parallèles (aussi communément appellés "DSP" pour *Digital Signal Processors*). Ensuite, il est précisé que ces méthodes doivent être étudiées à travers le développement d'une bibliothèque de traitement d'images destinée à la vision industrielle avec pour architecture cible, le TMS320C80 de Texas Instruments (TI). Enfin, sur la base de cette librairie, le cahier des charges stipule la réalisation de deux applications de références représentatives du domaine du traitement d'images pour la vision industrielle. Soulignons que les deux applications de référence que nous développons dans ce mémoire sont à l'origine issues de recherches conjointes (cf. thèse de L. Lacassagne sur l'algorithmique pour la rétro-vision automobile et le suivi de cibles [1]). Le travail présenté ici vise plus particulièrement les méthodologies logicielles pour la mise en œuvre sur C80 et moins directement sur le nouveau TMS320C62.

Ce mémoire, qui se compose de quatre chapitres, présente notre réponse aux objectifs donnés par le cahier des charges.

Le premier chapitre présente la spécificité des architectures DSP et leur intérêt face aux processeurs RISC (*Reduced Instruction Set Computer*), notre analyse introduisant une comparaison objective sur le plan des performances ainsi qu'une argumentation plus qualitative sur l'adéquation des DSP face aux contraintes des systèmes embarqués. De plus, nous justifions l'utilisation du C80 comme cible principale de nos travaux dans le domaine du traitement d'images. Nous soulignons également la difficulté de programmation des DSP et plus particulièrement celle du C80, avant de présenter plusieurs pistes qui, sur le plan pratique, permettent une simplification de la mise en œuvre logicielle des systèmes DSP complexes.

Ensuite, face à ces pistes que n'adressent pas les outils commerciaux actuels, nous développons tout d'abord un état de l'art comprenant les outils et librairies de traitement d'images pour la programmation spécifique du C80 et présentons les restrictions par rapport aux objectifs précédemment identifiés pour l'aide à la programmation. Nous poursuivons cette analyse avec un état de l'art plus général sur les pratiques académiques concernant l'implantation d'algorithmes sur des architectures complexes. Les limitations actuelles qui sont alors dégagées, nous permettent de conclure ce chapitre par des perspectives de contributions scientifiques visant, d'une part, à dépasser la difficulté spécifique de programmation sur C80 et, d'autre part, à proposer des améliorations ainsi que des réponses nouvelles plus générales aux pratiques de l'état de l'art pour l'aide à l'implantation logicielle. Le choix est alors fait d'explorer principalement l'optimisation de la gestion des flux qui s'appuie plus spécifiquement sur les co-processeurs DMA (*Direct Memory Access*) dans le cas des DSP.

Le deuxième chapitre présente les deux aspects essentiels de la mise en œuvre des algorithmes sur DSP qui sont la programmation des opérateurs de traitement et la gestion séparée des entrées/sorties au moyen des DMA. Nous détaillons tout d'abord quelques techniques d'optimisation génériques qui correspondent à celles jugées les plus

significatives pour les coeurs de traitement actuels et que, par ailleurs, nous utilisons dans les applications développées sur C80 et C60.

Ensuite, en accord avec ces techniques, nous développons une méthodologie originale pour la gestion des flux à travers la programmation automatique et optimisante des co-processeurs de transferts DMA. Notre approche se veut générique, flexible ; elle est conçue pour une amélioration des performances.

Après quelques définitions sur la nature des objets manipulés, nous précisons certains aspects pratiques de mise en œuvre avant d'aborder une présentation plus théorique de l'analyse mathématique sur laquelle s'appuie notre approche. Le formalisme détaillé s'appuie sur la gestion des traitements qui relève du domaine des flots de données synchrones. Enfin, nous développons un modèle de performance pour la prédiction des durées d'exécution d'algorithmes implantés suivant notre méthodologie de gestion des flux.

Le troisième chapitre permet de valider les méthodes de programmation introduites. Pour ce faire, nous présentons tout d'abord la structure de notre librairie de traitement d'images sur C80 qui met en œuvre notre approche originale pour la gestion des flux. De là, nous analysons les performances de quelques chaînes de traitement simples qui s'appuient sur notre librairie et au sein de laquelle nous avons développé un large ensemble d'opérateurs optimisés relevant du bas niveau. Ces exemples nous permettent à la fois de valider certains choix sur la gestion des flux et de vérifier les grandes lignes du modèle de performances proposé.

Ensuite, au moyen de cette librairie C80, nous présentons le résultat de l'implantation d'un algorithme de détection optimale de contours plus complexe (et autour duquel règne une grande effervescence académique au niveau national). Nous présentons la mise en œuvre des techniques d'optimisation pour la programmation des opérateurs, puis nous précisons les principaux calculs impliqués autour de l'utilisation automatique de notre méthodologie de gestion des flux. Par cet exemple, nous validons également plus avant notre modèle de performance. Enfin, pour souligner la généricité des techniques d'optimisation proposées et situer les performances du C80 par rapport à l'architecture du C6X qui lui succède

historiquement, nous présentons une mise en œuvre sur C62 des techniques d'optimisation pour l'implantation des opérateurs de ce même algorithme de détection de contours.

Enfin, le dernier chapitre présente notre deuxième étude de cas illustrant l'utilisation des méthodes de développement détaillées au second chapitre. Il s'agit d'une application plus complexe qui s'appuie sur notre librairie de traitement d'images C80 et qui vise l'introduction d'une technique originale de compression de séquences vidéo sur la base d'un algorithme de détection de mouvement connu. Dans un premier temps, nous présentons cet algorithme qui repose sur une approche markovienne. Ensuite, nous présentons l'implantation de cet algorithme sur C80 et analysons les performances obtenues. Dans un deuxième temps, nous détaillons notre nouveau principe de compression et présentons une technique concurrente issue d'un brevet industriel. Après une présentation de l'implantation de l'ensemble du système sur un prototype fonctionnel s'appuyant à la fois sur une librairie C80 JPEG (*Joint Photographic Expert Group*) existante et sur notre infrastructure logicielle, nous comparons expérimentalement les bénéfices des deux techniques et démontrons l'intérêt de notre approche. Soulignons ici que cette méthode fait actuellement l'objet d'un dépôt de brevet et constitue, après notre méthodologie de gestion des flux par DMA, la deuxième principale contribution scientifique de nos travaux.

1 Intérêt des DSP et état de l'art pour leur mise en œuvre

1.1 Les architectures DSP programmables

Les processeurs parallèles de traitement de signal ont connu un engouement considérable cette dernière décennie et la véritable concurrence que représentent les performances apportées par les plus récentes générations de processeurs RISC dites "grand-public" est sans aucun doute un élément expliquant cette constatation. Ce chapitre présente les principaux DSP du moment et analyse leur complexité de mise en œuvre logicielle. Nous exposons également l'intérêt que présente la réalisation de systèmes à base de DSP et nous expliquons plus particulièrement en quoi l'architecture TMS320C8X constitue une solution de choix pour le traitement d'images embarqué.

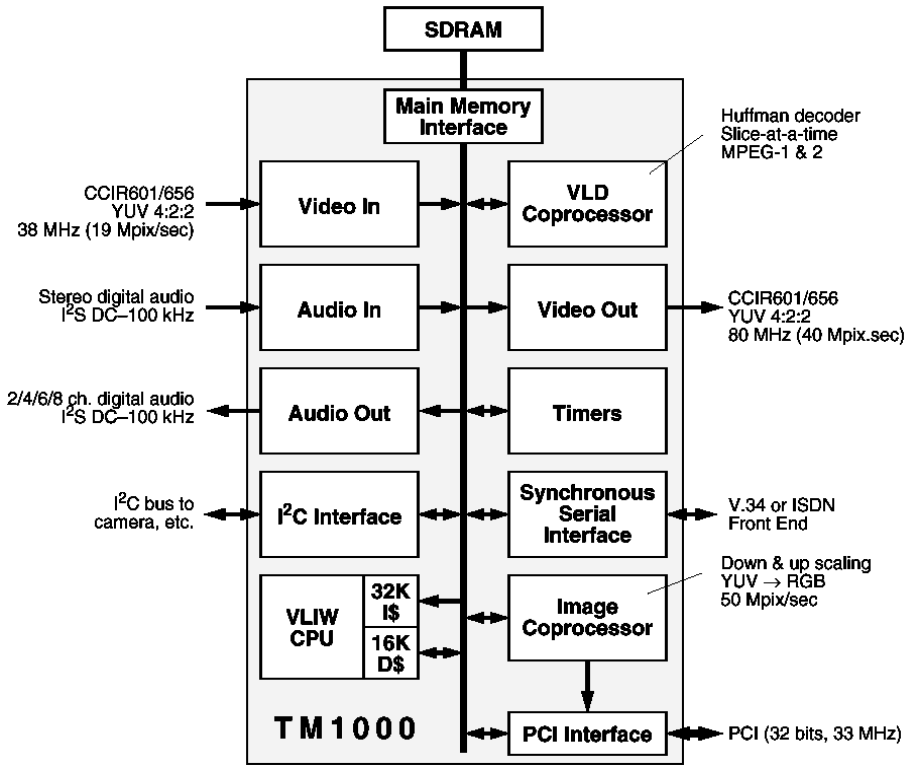
1.1.1 Un aperçu d'architectures DSP parallèles avancées

Pour mieux appréhender la spécificité des DSP (notamment face aux processeurs RISC), nous introduirons tout d'abord, dans cette partie, un éventail d'architectures de DSP programmables parmi les plus performantes du moment. Nous détaillons plus particulièrement l'architecture du TMS320C62 et celle du TMS320C8X qui sont les architectures cibles de notre étude pour l'implantation des algorithmes présentés dans les chapitres suivants.

1.1.1.1 Le Trimedia de Philips

Le TM1300 de Philips (Figure1-1) est le dernier membre de la famille des Trimédias 32 bits avec l'avènement annoncé du TM1400 qui constituera le premier composant de la gamme utilisant une architecture 64 bits (aussi appelée architecture "CPU64").

Figure 1-1. L'architecture du TM1000/TM3000 ou CPU32



Le TM3000 est une architecture VLIW entière et flottante destinée aux applications multimédias. Elle comporte un processeur central et des blocs connexes aux fonctionnalités spécifiques. Nous distinguons notamment des co-processeurs d'entrées/sorties (audio et vidéo), un co-processeur d'images, ainsi qu'un bloc dédié à la décompression MPEG-2 (VLD Coprocessor pour un premier niveau de décodage Huffman des données). Nous trouvons également 3 autres blocs d'E/S (Entrée/Sortie) facilitant l'intégration de ce composant qui sont, un bloc d'interface I^2C pour l'utilisation des caméras vidéo, un bloc d'interface série synchrone pour l'interconnection aux modems et une interface PCI pour permettre de facilement intégrer le Trimédia comme co-processeur dans des systèmes hôtes. Chacun de ces blocs est connecté au bus interne 32 bits qui permet d'accéder à la mémoire dynamique (SDRAM) externe (ils se partagent donc la bande passante).

L'architecture du TM1300 est aujourd'hui cadencée à 166 MHz et intègre un format d'instructions VLIW permettant le lancement conditionné d'un nombre maximum de 5 opérations à destination de 27 unités fonctionnelles pipelinées. Le coeur compte 128 registres et s'appuie notamment sur des opérations complexes pouvant fonctionner suivant le paradigme SIMD (*Single Instruction, Multiple Data*). Certaines de ces opérations correspondent à un nombre important de calculs élémentaires comme pour les opérations *quadavg* et *ume8* qui permettent, respectivement, de sommer les moyennes de 4 couples de 2 valeurs 8 bits - 7 additions, 1 division par 2 - ou le calcul de la somme de différences absolues pour l'estimation du mouvement de la norme MPEG-2 (Motion Pitcure Expert Group) - 7 soustractions/additions, 4 valeurs absolues. Ce type d'opérations permet ainsi d'atteindre la performance crête de 6.5 GOPS sans, par ailleurs, tenir compte des capacités de traitement des autres blocs (le co-processeur d'images permet en effet, par exemple, d'appliquer en parallèle un filtre FIR d'ordre 5 au plus par direction de l'image). Nous trouvons également plusieurs fonctionnalités pré-câblées au niveau des blocs annexes et couramment utilisées dans les systèmes de vision. Nous pouvons citer le sous-échantillonnage et le démultipexage des composantes YUV du domaine de représentation luminance-Chrominance (bloc Video In), le zoom des données et le filtrage FIR des données (Image Coprocessor) ou la conversion YUV/RGB, la superposition (alpha blending) et le masquage des données (bloc Video Out).

Le CPU (*Central Processing Unit*) se compose d'un cache de données distinct d'un cache d'instructions comportant respectivement 16 et 32 Ko. Le cache de données autorise deux accès concurrents et introduit un mécanisme permettant de désynchroniser par rapport au CPU la mise à jour des références externes. Nous trouvons également des mécanismes permettant de verrouiller certaines zones du cache (pour maintenir la présence de données critiques). Enfin, notons que le format des instructions est compressé afin d'augmenter la capacité du cache d'instructions.

Les blocs co-processeurs apparaissent comme des contrôleurs DMA (à deux dimensions pour les blocs vidéo qui permettent, par exemple, de dé-multipexer des composantes YUV) et offrent ainsi la possibilité de dé-synchroniser les transferts de données du séquencement du coeur CPU. Les données sont partagées au niveau de la mémoire externe dans le sens où

le résultat des traitements des blocs connexes transite nécessairement par la mémoire externe avant que le coeur du processeur ne puisse notamment y accéder (il n'y a pas de lien de communication interne entre les blocs). Des mécanismes d'arbitrage sont présents pour régir les nombreuses contentions d'accès que suscite ce mode de mise en œuvre.

La programmation s'effectue en C ou à l'aide d'un pré-processeur (C) de C++. Un optimiseur de code permet d'ordonnancer statiquement les opérations des instructions VLIW alors que des macro-fonctions (ou "intrinsèques") permettent d'exploiter les opérations multimedia spécifiques depuis le langage haut niveau.

Pour conclure, l'architecture Trimédia est complexe ; elle offre de très nombreuses fonctionnalités qui la définissent comme un véritable système autonome embarquable pour les applications vidéo nécessitant des capacités de calcul importantes.

1.1.1.2 Le Sharc 21160 et Tiger-Sharc d'Analog Devices

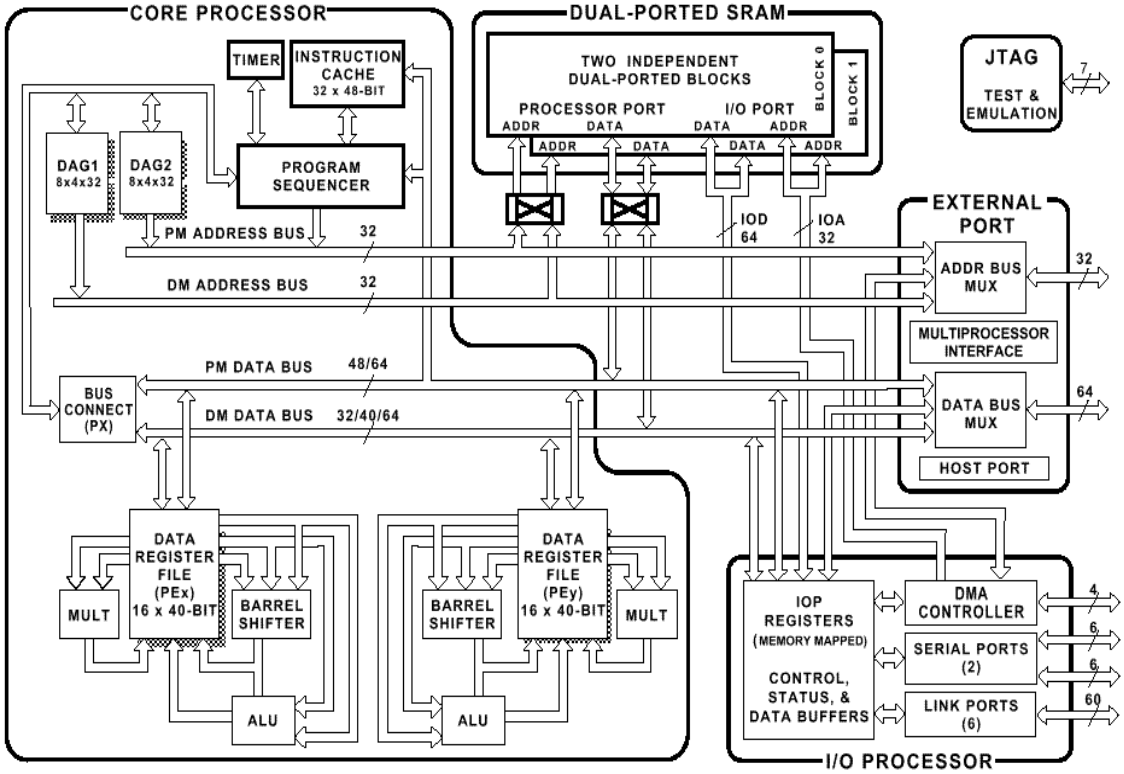
Les architectures Sharc, acronyme de *Super Harvard ARchitecture Computer*, constituent des architectures prisées pour les systèmes multi-processeurs parallèles à base de DSP (sur ce plan, nous pouvons également citer le TMS320C4X de Texas). Cette constatation traduit la présence de mécanismes performants et parfaitement intégrés pour l'inter-opérabilité de ce composant. Le Sharc 2106X représente la première génération des membres de cette famille. Récemment, une nouvelle architecture SIMD, compatible du point de vue du code avec la première, est apparue avec le Sharc 21160 (ou Sharc II). Présentons maintenant les principales caractéristiques ainsi qu'un bref aperçu de la troisième génération introduite avec l'architecture SIMD/VLIW du Tiger Sharc.

1.1.1.2.1 Le Sharc 21160

Le 21160 (Figure 1-2) permet d'atteindre des performances de 600 MFLOPS crête et de 400 MFLOPS en flux continu[1]. Il complète l'architecture scalaire du 2016X et double les unités fonctionnelles de traitement pour le mode SIMD (l'architecture devient symétrique).

1. Les performances en flux continu intègrent l'impact de la congestion des bus PM et DM détaillé plus loin.

Figure 1-2. L'architecture du Sharc 21160



Dans le même temps, la fréquence est multipliée par un facteur 2,5 par rapport au 21060 (40MHz → 100MHz), ce qui explique que les performances soient globalement 5 fois supérieures à celles de la première génération. Le pipeline d'instructions comporte 3 étages pour une latence d'instruction constante de 1 cycle (sauf pour les branchements). Les instructions peuvent regrouper plusieurs opérations dans le même cycle. Ainsi, pour chaque groupe d'unités fonctionnelles du mode SIMD, les ALU (*Arithmetic and Logic Unit*) supportent une multiplication, un calcul arithmétique du type addition/soustraction et une opération de décalage registre (ou une double addition/soustraction avec une multiplication). Parallèlement à ces calculs concurrents, les deux unités d'adressage mémoire (DAG1/2, *Direct Adress Generator*) autorisent le chargement simultané de deux opérandes de la mémoire interne.

Le chemin de données de l'un des opérandes partage le bus programme (ou bus PM) ce qui explique que ce double chargement n'est possible, sans pénalité de cycles, que lorsque le code opératoire de l'instruction associée aux deux chargements se trouve dans le cache interne de 32 instructions (séparé de la mémoire statique interne). Il s'agit d'un cache

sélectif associatif à 2 voies qui ne mémorise que les instructions pour lesquelles des contentions d'accès au bus PM existent. En l'absence de contention (chargement d'un seul opérande), le code opératoire est directement chargé depuis la mémoire statique interne avec un débit d'un cycle par instruction.

Pour la partie traitement, l'architecture du Sharc supporte un contrôleur de boucle matériel qui permet d'optimiser la gestion des boucles (en terme de nombre d'instructions nécessaires et de rapidité d'exécution). Elle supporte également des constructions de type "IF THEN ELSE" câblées. Par ailleurs, au niveau de l'ALU, certaines instructions autorisent un nombre élevé d'opérations. Le Sharc s'appuie ainsi sur des "macros" instructions comme pour l'instruction Ru=Rv+Rw ×Rx, Ry=(Ry+Rz)/2 qui dénombre 4 opérations (ou ',' schématise une exécution concurrente et R des registres). Ce type d'instruction permet une implantation performante des algorithmes de type convolution. Nous pouvons également citer les instructions de type Min(Rx,Ry) pour le seuillage ou Rx OR LSHIFT Ry BY Rz pour la génération de masques binaires (avec LSHIFT décalage binaire vers la gauche). Dans le mode SIMD, l'ensemble des opérations effectuées par ces macros-instructions est bien sûr doublé ce qui contribue à un nombre d'opérations parallèles important. De plus, au niveau des générateurs d'adresses (DAG1/DAG2), la pré/ post-incrémentation automatique avec modification des pointeurs est supportée ainsi que l'adressage circulaire des buffers. Le langage assembleur utilise les expressions arithmétiques et logiques qui définissent les instructions (plutôt que des mnémoniques) ce qui facilite le développement. Nous pouvons alors parler d'assembleur algébrique comme pour le TMS320C80.

Outre sa puissance, le succès de l'architecture Sharc provient notamment des mécanismes intrinsèques ("glue-less") permettant la réalisation de systèmes parallèles avec ce composant. Les modèles de mémoire partagée et distribuée sont simultanément supportés et l'architecture autorise des grappes (clusters) d'un maximum de 6 processeurs. Le modèle peut, par ailleurs, être étendu à une architecture multi-grappes, le bus dit "hôte" servant généralement de canal de communication inter-grappes.

Pour le modèle de mémoire partagée, chaque processeur de la grappe perçoit l'ensemble des mémoires internes des Sharcs comme un unique espace d'adressage unifié. Cette

unification permet, notamment, de diminuer la quantité de mémoire externe nécessaire au système. Le DMA d'un Sharc peut permettre la lecture/écriture de blocs de données dans la mémoire interne d'un autre Sharc à l'aide d'un unique bus partagé. Ce bus supporte des connections point à point entre 2 processeurs ainsi qu'un mode de diffusion à l'ensemble des processeurs de la grappe pour les opérations d'écriture (mode "*broadcast-write*"). Des mécanismes de préemption et de gestion de niveaux de priorité (fixes ou "round-robin") sont supportés nativement par chaque processeur pour le partage effectif et non-bloquant du bus. Par ailleurs, la mémoire interne de chaque processeur offre deux ports d'accès associés à deux bancs de mémoire interne distincts (pour un total de 4 Mbits), ce qui autorise un accès concurrent et sans contention entre le bus DM ou PM et le bus d'E/S associé au co-processeur DMA.

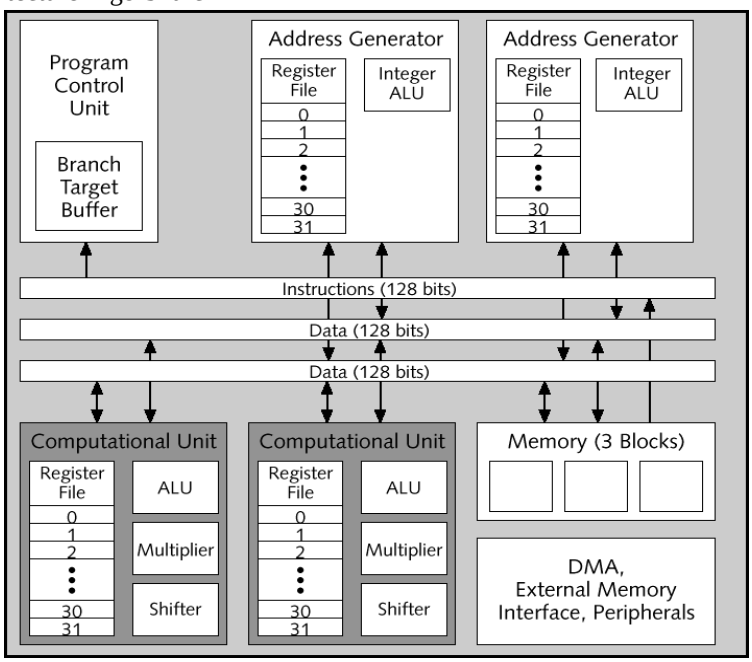
Pour limiter l'éventuelle congestion du bus partagé, un modèle de mémoire distribuée est également supporté avec les liens de communication (link ports) qui sont au nombre de 6. Chaque lien autorise un taux de transfert de 100 Mo/s (à 100 MHz) pour une largeur de bus de 8 bits. Ces mécanismes offrent un débit inférieur aux 528 Mo/s (à 100 MHz) qu'autorise le bus partagé (64 bits) et sont, par ailleurs, plus longs à initialiser. En terme de périphériques, le 21160 compte également 2 liens série à 50 Mbits/s (100 MHz) ainsi qu'un lien de communication avec une machine hôte (host port).

Pour l'ensemble des supports de communication (bus, liens, ports série), le DMA peut être utilisé pour le transfert de blocs de données. Il supporte l'adressage bi-dimensionnel et le chaînage des requêtes de transferts. Par ailleurs, le DMA autorise des transferts concurrents via 14 canaux distincts (4 pour les ports séries, 6 pour les liens et 4 pour le bus partagé). Ces canaux se partagent la bande passante des différents moyens de communication ainsi que celle du bus d'entrée/sortie interne. Là encore, des mécanismes de gestion de niveaux de priorités (fixes ou round-robin) ainsi que le découplage interne/externe des étapes élémentaires des transferts, permettent des taux de transferts soutenus ainsi qu'une bonne inter-opérabilité et scalabilité de l'architecture (ici la commutation d'accès au bus d'E/S interne entre les différents canaux intervient à chaque cycle et n'induit pas de contention).

1.1.1.2.2 Le Tiger Sharc

Le TigerSharc (Figure 1-3) apparaît en 1999 et introduit une nouvelle famille d'architectures, le code n'étant plus compatible avec les premières génération de Sharc [2]. Il atteint des performances crêtes de 1,5 GFLOPS et de 8 GOPS en arithmétique SIMD 8 bits.

Figure 1-3. L'architecture TigerSharc



Il s'agit d'une architecture symétrique SIMD/VLIW de 2×2 unités fonctionnelles (jusqu'à quatre opérations parallèles) initialement cadencée à 250 MHz et pour laquelle les huit étages du pipeline d'instructions sont totalement inter-bloqués. L'idée du constructeur est de garantir les dépendances registres du code VLIW au niveau même de l'architecture. L'ordonnancement des instructions est déporté au niveau du compilateur mais, à l'inverse de ce que nous rencontrons sur l'architecture C6X ou celle du Trimedia, cet ordonnancement statique des opérations favorise l'optimisation du code. En revanche, elle n'est pas nécessaire à la cohérence de l'exécution. Pour le TigerSharc, la latence des instructions est constante et le séquenceur s'assure que l'ensemble des opérations d'une instruction VLIW (appelé ligne d'instructions) termine au même instant[1].

1. Pour la génération d'adresse, la latence des opérations est de 1 cycle alors qu'elle est de 2 cycles pour les opérations de l'ALU ainsi que celles l'accès à la mémoire interne.

De part ces spécificités, les architectes d'Analog Devices préfèrent parler d'architecture superscalaire statique comme pour le futur Merced d'Intel plutôt que d'architecture VLIW où le pipeline n'est généralement pas interbloqué. En outre, le TigerSharc emprunte le mécanisme de prédiction de branchement que nous rencontrons dans la plupart des architectures RISC modernes et souligne le processus de fusion des coeurs RISC et DSP. Le constructeur souligne une simplicité de mise en œuvre ainsi qu'une bonne réactivité aux interruptions (argument qui est, en effet, pénalisant pour la gamme C6X[1]). Les très bonnes performances affichées au niveau du coeur s'appuient sur deux unités ALU, chacune capable d'exécuter huit opérations SIMD 8 bits en parallèle (ou quatre en 16 bits ou deux en 32 bits entier ou flottant). Le découplage des capacités SIMD autorise une grande souplesse dans l'implantation des algorithmes. Dans [2], l'auteur explique comment ce découplage autorise un code efficace pour l'implantation des filtres FIR.

L'architecture compte par ailleurs un nombre important de registres 64 bits (128) et supporte l'exécution conditionnelle de chaque opération du code VLIW. Pour alimenter cette puissance de calcul en terme de flots de données, l'architecture comporte trois bus de 128 bits de large associés à trois bancs de mémoire interne distincts. Deux accès concurrents 128 bits aux opérandes sont ainsi autorisés par cycle, pour une bande passante interne totale de 12 Go/s. Cette bande passante est encore augmentée avec la possibilité de diffuser le chargement de données aux unités ALU (mode " *broadcast*" où les mêmes données sont présentées aux 2 unités de traitement). Ces mécanismes, ainsi que l'ensemble des instructions arithmétiques et logiques s'appuient, au niveau de l'assembleur, sur une programmation algébrique.

Concernant les périphériques, l'architecture supporte un grand nombre de biais de communication tels que ceux introduits avec les premières générations de Sharc pour la réalisation de systèmes multi-processeurs performants. Les canaux DMA du TigerSharc supportent également les transferts multi-dimensionnels.

---

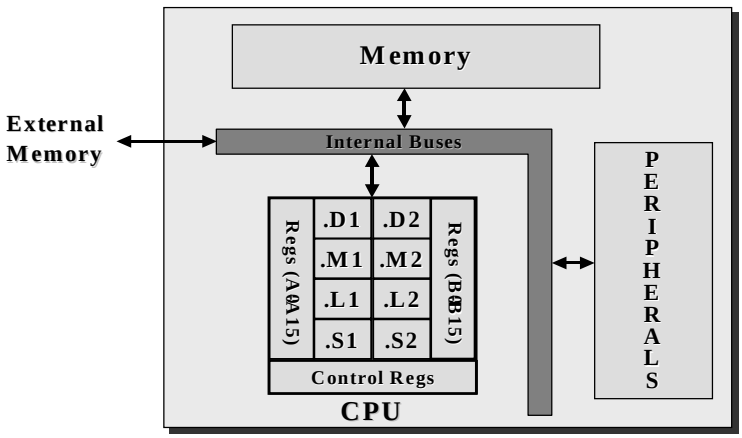
1. L'architecture ne peut traiter les interruptions lorsqu'une opération de branchement est en attente. Or, la technique essentielle d'optimisation dite du pipeline logiciel (chapitre 2), implique, généralement, que des opérations de branchement soient pendante durant toute l'exécution d'une boucle algorithmique. Dans ce contexte, la réactivité du C6X aux interruptions est bien sûr mise en cause.

1.1.1.3 Le TMS320C6X et TMS320C8X de Texas Instruments

1.1.1.3.1 Le TMS320C6X

Ce processeur apparaît en 1998 comme le successeur du TMS320C80. Il s'agit d'un mono-processeur VLIW large (256 bits) qui autorise jusqu'à 8 instructions en parallèle par cycle. A 250 MHz, l'architecture atteint ainsi 2 GOPS (Figure1-4).

Figure 1-4. Blocs fonctionnels du TMS320C6X



Le coeur s'appuie sur un unique super-pipeline de 12 étages pour la version entière du processeur - le C6202 - et de 16 étages pour la version flottante avec le C6701. A respectivement 250 et 166 MHz, la durée d'exécution de chaque étages est de 4 et 6 ns. Les unités fonctionnelles (arithmétiques et d'accès à la mémoire interne) sont au nombre de 2×4, l'architecture étant symétrique. Le jeu d'instructions est élémentaire et les opérations atomiques se regroupent selon les 4 unités fonctionnelles VLIW de l'architecture qui autorisent leur exécution (unités *.D*/*.M*/*.L*/*.S*). Chaque côté de l'architecture permet un accès mémoire (avec post-incrémentation du registre pointeur et support de l'adressage circulaire), une multiplication (16 bits) et deux opérations arithmétiques (32/40 bits) ou logiques. Chaque exécution d'opération peut, par ailleurs, être individuellement conditionnée par la valeur binaire[1] de registres.

1. Exécution si zéro ou pas zéro.

Le jeu d'instructions du C67 est un sur-ensemble de celui du C62 et apporte l'arithmétique flottante en précision simple et double (32/64 bits). Par ailleurs, l'architecture supporte l'addition et la soustraction entière SIMD sur 16 bits, ce qui permet ainsi d'atteindre un maximum de 10 opérations par cycle. Chaque côté de l'architecture inclut un banc de 16 registres 32 bits pour un total de 32 registres de données ou d'adresses. Le croisement ("crosspath") des opérandes registres d'opérations VLIW entre ces deux bancs distincts est possible (sans le recours à une opération de type "move" et sans pénalité de cycle) mais leur nombre est limité.

Si le pipeline est bien unique, la durée des instructions dans la phase d'exécution du pipeline varie et l'ordonnancement des instructions doit être figé au stade de la compilation (comme pour le TM3000). Par ailleurs, l'architecture ne garantit pas l'interdépendance des résultats asynchrones des opérations VLIW (il n'y a pas de mécanisme de "*scoreboarding*"). L'ordonnancement cohérent des opérations (intégrant la latence des instructions) est ainsi entièrement déporté au niveau du compilateur. De plus, malgré une largeur d'instructions fixe, l'architecture permet l'exécution partiellement parallèle ou totalement séquentielle des opérations qui composent le code VLIW (technologie "VelociTI"). Cet aspect permet la diminution de la taille du code et réduit du même coup l'impact de la gestion du cache d'instructions (d'autant que le bus d'instructions externe n'étant que de 32 bits, le chargement des instructions 256 bits est pénalisé).

Concernant les périphériques, le C6202 intègre 2 Mbits de mémoire programme (pouvant ou non être gérés par un cache matériel) et 1 Mbit de mémoire de données que l'utilisateur doit exploiter avec le processeur DMA multi-dimensionnel (3 dimensions). Ces canaux sont au nombre de 4 et partagent une bande passante de 1 Go/s. L'architecture intègre également 3 ports séries bi-directionnels.

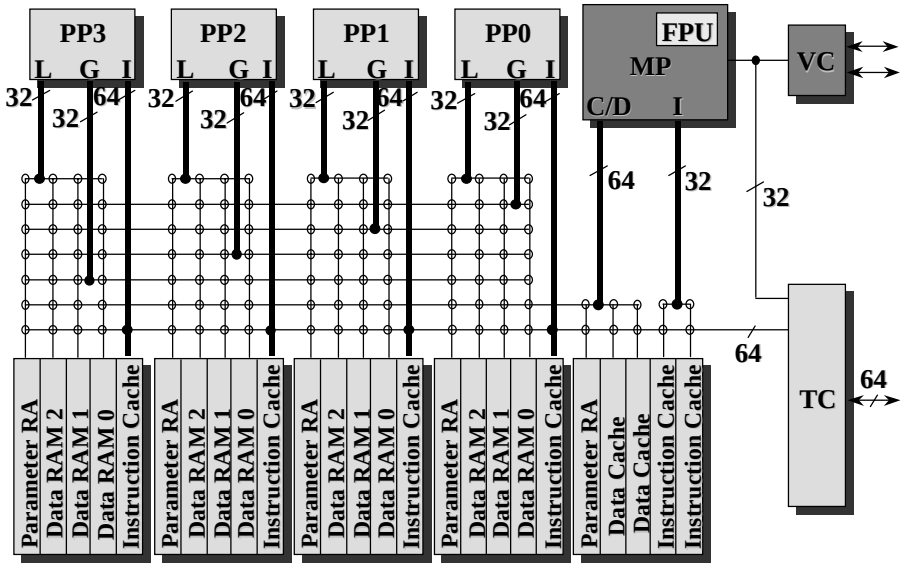
1.1.1.3.2 Le TMS320C8X

L'architecture TMS320C80 (C80) étant la cible privilégiée de ce mémoire, nous présentons ci-dessous ses caractéristiques majeures après avoir donné un aperçu des principaux blocs fonctionnels du composant. Nous montrons également en quoi cette architecture apparaît comme dédiée au traitement d'images.

Aperçu des blocs fonctionnels

Le TMS320C80 ou MVP (*Multimedia Video Processor*) de Texas Instruments (Figure1-5) est apparu en 1994 et correspond encore aujourd'hui à l'un des DSP commercialisés les plus performants du moment (2,4 GOPS à 60 MHz). Il s'agit d'une architecture multi-processeurs hétérogène de type MIMD à mémoire partagée.

Figure 1-5. Blocs fonctionnels du TMS320C80[1]



La puce regroupe 4 processeurs VLIW entiers 32 bits, ou PP pour *Parallel Processor*, et un processeur RISC flottant 64 bits, le MP, pour *Master Processor*. Un crossbar interne permet

1. L/G: port d'accès local/global, I port d'instructions, C/D port d'accès aux données (via le Cache ou Directement à la mémoire externe)

aux différents processeurs d'accéder à l'ensemble de la mémoire statique interne avec un minimum de contention. Tous les processeurs disposent d'un cache d'instructions, le MP bénéficiant en plus d'un cache de données. Un contrôleur de DMA particulièrement avancé (ou TC pour *Transfer Controller*) gère l'ensemble des requêtes de transfert de données et d'instructions entre la mémoire statique interne et la mémoire externe. L'architecture est particulièrement bien adaptée à l'imagerie bas et moyen niveau grâce aux possibilités offertes par le TC, au coeur de calcul des PP (ALU) qui est axé sur le traitement d'images, et à la conception générale de l'architecture, qui incite à un partitionnement de type ferme de processeurs (le MP étant le maître). Le processeur dispose, par ailleurs, d'un contrôleur vidéo (le VC) capable de générer l'ensemble des signaux nécessaires au pilotage d'un périphérique d'affichage ou de capture vidéo (ou les deux simultanément). Il existe une version allégée de l'architecture, le C82, qui n'intègre pas de VC et ne compte que 2 PP. Chaque PP dispose en revanche de plus de mémoire interne (taille du cache d'instructions et de la mémoire de données).

Table 1-1. Caractéristiques des coeurs de traitement C80

	MP (RISC)	PP (VLIW)
Performances crêtes à 60 MHz	120 MFLOPS 60 MIPS	540 MOPS par PP
Taille des entiers	32 bits	32 bits
Taille des flottants	64 bits	-
Taille du cache d'instructions	4Ko	2Ko (4Ko pour le C82) par PP
Taille du cache de données	4Ko	-
Taille mémoire de données	-	3×2Ko (2×4Ko pour le C82) par PP
Taille des instructions	32 bits	64/96 bits
Hauteur du pipeline d'instructions[a]	3 (FEA)	3 (FAE)
Hauteur des pipelines flottants	4 pour l'addition 3 pour ×	-
Nbr. de registres d'adresse+de données	35	17+8
Nbr. de modes d'adressage	3	12

a. FEA/FAE: 3 étages du pipeline: "Fetch"/"Access"/"Execute".

Table 1-2. Bande passante du C80

	Crossbar	TC (DMA)
Performance à 60 MHz	2,2 Go/s pour les instructions 2,9 Go/s pour les données	480 Mo/s
Nombre de modes d'adressage	-	46

Les deux tableaux précédents résument par quelques chiffres clefs les caractéristiques de cette architecture.

L'architecture des PP

Les performances du composant reposent largement sur l'architecture des PP et notamment sur les possibilités de calculs parallèles offertes par l'ALU. S'agissant d'un coeur VLIW, l'ordonnancement est statique et chaque instruction autorise l'utilisation d'un maximum de quatre unités fonctionnelles en parallèle (par PP) sur la base d'un unique pipeline de trois étages. Les instructions VLIW de longueur fixe, autorisent par cycle une multiplication, un ensemble de calculs au sein de l'ALU et deux accès mémoires concurrents (avec pré ou post incrémentation des registres pointeurs). Ces accès s'exécutent sans contention lorsqu'ils concernent des bancs de mémoires distincts et qu'il n'y a pas, par ailleurs, concurrence d'accès avec un autre acteur de l'architecture (TC/MP/autre PP). Dans ce dernier cas de contention, le crossbar applique un algorithme d'ordonnancement à priorité tournante ("round-robin") qui intervient lui-même après un mécanisme de gestion du niveau de priorité des accès en attente.

En outre, l'architecture autorise l'arithmétique SIMD que nous pouvons dynamiquement commuter à chaque instruction sans diminution de performance. Au niveau de l'ALU, les registres 32 bits peuvent ainsi être vus comme contenant quatre pixels 8 bits distincts pour l'addition, la soustraction, les décalages et le masquage des données. La multiplication offre également un mode SIMD autorisant deux multiplications 8 bits en parallèle.

L'exécution des opérations VLIW peut, par ailleurs, être conditionnée suivant le registre de statut arithmétique (exécution si le précédent résultat est >, =, <, < 0, etc), mais au prix d'un ensemble de restrictions sur le nombre d'opérations VLIW en parallèle.

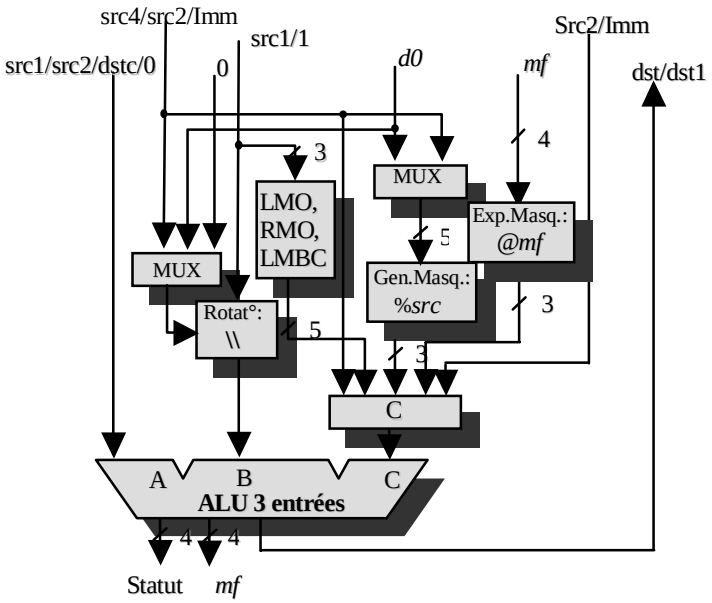
L'ALU des PP est une unité de calcul entière 32 bits triadique (3 ports) comme le montre la Figure 1-6 dont la formulation générale peut se résumer à l'équation arithmétique/ logique suivante :

Equation 1-1. Equation générale de l'ALU des P

$$dst \leftarrow A \,\&\, f_1(B, C) +/- f_2(B, C)$$

où *A,B,C* représentent les ports d'entrée de l'ALU, "&" le "et" logique, f_1 et f_2 constituent des opérations booléennes indépendantes sur les ports *B* et *C*.

Figure 1-6. L'ALU des PP



Ces deux derniers ports peuvent, par ailleurs, avoir comme entrée le résultat de pré-traitements de certains opérandes. Ces pré-traitements ont lieu dans le même cycle et autorisent des mécanismes de détection de bits (opérations de type RMO pour *Right Most One*) ou un décalage registre (opération "\\") au niveau du port B. Le port *C* peut, quant à lui, servir à la création d'un masque de bits, soit par génération directe suivant une longueur précisée (*%src*), soit par expansion (*@mf*). Nous pouvons donner une définition mathématique de ces deux types d'opérations. Avec "|" représentant le "ou" logique, nous écrivons :

Equation 1-2. Mécanismes de génération de masque

$$C = \left\{ \begin{array}{l} \%src = 2^{src} - 1 \\ \left(@mf = \left\{ \begin{array}{l} @mf_0 \leftarrow 0 \\ @mf_{i,\, i:1..} \leftarrow @mf_{i-1} \quad (([mf] \times 0xFF) \setminus\setminus (\quad \times (\quad -8\ )) \end{array} \right. \right) \end{array} \right.$$

En d'autres termes, la première équation permet de générer un masque d'une longueur de bits spécifiée alors que la définition récursive de l'expansion traduit la possibilité de générer un jeu de 4 masques 8 bits regroupés au sein du registre *@mf*. Ainsi, sur la base des quatre bits de poids faible du registre *mf* (*mf[num_bit]*), l'expansion met à jour le registre de masques intermédiaire "@mf" qui permet la sélection de pixels d'intérêts. Par exemple, avec *mf[1..4]=1011* en binaire, *@mf* donne 0xFF00FFFF en hexadécimal. Si le registre de donnée d1 a pour valeur 0xAABBCCDD, d1&*@mf* produit alors 0xAA00CCDD.

Ce mécanisme s'intègre parfaitement aux capacités SIMD de l'architecture, le registre *mf* pouvant, par ailleurs, être directement initialisé en fonction des résultats de l'arithmétique SIMD (il s'agit en fait du registre de statut multiple ("multiple flag") qui mémorise, pour chaque sous-opération SIMD parallèle, un des bits de statut comme le bit de détection du résultat à zéro). Nous notons que l'opération *%src* supporte également un traitement SIMD. Dans le cas 8 bits, nous aurions ainsi %3=0x07070707.

Globalement, le coeur autorise ainsi plus de sept opérations de calcul par cycle et par PP (2 multiplications, 4 additions et une opération de masquage/décalage). Le tableau récapitulatif qui suit donne un aperçu des combinaisons d'opérations supportées par l'ALU et traduit ainsi sa puissance fonctionnelle. Nous comptons plus de 4.000 combinaisons d'opérations arithmétiques et/ou logiques. Ce constat explique, à lui seul, que le langage assembleur mette en avant une programmation sous forme d'expressions algébriques qui emprunte la souplesse du formalisme des expressions du langage *C*. Ceci permet de s'affranchir d'une longue liste de mnémoniques spécifiques à chaque type de calcul et diminue la complexité de programmation.

Table 1-3. Ports de l'ALU triadique des PP[1]

A	B	C
Src2	Src1	@mf
Dstc	Src1\\d0	Src2
Dstc	Src1	%Src2
Dstc	Src1\\Src2	%Src2
Src2	Src1\\d0	%d0
Src2	Src1\\d0	@mf
Dstc	Src1	Src2
Src1	1\\Src2	Src2

Pour finir, il convient de souligner que la gestion des boucles est automatisée, à la manière du Sharc, avec le support matériel d'un maximum de trois niveaux d'imbrication. Ce support matériel se traduit par un ensemble d'instructions assembleur spécifiques qui permettent d'initialiser un compteur ainsi qu'une adresse de début et de fin de boucle. La gestion du compteur ainsi que la rupture du séquencement des instructions sont alors assurées d'une manière transparente au profit d'un coeur de boucle simplifié et plus performant.

Pour conclure sur les principaux dispositifs de traitements associés aux PP, nous pouvons souligner l'adéquation de l'architecture au traitement d'images bas niveau, grâce au support des calculs SIMD 8/16 bits combinés à de puissantes possibilités de masquage. La présence d'opérations spécifiques comme pour la détection de bits permet également d'optimiser l'étape du codage d'Huffman pour les algorithmes de compression MPEG et JPEG. Dans le même sens, des extensions aux mécanismes de multiplication favorisent le calcul rapide et précis de la transformée en cosinus discret (DCT pour *Discrete Cosine Transform*). Soulignons enfin le support optimisé des mécanismes de combinaisons d'images avec la structure triadique même de l'ALU qui permet la prise en compte simultanée de deux plans images et d'un plan de masquage pour l'incrustation sélective de pixels. Une telle approche permet ainsi une intégration performante de la notion de lutins (ou "*sprites*") pour la norme MPEG-4.

1. Ou *Src1, Src2, Dstc, d0* constituent des ensembles de registres de données (*dx*) autorisés pour les différentes classes d'opérations.

Le TC ou contrôleur de transfert

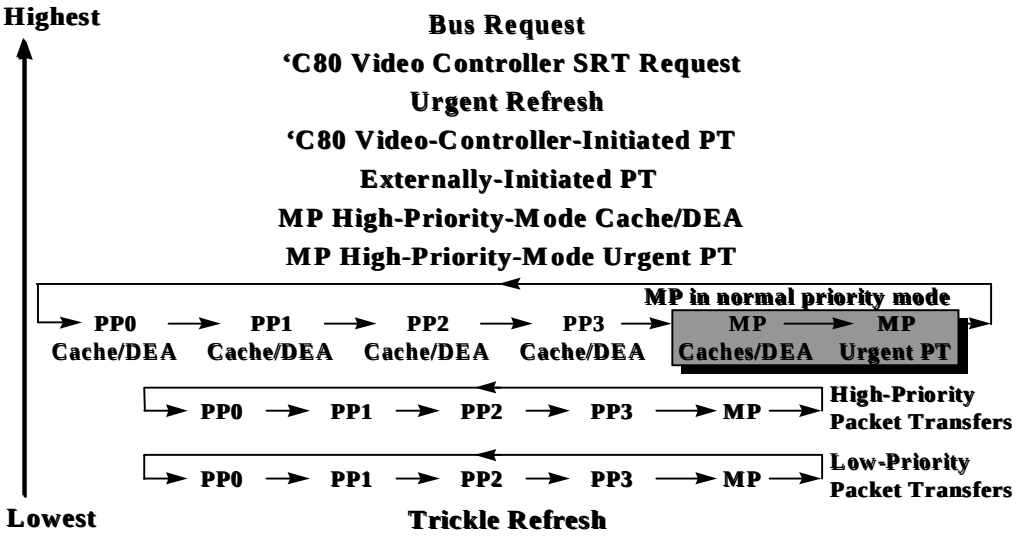
Le TC (*Transfer Controller)* est sans aucun doute le deuxième élément original de l'architecture C80. Il souligne, à son tour, l'adéquation du composant au domaine du traitement d'images. D'un point de vue fonctionnel, le TC constitue d'abord l'unique interface entre la mémoire statique interne associée au crossbar et la mémoire externe. Il s'agit d'un co-processeur DMA qui répond aux requêtes de transfert émanant du MP, des PP et éventuellement de périphériques externes. Sa bande passante maximale est de 480 Moctet/s à 60 MHz, débit effectif dépendant du type de mémoire interfacée au C8X et de la nature du transfert. Nous pouvons distinguer deux catégories de requêtes, ou PT (*Packet Transfer*) : les requêtes matérielles, implicites, et les requêtes logicielles, explicites.

En générant des requêtes matérielles, le TC prend en charge l'ensemble des transferts liés à la gestion des caches d'instructions des PP et du MP. Pour le MP, cette gestion s'étend au cache de données, les PP n'intégrant pas un tel cache. En revanche, une zone de mémoire comportant trois bancs de 2 Ko (pour le C80) est associée à chacun. Il appartient alors au programmeur de gérer ces bancs de manière logicielle en s'appuyant sur les transferts de blocs qu'initie le DMA. Cette gestion passe par le paramétrage manuel des PT pour lesquels la source et la destination du transfert peuvent indifféremment correspondre à une zone de mémoire interne ou externe. Un ensemble de mécanismes de scrutation ou d'interruption permet au programme de connaître l'état des transferts et de lancer ainsi des traitements asynchrones.

Par ailleurs, s'agissant d'une ressource partagée, des mécanismes poussés gèrent les niveaux de priorités associés aux requêtes logicielles et matérielles pendantes avec, comme pour le crossbar, un ordonnancement "*round-robin*" des requêtes concurrentes ayant un même niveau de priorité (cas typique des requêtes logicielles issues des PP). Ce principe est illustré avec la Figure1-7. Ces mécanismes permettent notamment de suspendre l'exécution d'une requête en cours moins prioritaire lorsque l'attente d'une requête pendante plus prioritaire dépasse un seuil critique paramétrable (nous pouvons citer l'exemple des cycles de rafraîchissement de mémoire dynamique ou "*refresh cycles*"). Ces

mécanismes permettent ainsi de réguler de manière cohérente les flux de l'architecture et préviennent les inter-blocages.

Figure 1-7. Niveaux de priorité du DMA C8X

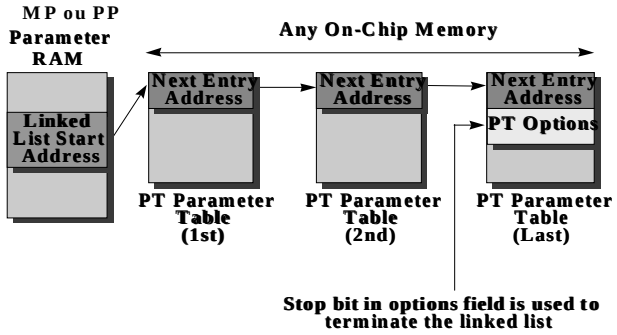


Paramétrage des requêtes de transfert

Pour comprendre en quoi l'architecture du TC est particulièrement innovante et en quoi elle répond aux besoins des algorithmes de vision, il convient d'appréhender les mécanismes de transfert logiciel multi-dimensionnels associés au paramétrage des PT.

Ces PT sont tout d'abord organisés en liste chaînée (Figure1-8) ce qui autorise la soumission d'une série de requêtes et minimise ainsi l'impact d'une gestion individuelle du lancement des PT (possibilité également offerte avec les DMA des processeurs Sharc 20160 et 21160). De plus, pour traiter le cas du lancement consécutif d'une même série de requêtes, la liste des PT peut s'organiser en liste chaînée circulaire de manière à ce qu'au terme de l'exécution de la chaîne de PT, le TC se charge de la ré-initialisation automatique du pointeur de tête de liste. Ces mécanismes apparaissent comme un premier niveau d'optimisation globale. Par la suite, nous présenterons d'autres mécanismes automatisant la gestion des paramètres PT à un niveau qui sera donc local.

Figure 1-8. Liste chaînée des requêtes de transfert (PT)



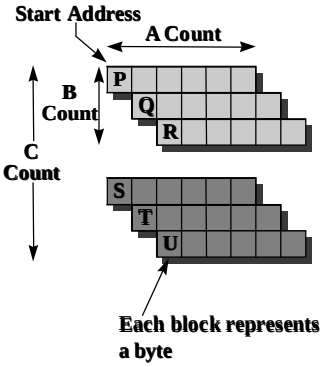
La nature des paramètres de chaque PT décrit la manière dont le TC parcourt les données de la source et de la destination d'un transfert (Fig u re1-9). Cette description offre un large éventail de possibilités (plus de 40 modes de transfert) et autorise deux schémas de parcours de la mémoire indépendants pour la source et la destination. Cette description s'appuie, par ailleurs, sur une organisation tri-dimensionnelle des données.

Figure 1-9. Structure d'un PT

Next Entry Address		PT Options	
Src Start Addres		Dst Start Address	
Src B Count	Src A Count	Dst B Count	Dst A Count
Src C Count		Dst C Count	
Src B Pitc		Dst B Pitch	
Src C Pitc		Dst C Pitch	
Reserve			
Reserve			

Ainsi, la première dimension permet de spécifier la taille d'une ligne de données définie comme une suite d'octets consécutifs en mémoire (il s'agit de la dimension *A*). Les dimensions 2 et 3 (respectivement *B* et *C*) détaillent, quant à elles, la taille de blocs que nous définissons comme regroupant un nombre spécifié de lignes, et, de manière récurrente pour la troisième dimension, la taille de paquets regroupant un nombre spécifié de blocs (ou "patch" selon la terminologie du constructeur). Ce regroupement des données prend tout son sens lorsque nous ajoutons que pour les dimensions *B* et *C*, les paramètres précisent également les distances (en octets) qui séparent deux lignes consécutives (il s'agit du pas *B* ou "B pitch") et celles séparant deux blocs consécutifs (pas *C*). Nous illustrons ce découpage avec la Figure1-10.

Figure 1-10. Paramétrage des 3 dimensions



Paramètre	*Description*
Start Addre	adresse source ou dest. (mémoire int. ou ext.)
A Count	nombre d'octects pour former une **ligne** (1er dimension)
B Count	nombre de ligne - 1 pour former un **bloc** (2ième dimension)
C Count	nombre de blocs -1 pour former un **paquet** (3ième dimension)
B Pitch	« Pitch » ligne : différence d'adresse entre 2 lignes (-P)
C Pitch	«Pitch » bloc : différence d'adresse entre 2 blocs (-P)

Avec ces paramètres, l'adressage de sous-matrices devient très souple et très performant puisque le calcul d'adresses est opéré à la volée et de manière intrinsèque par le TC. Concernant l'imagerie, de tels mécanismes peuvent servir pour l'adressage d'une zone d'écran contenue dans une mémoire VRAM, ou bien, plus simplement, l'adressage d'une région d'intérêt (ou ROI pour *Region Of Interest*) au sein d'une image. Plus encore, le support tridimensionnel permet l'extraction d'une composante de couleur d'une ROI où les données sont entrelacées comme pour le format YUV par exemple.

Pour aller plus loin dans la compréhension de la formidable souplesse des mécanismes de transfert qui est offerte, rappelons l'indépendance du format des données pour la source et la destination. Cette indépendance permet de manipuler l'organisation des données de sorte que nous puissions envisager d'adresser un ensemble de blocs 2D en mémoire externe (exemple de l'isolement des sous matrices 8×8 d'une image en vue d'un calcul de transformée en cosinus discrets) et de répartir ces données sous forme d'une suite de vecteurs 1D en mémoire interne (pour favoriser la vitesse du traitement par exemple).

En outre, la taille et le nombre des bancs internes étant limités, le transit des données via la mémoire interne est susceptible de se faire en plusieurs fois. Nous avons vu le support de listes chaînées circulaires qui nous permet, dans ce cas, de définir une liste constituée de 2 PT : l'un important les données à traiter (mémoire externe $\Rightarrow$ mémoire interne), l'autre exportant les données traitées (mémoire interne $\Rightarrow$ mémoire externe). Si nous considérons qu'une seule ligne de blocs 8×8 peut être traitée à la fois et que l'image compte plusieurs de ces lignes, des modifications sur l'adresse source et destination respective des 2 PT sont nécessaires pour assurer la bonne reprise du transfert entre 2 exécutions de liste. Or, cette

gestion peut être également automatisée par le TC. Concrètement, cela se traduit par la possibilité de modifier l'adresse de début du transfert avec l'adresse du dernier bloc ou paquet incrémenté du pas correspondant (*B* pitch ou *C* pitch). Cela donne :

Equation 1-3. Modification des adresses PT

$$\begin{cases} \text{Adresse Src/Dst} \leftarrow \text{Adresse Src/Dst dernier paquet} + B_{pitch} \\ \text{ou} \\ \text{Adresse Src/Dst} \leftarrow \text{Adresse Src/Dst dernier paquet} + C_{pitch} \end{cases}$$

Dans ce cas, nous ne parlons plus de calculs intrinsèques au TC puisque la structure des paramètres du PT est bien modifiée. Ce mécanisme peut également s'appliquer à la modification des pointeurs en mémoire interne. Ainsi, partant du principe que nous exploitons le fonctionnement asynchrone du DMA de telle sorte que les transferts soient lancés parallèlement au traitement, il convient alors de minimiser les contentions d'accès aux bancs de mémoire de données entre le TC et l'unité de traitement du PP. Si nous assignons des bancs séparés aux deux acteurs et que nous modifions temporellement le rôle de ces bancs, nous pouvons nous prémunir des phénomènes de contention. Ainsi, avec la technique du double buffering, l'utilisation alternée de deux bancs permet d'une part au DMA d'accéder à un unique banc pour sortir les données traitées et, en séquence, rapatrier les futures données à traiter et, d'autre part, l'utilisation d'un deuxième banc permet au PP de lire les données à traiter et d'écraser ces données avec les résultats du traitement (nous "pipelinons" ainsi les étapes du traitement). L'alternance du rôle des bancs dans cet exemple de double buffering peut être assurée automatiquement par le TC. Dans ce contexte, la 3ème dimension définissant le format des données internes n'est pas utilisée (nous considérons que les données sont regroupées au sein d'un même paquet en spécifiant *C count = 0*) pour permettre au pas *C* de définir le décalage d'adresse du deuxième banc que le TC devra utiliser. Pour assurer le mécanisme d'alternance du rôle des bancs qui se traduit, dans notre exemple, par un va-et-vient de la destination du premier PT, et, respectivement, de la source du second, le TC va plus loin dans la modification des paramètres PT. Il autorise en effet l'inversion automatique de la direction des pas après la modification d'adresse évoquée plus haut. Le mécanisme devient alors :

Equation 1-4. Double buffering et modification d'adresse

$$\begin{cases} \text{Adresse Src/Dst} \leftarrow \text{A} \quad \text{dresse Src/Dst dernier paqu} \quad \text{etC p} \quad \text{itch} \\ \qquad\qquad\qquad\qquad \text{C pitc} \quad \leftarrow \text{hC pitc} \end{cases}$$

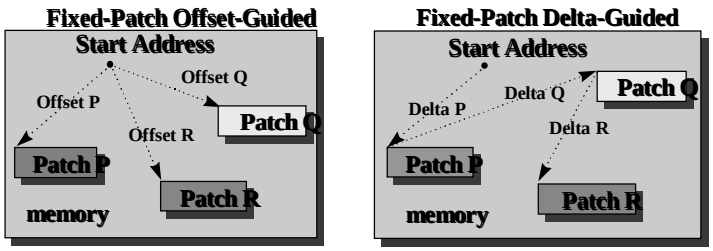
Ces mécanismes permettent une réelle optimisation dans la gestion des transferts. Ce gain en performance n'est pas tant lié à la complexité moyenne des calculs d'adresses qui sont généralement réalisés (même si l'exemple du plan YUV montre que cette charge de calcul est avantageusement déportée au niveau du TC) mais au fait qu'il favorise surtout la cohérence des caches d'instructions des PP. En effet, en déportant la gestion des requêtes au niveau du TC (gestion transparente de la liste et modification automatique des adresses après transfert), les caches d'instructions maintiennent la présence des blocs de programme associés au coeur du traitement. De cette manière, entre deux soumissions d'une liste de PT, le nombre de rechargements de blocs de cache est diminué. Ce gain est augmenté par la taille et la structure des caches d'instructions qui regroupent un total de 256 instructions VLIW pour une associativité à quatre voies (le remplacement des blocs étant géré selon une politique LRU pour *Least Recently Used*). Ainsi, chaque bloc ne peut contenir qu'un total de 64 instructions VLIW contiguës (instructions 64 bits), ces instructions étant elles-mêmes chargées par sous-bloc de seize instructions.

L'impact de la gestion automatique des PT permet, notamment, l'optimisation d'algorithmes rapides pour lesquels la durée des transferts (données + instructions) dépasse la durée des traitements. En outre, ce gain est fortement corrélé à la quantité de blocs occupés par l'algorithme. De plus, en cas de faute de cache, la latence de chargement d'un sous-bloc est variable et dépend du niveau de priorité de l'ensemble des requêtes de transfert pendantes (matérielles et logicielles). Plusieurs milliers de cycles sont ainsi parfois nécessaires avant la mise à jour d'un sous-bloc, ce qui justifie pleinement l'utilisation de l'ensemble des mécanismes que nous venons de présenter.

Les transferts variables et spécifiques

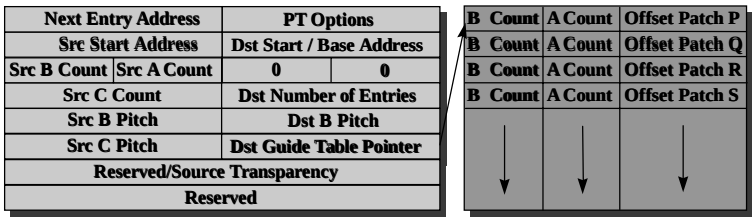
Nous avons, jusqu'à présent, introduit des transferts pour lesquels la répartition spatiale des données est uniforme et régulière pour les trois directions. Nous parlons aussi de transferts dimensionnés. Les capacités d'adressage du TC vont encore plus loin avec les transferts variables qui permettent de s'affranchir des contraintes d'uniformité. Pour cela, l'idée consiste à compléter la définition de la structure d'un bloc par un pointeur sur une table de mise en correspondance (LUT pour *Look-Up-Table*) qui précise la position de chaque bloc dans l'image. Il n'y a plus de notion de paquets (nous perdons une dimension), les paramètres de la dimension *C* étant alors utilisés pour préciser l'emplacement de la LUT ainsi que sa taille. La position des blocs peut être relative à l'adresse de la source ou de la destination du transfert (*Offset-guided*), ou relatif à la position du dernier bloc (*Delta-guided*) comme l'exprime la Figure1-11. En outre, le format de la source étant indépendant de celui de la destination, nous pouvons ainsi avoir une source dimensionnée alors que la destination sera variable. Un tel mécanisme permet de gérer aisément l'étape de compensation de mouvement lors de la reconstruction d'un flux MPEG, les blocs 8×8 reconstruits étant directement positionnés dans le plan image résultant.

Figure 1-11. Transferts variables



Les transferts variables vont encore plus loin avec la possibilité de modifier la taille individuelle de chaque bloc que nous avons supposé fixée dans l'exemple précédent (Figure 1-12). Dans ce cas, la LUT précise les dimensions exactes des blocs ainsi que leur emplacement dans le plan (*"offset-guided"* ou *"delta-guided"*). Nous parlons alors de blocs variables guidés par décalage si nous tentons de donner une traduction à *"Offset/ Delta guided Variable-Patch"*.

Figure 1-12. Transferts variables et géométrie variable des blocs



Pour terminer la description des principaux types de transferts, nous pouvons parler des transferts de type “remplissage” ainsi que des transferts en mode transparent.

Les transferts de type remplissage permettent de préciser, à la place des paramètres dimensionnels de la source, une valeur de remplissage 64 bits que le TC utilise pour initialiser la destination du transfert qui, comme précédemment, peut être soit dimensionnée soit variable.

Par ailleurs, chaque PT dispose d’un champ permettant de filtrer les valeurs de la source. Ce mécanisme consiste à comparer les données successives de la source à une valeur de référence 8/16/32 ou 64 bits. Lorsque la donnée source est égale à cette valeur de référence, l’écriture vers la destination devient effective. L’égalité stricte empêche l’utilisation de ce mécanisme pour seuiller une image, mais il permet, par exemple, d’incruster très facilement du texte dans une image.

Pour conclure, nous pouvons souligner le véritable rôle de co-processeur de transfert qu’incarne le TC (rôle que nous avons par ailleurs justifié) et mettre à nouveau en avant l’ensemble des fonctionnalités offertes par ce DMA. Il s’agit bien d’un cas unique même si d’autres processeurs reprennent une partie de ces fonctionnalités, comme par exemple les processeurs C6X. Enfin, l’ensemble de ces mécanismes souligne l’adéquation de l’architecture à la plupart des besoins en matière de flux de données pour l’imagerie bas niveau.

1.1.2 Programmation des DSP

> *Prospects to finding ever-increasing amounts of instruction-level parallelism in a manner that is efficient to exploit are somewhat limited*. Hennessy et Patterson [3].

Les constructeurs de DSP mettent en avant les performances crêtes de leurs processeurs mais ne font que très rarement part de l'effort nécessaire pour leur mise en œuvre et de la difficulté à les programmer pour atteindre une fraction significative de leurs performances. Or, si nous considérons la durée de vie des processeurs actuels et le contexte de concurrence qui poussent à la diminution des durées de mise en œuvre de ces composants, nous comprenons qu'il s'agit pourtant bien là d'un critère économique essentiel. Dans ce sens, il faut souligner que la programmation des DSP avancés tels que ceux présentés dans les paragraphes précédents constitue une étape longue et difficile tant l'expertise requise du fonctionnement intrinsèque de l'architecture est à la fois grande et incontournable pour atteindre les performances dictées par des algorithmes qui, par ailleurs, gagnent en complexité.

Les compilateurs/optimiseurs de langage haut niveau (la plupart du temps le langage *C* avec le début du support pour le *C++* détaillé dans [3]) n'offrent, jusqu'à présent, que des performances très éloignées des performances crêtes. A titre d'exemple, aucun compilateur des principaux DSP commercialisés n'utilise, de manière automatique, les capacités SIMD des coeurs actuels (C8X, C6X[1], Sharc 21160[2], Lucent DSP16XXX). Rares sont ceux qui font un usage automatique des instructions de type VMAC (chargement, multiplication et addition en un minimum de cycles ou *Vector, Multiply and ACcumulate*), macro-instructions qui ont paradoxalement largement contribué à l'essor des DSP. Par ailleurs, la complexité matérielle du coeur de certains processeurs comme celui des PP du C8X rend extrêmement difficile voire impossible à résoudre, dans un temps raisonnable, le problème

1. L'utilisation des opérations SIMD en C peut être manuellement spécifiée au moyen de fonctions C dites intrinsèques qui obligent le compilateur à faire usage, entre autres, de ces opérations assembleurs SIMD. Cette possibilité existe également pour le Trimedia ou le compilateur MMX d'Intel.

2. Ici, le compilateur permet une approche semi automatisée puisque le programmeur doit spécifier et modifier la section de code C pour laquelle les opérateurs arithmétiques et les accès mémoire doivent s'exécuter dans le mode SIMD de l'architecture.

NP-complet[1] que constitue la projection du graphe de dépendance algorithmique sur les ressources de l'architecture.

Cette situation oblige, pour les parties algorithmiques les plus sensibles, à exploiter un parallélisme de données ou de flux directement au niveau du micro-code grâce à des instructions assembleur spécifiques que le programmeur doit connaître et mettre en œuvre. Le développement est alors long et difficile d'autant que, pour favoriser les deux grandes sources de parallélisme (flux/données), plusieurs techniques doivent être combinées pour atteindre les meilleures performances. Ainsi, le niveau d'expertise nécessaire pour le développement sur C8X est tel qu'il explique largement un succès commercial modéré.

Ce tableau ne doit pourtant pas ternir les récentes évolutions des techniques de compilation avec notamment la technique dite du "pipeline logiciel" (*software pipelining*) que met en œuvre le compilateur C des coeurs C6X, et dont l'efficacité s'améliore au fil des versions de l'outil. Cette efficacité grandissante est sans aucun doute liée à une architecture de coeur simplifiée qui, grâce à un jeu d'instructions réduit combiné à un ordonnancement d'opérations VLIW figé au stade de la compilation (on parle de " *compile-time scheduling*"), permet d'automatiser cette technique d'optimisation générique tout en dopant les fréquences d'horloge de ces coeurs simplifiés.

Si des progrès notables sont ainsi relevés pour l'optimisation automatique des ressources processeurs, progrès qui reposent donc essentiellement sur un parallélisme de flux, les techniques permettant l'optimisation des flots d'entrée/sortie de données comme celles favorisant la localité des références (blocage de boucles [5], chaînage des opérateurs algorithmiques ou préservation de la cohérence des caches par la mise en œuvre d'instructions de référencement spécifiques outrepassant le cache comme pour le C8X ou Trimedia), ne sont, pour l'instant, que partiellement et très rarement automatisées dans le monde des RISC [6][7] comme dans celui des DSP.

A cet égard, il faut souligner la différence architecturale majeure que constitue l'emploi généralisé pour les DSP de co-processeurs de transfert (ou DMA) couplés aux mémoires internes de données qui s'opposent aux caches de données matériels qu'intègrent la plupart

1. Un problème "NP-complet" se dit d'un problème qui ne peut être résolu suivant une durée bornée.

des processeurs RISC. Dans un contexte où les coeurs DSP et RISC convergent technologiquement [8][9][10][11] (la piste des coeurs simultanément VLIW et SIMD semble être communément privilégiée avec, notamment, le futur Merced d'Intel/HP ou, pour les DSP, le TigerSharc d'Analog Devices et le StarCore140 de Lucent/Motorola), les DMA constituent l'un des rares discriminants subsistant encore entre ces deux classes d'architectures. Il s'agit effectivement d'une spécificité qui ne concerne que les caches de données, beaucoup de DSP intégrant des caches d'instructions matériels comme leurs homologues RISC. Les DSP présentés dans cette partie du mémoire en constituent des exemples. L'utilisation des DMA permet une réelle optimisation des flux d'entrées/sorties (E/S) mais au prix d'un effort de programmation supplémentaire. Par ailleurs, cet effort de programmation est bien souvent incontournable compte tenu du coût prohibitif des accès directs à la mémoire externe, coût qui s'explique en partie par le pipeline des étapes élémentaires d'un accès et la latence nécessaire pour remplir ce pipeline. Ainsi, les données doivent être cachées par bloc de manière logicielle grâce au DMA, au risque d'une chute de performance importante (facteur 10 et plus avec le C8X/C6X). Compte tenu de l'évolution récente des potentialités offertes par les DMA avec, notamment le support de données multi-dimensionnelles (C8X/C6X/Sharc), la lourdeur de la mise en œuvre est ainsi accentuée alors que les outils génériques et flexibles sont inexistants.

En terme de productivité, le but ici affiché est d'asseoir les développements logiciels sur une gestion transparente (automatique) et optimisante des flux de données. Une approche générique du problème est nécessaire pour pouvoir gérer l'éventail des besoins des algorithmes en matière d'E/S. La notion de flexibilité, quant à elle, résume la capacité à modifier les paramètres associés aux flux. La possibilité de reconfigurer dynamiquement le nombre de processeurs affectés à un traitement pour les architectures multi-processeurs, ou simplement de modifier la quantité de données traitées, constitue des exemples de flexibilité qu'aucun outil DSP ne propose aujourd'hui.

Face à ce déficit global de techniques d'aide au développement logiciel et compte tenu des nouvelles performances introduites par les $5^{ème}$ et $6^{ème}$ générations de processeurs RISC qui sont, dans le même temps, moins difficiles à mettre en œuvre sur le plan logiciel, il apparaît légitime de s'interroger sur l'intérêt de la mise en œuvre de DSP.

1.1.3 Intérêts des DSP

1.1.3.1 Concurrence des processeurs RISC

Le constat de la difficulté à programmer les dernières architectures DSP étant établi, il convient d'opposer cette situation au rapport des performances mesurées ramené à l'effort de développement nécessaire (niveau d'expertise et durée de développement) que proposent d'autres alternatives matérielles. Ainsi, avec un grand nombre d'outils de développement et une programmation haut niveau plus efficace, ce ratio semble aujourd'hui favorable aux architectures de type RISC.

Table 1-4. Performances des processeurs DSP et RISC en 1999

Processeur	Nb. d'opérations max. en || sans/avec SIMD : (alu+mem.+@mod.)[a]	Freq. MHz	Performance crête maintenable (hors modification d'adresse @mod) : E = Entier, F=Flottant
TMS320C80 (entier/flottant)	14+9+9 /30+9+9	60	120 MFLOPS (MP), 1.4 GOPS /2.5 GOPS (SIMD 8 bits)
TMS320C6202 (entier)	6+2+2 /8+2+2	250	0 FLOP, 1.6 GOPS, 500 MMACS (E.16b) /2.5 GOPS (SIMD 16 bits)
Sharc 21160 (flottant)	4+2+2 (idem 21060) /8+2+2	100	600 MFLOPS, 100 MMACS (F), 800 MOP /1,3 GOPS, 200 MMACS (F)
TMS320C6701 (flottant)	6+2+2 /8+2+2	166	1 GFLOPS, 332 MMACS (F), 1.3 GOP /1.6 GOPS
Tiger Sharc (flottant)[12]	6+2+2 /32+2+2	250	1.5 GFLOPS, 1 GMACS (F), 2 GOPS /8 GOPS (SIMD 8 bits), 2 GMACS (E. 16b)
Trimedia TM1300 (entier/flottant) [8][13]	5+0+0 ou 3+2+0 /5*4*2+0+0	166	400 MFLOPS, 166 MMACS (E/F), 830 MOP /6,5 GOPS (SIMD 8 bits)
Alpha 21264 (flottant)[14]	4(peak)/2(sust.)+2+0? /?+2+0	>600	>2.4 GOPS
Power PC G4 (flottant)[15][16]	3+? /3+	500	? /4.5 GFLOPS & 4 GMACS (F. 32b),
Pentium III (flottant)[17][18]	2+2+0 /16+2+0(MMX-1)	>500	>1 GFLOPS, 1 GMACS[b], 2 GOP /9 GOPS (SIMD 8 bits)

a. alu représente le nombre d'opérations arithmétiques (multiplication, addition) et logiques parallèles : mem. représente le nombre d'accès mémoire et @mod est le nombre d'opérations arithmétiques possibles sur les registres pointeur en parallèle des accès (comme la pré/post-incrémentation/décrémentation avec modification).

b. Avec certaines restrictions

L'ordonnancement dynamique d'instructions dans le désordre qui s'appuie sur la présence de multiples pipelines dédiés, déporte, au niveau de l'architecture et de manière transparente, une grande partie des difficultés logicielles rencontrées avec les DSP et plus particulièrement avec les architectures VLIW. Par ailleurs, nous pouvons schématiquement prétendre que le nombre maximum d'instructions exécutées en parallèle qui est en faveur des DSP (hors opérations SIMD) se trouve compensé par des fréquences d'horloge plus élevées pour les RISC.

Il faut également faire mention des capacités SIMD de la plupart des processeurs RISC récents qui, lorsqu'elles sont utilisées, permettent des performances comparables et parfois supérieures à celles des meilleurs DSP du marché actuel. Il convient cependant de souligner l'actuelle difficulté d'une mise en œuvre haut niveau de ces extensions de jeux d'instructions, observation commune avec les DSP. Pour l'architecture Pentium d'Intel, nous notons l'existence de librairies performantes exploitant ce parallélisme de données (il s'agit des extensions MMX) parmi lesquelles, la librairie de traitement de signal/d'image d'Intel, la librairie MIL de traitement d'images de Matrox et les librairies de modélisation 3D (Direct3D de Microsoft ou bien OpenGL de Silicon Graphics).

1.1.3.2 Adéquation des DSP au marché de l'embarqué

Cet état des lieux, favorable aux RISC, ne doit pas masquer l'adéquation des DSP à d'autres critères comme ceux liés aux contraintes des systèmes embarqués. Le prix des DSP ainsi que leur consommation d'énergie ou leur encombrement en font ainsi des solutions de choix pour des systèmes autonomes. Si les chiffres sont en faveur des DSP, il faut mentionner la récente offensive des solutions RISC dédiées au monde de l'embarqué avec le Mobile PII d'Intel ou le K6E d'AMD. Ainsi, ces versions spécifiques de processeurs offrent de bonnes performances pour une consommation diminuée et un encombrement réduit.

Table 1-5. Consommation, encombrement et prix des processeurs DSP et RISC en 1999

Processeurs[a]	Consommation en Watts	Taille en centimètres L×H×(P+epaisseur radiateur/ ventilateur)	Ordre de grandeur du Prix
TMS320C80	4 (@60 MHz)	4,7x4,7x(0.3+0.7)	$128
TMS320C6202*	2 (@ 200 MHz)	1,8x1,8x(0.3+0)	$130
TMS320C6211*[b]	1,5 (@ 150 MHz)	2,7x2,7x(2,3+0)	$25 pour 25000
TMS320C6701*	2 (@ 167 MHz)	3,5x3,5x(0,35+0)	$145 pour 10000
Sharc 21160	2 (@100 MHz)	2,7x2,7x(0,23+0)	$10 par volume
Tiger Sharc	? (@200 MHz)	?	?
Alpha 21264	100 (@1 GHz)	6x6x(0.2+(5,5 @ 600 Mhz))	>$400
PII*	27 (@450 MHz)	11,5x5,5x(1+6)	>$400
Mobile PII*	6,6 (@ 366 MHz)	3,5x3,5x(0,15+0,7) (pour la version à 266 MHz)	<$160
K6E*	8,5 (@ 266MHz)	4,8x4,8x(0,3+?)	$80 pour 1000

a. *: présence d'un mode de fonctionnement dégradé pour diminuer la consommation.

b. Le C6211 intègre un cache de donnée L1 et L2 (unifié avec le cache d'instructions).

Par ailleurs, il convient de souligner, en faveur des DSP, que le vaste éventail de prix, performances et configurations en terme de quantité de mémoire interne ou de périphériques annexes (DMA / ports série / liens de communication) qu'offrent les constructeurs, permet d'optimiser au mieux les coûts d'un système. Il s'agit bien là d'une différence importante avec le marché des processeurs RISC qui focalise son offre sur des solutions à haute performance et qui n'a pas, par ailleurs, de véritable politique de continuité dans la fabrication d'anciennes générations de processeurs devenues moins performantes.

Dans le domaine du traitement d'images industriel qui nous intéresse, la puissance des composants et leur difficulté de mise en œuvre constituent les éléments clés de la problématique des travaux de cette thèse. Le traitement à cadence vidéo d'une image 512×512 nécessite une capacité de traitement de 10 millions de pixels par seconde ce qui représente 100 ns de durée de traitement par pixel. Or, compte tenu de la complexité croissante des algorithmes, leur implantation, sous de telles contraintes temporelles, constitue encore bien souvent un défi à relever. Ce défi passe notamment par l'optimisation

des flux d'information, aspect difficilement maîtrisable sur le plan logiciel avec les caches de données des processeurs RISC [19][20][21][22][23] alors qu'en parallèle, nous rencontrons des tentatives sur le plan matériel [24][25].

Sur ce point, l'effort de programmation des DMA est payant. L'amélioration des performances s'explique ainsi par le fait que d'une part, seules les données d'intérêt sont a priori transférées alors que d'autre part, les DMA permettent le recouvrement temporel des phases de calculs et de transferts avec, en général, un minimum de contention. Cette véritable opportunité d'optimisation des flux de données s'accompagne d'une prédictabilité accrue des performances. Il s'agit bien là d'un avantage par rapport aux RISC pour lesquels les mécanismes de caches sont difficilement modélisables et optimisables dans le cas d'algorithmes quelconques [26][27]. Une étude pour l'implantation d'un décodeur MPEG-2 sur des architectures RISC SIMD [28], suggère l'introduction d'extensions aux mécanismes de cache matériels pour permettre, notamment, un adressage semi-automatique des données suivant leur nature (verrouillage de blocs, accès non caché de certaines données, allocation de zones de cache pour certaines données). Ainsi, selon l'étude, ces extensions spécifiques à chaque catégorie de variables impliquées dans l'algorithme MPEG-2 permettent une réduction de plus de 50% des transferts et une meilleure prédiction des performances accessibles. Nous pouvons également citer comme références allant dans ce sens [29][30].

De plus, l'ordonnancement statique des instructions VLIW des DSP permet une bien meilleure estimation des performances que n'offre pas la simple lecture d'un code assembleur de RISC superscalaire, où ne transparaissent pas les ressources de calcul utilisées ainsi que leurs éventuels conflits d'accès. Cet aspect rend difficile l'optimisation au niveau du micro-code [18] et fait, dans ce cas, appel à des outils de simulation spécifiques comme l'outil VTUNE d'Intel [31].

La question de la prédictabilité des performances amène à celle de l'inter-opérabilité des composants et de leur scalabilité dans le cas d'architecture multi-processeurs. Sur ces points, les DSP constituent des solutions de choix. Ils proposent ainsi un éventail de mécanismes embarqués pour la gestion des flux comme les liens de communication (Sharc/C6X) ou les ports série (Sharc/C40/C6X) pour le modèle de mémoires distribuées qui

s'opposent aux mécanismes pour la gestion de mémoires partagées comme le crossbar du C8X ou l'adressage virtuel partagé des Sharcs. Ces canaux spécifiques pour le transfert de flux sont autant de périphériques qui, d'une part, permettent la réalisation "simple" (dans le sens "*glueless*" : qui ne nécessite pas l'apport d'une électronique supplémentaire) de machines parallèles, et d'autre part, favorisent une bonne scalabilité. De plus, ces voies de communication étant la plupart du temps couplées aux co-processeurs DMA, la prédictabilité mise en avant combinée à l'optimisation des flux peut également s'appliquer au contexte multi-processeurs.

1.1.3.3 Intérêt du C80

Le C80, apparu en 1995, constitue encore l'un des processeurs les plus performants du moment, preuve d'une avance technologique considérable étant donné le contexte très concurrentiel du marché des DSP. Cette avance souligne également l'aspect historique selon lequel ce processeur représentait la seule solution programmable mono-composant commercialisée et suffisamment performante pour atteindre le temps-réel avec des algorithmes lourds tel que le JPEG ou le MPEG-1.

Par ailleurs, nous avons vu l'adéquation du C80 au traitement d'images bas niveau et à l'imagerie en général (ALU des PP, présence du contrôleur vidéo et DMA particulièrement fonctionnel pour ce domaine). Or, cette adéquation ne transparaît pas avec la génération suivante incarnée par la gamme de processeurs C6X qui apparaît avant tout, jusqu'à ce jour, comme des architectures 16 bits dédiées aux télécommunications (ADSL - *Asynchronous Digital Suscriber Line* - compression audio). Ainsi, l'architecture ne supporte pas une arithmétique SIMD sur 8 bits et les accès concurrents d'octets contigus dans la même instruction VLIW posent des problèmes de contention. En outre, le DMA multi-dimensionnel du C6X ne permet pas les transferts variables du C80 et l'absence d'un cache de données (que nous souhaiterions commutable dynamiquement en zone de mémoire non cachée gérée explicitement avec le DMA [1]), rend difficile et peu efficace l'implantation d'algorithmes moyen/haut niveau. En effet, si nous considérons que cette classe

1. Possibilité offerte pour les caches d'instructions du C6201, C6202 et C6211. Par ailleurs, le C6211 ne permet pas la commutation de mode évoquée pour le cache de données qu'il intègre.

d'algorithmes est générateur de références mémoires aléatoires dans le sens où celles-ci dépendent des données et peuvent être très éloignées temporellement et spatialement, l'utilisation du DMA, délicate dans ce contexte, peut se révéler inefficace alors que dans le même temps, les accès directs à la mémoire externe sont très lourdement pénalisés (jusqu'à 40 cycles par accès pour le C62 [32]). Sur ce point, l'architecture C80 et son processeur RISC (MP) associé à un cache de données offre une meilleure adéquation aux applications pour lesquelles la part du moyen et haut niveau est susceptible de nuire aux performances.

L'architecture du C80 apparaît également extrêmement flexible si nous soulignons l'indépendance et l'asynchronisme des différents coeurs de traitement autorisant ainsi une très grande liberté dans la mise en place de sources de parallélisme (partitionnement SPMD - *Single Process, Multiple Data,* MPMD - *Multiple Processes, Multiple Data*, pipeline des traitements, etc.). Cette situation s'oppose, bien sûr, aux architectures mono-coeur telles que celle du C6X, et accentue par là même la complexité de mise enœuvre déjà largement soulignée. En effet, dans un tel contexte, une part non négligeable du développement est consacrée au partitionnement des traitements et à la synchronisation des processeurs.

1.1.4 Conclusions

Dans ce chapitre, nous avons démontré la particularité ainsi que l'intérêt des architectures DSP face aux architectures RISC concurrentes. Nous avons, plus particulièrement, justifié le choix du TMS320C80 comme architecture cible de cette thèse étant donné le domaine d'application du traitement d'images embarqué initialement spécifié. Nous avons, pour l'ensemble des architectures DSP présentées, souligné le manque actuel d'outils logiciels d'aide au développement pour l'extraction automatique des nombreuses sources de parallélisme que permettent les nouvelles architectures DSP.

Si des efforts scientifiques et industriels importants sont déployés pour l'amélioration des techniques de compilation, il convient de souligner l'inexistence d'outils et méthodes pour le problème spécifique de l'exploitation flexible et générique des co-processeurs DMA. Il nous semble en effet que cet aspect du développement ne doit pas être sous-estimé devant l'évolution des DMA (avec le support des données multi-dimensionnelles ou la présence de mécanismes d'adressage complexes comme ceux du TC du C8X) qui induit une étape

de mise en œuvre logicielle de plus en plus fastidieuse. Cette remarque est d'autant plus vraie dans le contexte multi-processeurs du C8X où l'expérience montre que le processus du partitionnement des traitements et des données suscite notamment une programmation des requêtes de transfert qui engendre une part importante des durées de développement.

Dans cette optique, nous recherchons une méthodologie de développement capable d'améliorer la **productivité** de l'étape de mise en œuvre logicielle avec l'exploitation transparente du DMA et des flux de données algorithmiques. Un gage de cette productivité consiste à appuyer les développements sur une méthodologie ayant les caractéristiques suivantes :

- support **générique** des algorithmes,
- optimisation des **performances**,
- **flexiblité**,
- **portabilité** des mécanismes.

Pour être plus précis, nous définissons chacune de ces notions, essentielles à la réduction des durées de développement.

La *généricité* évoquée peut se définir comme la possibilité de spécifier le plus grand éventail d'algorithmes possibles à l'aide d'une représentation unifiée des algorithmes. Cette unification permet de simplifier l'utilisation de la méthodologie de modélisation favorisant ainsi la productivité sur le plan de l'apprentissage et de la mise en œuvre. Pour une classe d'algorithmes, cela sous-entend une analyse des besoins en terme de gestion des flux ainsi qu'un formalisme permettant de décrire et d'intégrer les besoins des différentes applications visées.

La recherche d'une méthodologie de modélisation capable d'*optimiser* la gestion des flux découle de notre contexte pour la mise en œuvre d'algorithmes en temps réel sur des systèmes parallèles embarquables. Dans le cas des multi-processeurs, les contraintes temporelles nous incitent à exploiter le maximum de sources de parallélisme. Plus précisément, l'exploitation du parallélisme des instructions (avec l'exemple des capacités de traitement SIMD) pour l'implantation des opérateurs de traitement semble être une piste

incontournable pour la maximisation de l'utilisation des ressources intrinsèques à chaque processeur. A un niveau plus élevé, l'utilisation parallèle de mutiples processeurs est également primordiale. Dans ce sens, le schéma de parallélisation SPMD pour lequel chaque processeur exécute de manière concurrente un unique algorithme sur un sous-ensemble des données du problème apparaît comme le plus intuitif sur le plan conceptuel. Il s'agit également de la forme de parallélisme la plus simple à mettre en œuvre dans le cas du C80 et de l'architecture homogène de ses PPs. En visant ce modèle de macro-parallélisme, la question de l'optimisation des flux d'entrées/sorties pris en charge par le DMA et partitionnés entre les différents processeurs, devient prépondérante. En effet, pour que le paradigme SPMD s'avère véritablement bénéfique, nous devons nous assurer que le coût des transferts DMA partitionnés (communications) n'occulte pas la capacité brute de traitement qui est découplée par l'utilisation de plusieurs processeurs.

Par *flexibilité*, nous entendons la nécessité de pouvoir traiter des images de taille variable (avec le support de régions d'intérêt). Dans le cas des multi-processeurs, il s'agit de permettre le paramétrage du nombre de processeurs assignés à la parallélisation. Dans une autre optique, une caractéristique de cette flexibilité transparaît avec le support d'une gestion des flux qui puisse être reconfigurable dynamiquement afin de diversifier l'utilisation d'un système et d'optimiser ainsi au mieux les ressources.

La *portabilité* que nous recherchons consiste à appréhender les différentes architectures DMA tout en autorisant le support du plus grand nombre de fonctionnalités matérielles offertes par ces périphériques. Cette orientation permet de favoriser les performances en exploitant au mieux les caractéristiques architecturales. De plus, cet aspect augmente et pérennise l'intérêt d'une méthodologie pour la gestion des flux avec le paradigme matériel des DMA. La méthode recherchée doit ainsi permettre d'appréhender les riches mécanismes du DMA C8X comme un sur-ensemble de fonctionnalités offertes par d'autres DMA moins fonctionnels (comme celui du Sharc 21060/21160 par exemple). Intuitivement, nous pouvons ainsi percevoir le support des transferts 3D (C8X/C6X) comme un sur-ensemble des paramètres DMA nécessaires à la spécification d'un transfert

2D (Sharc). Ce même raisonnement doit permettre le support de DMA mono-dimensionnels classiques.

Pour conclure, nous avons défini la portée algorithmique et architecturale des travaux que nous visons dans cette thèse en ciblant le traitement d'images bas niveau dans le contexte de DSP mono/multi-processeur(s) utilisant un DMA. La recherche d'une méthodologie permettant l'exploitation générique, performante et flexible des flux d'entrées/sorties sera décrite au chapitre 2 après une présentation de l'état de l'art des techniques de développement logiciel. Cet état de l'art souligne les limitations actuelles face aux objectifs que nous venons d'établir et qui nous semblent particulièrement importants en vue de l'implantation des deux applications de référence stipulées dans le cahier des charges de nos travaux (traités à la fin du mémoire). Par ailleurs, cette étude des limitations actuelles sur le plan de la gestion des flux nous permettra d'introduire des objectifs de recherche exprimés en termes *scientifiques* face aux besoins logiciels ici exprimés sur le plan de *l'ingénierie* des systèmes embarqués complexes.

1.2 Etat de l’art des outils et méthodes de programmation parallèles

Dans cette partie, nous présentons un état de l’art des outils et méthodes de programmation pour les systèmes temps réel parallèles. Notre démarche a pour but de tirer des enseignements sur les pratiques et limitations actuelles afin de proposer et de justifier les orientations de recherche développées dans la suite de ce mémoire.

Il s’agit d’étudier les pistes qui nous sont offertes pour apporter des réponses aux notions de généricité, flexibilité et d’optimisation des performances dans la programmation de systèmes embarqués complexes, telles que nous les avons définies dans la conclusion du précédent point.

Trois axes sont étudiés : les outils et librairies de traitement d’images spécifiquement dédiés au C80, les langages de programmation et, enfin, les ateliers de génération de code DSP pour des architectures complexes. Nous concluons cette partie par la justification du choix de quelques perspectives d’innovation qui tiennent compte de l’état de l’art dressé et du cahier des charges initial de nos travaux. Cette approche permet notamment de mieux appréhender la méthodologie de programmation développée au chapitre suivant et sur laquelle repose l’implantation des deux applications de référence présentées à la fin de ce mémoire.

1.2.1 Outils et librairies pour le TMS320C80

Dans ce paragraphe, nous introduisons les principaux outils et librairies de traitement d'images existants pour l'architecture C8X.

1.2.1.1 Les outils de développement Texas Instruments

S'agissant d'une architecture hétérogène, le C8X est fourni avec deux compilateurs/ assembleurs, l'un est dédié au MP, l'autre aux PP. Ces outils sont complétés par un éditeur de liens qui est commun aux fichiers binaires intermédiaires générés par les compilateurs MP/PP, afin de ne produire qu'un seul exécutable à télécharger dans le système. Si

l'optimisation du code assembleur est bien supportée par ces outils, la capacité à exploiter les spécificités de l'architecture et notamment le format VLIW des instructions PP ainsi que le potentiel du traitement de l'ALU est très limitée (pour les cas simples, l'optimiseur C met essentiellement en œuvre les contrôleurs de boucle matériels). De là, le recours à une programmation s'effectuant au niveau du micro-code s'avère souvent nécessaire. A titre d'exemple, l'auteur de [33] montre une accélération d'un facteur 50 sur C80 entre les performances d'un filtre médian 3×3 écrit en C et sa version manuellement optimisée en assembleur[1].

Le constructeur fournit également un noyau exécutif principalement destiné au MP - le MTE (*Multi-Tasking Executive*) - qui permet l'exécution de tâches logicielles concurrentes sur le MP (elles ne sont cependant pas automatiquement préemptées) et la synchronisation de celles-ci avec les mécanismes traditionnels de type sémaphore ou envoi de messages (s'agissant d'une architecture à mémoire partagée, le terme de partage de message s'avère plus approprié). Cet exécutif permet également la synchronisation du processeur maître (le MP) avec les PP grâce à des mécanismes de scrutation ou d'interruption. Dans un contexte multi-C80, cette couche logicielle offre également des mécanismes standards d'échanges et de synchronisation entre les DSP. L'utilisation d'une telle librairie permet alors un premier niveau d'abstraction matérielle et simplifie le développement.

En 1998 (soit environ 3 ans après la sortie des premiers composants et outils), Texas Instruments, en partenariat avec l'université Nationale de Séoul, introduit un nouvel outil dédié au codage sur PP : le compacteur de code [33]. Il s'agit d'un pré-processeur de fichiers source assembleur qui permet d'exploiter les unités du code VLIW qui ne sont pas utilisées dans le source d'origine. Ce mécanisme repose sur l'analyse des dépendances des différentes ressources matérielles utilisées. Parmi les ressources considérées, l'outil s'intéresse bien sûr aux durées de vie et dépendances des registres de données de manière à favoriser l'entrelacement des opérations assembleur au sein d'instructions VLIW. Dans le

1. Dans [34], nous trouvons quelques techniques d'optimisation d'un tel filtre.

même temps, cet outil introduit également l'allocation automatique des registres pour favoriser la compaction des opérations assembleur.

Les restrictions au niveau de la construction manuelle des instructions VLIW sont telles sur cette architecture (page D3-D4 de [35]) que cet outil apparaît comme véritablement précieux malgré ses lacunes. Ainsi, si l'allocation des registres est effectivement assurée par ce compacteur, il ne supporte pas l'empilement et le dépilement automatique " des registres. De même, il ne modifie pas les opérations assembleur et ne restructure pas le graphe de dépendance algorithmique (il modifie "simplement" la projection des opérations assembleur sur les instructions VLIW). Dans ce sens, et à l'inverse de l'optimiseur de code assembleur de la série des C6X, il ne met pas en place des techniques d'optimisation comme le pipeline logiciel. Certaines directives doivent être insérées dans le code pour renseigner le compacteur sur le comportement dynamique du programme (notamment en ce qui concerne les ruptures de séquences des contrôleurs de boucle), ce qui rend l'approche semi-automatique.

Malgré ces insuffisances, l'outil permet une meilleure productivité et simplifie l'écriture du code assembleur. Le format d'entrée adopte une syntaxe dite d'assembleur "linéaire". Sous cette forme, des noms symboliques de variables remplacent les étiquettes de registres (à ceci près que la nature des registres doit tout de même être spécifiée suivant que l'on considère un registre d'adresses, d'index ou de données). Par ailleurs, le code ne compte idéalement pas d'opérations en parallèle mais détaille le séquencement logique de celles-ci. Des directives spécifiques au compacteur de code sont nécessaires pour la cohérence de certaines constructions VLIW liées aux ruptures de séquence (de type branchement). Par ailleurs, pour les cas où l'allocation de registres ne peut aboutir, des informations précises sont retournées à l'utilisateur pour la mise au point du code (notamment au niveau du spilling). Un mode dégradé permet d'exploiter l'allocateur de registres indépendamment de la compaction, ce qui autorise une meilleure analyse des cas où les ressources sont insuffisantes.

1.2.1.2 Les librairies de traitement d'images

Les deux principales librairies de traitement d'images disponibles pour C80 sont celles, sous licence, de l'université de Washington et celle, concurrente, de la société Matrox qui accompagne la solution matérielle d'acquisition/traitement "Genesis".

1.2.1.2.1 La librairie de l'université de Washington

La première librairie est issue d'un partenariat avec Texas Instruments qui démarre dès 1994 et permet notamment la mise au point du composant autour d'une carte d'évaluation : la MediaStation 5000 [36][37][38]. Les caractéristiques de cette carte ISA pour PC qui s'appuie sur un C80 cadencé à 40 MHz sont largement reprises au niveau de la carte d'évaluation officielle de Texas utilisée dans le cadre de ce mémoire (la SDB pour*Software Development Board*). Il s'agit d'un vaste projet mobilisant près de 16 universitaires et visant la réalisation d'une librairie optimisée qui compte aujourd'hui près de 200 opérateurs de traitement. Nous y trouvons essentiellement des fonctions bas niveau parmi lesquelles les traitements point à point et ceux orientés voisinages (filtrage/convolution/ opérateurs morphologiques), les traitements statistiques ainsi que les opérateurs de type "transformées" (géométriques avec le warping, la rotation, le redimensionnement ou de domaine avec la DCT, FFT - *Fast Fourier Transform*, DWT - *Discrete Wavelet Transform*, YUV↔RGB). Cette librairie supporte également la détection de motifs ("pattern matching" par corrélation exhaustive ou multi-résolution), la segmentation par Snakes et le filtrage optimal de Canny. Enfin, nous trouvons certaines fonctions plus spécifiques comme pour la compression de données (VQ - *Vector Quantization*, RLE - *Run Length Encoding*, Huffman) ou le graphisme 3D.

Sur le plan de la mise en œuvre, si le macro-parallélisme des traitements (SPMD, MPMD ou pipeliné) est à programmer explicitement par le développeur, la plupart des n œuds supportent nativement le partitionnement SPMD entre les processeurs PP. Par ailleurs, l'infrastructure logicielle est conçue de manière à pouvoir enrichir la librairie en offrant la possibilité de récupérer les mécanismes de gestion des transferts de données par

l'intermédiaire du DMA. Ces mécanismes sont limités dans le sens où ils ne permettent pas un niveau d'encapsulation suffisant pour le traitement d'images de taille variable ou par rapport à un sens ou à un ordre de parcours spécifique des données de l'image. Ces mécanismes sont par ailleurs figés pour chaque opérateur au moment de la compilation. De ce fait, la librairie ne permet pas de chaîner simplement et dynamiquement l'exécution des opérateurs afin d'améliorer la localité des données dans les mémoires internes tampons. Cela signifie que l'application de deux opérateurs bas niveau sur une même image nécessite de parcourir l'image à deux reprises ce qui engendre généralement une diminution globale des performances. L'alternative consiste alors à reprogrammer les différents opérateurs pour chaîner statiquement l'exécution des fonctions de traitement (c'est-à-dire à créer des macro-opérateurs). De plus, cette étape nécessite généralement de repenser la programmation DMA afin de véritablement optimiser les performances. Dans ce sens, le chaînage ou l'extension même des opérateurs de la librairie accusent le manque d'une procédure unifiée (générique) et véritablement flexible.

1.2.1.2.2 Le système Genesis/MIL de Matrox

La librairie commerciale MIL de Matrox pour la carte Genesis apparaît sur le marché en 1996 et s'articule autour d'une offre professionnelle pour PC autorisant le chaînage de plusieurs cartes PCI comportant chacune un ou deux nœuds C80. Chaque nœud C80 est par ailleurs épaulé par un ASIC propriétaire pour l'accélération des traitements bas niveau : le NOA. Selon le constructeur, la puissance crête théorique de traitement s'élève à 100 GOPS pour une configuration optimale de 13 processeurs C80 et autant de composants NOA. La librairie associée à ce système offre un ensemble de fonctions pouvant opérer sur des images de taille variable et suivant un partitionnement défini par l'utilisateur (SPMD, MPMD, pipeliné). De plus, nous trouvons des mécanismes par défaut qui permettent de répartir les traitements sur les processeurs des différentes cartes et d'automatiser l'utilisation du partitionnement SPMD. L'essentiel du bas niveau est implanté et correspond à la plupart des fonctions optimisées présentes dans la librairie de Washington. Comparé à cette dernière, Matrox apporte essentiellement l'analyse statistique de régions

("blob analysis") ainsi qu'une librairie de compression JPEG sans perte qui s'appuie par ailleurs sur le NOA à l'instar de la plupart des opérations de voisinage de la librairie.

La formidable capacité de traitement de la solution Genesis est accessible depuis l'hôte PC grâce à une architecture en client/serveur où le programme PC transmet les commandes de traitement au serveur C80 de la carte maîtresse du système chargée de distribuer l'exécution des opérateurs (le partitionnement pouvant être totalement paramétré).

Si Matrox propose une architecture logicielle et matérielle souple et très performante, nous pouvons lui reprocher, comme pour la première librairie, de ne pas permettre le chaînage des opérateurs au niveau du C80. Un kit de développement permet de récupérer les mécanismes de transfert DMA associés à différentes familles d'opérateurs pour enrichir la librairie native avec de nouvelles fonctions PP. Ces mécanismes sont généralement plus performants que ceux de la librairie de Washington (avec le support systématique de taille d'image variable ainsi que le partionnement SPMD entre un nombre variable de PP répartis sur l'ensemble des processeurs C80 de la configuration). Ils montrent en revanche les mêmes limitations que celles déjà évoquées lorsque nous cherchons à compléter la librairie, ou lorsque nous souhaitons optimiser les traitements avec l'exemple du chaînage des opérateurs au niveau des PP.

1.2.1.2.3 Conclusions

Pour conclure, nous constatons que malgré un contexte matériel très inégal, l'infrastructure logicielle de ces deux librairies montre essentiellement les mêmes limitations pour le développeur. Elles sont principalement de deux ordres. Premièrement, l'extension de ces librairies apparaît comme une opération qui manque de souplesse et réclame un effort de programmation important. Cette constatation souligne l'inexistence d'une méthodologie de développement véritablement structurée et unifiée (générique) pour l'ensemble des traitements bas niveau. Cette lacune met en avant le deuxième aspect relatif à l'optimisation des flux de données que ne permettent pas aisément l'une ou l'autre des librairies. Là encore, seule une approche globale et unifiée de la gestion des flux au plus bas niveau (qu'il s'agisse des flux de données ou d'instructions) permet une meilleure optimisation de la

vitesse de traitement des algorithmes. Au regard de la puissance calculatoire de la carte Genesis, ce deuxième point n'a bien sûr de sens que dans la mesure où nous visons un contexte temporel contraignant et où le coût du matériel doit également être optimisé.

1.2.2 Les langages de programmation dédiés

Dans cette partie, nous présentons brièvement les caractéristiques de quelques langages conçus pour l'expression du parallélisme ainsi que les formalismes synchrones qui permettent, plus spécifiquement, de programmer des systèmes réactifs.

Comme le souligne l'auteur de [39], avec l'avènement de l'informatique et du modèle de programmation séquentiel, nous assistons depuis près de 30 ans à une abstraction croissante de la sémantique des langages de programmation. Nous avons ainsi pu voir apparaître des langages fortement structurés (Fortran, Pascal), des langages logiques (PROLOG) et fonctionnels (LISP), pour aujourd'hui vivre l'ère de la programmation objet avec des langages comme SMALLTALK (1978), C++ (1986) ou Eiffel (1992).

Pour le contexte spécifique de la programmation temps-réel de systèmes parallèles, les langages de programmation ont également évolué vers des sémantiques descriptives simplifiant la modélisation de système réactif comme pour l'exemple des langages synchrones (SIGNAL, LUSTRE), ou bien celles favorisant l'expression d'un parallélisme de données ou de flux (HPF, pC++, etc.) pour les architectures parallèles. L'élévation du niveau d'abstraction offre des solutions intéressantes à des problèmes très variés comme la portabilité des programmes et leur fiabilité, le rendement de programmation ou l'optimisation des performances. Le rôle des techniques de compilation, dans ce contexte où le niveau de modélisation cherche à s'éloigner des modèles d'exécution, devient alors particulièrement critique et complexe.

1.2.2.1 Les langages parallèles

1.2.2.1.1 Quelques exemples et critères de classification

Nous pouvons constater que les langages parallèles ont jusqu'à présent essentiellement servi à la programmation de systèmes massivement parallèles (basés sur des super-calculateurs ou, plus récemment, sur des réseaux de stations de travail distribuées). Ils se distinguent soit par des enrichissements fonctionnels apportés aux langages séquentiels structurés existants, principalement le Fortran et le C, soit par des langages (ou extensions) plus spécifiquement dédiés introduisant généralement un fort niveau d'abstraction. Parmi les formes de parallélisme que cherchent à exprimer ces langages, étudions plus particulièrement le paradigme SPMD qui semble être le plus naturel et le mieux exploré.

Nous constatons un très grand nombre d'extensions parallèles aux langages séquentiels comme HPF, Vienna Fortran, Fortran D, MP Fortran, Fortran 90 pour le Fortan ainsi que MPL, C*, HyperC, Split-C, DPCE pour le C (et pC++, Mentant pour le C++). Certaines de ces extensions se destinent plus particulièrement à des formes ciblées de parallélisme comme le partitionnement SPMD (PCF Fortran) ou le parallélisme de type SIMD (MP Fortran, MPL) ou MPMD (pC++, Mentat). Nous pouvons également citer les extensions de langages structurés sous forme de librairies de fonctions dédiées à l'exploitation du parallélisme (généralement de manière très explicite ce qui situe ce type d'extension comme faiblement abstraite). Citons, pour exemple, les librairies PVM ou MPI qui introduisent par ailleurs la notion de machine virtuelle plus spécifiquement destinée aux systèmes distribués. Par ailleurs, certaines extensions se destinent plus spécifiquement à des catégories d'architectures comme celles à mémoire partagée (PCF Fortran) ou distribuée (HPF). Dans [40] (p. 61-75), nous trouvons également une classification de ces extensions de langage suivant le niveau d'abstraction qu'ils offrent alors que dans [41], l'auteur préfère classer les langages parallèles par domaine d'abstraction : logique avec le PARLOG comme extension du PROLOG, fonctionnel (basé sur les flots de données) ou orienté-objets. Ces remarques soulignent que les critères de classification sont nombreux et leur portée souvent difficile à cerner tant les langages évoluent rapidement (et viennent

souvent emprunter des particularités tirées d'autres langages). En revanche, nous pouvons constater que les extensions de langage introduisant un premier niveau d'abstraction, s'appuient sur l'idée récurrente de simplifier l'expression du calcul matriciel (multi-dimensionnel) ce qui suggère ainsi naturellement un parallélisme de données.

Au-delà des extensions de langages séquentiels, nous trouvons un nombre important de langages parallèles aux noms aussi exotiques que OCCAM, ABCL/1, Prelude, PCN, Linda, Parallel Sets, SequenceL, etc. Le haut niveau d'abstraction généralement constaté pour ces langages permet d'exprimer aussi bien un parallélisme de contrôle que de flux avec le support de plus en plus répandu de la notion d'objets concurrents (ou encore acteurs, tâches) combinés à des formes plus ou moins explicites de parallélisme de type MPMD, pipelinées, etc. Leur étude et leur comparaison sortent du cadre de ce mémoire. Nous pouvons cependant constater que certaines spécificités de ces langages sont progressivement intégrées dans les mises à jour d'extensions de langages comme pour le récent standard HPF v2 qui intègre les constructions du type TASK_REGION permettant de regrouper l'exécution de certaines tâches sur un sous-ensemble de processeurs du système parallèle.

1.2.2.1.2 Le parallélisme de données

Arrêtons nous un instant sur les mécanismes mis en place par les extensions du C/Fortran afin de supporter un parallélisme de données (SIMD/SPMD). Cela nous paraît être une piste logique pour l'exploitation du macro-parallélisme offert par le C80 et l'utilisation conjointe des PP et du DMA. Nous nous intéressons plus particulièrement à la portée des constructions syntaxiques permettant d'exprimer la notion de localité des données avec la notion d'alignement ou celle guidant la distribution transparente des données sur les différents processeurs de l'architecture que nous offrent plus particulièrement les extensions HPF.

Historiquement ([42]), l'avènement du FORTRAN 90 dérivant des travaux autour du langage APL a permis (entre autres choses) de traiter d'une manière simplifiée les tableaux (aspect que nous retrouvons désormais communément dans le contexte du calcul scientifique avec l'exemple de MATLAB). Ensuite, diverses extensions au Fortran 90 ont

été proposées avec l'introduction des directives pour l'alignement et la distribution des données, le Fortran HPF v1 unifiant et standardisant les mécanismes proposés. La première mouture de HPF vise un parallélisme SIMD et s'appuie sur une machine virtuelle correspondant à un tableau rectangulaire (ou rectiligne) de processeurs. Dans ce contexte, le rôle du compilateur consiste à projeter les calculs et les données associées sur cette machine virtuelle, afin, dans une deuxième étape, d'optimiser l'implantation physique des traitements et données et d'encapsuler les entrées/sorties nécessaires.

Des directives explicites comme FORALL ou INDEPENDENT permettent d'imposer au compilateur d'effectuer les calculs et assignations des tableaux pour le bloc de programme considéré de manière parallèle sur un intervalle paramétrable d'indices (dans chaque dimension). Pour permettre cette parallélisation, le langage définit la notion de fonction PURE qui n'introduit pas d'effet de bord (comme le cas des dépendances de données entre les calculs parallèles d'une même expression).

Les directives ALIGN et DISTRIBUTE sont certainement les plus complexes à manipuler mais elles introduisent de puissants mécanismes pour structurer le partitionnement des données dans le système. Dans cette optique, l'alignement des données a pour but de définir un espace d'adressage virtuel permettant de regrouper les données adjacentes nécessaires à l'exécution d'une boucle ou d'une procédure afin de réduire le coût des communications inter-processeurs lorsque certaines valeurs de variables ne résident pas dans la mémoire locale. Cette notion d'alignement existe également pour les extensions C. Citons notamment l'exemple du qualificatif *shape* de DPCE (Data Parallel C Extensions). Par comparaison, HPF va plus loin dans les mécanismes d'alignement avec le support des décalages (les valeurs d'indices I du tableau A sont alignées avec les valeurs des indices I+2 du tableau B), d'expressions arithmétiques indicielles complexes (les données du tableau A sont alignées avec celles des indices impairs du tableau B), avec sa capacité à combiner les dimensions (alignement de lignes d'un tableau 2D avec les colonnes d'un autre tableau), ou avec la notion de réplication des données d'une dimension d'un tableau sur la dimension d'un autre tableau. Par ailleurs, nous trouvons également la notion de *Template* qui permet d'aligner la distribution des éléments d'une variable multi-dimensionnelle sur un espace d'adressage virtuel abstrait dans le sens où l'alignement ne

dépend pas de celui d'une variable existante. Techniquement, cette notion trouve notamment un intérêt pour l'alignement de tableaux définis par rapport à leur zone d'intersection.

Une fois les relations d'alignement définies, la primitive DISTRIBUTE permet de préciser la manière dont les groupes de données sont distribués sur l'architecture virtuelle. Nous pouvons citer l'exemple d'une distribution consistant à diviser par bloc (1D ou 2D) la quantité de données alignées, chaque bloc étant assigné à un processeur. Nous trouvons également la distribution cyclique qui consiste à espacer l'affectation des indices consécutifs d'un tableau par le nombre de processeurs (si 3 processeurs sont présents, le premier processeur reçoit les données d'indices 1,4,7, le suivant 2,5,8,...).

La directive DISTRIBUTE se situe à un niveau d'abstraction moins élevé que ALIGN et permet d'optimiser le coût des échanges de données inter-processeurs (suivant la topologie de la machine ou les spécifications techniques des liens de communication) comme pour le cas des algorithmes de type voisinage ayant des effets de bords ou bien pour le cas de variables répliquées sur l'ensemble des processeurs (exemple de LUT) et qui doivent être mis à jour afin de refléter une modification.

Pour l'exemple des effets de bord, nous soulignons que les compilateurs FORTRAN actuels sont capables d'analyser statiquement la portée des accès faisant référence à des données qui sortent de celles alignées en mémoire locale et, de là, optimisent automatiquement la distribution en répliquant les données périphériques en fonction de la largeur/hauteur de la fenêtre de convolution pour notre exemple (les spécifications HPF parlent de *stencil* ou élément structurant d'un opérateur). Lorsque l'estimation de la quantité de données à recopier est difficilement quantifiable (cas d'un stencil dont la géométrie est établie pendant l'exécution), le programmeur dispose de l'attribut optionnel SHADOW pour dimensionner la quantité exacte de données périphériques à dupliquer afin de mieux maîtriser les communications et de réduire leurs coûts. Ces quelques exemples montrent bien la souplesse et la complexité des mécanismes que nous rencontrons pour les langages parallèles "à parallélisme de données".

1.2.2.2 Les langages synchrones

Nous pouvons définir les langages synchrones tels que ESTEREL, LUSTRE ou SIGNAL comme des langages (ou formalismes dans la mesure où certains exemples s'appuient sur une description graphique comme les Statecharts) ayant pour vocation de favoriser un haut niveau d'abstraction pour la description de systèmes synchrones et réactifs. Pour ces systèmes, la notion de temps apparaît comme une suite d'événements à partir desquels sont définies des relations temporelles comme la notion de synchronisme (simultanéité de deux événements), des relations d'ordre (avant/après) ou de type sous/sur-échantillonnage des instants. Le niveau d'abstraction recherché permet ainsi essentiellement d'exposer un parallélisme de flux (notamment pour ESTEREL). Les notions de cadence et de latence, plus particulièrement caractéristiques des systèmes communément désignés comme "temps réel" (ou "sous contraintes temps réel"), complètent (plutôt que s'opposent à) la notion de système réactifs. Dans ce sens, de tels langages se distinguent conceptuellement des langages parallèles bien que nous trouvions des techniques de génération de code sous-jacentes qui permettent d'optimiser l'implantation du langage sur des systèmes parallèles (exemple de SynDEx décrit plus loin).

Les langages synchrones décrivent le comportement possible du système (en termes d'états et de règles de transition). La notion de temps physique (notamment au sens de la durée d'exécution) n'existe pas dans la mesure où les événements en sortie que nous pouvons comprendre comme des réactions du système (ou flots de données pour le cas de SIGNAL et de LUSTRE) sont considérés parfaitement synchrones des stimuli en entrée du système.

Cette notion de temps logique trouve un sens par la discrétisation de l'espace des transitions possibles. Le mécanisme se base sur la notion d'horloge virtuelle sur laquelle sont dérivées des horloges logiques associées à chaque expression du programme (exemple de SIGNAL). De là, chaque expression a le potentiel d'être exécutée d'une manière véritablement concurrente. Cette approche permet ainsi de séparer l'expression du parallélisme potentiel de celui effectivement mis en œuvre par des techniques de compilation et d'optimisation du code. Conceptuellement, c'est à ce niveau que sont déportées les véritables contraintes temporelles du système.

Les langages synchrones s'appuient sur une sémantique temporelle rigoureuse qui permet de vérifier la validité du système (ou de valider la vie du graphe des transitions si nous nous référons aux formalismes des réseaux de Pétri). Cette vérification est essentielle pour garantir la cohérence de la modélisation et s'appuie sur le déterminisme statique de la modélisation (au sens où, par exemple, la notion de rupture du séquencement logique des différents états possibles avec le support des mécanismes d'interruption par exemple, est proscrite de la modélisation au risque d'introduire un blocage du système). La compilation de ces langages produit généralement des automates à états finis implantés avec des langages séquentiels comme le C (à la manière de Lex et Yacc) et fige le comportement du système. Historiquement, les langages LUSTRE et SIGNAL font suite à ESTEREL et apportent la notion de flots de données synchrones qui introduit notamment des règles plus strictes sur les conditions de déclenchement des nœuds. Nous pouvons citer l'article [42] comme référence pour une comparaison plus fine de ces langages.

1.2.3 Les plates-formes de développement intégrées

Depuis quelques années, nous voyons l'émergence d'un grand nombre de plate-formes pour la simulation et la génération automatique de programmes plus particulièrement destinés aux applications de traitement de signal 1D. La plus populaire d'entre elles est sans doute le module Simulink de l'environnement MATLAB (essentiellement pour sa fonction de simulation). Nous trouvons également d'autres ateliers comme Hypersignal/OORVL d'Hyperception [44], SPW de Cadence, COSSAP de Synopsis, CapCASE de Matra [45], ou encore DSP Station de Mentor Graphics, ou bien encore LabView de National Instruments.

Ces systèmes s'appuient sur une modélisation graphique à partir de blocs élémentaires pré-programmés que l'utilisateur inter-connecte. En outre, ils permettent désormais la génération de code sur des cibles matérielles de type station de travail RISC, DSP ou FPGA (*Field Programmable Gate Array*). Ces plate-formes s'appuient généralement sur une description à base de flots de données synchrones introduits par le monde académique et exploités avec des ateliers tel que Ptolemy (simulation/génération de code) ou les outils GRAPE et SynDEx plus particulièrement destinés au prototypage de systèmes complexes

(génération de code). Nous donnons un bref aperçu de ces plate-formes qui résument assez bien l'état de l'art des techniques pour la programmation de systèmes hétérogènes complexes (nous trouvons également d'autres systèmes tels que CODE de l'université du Texas à Austin ou HeNCE qui reposent tous deux sur la librairie PVM et se destinent donc plus particulièrement aux systèmes distribués).

1.2.3.1 L'approche SynDEx

La méthodologie A^3 (acronyme d'Adéquation, Algorithme, Architecture) sur laquelle l'atelier SynDEX repose (pour EXécutif Distribué SYNchrone) est développée sous la direction de Yves Sorel à l'INRIA [46][47]. Elle vise l'implantation statique d'algorithmes temps réel sur des systèmes parallèles comportant plusieurs composants hétérogènes. Du point de vue théorique, le processus d'adéquation s'appuie sur un formalisme unifié de transformations de graphes vers la parallélisation d'algorithmes et la distribution/le partionnement, l'ordonnancement des grains de traitement ainsi que le dimensionnement des ressources matérielles [48]. Le domaine de modélisation repose sur le langage synchrone SIGNAL déjà évoqué et qui, pour résumer, permet de décrire les différents flots de données et les relations temporelles des opérations constitutives du traitement. La génération de code depuis l'atelier SynDEx produit alors un exécutif reflétant l'ordonnancement performant issu du processus d'adéquation entre le potentiel de parallélisme de la description et celui, effectif, du graphe matériel de l'architecture considérée. L'ordonnancement qui intègre le coût de communication et l'exécutif généré dans le cas des DSP s'appuie notamment sur l'utilisation des DMA (le choix du partitionnement et de l'ordonnancement des opérations du programme intègre dans ce sens le recouvrement temporel des calculs avec celui des transferts afin d'optimiser l'utilisation du système). Actuellement, SynDEx supporte principalement les architectures C40, Sharc 21060, quelques processeurs Intel ainsi que les stations Unix. L'infrastructure se veut ouverte vers d'autres plate-formes (nous pouvons citer les recherches en cours pour la génération de code VHDL pour les ASIC et FPGA).

1.2.3.2 Le projet Ptolemy

Le très vaste projet Ptolemy mené sous la direction du professeur Edward A. Lee de l'université de Berkeley, débute en 1990[1] et introduit la notion de domaine de modélisation et de programmation graphique pour la simulation et la génération de code [49][50]. La plate-forme se veut à la fois fonctionnelle (elle sert à l'enseignement), ouverte et expérimentale. Elle se distingue par le fait qu'elle permet d'explorer différents domaines de représentation pour la description d'algorithmes. Elle autorise une modélisation hétérogène et hiérarchique d'applications de traitement de signal sur la base de différents domaines supportés. Cette idée, qui est reprise au sein des plate-formes commerciales actuelles (comme avec le produit SPW de la société Cadence), est motivée par le fait que les différents modules d'un algorithme complexe s'adaptent généralement mieux à telle ou telle méthode de modélisation suivant la nature des flots de données ou des contrôles nécessaires pour l'exécution des nœuds de traitement. Les ordonnanceurs (*schedulers*) des nœuds (aussi appelés "acteurs") sont perçus comme des cibles paramétrables pour les différents domaines (nous trouvons généralement différentes cibles par domaine en fonction de la nature même des données). Parmi les domaines supportés, le plus mature correspond au modèle de flots de données synchrones à fréquence variable (*Multi-rate Synchronous Data Flows* ou SDF) initié par le professeur Lee [51]. De plus, pour les algorithmes dont la durée des étapes de traitement varie en fonction de la nature des données, nous trouvons le domaine des flots de données dynamiques (DDF pour *Dynamic Data Flow*) qui n'est supporté que pour la simulation. Lorsque les acteurs du traitement nécessitent un niveau important de contrôle, la plate-forme autorise une description à partir de la spécification d'événements discrets ou des machines à états finis. Nous trouvons également des modèles intermédiaires entre une description à base de flux régulier et des modèles nécessitant un contrôle accru lors de l'exécution, de même qu'un modèle expérimental pour la gestion de flots synchrones et multi-dimensionnels [52][53]. La modélisation proposée pour le support des tableaux 2D va au-delà de la simple perception des matrices comme une simple suite de vecteurs 1D. Il s'agit en effet d'une approche

1. Et correspond au prolongement du projet Gabriel initié près de 5 années auparavant.

dégradée du problème que suggère un bon nombre de plate-formes comme pour Simulink ou Hyperception. Ici, la modélisation proposée intègre la notion de sous-matrice et favorise l'expression du parallélisme de données vers une génération ou simulation de code performant sur des systèmes multi-processeurs. Dans ce sens, l'approche permet d'ordonnancer d'une manière quasi-optimale les nœuds en fonction des contraintes sur la quantité minimale de données pour les directions horizontale et verticale nécessaires à l'exécution d'une itération de traitement. Elle vise également à dimensionner les buffers d'une manière optimisée ainsi qu'à étendre la notion de retard du domaine définie dans le domaine SDF 1D qui permet d'initialiser les flux de données avec certaines valeurs et d'intégrer la notion de données passées et futures. Ce dernier point sous-entend également la volonté d'intégrer, dans la définition des flux, les besoins d'adressage spécifiques des nœuds de traitement. Nous trouvons un résumé des orientations prises pour le support de ce modèle en cours d'exploration [53] ainsi qu'une approche expérimentale vers une technique permettant la définition compacte de graphe de dépendance ayant une configuration des nœuds et arcs (dépendances de données) régulière. Ces recherches visent notamment l'ordonnancement optimal d'applications pour le calcul matriciel sur des architectures de type systolique [54], et plus généralement, l'introduction de techniques d'ordonnancement véritablement performantes pour les flots multidimensionnels (MDSDF). Par ailleurs, l'intégration des différents domaines repose sur des recherches théoriques préalables sur l'intérêt pour la modélisation et, notamment, le déterminisme de l'exécution ou la possibilité de dériver des ordonnanceurs performants et dépourvus de sources de blocages.

Pour chaque domaine, la plate-forme propose des techniques génériques de partitionnement, d'ordonnancement, d'encapsulation des entrées/sorties, et, pour la simulation (ou la génération de code pour système Unix) d'intégrer des mécanismes de communication entre les différents domaines de représentation utilisés.

Pour la génération de code sur des processeurs spécialisés, le système ne s'appuie pour l'instant que sur le domaine SDF 1D et permet l'implantation sur TMS320C50/Motorola 56000 et FPGA à l'aide du C, de l'assembleur ou du VHDL. L'ordonnancement employé lors du processus de génération de code est ici statique et l'approche est conçue pour

supporter des systèmes multi-composants. Les heuristiques employées permettent d'optimiser la taille des tampons utilisés de même que, tout comme SynDEx, la génération de code pour DSP permet l'utilisation des DMA afin d'optimiser les performances.

1.2.3.3 Le système Grape II

La plate-forme expérimentale GRAPE pour *Graphical RApid Prototyping Environment* est développée depuis près de 10 ans au laboratoire ACCA (*Automatic Control and Computer Architectures*) de l'université catholique de Louvin sous la direction des professeurs Lauwereins et Peperstraete. Elle est intéressante à double titre.

Tout d'abord, il s'agit de la seule plate-forme supportant la génération de code pour l'architecture C80 ([55]). Ainsi, sur la base des entités de traitement en C ou assembleur PP que fournit l'utilisateur, le système permet de répartir les tâches aux différents processeurs, d'encapsuler l'utilisation des canaux de communication en supportant le DMA pour l'accès à la mémoire partagée et d'ordonnancer l'application.

Le deuxième niveau d'originalité concerne la modélisation des algorithmes qui se base sur une extension du domaine de représentation à base de flots de données synchrones [56]. La nature des flux doit être statique mais la fréquence de consommation/production des jetons (données) peut varier de manière cyclique et, dans ce sens, les auteurs parlent de modélisation CSDF (*Cyclo-Static Data Flow*). Il s'agit d'une première étape vers le support de flots ayant une dynamique de comportement complexe. Dans ce sens, nous pouvons souligner les recherches entreprises vers le support d'un modèle vers l'ordonnancement d'algorithmes complexes pour lesquels la durée d'exécution des n œuds dépend des données (il s'agit du modèle CDDF pour *Cyclo-Dynamic Data Flow*). Des travaux théoriques consistants entourent ces modèles et visent à garantir le déterminisme de la description ainsi que le dimensionnement optimal des tampons de données comme critère des heuristiques d'ordonnancements spécifiquement développées pour ces extensions du domaine SDF.

Dans la pratique, tout comme SynDEx et Ptolemy, l'environnement GRAPE est capable de projeter le graphe d'algorithmes sur les ressources matérielles d'une manière optimisée mais statique (des recherches en cours ont pour but d'apporter au système le support de

traitements reconfigurables). L'outil se base sur l'information des durées d'exécution des nœuds de traitement que doit fournir le développeur. De même, la conception générale de la plate-forme permet à l'utilisateur de contrôler et de raffiner la projection automatique de l'application. Le système se veut ouvert en terme d'architectures supportées et vise la réalisation de systèmes hétérogènes complexes de type FPGA/DSP. Pour les DSP, l'atelier supporte la génération du code pour le TMS320C40 et le Motorola 56000. Ainsi, à l'inverse de Ptolemy (et comme pour SynDEx), cette ouverture ne concerne pas le domaine de la modélisation qui se cantonne au domaine CSDF dans le cas mono-dimensionnel. Nous trouvons plus de détails sur cette approche dans [57].

1.2.3.4 Conclusions

Les trois plate-formes détaillées affichent des objectifs comparables et traduisent les véritables progrès réalisés en matière d'implantation de systèmes embarqués temps-réel. Si l'effort de recherche est considérable (déployé sur des dizaines d'années), il apparaît que ces ateliers requièrent encore des avancées sensibles pour une implantation (génération de code) véritablement flexible et performante d'applications complexes quelconques. A cet égard, il convient de souligner les efforts restant à faire pour le support performant de systèmes dynamiques, d'algorithmes à base de flots de données multidimensionnelles (dans le sens des récentes recherches entreprises pour la simulation d'algorithmes de traitement d'images avec Ptolemy) ou bien encore, de techniques automatisant l'extraction du parallélisme potentiel de la description algorithmique[1].

1. Ainsi avec l'exemple de SynDEx, dans [47], l'espace des combinaisons pour la description de l'algorithme est exploré manuellement en modifiant la granularité des opérations ou en orientant la description du langage SIGNAL.

1.3 Conclusions : Perspectives de contribution

Pour offrir des pistes de réflexion aux objectifs de généricité, de flexibilité et d'optimisation des performances définis au point 1.1.4 que ne proposent pas les solutions d'imagerie C80 existantes en faveur d'une productivité accrue du développement, nous avons donné un bref aperçu des pratiques qui définissent l'état de l'art de la programmation des systèmes parallèles. A ce stade, nous pouvons par conséquent dresser une synthèse des limitations actuelles et dégager des perspectives de contribution vers la recherche d'une méthodologie de développement originale et profitable.

Intérêt/faisabilité de la réalisation d'un compilateur

Les langages et principes de compilation associés que nous avons introduits, tentent d'apporter une réponse globale à la mise en œuvre logicielle en intégrant d'une part, l'extraction automatique du parallélisme à différents niveaux de granularité (au niveau des instructions (ILP pour Insutrction-Level Parallelism) pour le cas du Fortran par opposition à l'ordonnancement de fonctions C que vise SynDEx à travers SIGNAL) et, d'autre part, en permettant l'encapsulation plus ou moins explicite des mécanismes de communication qui interviennent entre ces nœuds.

Pour nos travaux, il nous semble que la piste de l'utilisation d'un langage haut niveau spécifique associée à diverses techniques de parallélisation/d'optimisation automatiques nécessite des compétences particulières (l'écriture d'un compilateur HPF est particulièrement ardue et requiert d'importants moyens humains) qui sortent des objectifs de ce mémoire et du cahier des charges initial.

Dans le contexte CIFRE, se pose également la question de l'intérêt commercial de l'introduction d'un nouveau langage, ainsi que de l'intérêt scientifique de l'adaptation d'une syntaxe existante au contexte de l'architecture C80 ou du "simple" portage d'un compilateur donné (ne fut-il qu'un pré-processeur s'appuyant sur les outils standards de compilation du processeur). De plus, l'expérience montre qu'il apparaît difficile de

combiner la généricité de l'abstraction des langages parallèles avec une exploitation véritablement performante des spécificités des architectures, ce qui tend alors à réduire la portée des travaux et limite les gains en performances. Pour cette raison, nous préférerons exposer au chapitre 2 un ensemble de techniques d'optimisation génériques connues qui nous semblent les mieux adaptées à l'exploitation du micro-parallélisme des nouvelles architectures DSP. Nous nous concentrerons davantage sur la parallélisation des "n œuds" de traitement ayant une granularité moyenne (qui correspond à l'implantation assembleur d'opérateurs de traitement d'images bas niveau). Le choix de porter l'étude scientifique à ce niveau de granularité correspond à la démarche également proposée à travers les ateliers de génie logiciel (AGL) pour DSP que nous avons évoqués.

La piste des AGL pour DSP

Concernant ces plate-formes, soulignons tout d'abord que la génération automatique de code en est encore au stade expérimental, notamment dans le domaine du traitement d'images. Elles visent essentiellement le placement et l'encapsulation optimisés des nœuds de traitement avec les moyens de communication existants pour une architecture parallèle donnée. Sur le plan théorique, ces systèmes s'appuient, de manière récurrente, sur la modélisation par flots de données synchrones monodimensionnels qui semble être la plus mature. Ils utilisent cette représentation pour projeter le graphe de dépendance algorithmique sur le graphe de ressources matérielles. Si les heuristiques de cette transformation visent notamment à exploiter au mieux l'utilisation des processeurs en optimisant les moyens de communication (avec des notions comme le *clustering* des nœuds pour l'amélioration de la localité des données et le *buffering* des flux pour favoriser l'exécution parallèle des traitements), ces systèmes ne supportent pas de manière transparente la transformation du graphe algorithmique vers une première étape d'optimisation de l'algorithme. Citons pour exemple le schéma de parallélisation SPMD qui, sur ces plate-formes, nécessite le partitionnement manuel des données et l'instanciation explicite des différentes occurrences du nœud ayant en charge de traiter les sous-blocs d'informations. C'est après cette étape manuelle qu'intervient l'automatisation

des processus de placement et d'ordonnancement. L'exploration de l'espace des solutions possibles pour l'étape de projection matérielle est bien contrainte par la forme explicite du parallélisme qu'affiche la représentation initiale de l'algorithme. Cette situation est en tout point comparable à la programmation depuis un langage haut niveau qui nécessite la plupart du temps de guider les outils pour obtenir de meilleures performances en modifiant la forme du programme sans changer sa substance (exemple du déroulage de boucle). Par ailleurs, si, sur le plan des performances, ces ateliers sont capables de générer un code intégrant le fonctionnement asynchrone des DMA (avec une gestion de type double buffering), ils établissent généralement des communications de type point-à-point entre les nœuds. Or, le modèle de mémoire partagée illustré avec l'architecture du C80 ne se prête pas à ce type d'encapsulation des mécanismes de gestion des flux.

Ces plateformes apportent donc une aide intéressante au développement de systèmes complexes mais ne résolvent pas la problématique essentielle de l'aide au partitionnement SPMD et de l'aide à la mise en œuvre du DMA qui, pour le cas du C80, nous apparaissent comme des orientations incontournables en faveur d'une simplification de la programmation et d'une meilleure utilisation des ressources. Dans ce contexte, une piste intéressante consiste à apporter des extensions directes à l'une ou l'autre des plate-formes pour le C80. Sur le plan pratique pourtant, il nous semble qu'une telle orientation impliquerait une durée de prise en main des outils qui n'est pas compatible avec la charge de travail induite par les objectifs initiaux de cette thèse visant également à explorer le domaine applicatif. Le support inachevé du C80 dans l'environnement GRAPE et l'expérience aux résultats mitigés (de l'aveu même des responsables du projet) en terme de performances et de simplicité de modélisation pour la génération de code à destination d'une architecture multi-processeurs MIMD à mémoire partagée (comparable au C80) sur Ptolemy [58], sont autant d'éléments qui corroborent cette analyse.

Orientations de recherche et perspectives de contribution

Après l'argumentation que nous venons de développer autour du problème spécifique de la gestion des flux et de la parallélisation SPMD pour le C80, nous proposons d'introduire nos orientations de recherche à l'aide du tableau 1-6.

Table 1-6. Améliorations de la gestion des entrées/sorties : objectifs et perspectives

Objectifs pratiques généralistes	Réponses/Orientations scientifiques (✞Choix pratiques)	Apports à l'état de l'art
Généricité : - Méthodologie de **modélisation unifiée** : support des principales classes d'opérateurs de **traitement d'image bas niveau (**point à point, diadique, filtrage FIR, IIR, convolution&correlation par masque 2D)	- Modélisation par **flots de données synchrones à cadence variable (MSDF ou Multi-rate Synchronous Data Flow) pour le cas 2D**	Extension de la **modélisation MSDF pour le cas 2D (MSD2DF)**
Optimisation des performances : - **Parallélisation des traitements** pour le contexte **multi-processeurs** - **Réduction du coût des communications** (transferts DMA)	- **Partitionnement SPMD automatique** : découpage des buffers et utilisation synchrone des requêtes DMA (asynchrone) - Exploitation de l'asynchronisme du DMA : **récouvrement temporel des phases de calcul et de transfert** avec le support du double ou triple buffering - Introduction d'**un modèle de performance des** traitements et **transferts** - Réduction des transferts : amélioration de la localité des données par **chaînage des opérateurs** - **Minimisation de l'impact des caches d'instructions**	- Extension à la modélisation MSDF de la notion HPF d'**alignement des données** pour le découpage des buffers 2D **lors de la parallélisation SPMD** (notion de *stencil* ou *d'élément structurant*) - **Modèle de performance unifiant performances des opérateurs** et **coût des transferts** (issu d'une représentation MSD2DF des flux mise en œuvre en // avec le partitionnement SPMD ⟹ "MSD2DF//") - Réduction de l'impact des transferts : support du **triple buffering** et **minimisation de l'impact des caches d'instructions lors du chaînage**
Flexibilité : - **Indépendance** de la **taille des images** et du **nombre de processeurs** - **Reconfiguration dynamique**	- **Modélisation analytique du coût des transferts en fonction du nombre de processeurs, de la taille des images et de celle des tampons internes** (paramétrés par l'utilisateur) (✞ Implantation des mécanismes de gestion des E/S sous forme d'une**librairie en C** mettant en œuvre la méthodologie "MSD2DF//" au "***runtime***")	- **Reconfiguration dynamique optimisée**
Portabilité : - Support de plusieurs architectures DMA/SRAM	- **Analyse des caractéristiques architecturales** des DMA et structures de mémoires internes associées (✞ Implantation du coeur des mécanismes de partitionnement en langage C portable)	- Plus grande **ouverture dans l'intégration des spécificités des architectures DMA** (plus grande unification dans la méthodologie de développement, exemple du support paramétrable du nombre de dimensions qu'exploite le DMA ou du nombre de bancs de mémoire interne)

A gauche, ce tableau reprend les quatre objectifs principaux en faveur d'une simplification de la mise en œuvre définis au point 1.1.4 (généricité, amélioration des performances, flexibilité et portabilité) et propose des réponses précises à ces différentes

recommandations avec la colonne du centre. Aussi, par souci de clareté, nous précisons avec la colonne de droite les apports attendus de nos orientations scientifiques compte tenu de l'état de l'art.

Pour répondre à la question de l'unification de la représentation que soulève la notion de *généricité*, nous nous inspirons de l'état de l'art et choisissons une représentation par flots de données synchrones multi-cadences (" *multi-rate*") qui nous semble bien adaptée aux opérations bas niveau offrant une certaine régularité dans l'accès aux données. Celle-ci permet, grâce à cette régularité, de réduire le coût de la gestion de l'encapsulation des entrées/sorties. Dans notre contexte de traitement d'images, il convient d'étendre la modélisation traditionnelle par flots de données au cas 2D, notamment afin de tirer au mieux parti du support multi-dimensionnel qu'offrent aujourd'hui les DMA et celui du C80 en particulier.

Sur le plan de *l'optimisation des performances* pour laquelle nous cherchons à paralléliser les traitements avec le schéma de partitionnement SPMD ainsi qu'à réduire le coût des transferts, nous proposons tout d'abord un découpage transparent des données qui conduit à une gestion synchrone des requêtes DMA (lesquelles conservent leur caractère asynchrone) afin de minimiser le coût de la gestion des entrées/sorties. Concernant l'opération de partitionnement, nous proposons d'intégrer la notion d'alignement des données du langage HPF qui entoure l'idée d'élément structurant (ou *stencil*). Cette démarche a pour but de diviser, de manière cohérente, les flux bidimensionnels issus de la représentation par flots synchrones. Dans ce contexte et à notre connaissance, il s'agit d'une problématique véritablement innovante puisque non encore abordée par l'état de l'art.

Le deuxième aspect sur lequel nous voulons concentrer nos travaux concerne l'optimisation du coût des transferts partitionnés. Pour cela, nous proposerons un modèle de performance des communications (et des traitements) issu de la gestion synchrone des flux que nous visons. Pour optimiser ce modèle, nous chercherons à chaîner l'exécution des opérateurs de traitement dans le but d'exploiter au mieux la localité des données et de

réduire les transferts explicites. Dans la modélisation, nous intégrerons également l'impact implicite de la gestion des caches d'instructions. Sur le plan de l'apport scientifique, nous choisissons de proposer un modèle de performance qui dérive des extensions originales que nous donnerons à la description par flots synchrones. En outre, nous introduirons des pistes pour la réduction du coût de la gestion des caches d'instructions que ne proposent pas les AGL présentés dans ce chapitre (avec notamment l'exemple du support pour le triple buffering et, nous le verrons, avec l'intégration des caractéristiques de certaines techniques d'optimisation au niveau instructions).

Pour satisfaire à la notion de *flexibilité*, nous proposons d'orienter notre modèle analytique de performances de manière paramétrée. Le coût des communications partitionnées sera notamment fonction de la taille des images traitées, du nombre de processeurs impliqués dans le partitionnement et de la taille des buffers internes pour la gestion des caches avec le DMA. Concernant ce dernier paramètre, soulignons que nous ne chercherons pas à dimensionner automatiquement la taille des tampons internes (l'état de l'art nous enseigne en effet qu'il est possible de déduire la taille des buffers d'après la représentation multi-cadence synchrone). Il s'agira pour nous d'une donnée du problème.

Sur un plan plus pratique, pour répondre à la question de la reconfigurabilité dynamique des traitements, nous implanterons notre approche pour le partitionnement et l'encapsulation transparente de la gestion des flux (issue de notre modèle de performance) sous forme d'une librairie de fonctions C pré-compilées mettant en œuvre l'analyse automatique et dynamique de la modélisation par flots synchrones. Une telle architecture logicielle dépasse ainsi l'état de l'art actuel qui vise une génération de code statique.

Pour finir, sur le plan de la *portabilité*, nous chercherons à analyser plusieurs types d'architectures DMA/mémoires internes afin de favoriser la portée de notre méthodologie et d'intégrer au mieux les capacités architecturales (avec, par exemple, le support du triple buffering ou l'analyse des besoins pour la gestion des flux dans le cas du C6X). Par ailleurs, le choix de la mise en œuvre en C des mécanismes de partitionnement et d'encapsulation des flux nous offre un gage de portabilité.

2 Méthodologies de développement

Ce chapitre introduit plus particulièrement la motivation et les fondements théoriques d'une méthode originale de génie logiciel ayant pour but l'automatisation et l'optimisation de la gestion des flots de données pour des architectures multi-DSP à mémoire partagée. L'objectif de cette méthodologie est d'exploiter au mieux le couplage des co-processeurs de transferts mémoire avec les mémoires statiques internes dédiées aux données. Les enjeux sont multiples puisque nous visons à la fois une modélisation générique et flexible pour les algorithmes de traitement d'images bas niveau ainsi qu'une amélioration des performances avec, notamment, le chaînage des opérateurs de traitement. Pour cela, nous introduisons une méthodologie en faveur d'une gestion transparente, optimisée et structurée des caches de données logiciels. Les applications de référence présentées par la suite reposent sur la mise en œuvre de cette méthodologie et constituent alors des études de cas de notre approche.

L'implantation d'algorithmes sur DSP se décompose généralement en deux étapes : d'une part la programmation des opérateurs de traitement et, d'autre part, la gestion des flux avec le DMA. Nous présentons dans une première partie quelques techniques d'optimisation génériques pour une implantation performante des opérateurs. Cette présentation nous permet de souligner que ces méthodes augmentent généralement la taille du code, ce qui a pour conséquence d'induire un surcoût dans la gestion implicite des caches d'instructions. Nous soulignerons que ce phénomène apparaît notamment dans le cas de chaînes d'opérateurs algorithmiques que nous chercherons plus particulièrement à implanter. Dans ce contexte, nous proposerons des solutions pour réduire la granularité des opérateurs optimisés suivant les techniques présentées.

Dans une deuxième partie, nous présentons notre approche originale de gestion des flux pour l'encapsulation des entrées/sorties de données des opérateurs de traitement qui

composent les chaînes algorithmiques. Pour ce faire, nous présentons les éléments de la modélisation qui permettent d'établir la portée générique de la méthode visée et adressons plusieurs problèmes relatifs à la mise en œuvre pratique de l'approche. Ensuite, nous développons un modèle mathématique original pour le partitionnement des données et la gestion synchrone des requêtes de transfert DMA. Nous introduisons plusieurs pistes pour minimiser le coût des transferts en intégrant notamment, dans notre modèle, les recommandations en faveur d'une minimisation de la granularité des nœuds présentée dans la première partie.

Enfin, nous consacrons la dernière partie de ce chapitre à la présentation d'un modèle de performance des durées d'exécution globale des chaînes algorithmiques (sur le plan des traitements et des entrées/sorties) qui dérive de notre approche originale pour la gestion des flux bi-dimensionnels dans le contexte multi-processeurs du C80.

2.1 Techniques d'optimisation pour la programmation des opérateurs

Les techniques que nous détaillons ici sont génériques et s'appliquent aussi bien aux architectures RISC récentes qu'à celles des DSP. Les gains qu'offrent ces méthodes dépendent de l'application et des ressources matérielles du processeur ciblé. Une combinaison de ces techniques est souvent nécessaire pour atteindre les meilleures performances (comme il est souligné dans [59][60][61]), nous illustrerons cette remarque par la suite. Soulignons ici que l'ordre dans lequel il convient d'appliquer ces techniques pour une performance optimale dépend véritablement de l'algorithme étudié. Dans cette partie, nous détaillons individuellement les 4 techniques d'optimisation les plus significatives que nous avons mises en œuvre pour l'implantation de la librairie ainsi qu'au sein des deux applications de référence. Ces méthodes sont : le déroulage de boucle, l'utilisation des capacités SIMD, le pipeline logiciel et l'utilisation de tables pré-calculées (LUT). Pour chaque technique, nous nous efforçons de présenter les principales caractéristiques de mise en œuvre. Dans cette démarche, nous montrons que l'utilisation

des trois premières techniques ne favorise pas la compacité du code et proposons différentes approches pour y remédier. Mais d'abord, justifions brièvement l'intérêt de l'optimisation de la granularité des opérateurs.

2.1.1 Intérêt de l'optimisation de la granularité des nœuds

L'optimisation de la granularité des opérateurs algorithmiques (ou nœuds) ne vise pas tant l'optimisation de l'espace de la mémoire programme qu'occupent l'implantation (bien qu'il s'agisse d'un critère économique généralement important pour les systèmes embarqués et le coût des mémoires connexes), mais plutôt l'amélioration des performances par la réduction de l'impact de la gestion implicite des caches d'instructions matériels. Pour justifier ce point, nous pouvons intuitivement penser que plus les opérateurs comptent d'instructions, plus les mécanismes de mise à jour du/des cache(s) d'instructions risquent d'être solliciter et d'induire une réduction des vitesses de traitement.

Dans notre contexte, le gain en performance que permet la recherche d'une faible granularité est particulièrement important dans notre démarche globale pour l'optimisation des flux (abordée au point 2.2) selon laquelle nous chercherons à chaîner séquentiellement l'exécution des nœuds de manière à exploiter la localité des données (réduction du coût des transferts de données explicites (données traitées par l'algorithme).

En effet, afin d'augmenter les performances, des sous-blocs de l'image externe sont "cachés" par le biais du DMA dans la mémoire interne où les accès aux données sont nettement moins coûteux (souvent d'un ou plusieurs ordres de grandeur). La mise enœuvre logicielle de ce principe avec l'objectif du chaînage nécessite toutefois d'exécuter l'ensemble de la chaîne après le traitement de chaque bloc de données en mémoire interne (après chaque jeu de transferts en entrée et en sortie). Or, à chaque re-démarrage de la chaîne, le contenu du cache d'instructions se trouve modifié (il contenait les instructions des derniers n œuds exécutés qu'il faut remplacer avec le code associé aux premiers opérateurs de la chaîne). Cette gestion peut induire une diminution significative des performances proportionnelle à la granularité moyenne de l'ensemble de la chaîne.

A titre d'exemple, le rechargement d'un bloc de cache d'instructions du PP C80 par requête DMA diminue de 30 % les performances d'une chaîne ne comportant qu'un n œud implantant un "NON" logique[1]. Pour le C80, l'importance de l'optimisation de la taille du code est encore soulignée au regard de la petitesse du cache d'instructions des PP (256 instructions VLIW pour le C80) et du nombre de blocs nécessaire à l'exécution du patron de traitement que nous estimons à un bloc (64 instructions) en fonctionnement normal (hors initialisation du traitement).

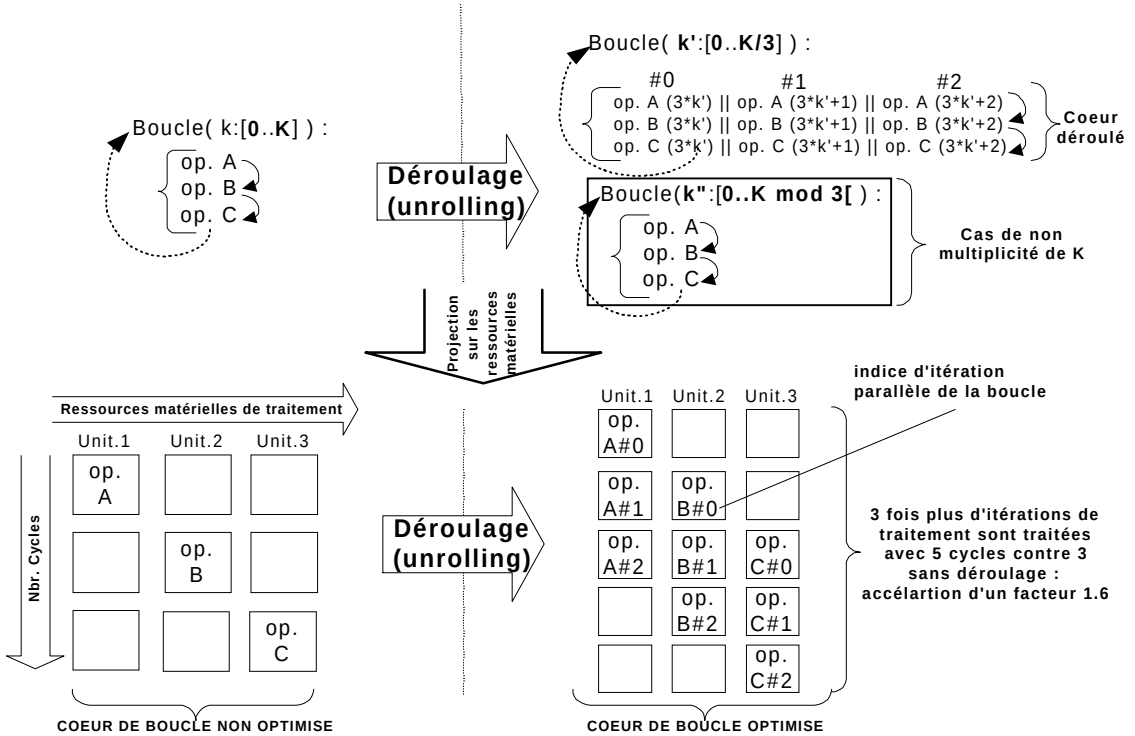
La recherche d'une optimisation de la granularité vise donc bien à optimiser le coût caché (explicite) de la gestion des caches d'instructions, notamment dans le contexte du chaînage des opérateurs. La présentation des techniques d'implantation que nous menons avec les paragraphes suivant intègre cette dimension non-triviale et, anticipant quelque peu la section 2.2, propose dans ce sens des aménagements aux techniques de mise en œuvre en faveur d'une réduction de la granularité. Etudions la première technique d'implantation très connue : le déroulage de boucle.

2.1.2 Le déroulage de boucle

Le déroulage de boucle est une technique ancienne dont le principe, très simple, consiste à dupliquer un certain nombre de fois l'écriture du contenu d'une boucle avec pour objectifs d'une part de réduire le coût des opérations ayant en charge la gestion de la boucle en l'absence de contrôleur de boucle matériel (décrémentation du compteur de boucle et branchement conditionné par cette même variable) et, d'autre part, de suggérer une meilleure utilisation des ressources parallèles avec l'opportunité généralement laissée au compilateur ou au processeur (dynamiquement) de mieux entrelacer l'exécution temporelle des opérations du coeur de boucle déroulé. Sans cette approche explicite, l'entrelacement des opérations reste limité par la rupture du séquencement de l'exécution liée à l'opération de branchement conditionnel. La latence d'une itération de boucle peut ainsi être réduite, le nombre d'itérations total de la boucle étant divisé par le nombre d'itérations déroulé comme l'illustre la Figure2-1. Les performances sont clairement améliorées.

1. Il s'agit d'un exemple d'autant plus significatif que la durée des transferts est supérieure à celle du traitement.

Figure 2-1. Principe du déroulage de boucle (*loop unrolling*)



A contrario, cette technique engendre un code moins compact du fait de la réplication des instructions du coeur de boucle ainsi que le traitement optionnel pour le cas où le nombre effectif d'itérations de boucle n'est pas multiple du nombre de déroulage. En effet, dans le cas où la quantité *K* de données à traiter n'est pas multiple du nombre d'itérations *(K/3)* physiquement exécutées par la boucle déroulée, nous devons, en toute rigueur, compléter l'implantation de la boucle par une partie non-déroulée qui gère les *k"* itérations non multiples (sur l'exemple, *k"=[0 .. K mod 3]*). Il en résulte que le code déroulé occupe généralement 3 fois plus d'espace mémoire que sa version non optimisée.

Pour favoriser la granularité de l'implantation des opérateurs, nous préconisons alors de ne pas gérer le cas de non multiplicité. Pour ce faire, nous imposons des contraintes sur la multiplicité ainsi que sur la quantité minimum des données traitées.

Par ailleurs, dans la mesure où, dans la suite de ce mémoire, la programmation des opérateurs est directement réalisée en assembleur pour les DSP, nous avons pu contrôler précisément l'impact de cette technique sur l'utilisation des ressources registres. Il faut, en effet, souligner que la mise en œuvre de cette approche depuis un langage haut niveau sans connaissance a priori des capacités architecturales peut, à l'inverse, engendrer une

diminution importante des performances lorsque le coeur de boucle est déroulé un trop grand nombre de fois et que la quantité de registres disponibles devient insuffisante. Cette situation engendre l'intégration, par le compilateur, d'instructions pour l'empilement/le dépilement (ou "*spilling*") des résultats intermédiaires ce qui induit la plupart du temps une dégradation importante des performances. Pour cette raison, il convient, pour chaque architecture et chaque coeur de boucle, d'évaluer l'opportunité du déroulage de boucle et de borner le nombre de réplications en faveur d'un véritable apport de performances. Cette délicate analyse nécessite bien souvent l'intervention de l'utilisateur. De même, pour le cas des boucles imbriquées, nous avons parfois intérêt à favoriser le déroulage de la boucle la plus externe afin d'améliorer la localité des données. Si des outils précurseurs comme KAP [6][7] permettent ce niveau d'analyse, l'intervention du programmeur reste généralement incontournable. Nous concluerons en soulignant que cette technique générique peut aussi bien s'appliquer aux itérations de boucles bornées (statiquement ou dynamiquement) qu'aux boucles non bornées de type *do while* [62].

2.1.3 Utilisation des capacités SIMD

A l'instar du déroulage de boucle, l'utilisation des capacités SIMD (*Single Instruction, Multiple Data*) désormais généralisées au sein des processeurs RISC et DSP permet de traiter plusieurs itérations algorithmiques en parallèle et réduit le nombre d'itérations globales du coeur de boucle. Ici, les instructions ne sont cependant pas dupliquées puisque le parallélisme repose bien sur les capacités intrinsèques des instructions SIMD employées.

Tout comme pour le déroulage logiciel, l'exécution de plusieurs itérations en parallèle nécessite une gestion spécifique pour les cas où la quantité de données traitées ne s'avère pas multiple du nombre d'itérations traitées en parallèle. Là encore, afin de réduire l'occupation du code, nous imposerons des contraintes de taille qui seront par ailleurs parfaitement intégrées dans notre méthodologie de gestion des flux du point 2.2.

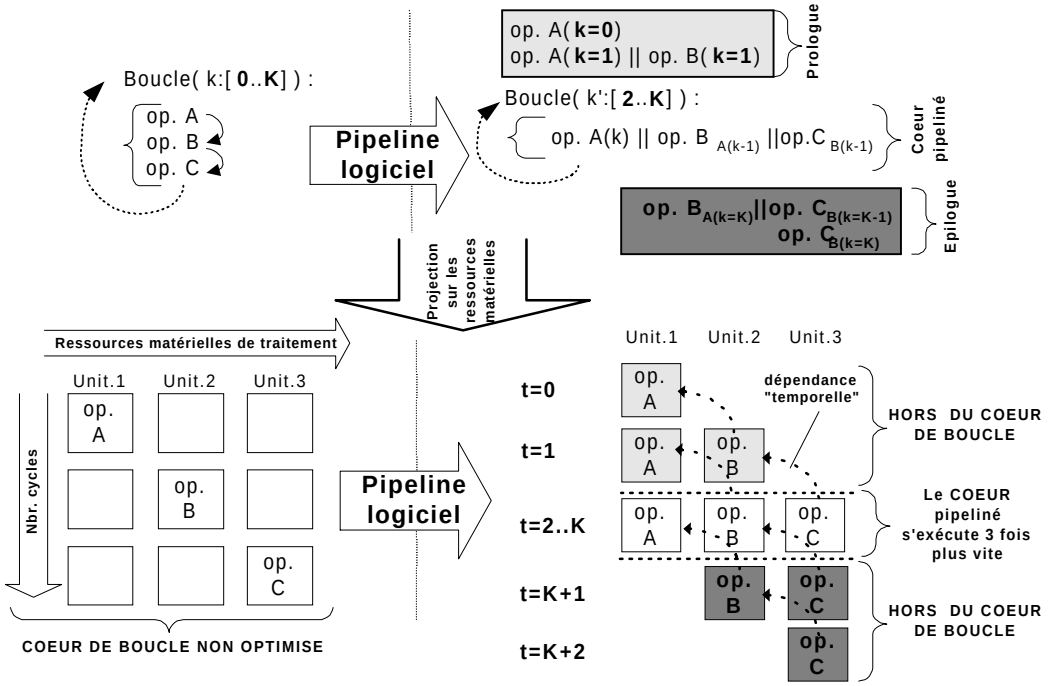
Suivant le niveau d'intégration des calculs SIMD de la cible, soulignons qu'il s'avère parfois nécessaire d'introduire une part non négligeable d'opérations spécifiques pour la gestion des sous-résultats au sein des registres de données (du type décalage ou masquage).

Cette gestion engendre un surcoût qu'il convient d'appréhender avant d'établir les bénéfices réels de l'approche. A l'inverse, soulignons que certaines optimisations deviennent généralement possibles pour compenser cet éventuel surcoût, avec l'exemple du chargement d'un mot qui permet de réduire l'impact des multiples chargements d'octets consécutifs.

2.1.4 Le pipeline logiciel

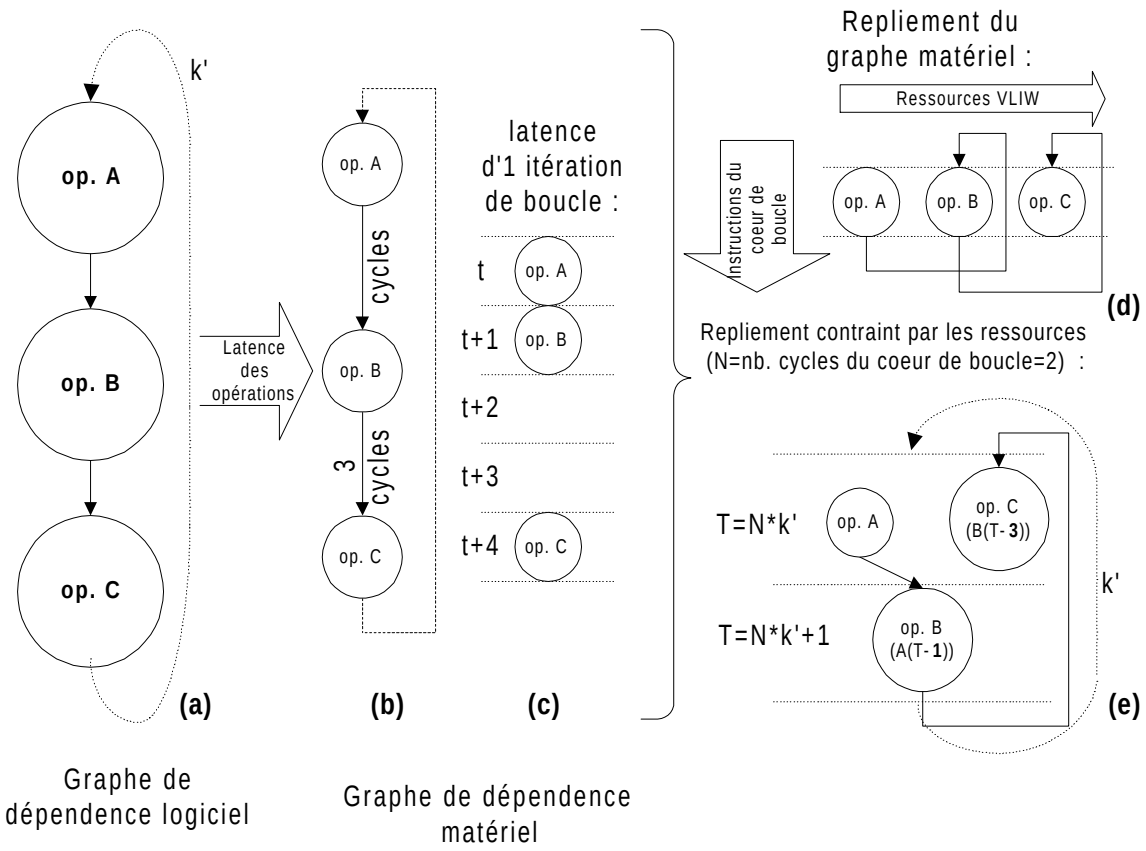
Le pipeline logiciel est historiquement lié aux architectures VLIW qui apparaissent dès le début des années 80, l'article de Lam [63] ayant intronisé le terme anglais de "*software pipelining*". Cette technique vise l'optimisation des traitements itératifs définis au sein de boucles logicielles tout en favorisant une meilleure utilisation des ressources des processeurs. Elle permet un gain de performances très important dans le cas du C6X et s'applique également à l'architecture des PP du C8X. L'idée consiste à désynchroniser ("*retiming*") l'exécution des opérations de la boucle de telle sorte que plusieurs parties d'une itération de boucle évoluent en parallèle. Comme pour l'exemple du déroulage de boucle sur la Figure 2-1, il en résulte un entrelacement des opérations qui portent sur différentes itérations "logiques" de la boucle initiale tel que décrit avec la Figure2-2.

Figure 2-2. Principe du pipeline logiciel (*software pipelining*)



Sur ce graphique, nous comprenons que le principe consiste à maximiser l'utilisation temporelle des unités de calcul, exactement à la manière d'un pipeline d'instructions matériel. Le coeur de boucle traitant différentes itérations logiques (k) en parallèle, le traitement nécessite une étape progressive d'initialisation appelée prologue (nous trouvons également le terme "prélude" dans la littérature), de même que, symétriquement, une partie du code vient graduellement terminer le traitement (épilogue/"épilude"). Par comparaison avec le déroulage de boucle, le pipeline logiciel va plus loin dans le réordonnancement des opérations du coeur itéré dans la mesure où les étapes d'initialisation/de terminaison de la boucle sont déportés à l'exterieure de celle-ci. Conceptuellement, la méthode du pipeline logiciel revient à replier le graphe de dépendance des données du coeur de boucle sur lui-même comme le montre la Figure 2-3.

Figure 2-3. Principe du repliement du graphe de dépendance

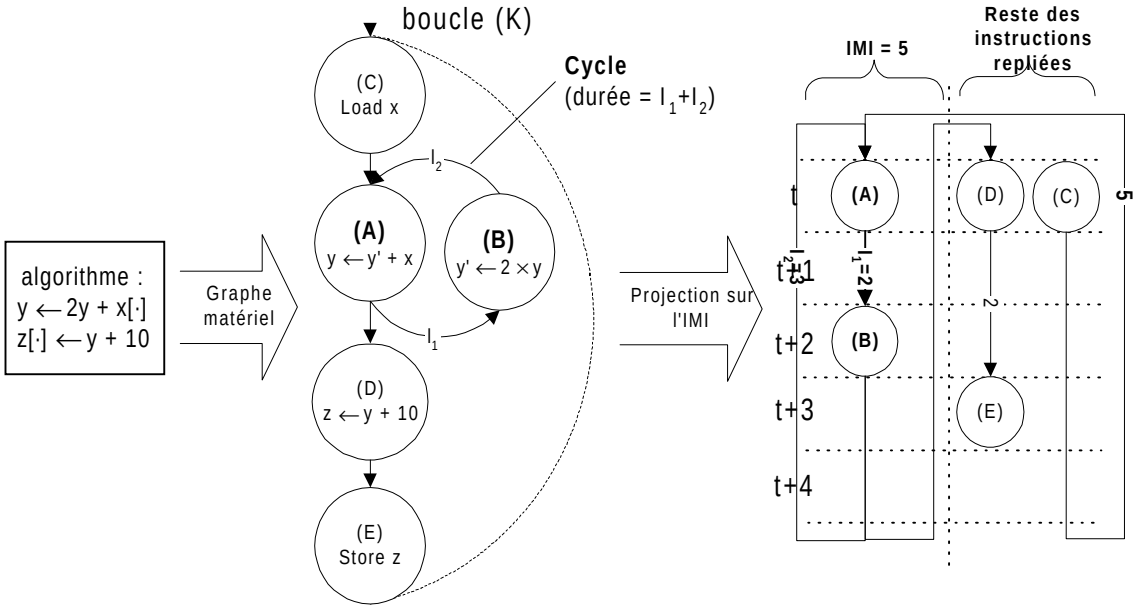


Partant d'un graphe de dépendance logicielle (a) qui précise les opérations constitutives de l'algorithme en situation de dépendance de données (au niveau des registres ou de données en mémoire externe), nous obtenons le graphe de dépendance matérielle (b) en pondérant

chaque arc de dépendance par la latence des opérations. Cette approche nous permet alors d'obtenir une première estimation de la durée globale d'une itération de traitement (c) que nous cherchons à optimiser avec le pipeline logiciel. Si, parmi les opérations du graphe du coeur de la boucle, nous ne rencontrons pas de cycle (au sens de la théorie des graphes), alors le coeur pipeliné peut être réduit à une instruction VLIW (d) en l'absence de contrainte de ressources, qu'il s'agisse des unités de calcul disponibles ou des ressources registres. Bien entendu, les ressources étant limitées, le nombre d'instructions VLIW du coeur de boucle est adapté (e).

En présence de cycles algorithmiques tel que montré sur l'exemple de la Figure2-4, nous pouvons borner l'intervalle minimum d'itérations (IMI) qui permet d'établir la latence minimum du coeur de boucle replié (ici, en nombre d'opérations VLIW, qui, sans les contentions, s'exécutent chacune en un cycle d'horloge) et ce, indépendamment des contraintes de ressources. Ce seuil traduit le délai minimum nécessaire au lancement d'une nouvelle itération repliée de manière à ce que les nœuds des cycles aient l'inter-dépendance temporelle des données satisfaite. Sur l'exemple, l'exécution du n œud A est bien subordonnée au résultat produit par le nœud B empêchant ainsi de débuter un nouvel ensemble d'opérations désynchronisées tant que la latence du cycle n'est pas respectée. La valeur du IMI se déduit alors de la durée maximale des cycles présents dans le graphe, la latence des cycles du graphe s'obtenant par sommation de la durée des arcs reliant les nœuds qui composent chaque cycle. Précisons que dans certains cas, l'IMI peut être optimisé en réduisant les dépendances, par exemple, avec l'utilisation de registres intermédiaires.

Figure 2-4. Pipeline logiciel et définition de l'IMI



Soulignons également que la technique de l'ordonnancement "modulo" (modulo scheduling) permet de projeter le graphe matériel replié sur l'intervalle minimum d'itérations tout en optimisant la durée du prologue et celle, équivalente, de l'épilogue avec un ordonnancement de type EDF (*Earliest Deadline First*) des instructions du plus long chemin de la chaîne. Pour informations complémentaires, le lecteur pourra consulté [59] (chapitre "Assembler") ainsi que [64] pour la description d'une technique permettant l'optimisation de l'impact du prologue et de l'épilogue dans le cas de boucles imbriquées. Dans la suite de ce mémoire, nous nous concentrons plus particulièrement sur l'optimisation des performances des coeurs de boucle interne. Pour l'application de la technique du pipeline logiciel (notamment pour l'exemple du filtrage optimal sur C62 détaillé ultérieurement), nous relevons alors 3 grandes étapes qui sont :

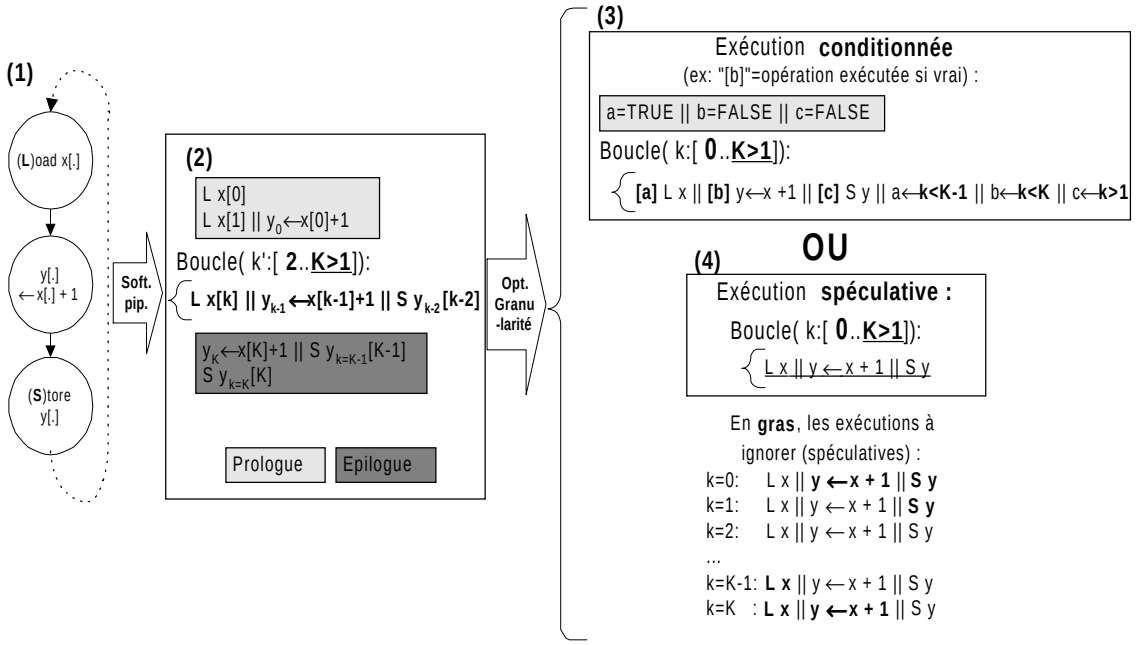
- L'optimisation des calculs algorithmiques (étape non spécifique au pipeline logiciel) et qui vise à réduire le nombre d'opérations du traitement,
- Projection des opérations algorithmiques sur les opérations du processeur, la granularité des opérations s'entendant ici au niveau du micro-code. Nous obtenons ainsi le graphe de dépendance matérielle,
- Le repliement du graphe de dépendance matérielle en fonction du IMI et de la latence des instructions du graphe avec le partitionnement des ressources pour effectivement permettre le repliement, cette étape modifiant le nombre de cycles effectif du coeur de boucle, la mesure du IMI apparaissant comme une borne théorique inférieure.

Du point de vue des entrées/sorties, nous pouvons réduire l'impact des contentions de type CPU/CPU en répartissant, dans la mesure du possible, les accès du coeur replié.

Pour diminuer la granularité de l'implantation des opérateurs, soulignons que le pipeline logiciel nécessite, comme pour le déroulage de boucle ou l'utilisation des capacités SIMD, une gestion spécifique lorsque la quantité de données traitées est inférieure au nombre d'itérations évoluant en parallèle. Dans le cas du C6x, l'optimiseur de C duplique la boucle et pipeline un des exemplaires obtenus ce qui, au minimum, double l'espace occupé par le code. La version appropriée est ensuite exécutée suivant le nombre d'itérations dynamiquement spécifié en paramètre.

Pour éviter cet impact, nous pouvons imposer une quantité minimum de données à traiter. De plus, nous pouvons optimiser la longueur du prologue et de l'épilogue en introduisant l'exécution conditionnelle totale ou partielle des opérateurs de traitement. L'idée est illustrée avec le schéma de principe de la Figure2-5.

Figure 2-5. Minimisation de la longueur du prologue et de l'épilogue



Sur l'exemple élémentaire qui est proposé, le graphe de dépendance (1) est replié (2) (l'IMI est de 1) et nous supposons qu'aucune contrainte de ressources ne vient entraver l'exécution parallèle de l'ensemble des opérations au moyen d'une unique instruction VLIW. Pour supprimer ou réduire la longueur de l'épilogue/ du prologue, nous

conditionnons l'exécution des opérations avec des drapeaux binaires (flags ou guard bits) permettant d'appréhender d'une manière spécifique, à chaque instruction, le désynchronisme du lancement des opérations (3). Nous rencontrons le support matériel de ce type d'exécution conditionnée sur le C6X et le C8X (la syntaxe "[a]" est ici empruntée au C6X), alors qu'il convient de souligner que ce principe se rencontre de manière récurrente sur les processeurs VLIW (Merced, Trimedia). Sur notre exemple, le prologue se réduit à l'initialisation des tests qui sont ensuite ré-évalués à chaque nouvelle itération de boucle à l'aide du compteur *k*. Comme le montre notre exemple, cette approche augmente la compacité du code mais induit l'utilisation de nouvelles unités de traitement (ainsi que de nouveaux registres) au niveau du coeur pipeliné. Afin de réduire la granularité sans diminuer les performances, ce que peut engendrer l'utilisation de ces nouvelles ressources, l'alternative consiste à ne mettre en œuvre que partiellement l'approche. Nous diminuons alors la longueur du prologue et/ou de l'épilogue sans toutefois l'annuler à l'inverse de ce que suggère notre exemple graphique.

Une deuxième solution consiste à supprimer le prologue et/ou l'épilogue et à itérer la boucle sur l'intervalle initial sans apporter de restriction aux instructions du coeur pipeliné (4). Pour appliquer cette démarche, il convient cependant de gérer les effets de bords liés à l'exécution spéculative des opérations. En effet, nous pouvons comprendre sur l'exemple que pour l'itération *k=0*, les opérations parallèles "*y←x+1 || S y*" sont à ignorer puisque *y* et *x* n'ont pas encore eu l'occasion d'être évalués. A ce stade, la seule opération significative de l'instruction VLIW consiste à charger la valeur *x* ("*L x*"). D'une manière similaire, l'opération "*L x*" de l'itération de fin de boucle *K-1* n'offre pas d'intérêt puisque le résultat de l'opération "*y←x+1*" lors de l'itération suivante (K) n'est jamais utilisé. Pour appliquer l'approche de manière cohérente, il convient alors de vérifier que les 2 derniers chargements sont possibles dans le sens où il doit physiquement exister de la mémoire pour chacun des 2 accès "spéculatifs". Concernant l'opération de sauvegarde des résultats en mémoire externe ("*S y*"), il convient, dans la même optique, de permettre l'écriture de 2 résultats supplémentaires en mémoire. La quantité globale de données produites sur notre exemple correspond alors bien à K+2 valeurs de *y*, les deux premiers accès n'étant pas significatifs. Soulignons que ce type d'optimisation sur la granularité n'a pas d'incidence

sur la performance des opérateurs (le nombre et la latence des itérations du coeur de boucle sont toujours conservés).

Nous concluerons en soulignant que la combinaison de l'exécution conditionnelle et spéculative des opérations permet de diminuer la granularité tout en réduisant la complexité de mise en œuvre qu'induit l'approche plus radicale des opérations spéculatives.

2.1.5 Utilisation de table de transcodage (LUT)

La dernière technique générique que nous présentons concerne l'utilisation de tables nous permettant de stocker un certain nombre de calculs récurrents évalués une seule fois dans une phase d'initialisation. A la manière des tables de transcodage colorimétrique, nous associons à un indice de la table une valeur pré-établie. Cette optimisation, intuitive et simple à mettre en œuvre, permet d'améliorer les performances dans la mesure où les coûts d'accès à la table sont, bien sûr, plus réduits que le coût des calculs correspondants qui, sinon, seraient exécutés dynamiquement (ce qui, avec les capacité SIMD ou les multiples unités fonctionnelles des architectures actuelles, n'est pas toujours le cas). Cette remarque est d'autant plus significative que cette technique présente l'inconvénient de nécessiter une quantité de mémoire accrue. Pour l'exemple de la détection de mouvement sur C8X de notre deuxième application de référence (chapitre 4), nous avons employé cette technique qui permet des gains de performances importants pour un espace occupé compatible avec la mémoire interne vacante des PP.

2.2 Une approche logicielle originale pour l'optimisation des flots de données

> *Over the coming decade, memory subsystem design will be the* ***only*** *important design issue for microprocessors*. Dick Sites (1996), Architecte des processeurs Alpha de Digital [9].

Après notre introduction des techniques d'optimisation que nous jugeons les plus significatives pour une implantation performante des nœuds, nous développons, dans cette partie, la deuxième principale étape de la mise en œuvre d'algorithmes sur DSP qui consiste à programmer le(s) co-processeur(s) DMA pour la gestion des entrées/sorties de données. A cette fin, nous détaillons tout d'abord les objets génériques que nous manipulons ensuite pour la modélisation des chaînes de traitement. De là, nous introduisons sur un premier plan pratique, la manière de manipuler ces entités pour permettre la mise en place d'une gestion optimisée des flux. Cette démarche nous permet ensuite de présenter un modèle théorique visant à la fois le partionnement des données et la spécification synchrone des requêtes de transfert avec le DMA.

2.2.1 Une approche générique

Dans cette partie, nous définissons les différentes entités impliquées dans la description générique d'algorithmes composés d'opérateurs de traitement d'images bas niveau. Nous présentons les paramètres retenus pour chaque entité afin d'exploiter au mieux les caractéristiques de la classe d'architecture ciblée et notamment, celle du C80. Nous présentons également les mécanismes pratiques d'utilisation des ressources pour la gestion des entrées/sorties et les justifions face aux alternatives qui nous sont offertes.

2.2.1.1 Définition des objets élémentaires

Notre approche s'appuie sur la combinaison de plusieurs types d'entités permettant la composition et la manipulation de chaînes de traitement. Définissons tout d'abord ces entités.

2.2.1.1.1 Les buffers image

Dans notre contexte de traitement d'images, un "buffer" est synonyme de région d'intérêt 2D d'une image (ou ROI pour Region Of Interest). Nous associons un total de sept paramètres à chaque buffer avec, tout d'abord :

- sa taille définie par la largeur W_{ref} et la hauteur H_{ref},
- son emplacement en mémoire externe (*@BUF*),
- le pas image W_{pitch} défini comme étant la différence d'adresse entre deux lignes d'image.

W_{pitch} est utilisé par le DMA multi-dimensionnel de manière à pouvoir adresser une quelconque sous-matrice, ou ROI, sans qu'un transfert supplémentaire ne soit nécessaire pour isoler la sous-image avant son utilisation.

Les accès directs à la mémoire externe étant généralement particulièrement coûteux pour les DSP (plus de dix cycles par accès avec les PP et quelques 40 cycles dans le cas du C6X suivant les types de mémoire externe interfacée [32]), nous programmons le DMA de manière logicielle pour "cacher"[1] les données des buffers externes en spécifiant et en synchronisant (au niveau du programme) les requêtes de transfert de la mémoire externe vers la mémoire interne et vice et versa. Avec cette approche, les accès aux données en mémoire interne s'exécutent beaucoup plus rapidement (un cycle pour les PP C8X et le C6X lorsqu'il n'y a pas contention d'accès au banc).

A chaque buffer, nous associons alors trois paramètres pour cette gestion de cache. Tout d'abord, nous définissons une adresse en mémoire interne pour cacher les données (*@CACHE[0]*) ainsi qu'une taille de cache associée (*S*). Un des aspects particulièrement

1. Du verbe anglai *to cache* qui ne bénéficie pas d'autre équivalent en français que la traduction "de mettre en antémémoire" selon le dictionnaire anglais-français Harraps.

attrayants des DMA est de pouvoir fonctionner de manière asynchrone par rapport au coeur CPU. Cela signifie donc que le traitement peut se faire en parallèle des transferts. Dans le cas du C8X (et du C6X), ce fonctionnement concurrent est possible sans contention lorsque le DMA et les PP accèdent à des bancs de mémoire interne distincts. Ce recouvrement temporel des phases de calcul et de transfert a lieu sans contention si nous utilisons la technique dite du “double buffering” qui repose sur l’existence d’une deuxième zone de cache qui, dans notre contexte, s’associe à l’utilisation d’un deuxième banc à l’adresse *@CACHE[1]*. Pour simplifier la modélisation, nous supposons que la région ainsi pointée occupe également un espace de *S* octets.

Dans un contexte multi-processeurs, nous associons enfin un dernier paramètre au buffer, σ_{ref}, qui représente le nombre de PP utilisés pour le partitionnement SPMD des données du buffer. Les paramètres du cache (*S* et *@CACHE[]*) sont par ailleurs communs pour chaque processeur mais relatifs à l’adresse du premier banc de mémoire interne associé à chaque processeur. Ainsi, alors que le C8X est défini comme une architecture à mémoire partagée, le mécanisme de cache logiciel mis en place s’apparente à un modèle de mémoire distribuée ou chaque processeur accède uniquement à sa mémoire privée. Le DMA apparaît donc ici comme le seul véritable acteur de l’architecture utilisant le partage de la mémoire commune.

Le tableau suivant résume les sept paramètres que nous associons à chaque buffer.

Table 2-1. Récapitulatif des paramètres d’un buffer

Paramètre	Définition
W_{ref}	Largeur du buffer en octets
H_{ref}	Hauteur du buffer en nombre de ligne
@BUF	Adresse du début du buffer en mémoire externe
W_{pitch}	Différence d’adresse entre deux lignes de buffer (en octets)
@CACHE[..]	Tableau de pointeurs sur les zones de tampon interne employées comme cache
S	Taille des zones de cache
σ_{ref}	Nombre de processeurs de référence pour le partitionnement des données du buffer

2.2.1.1.2 Nœud et chaîne de traitement

Un nœud de traitement est une fonction exécutant une ou plusieurs itérations d'un opérateur algorithmique. L'exécution séquentielle d'une suite de n œuds définit une chaîne de traitement qui consomme et produit des données depuis et vers les buffers. Pour l'architecture C8X, les nœuds s'exécutent sur les PP et sont, la plupart du temps, écrits en assembleur algébrique pour tirer au mieux parti des capacités parallèles du coeur de l'architecture.

2.2.1.1.3 Le patron de traitement

Le patron de traitement (*Processing Template*) est un programme fonctionnant de manière indépendante sur chaque PP et qui se charge du lancement séquentiel des nœuds ainsi que de la synchronisation de la chaîne avec les transferts DMA. Cette encapsulation des traitements algorithmiques avec la gestion des caches de buffers offre une base commune à l'exécution de chaînes génériques.

2.2.1.1.4 La notion centrale des patrons de données

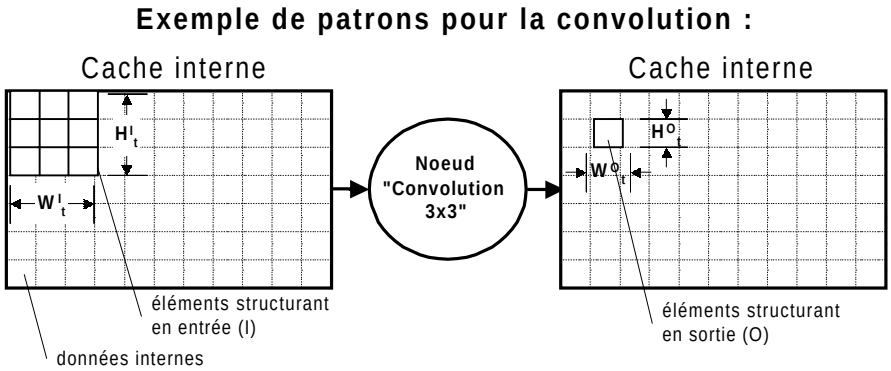
L'élément structurant d'un opérateur

Les patrons de données ou patron (*Data Template*) apparaissent comme une notion centrale pour notre modélisation des flux de données associés aux chaînes de traitement. C'est pourquoi, les concepts et paramètres introduits dans cette section importent pour la compréhension du reste de ce chapitre.

Dans un premier temps, nous pouvons définir les patrons de données comme des entités associées à chaque nœud décrivant l'élément structurant consommé ou produit par un opérateur algorithmique. Un opérateur de convolution $m \times n$ définit le patron de données du nœud associé qui possède pour géométrie $W^I_t \times H^I_t$ ($W^I_t = n$, $H^I_t = m$) - l'indice t caractérisant la quantité minimale de donnée nécessaire au nœud pour la direction horizontale (W) et verticale (H), l'indice I précisant qu'il s'agit de la géométrie des données

en entrée du nœud (*(I)nput*). Dans cet exemple illustré avec la , l'élément produit en sortie (*(O)utput*) de l'opérateur de convolution a pour géométrie de patron $W^O{}_t \times H^O{}_t = 1 \times 1$, soit un simple pixel. La notion de patron de sortie va permettre une représentation des chaînes de traitement qui s'appuie sur une description synchrone des flux de données (SDF). Cette synchronisation des flux intervient à la fois entre les nœuds d'une chaîne mais également au niveau des requêtes DMA nécessaires pour cacher les différents buffers impliqués dans l'algorithme.

Figure 2-6. Formatage des éléments structurant : les patrons synchrones d'entrée/sortie



Extension de la notion d'élément structurant avec les contraintes d'implantation

La définition du patron de données va plus loin que le simple élément structurant dans la mesure où nous intégrons, au sein des paramètres géométriques, des contraintes sur la taille des données en entrée ou en sortie. Ces contraintes sont liées à la manière dont les opérateurs sont implantés. Ici, nous pensons aux simplifications mises en place en faveur de l'optimisation de la granularité lors de la mise en œuvre des techniques d'optimisation génériques pour la programmation performante des nœuds (déroulage de boucle, capacité SIMD, pipeline logiciel). En effet, avec l'exemple du déroulage de boucles ou celui de la mise en œuvre des instructions SIMD, nous avons notamment vu que ces techniques ne favorisent pas la compacité du code et que, dans l'optique du chaînage des opérateurs que nous visons afin de réduire la quantité globale des données algorithmiques transférées, la non-optimisation de cette granularité risque paradoxalement d'inhiber l'amélioration des performances avec le coût accru de la gestion des caches d'instructions (Cf 2.1.1) [1]. Pour

ces techniques, nous avons alors suggéré que le nombre d'itérations de traitement algorithmique (logique) soit multiple du nombre d'itérations parallèles du n œud (physique). L'idée consiste à exclure du nœud la gestion spécifique des quantités de données que ne prend pas en charge le coeur de boucle optimisé. Avec le même but, nous avons suggéré que la quantité minimale de données à traiter soit en accord avec le nombre d'itérations déroulées (ou traitées en parallèle avec les instructions SIMD). Aussi, nous avons montré que ces mêmes contraintes peuvent s'appliquer à la réduction de la taille du code qu'engendre également la technique du pipeline logiciel très importante pour le cas d'architectures VLIW. Outre l'intérêt de l'optimisation de la compacité du code programme, soulignons enfin que les contraintes de taille introduites sur les quantités de données qui limitent ainsi la taille $W_{ref} \times H_{ref}$ des images réellement traitées, permettent de simplifier l'écriture des opérateurs en ne gérant pas les cas spécifiques. Pour le C80, cette remarque est accentuée par le fait que l'assembleur algébrique de PP est réputé complexe et s'affranchir de l'implantation de la gestion de la non-multiplicité (ou de l'épilogue et du prologue des nœuds pipelinés) apporte un gain important de productivité.

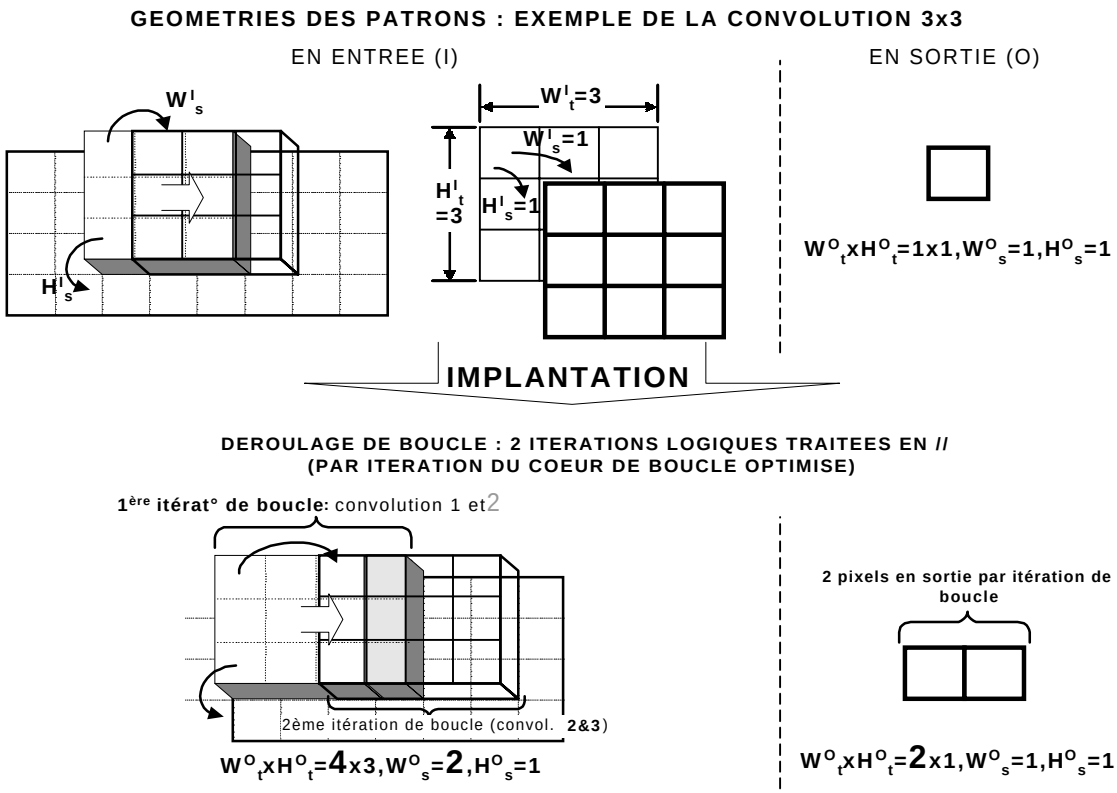
Les paramètres géométriques fondamentaux du patron de données

Au regard des considérations de granularité/productivité introduites, nous proposons d'intégrer les contraintes de tailles évoquées au sein des paramètres géometriques des patrons. Nous donnons par conséquent une extension à la configuration spatiale de l'éléments structurant avec deux types de contraintes sur la taille des données.

La première contrainte concerne la quantité minimale de données nécessaires pour une itération de traitement. Si nous reprenons l'exemple de la convolution $m \times n$ dont nous supposons que l'implantation déroulée traite deux itérations de noyau par itération de boucle dans la direction horizontale, le patron en entrée devient $(n+1) \times m$ alors que la géométrie des données produites en sortie devient 2×1.

1. Par la suite, nous aurons l'occasion de mesurer précisément l'impact du coût de la gestion des caches d'instructions avec l'implantation C80 de l'algorithme de détection de contours (Cf 3.3.2.1).

Figure 2-7. Intégration des contraintes de taille dans les patrons



La deuxième contrainte est liée à la multiplicité de la taille des données à traiter. Nous pouvons également appréhender cette contrainte comme la quantité de données nécessaires pour une nouvelle itération de traitement pour la direction horizontale et verticale. Prenons l'exemple simpliste d'un traitement point à point comme l'addition d'une constante à une image. Si nous implantons cet opérateur en utilisant les capacités SIMD de l'architecture PP qui autorise quatre additions 8 bits en parallèle, le patron en entrée/sortie devient 4×1 et la quantité de données nécessaires pour une nouvelle itération de traitement suivant la direction horizontale est alors de quatre octets (W^I_s, l'indice *s* dénotant la notion de pas ou "*step*") et d'une ligne pour la direction verticale (H^I_s). Nous définissons ainsi les "pas" de la boucle qui coïncident avec la multiplicité des données traitées. Moins trivialement, pour le nœud de convolution $(n+1)\times m$, ces pas deviennent $W^I_s=2$ et $H^I_s=1$, indiquant ainsi qu'en décalant le centre du noyau structurant, deux nouvelles colonnes de *n* lignes de pixels sont nécessaires pour le calcul des nouveaux résultats suivant la direction horizontale alors qu'une seule ligne de données supplémentaire est nécessaire pour convoluer les données des lignes successives. Ce principe est illustré avec la Figure2-7.

Nous introduisons donc les contraintes $W_{ref} - (W^I_t + 1) \bmod W^I_s = 0$ et $H_{ref} - H^I_t \bmod H^I_s = 0$ sur la taille des buffers avec, bien sûr, $W_{ref} \geq W^I_t + 1$ et $H_{ref} \geq H^I_t$.

Lorsque plusieurs itérations d'un opérateur algorithmique sont ainsi traitées en parallèle au sein d'une itération du coeur de traitement du nœud, nous pouvons alors définir un premier modèle simple pour la définition des paramètres $(W'_t \times H'_t, W'_s, H'_s)$ d'un patron compte tenu de la géométrie initiale de l'opérateur algorithmique $(W_t \times H_t, W_s, H_s)$:

Equation 2-1.

$$\begin{cases} W'_t = (p-1) \times W_s, & W'_s = p \times W_s \\ H'_t = (q-1) \times H_s, & H'_s = q \times H_s \end{cases}$$

où, p et q correspondent au nombre d'opérateurs traités en parallèle pour la direction horizontale et, respectivement, pour celle verticale.

Par ailleurs, nous soulignons que les égalités précédentes s'appliquent de façon synchrone à la géométrie du patron en entrée (d'indice I) et à celle en sortie (indicé par O). Par là, nous entendons notamment qu'il doit y avoir une correspondance 1 pour 1 entre la quantité W_s de données permettant une nouvelle itération du n œud (complémentaire à celle déjà permise par la présence de W_t éléments) et celle produite par le nœud en sortie. Ainsi, en prenant l'exemple d'un n œud ayant pour but de diviser par deux la taille d'un signal d'entrée et pour lequel deux itérations algorithmiques sont traitées en parallèle ($n=2$), nous avons alors $W^I_t = W^I_s = 4$ et, de manière synchrone $W^O_t = W^O_s = 2$. Cette remarque s'applique également à l'exemple du nœud diadique (faisant intervenir deux patrons en entrée) où nous employons une même valeur de n pour les trois patrons du nœud. Par la suite, nous verrons l'intérêt de cette approche qui vise la synchronisation cohérente des requêtes DMA.

2.2.1.2 Cas des patrons de données se recouvrant

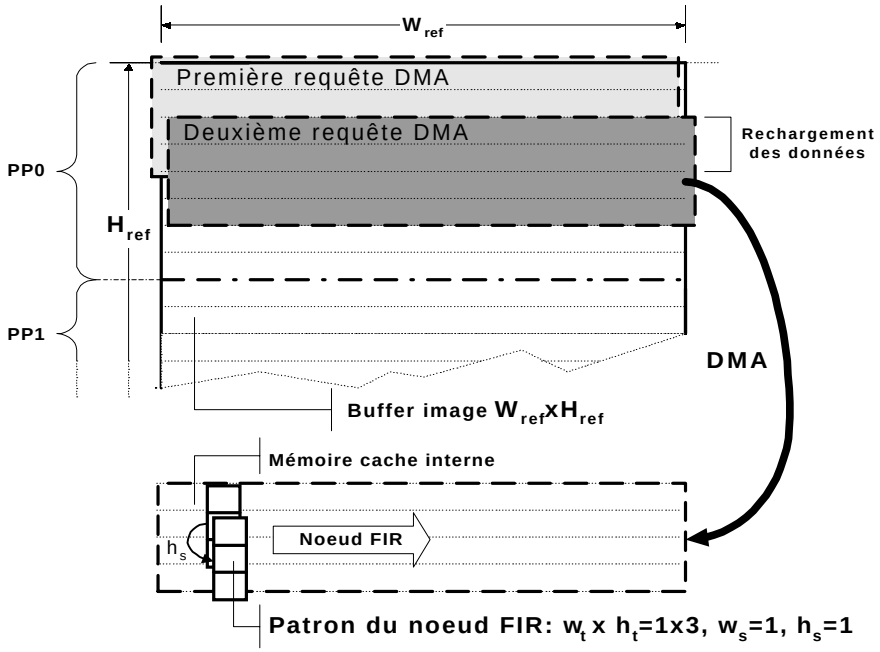
L'exemple du nœud de convolution introduit la notion de patron de données se chevauchant entre deux itérations successives du nœud. Un tel recouvrement intervient donc lorsque $W_t > W_s$ et/ou $H_t > H_s$. Or, ce chevauchement des patrons pose la question des effets de bord entre les PP assignés au partitionnement SPMD alors que ce phénomène intervient déjà

entre les requêtes DMA d'une même bande d'image assignée au même processeur (dans l'hypothèse fréquente où toutes les données de l'image ne "tiennent" pas dans la zone de cache *S*).

Nous pouvons envisager différentes approches pour traiter l'effet de bord. Tout d'abord, nous pouvons recharger les données du patron qui sont à cheval sur deux requêtes DMA tel qu'illustré sur la Figure2-8. Sur cet exemple, nous considérons que le partitionnement des données est conçu de telle sorte que chaque processeur traite une bande horizontale de l'image nécessitant chacune plusieurs requêtes DMA afin d'être parcourue.

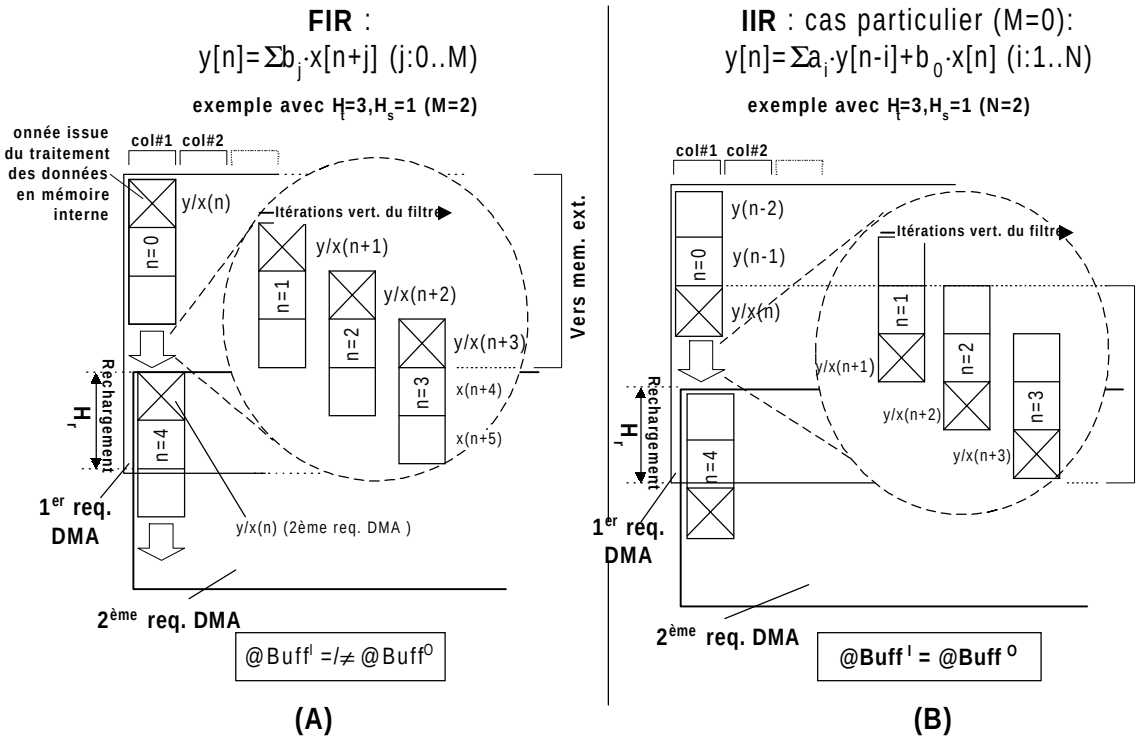
Nous pouvons imaginer l'application d'un filtre FIR à trois coefficients sur un buffer ($y(n) = \sum_{j=0}^{2} a_j \cdot x(n+j)$). A chaque requête DMA, deux lignes d'image sont ainsi rechargées pour l'application verticale de ce nœud par bande d'image horizontale.

Figure 2-8. Effet de bord et rechargement des données



La partie gauche (A) de la Figure2-9 donne un détail du rechargement pour le cas des FIR. Ici, les données rechargées correspondent aux valeurs de *x(n+4)* et *x(n+5)* ayant servi au calcul de *y(n+3)* lors de la dernière itération interne du filtre associée aux dernières données cachées (c-à-d pour *n=3*).

Figure 2-9. Rechargement des données pour les filtres FIR et IIR (cas vertical)



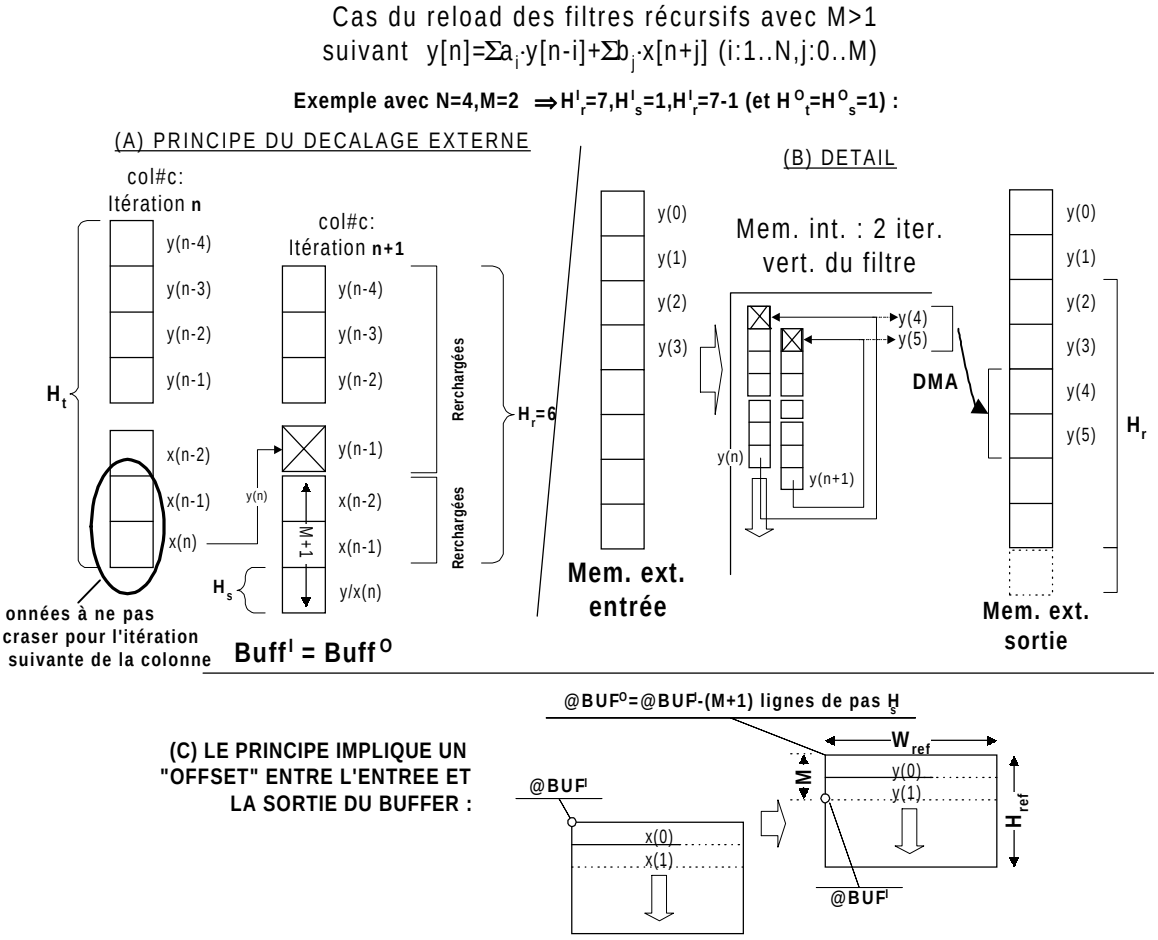
Pour le cas des filtres récursifs (IIR), l'approche du rechargement permet également de ré-intégrer les données *y(n-i)* précédemment traitées (ou données "passées" au sens de z^{-1}). Nous illustrons tout d'abord ce mécanisme dans le cas particulier des filtres IIR ayant comme caractéristique $M = 0$ pour . $y(n)= \sum_{j=0}^{M} b_j \cdot x(n+j)+ \sum_{i=1}^{N} a_i \cdot y(n-i)$

La partie droite (B) de la Figure 2-9 montre le rechargement des valeurs *y(n+2)* et *y(n+3)* issues des dernières itérations de traitement appliquées aux données de la précédente requête DMA. Ce rechargement est donc nécessaire à l'exécution des deux premières itérations de traitement de la nouvelle requête DMA. Pour que le raisonnement soit cohérent, ce principe implique de faire coïncider le buffer d'entrée et de sortie des données comme il est souligné dans la figure (nous écrivons $@BUF^I = @BUF^O$).

Pour le cas de la formulation générale des filtres IIR (c-à-d $N \neq 0$), nous proposons de décaler l'emplacement du résultat *y(n)* en mémoire externe par rapport à la valeur *x(n)* afin de ne pas écraser cette dernière. Cette approche est nécessaire pour appliquer de manière cohérente les itérations du filtre. Ce principe est détaillé dans la Figure2-10 (A). Là aussi, nous faisons coïncider les buffers d'entrée et de sortie, mais nous modifions simplement l'adresse du buffer de sortie (C) de manière à ce que les résultats de traitement (*y(n)*)

n'écrasent pas les *M* valeurs de *x(n+j)* qui font partie des données rechargées entre deux requêtes DMA (avec les N valeurs de la dernière itération du calcul de *y*).

Figure 2-10. Cas général du rechargement des filtres IIR pour la direction verticale



Nous pouvons proposer des alternatives à l'approche du rechargement. Tout d'abord, nous pourrions faire en sorte que les nœuds gèrent les transferts DMA après qu'ils aient terminé de ré-utiliser les précédentes données. Dans une variante, nous pourrions envisager que le nœud isole temporairement, dans la mémoire interne, les données à ré-utiliser (nous ajouterions par exemple une copie des données vers une zone spécifique).

Nous préférons cependant l'approche du rechargement dans la mesure où l'implantation des nœuds est simplifiée (nous favorisons ainsi une faible granularité des n œuds) et l'approche plus modulaire dans le sens où la gestion des flux externes reste décorrélée des opérateurs de traitement. Ainsi, la gestion des entrées/sortie est volontairement déportée au

niveau du patron de traitement dont le rôle est bien d'unifier l'encapsulation des opérateurs avec la gestion du DMA. Dans la pratique, les nœuds sont "passifs" et reçoivent de multiples paramètres parmi lesquels les spécifications de l'emplacement en mémoire interne des différents flux (ainsi que leur taille et leurs caractéristiques). Quant à la deuxième solution (celle de la copie interne des données que nous aurions, sinon, à recharger), elle nécessite également une implantation plus complexe des nœuds ainsi que de l'espace mémoire supplémentaire, espace qui devient critique si nous considérons que deux des trois bancs du C80 sont assignés au double-buffering. Nous verrons dans le paragraphe 2.2.2 que la technique du triple buffering va également dans le sens du maintien temporaire des données tout en permettant d'éviter l'opération de recopie.

Il convient toutefois de citer l'inconvénient majeur de l'approche du rechargement qui est d'induire un coût de transfert supplémentaire. Néanmoins, dans la mesure où les patrons se recouvrant sont généralement associés à une charge de calcul importante (de type convolution), ce surcoût est généralement atténué voire sans incidence sur la durée globale de traitement lorsque la vitesse de calcul est supérieure ou égale à celle des transferts. Par la suite, nous vérifierons cette remarque sur un exemple précis.

Nous pouvons conclure que l'approche du rechargement semble offrir un nombre important d'avantages et constitue l'approche privilégiée dans la suite de ce mémoire. Nous lui associons deux nouveaux paramètres qui servent à définir la quantité de données rechargées entre chaque requête DMA avec W_t - W_s pour la direction horizontale et H_t - H_s pour la direction verticale. Ces valeurs apparaissent comme des quantités par défaut qui conviennent aux exemples de filtres évoqués. Par la suite, nous permettons un paramétrage spécifique du rechargement au travers des paramètres W_r (l'indice r dénotant le rechargement ou "*reload*") pour la direction horizontale et H_r pour celle verticale. Nous adoptons la notation $W_s[\backslash W_r]$ et $H_s[\backslash H_r]$ pour préciser que la quantité de données n'est pas celle par défaut (3\1 signifiant, pour l'une ou l'autre des directions, "pas" de 3 et "rechargement" de 1). Par ailleurs, nous pouvons souligner que, du point vue des transferts exportant les données traitées vers les buffers en sortie, la notion de rechargement n'intervient pas ; elle ne concerne intuitivement que les données en entrée du traitement.

2.2.1.3 Balayage des données internes/externes

Nous avons vu que le DMA du C8X autorise un paramétrage indépendant des dimensions du transfert pour la source et la destination. Nous pouvons ainsi décorréler la manière dont les données sont parcourues en mémoire externe par rapport à la mémoire interne (fonctionnalité également offerte par le DMA du C6X et celui du Sharc). L'intérêt de l'approche est de permettre à un même nœud de traitement de s'appliquer indépendamment du sens de parcours ou de l'organisation des données. Ainsi, pour les filtres séparables, le même nœud de traitement peut s'appliquer au parcours horizontal et vertical de l'image. Nous pouvons également songer au cas des images couleur au format YUV ayant une organisation alternée des composantes dans le buffer image. Ainsi, s'il s'agit d'appliquer un quelconque opérateur à la seule composante de luminance, le parcours de la mémoire externe peut effectivement filtrer la composante d'intérêt grâce au support multi-dimensionnel du DMA. Cet exemple souligne la polyvalence d'un nœud qui peut donc par exemple s'appliquer indifféremment à une image couleur ou noir et blanc. Pour effectivement filtrer une composante d'intérêt comme pour l'exemple du plan YUV, nous introduisons deux nouveaux paramètres décrivant le patron de données. D_s représente la taille des données (ou pixels) en octets, alors que D_p correspond au nombre d'octets séparant deux pixels. Lorsque $D_s \neq D_p$, nous nous reposons sur l'utilisation d'une dimension du DMA pour l'adressage des données. Pour le C8X, le cas où $D_s < 8$ induit un taux de transfert réduit de $D_s/8$, ce qui limite la portée de l'approche[1].

En outre, l'organisation des données en mémoire interne est, dans la mesure du possible, toujours linéaire (les données consécutives correspondent aux pixels des lignes successives de la région d'intérêt traitée). Ici, l'objectif est de favoriser le chaînage des opérateurs ce qui suppose que l'organisation des données consommées et produites par les nœuds soit la plus homogène possible. Aussi, pour les opérations par blocs de type DCT 8×8, le parcours des données ne privilégie pas le regroupement des 64 données des blocs mais charge simplement les lignes ou colonnes d'images dans la mémoire interne. Le nœud de traitement reçoit l'ensemble des caractéristiques du nombre de lignes ou colonnes présentes

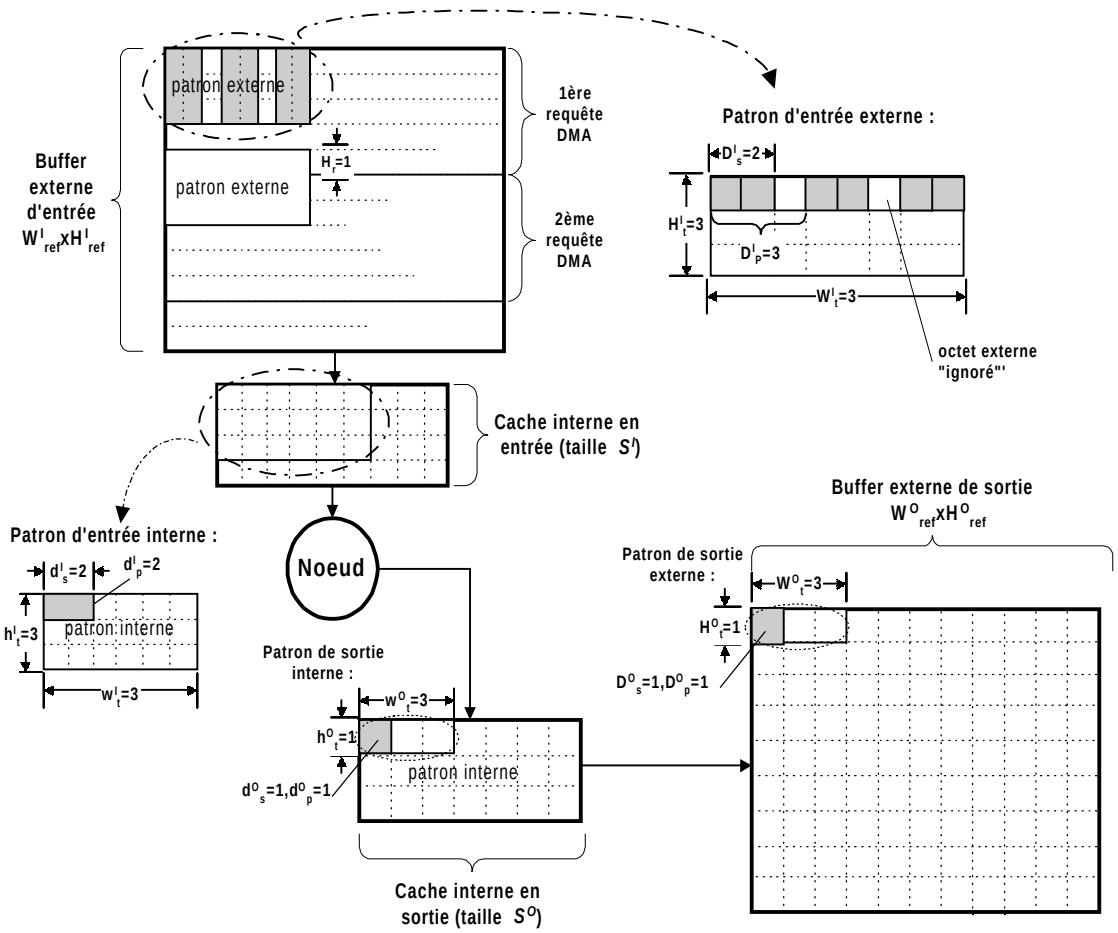
1. Le bus d'E/S est un bus 64 bits.

à un moment en mémoire interne, ce qui permet au nœud d'adresser les données de chaque bloc selon les besoins de l'algorithme.

Cette discussion suggère l'utilisation de deux types de patrons de données : le patron externe et interne. Pour les distinguer, nous utilisons les lettres majuscules pour le patron externe et les minuscules pour la contrepartie interne. Par la suite, nous associons notamment une définition transposée des paramètres internes/externes suivants $W_t=h_t$ et $H_t=w_t$, à l'application dans le sens vertical d'un opérateur de traitement (cf 2.2.3.2.2).

Par ailleurs, par souci de simplification, nous contraignons la valeur de D_p à être multiple de celle du pas horizontal (W_s) des données. Nous autorisons également la définition d'un pas interne de données (d_p) dans le but de pouvoir "réserver" de l'espace en mémoire interne pour permettre, par exemple, l'expansion de données 8 bits en 16 bits.

Figure 2-11. Récapitulatif des paramètres géométriques des patrons : un exemple



2.2.1.4 Conclusions

Nous choisissons de conclure cette partie en illustrant la définition de quelques patrons pour des exemples très simples d'opérateurs de traitement sur C80. Nous précisons également les différentes techniques d'optimisation employées :

Table 2-2. Exemples de patrons internes pour divers nœuds

Nom du nœud assembleur	Patron d'entrées : $w^I_t \times h^I_t, d^I_s, d^I_p, w^I_s, h^I_s$	Patron de sorties : $w^O_t \times h^O_t, d^O_s, d^O_p, w^O_s, h^O_s$	Technique d'optimisation
Histogramme	1×1,1,1,1,1	Aucun	
Lisseur IIR du premier ordre de FGL[a]	4×1,1,1,1,1	3×1,1,1,1,1	Pipeline logiciel
Non binaire (NOT)	4×1,1,1,4,1	4×1,1,1,4,1	SIMD
Convolution 3x3	4×3,1,1,2,1	2×1,1,1,2,1	SIMD & pip. logiciel
Gradient 2×2 de FGL[a]	10×2,5,1,4,1	9×1,1,1,5,1	SIMD & pip. logiciel
Troncature 8 bits/16 bits	4×1,1,2,8,1	4×1,2,2,8,1	SIMD

a. Cf. paragraphe 3.3

2.2.2 Gestion performante des bancs de mémoire internes

Dans cette partie, nous discutons des techniques logicielles que nous sommes en mesure d'employer pour une utilisation véritablement optimisée des bancs de mémoire interne. L'optimisation vise notamment l'exploitation des capacités des DMA qui permettent le recouvrement temporel des phases de calcul et de transferts. Dans ce sens, le double buffering constitue un moyen simple d'optimiser la gestion logicielle des caches de données. Ici, nous introduisons tout d'abord le triple buffering qui peut notamment favoriser les performances des chaînes de traitement.

2.2.2.1 Intérêt du triple buffering

Nous distinguons 3 avantages à l'utilisation du triple buffering. Tout d'abord, l'approche permet de dissocier la zone des données en entrée de celle en sortie en mémoire interne, ce que nécessitent certains opérateurs. Ensuite, elle peut également permettre d'éviter le rechargement des données en temporisant la présence des données en mémoire interne. Enfin, cette technique permet également de réduire les contentions d'accès à la mémoire.

Différencier le banc de mémoire d'entrée et de sortie

Nous pouvons reprendre l'exemple du nœud de convolution qui pose le problème de l'écriture du résultat de l'opérateur en mémoire interne. Intuitivement, ce calcul ne doit pas "écraser" la donnée centrée sur le noyau au risque de rendre incohérentes les itérations successives de l'algorithme. Nous envisageons alors deux techniques. La plus intuitive consiste à utiliser un banc de mémoire pour sauvegarder les résultats de l'opérateur autre que celui contenant les données en entrée du traitement. Dans cette optique, si nous souhaitons maintenir le transfert parallèle des données, nous sommes amenés à considérer le triple buffering combiné au pipeline du traitement tel que décrit à l'aide des tables 2-3 et 2-4. Cette approche vise à faire tourner le rôle des bancs de telle sorte que nous ayons toujours deux bancs consacrés au traitement et un banc pour le transfert concurrent (qui, en séquence, exporte les données traitées et importe les nouvelles données à traiter).

Table 2-3. Rôle des bancs internes lors du triple buffering

Banc de mémoire interne concern	**Jeu de requêtes DMA et traitement parallèl** • D: rapatriement de données (Download) par le DMA • P: traitement (Processing): indice 'I', données en entrée, 'O', données en sortie • F: exportation des données traitées (Flush) par le DMA • P_t/D_t: t=indice de données (i.e. num. de requête DMA) • "F_t,D_t" requêtes DMA en séquence								
	t=0	t=1	t=2	t=3	t=4	....	t=N-1	t=N	t=N+1
Banc 0	D_0	P^I_0	P^O_1	F_1, D_3	P^I_3		P^O_{N-2}	F_{N-2}	
Banc 1		D_1	P^I_1	P^O_2	F_2, D_4		P^I_{N-2}	P^O_{N-1}	F_{N-1}
Banc 2		P^O_0	F_0, D_2	P^I_2	P^O_3		F_{N-3}, D_{N-1}	P^I_{N-1}	

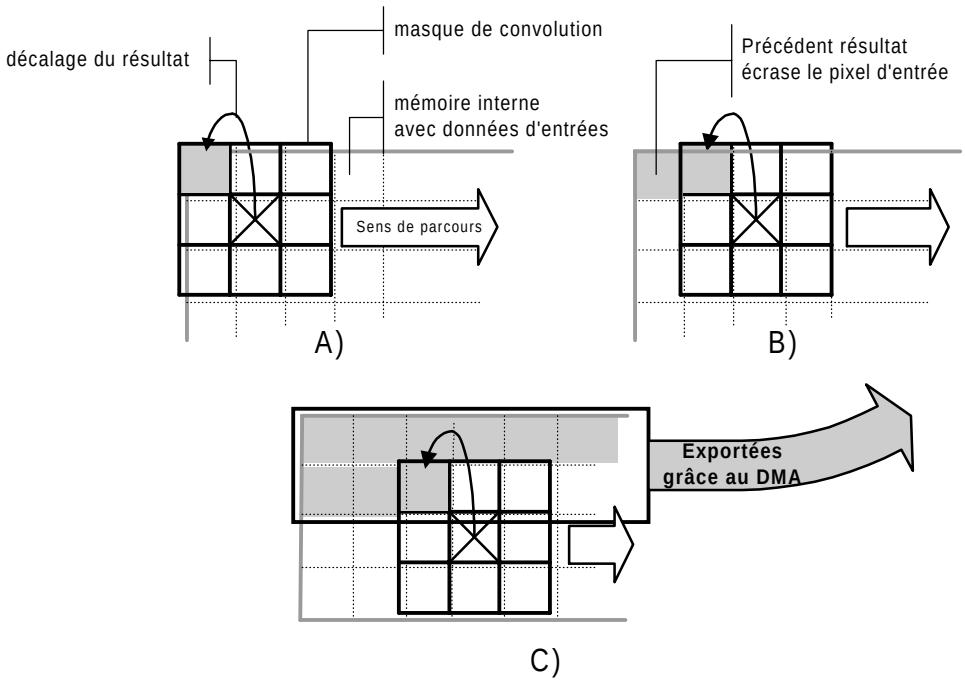
Table 2-4. Tour de rôle des buffers pour le triple buffering

Jeu de requêtes	**Action par banc de données (t:2..N-1)** • '%', fonction modulo (notation du langage C)		
	Banc 0	Banc 1	Banc 2
t % 3 = 0	F_{t-2}, D_t	P^O_{t-1}	P^I_{t-1}
t % 3 = 1	P^I_{t-1}	F_{t-2}, D_t	P^O_{t-1}
t % 3 = 2	P^O_{t-1}	P^I_{t-1}	F_{t-2}, D_t

Nous pouvons souligner, par ailleurs, que cette gestion des bancs offre également l'avantage de permettre à un nœud d'une chaîne qui en comporterait plusieurs, d'accéder aux valeurs initiales des données malgré la présence, en aval de la chaîne, d'un autre nœud susceptible d'écraser les données.

Nous rappelons que l'architecture C80 autorise ce type de gestion d'une manière performante avec le support de trois bancs de mémoires de données distincts (il n'y en a que deux pour le C82). En revanche, si le double buffering est géré de manière automatique par le DMA du C8X, le triple buffering nécessite une implication plus grande dans la gestion des transferts de la part du patron de traitement. Cette situation augmente notamment la quantité de blocs du cache d'instructions nécessaires au patron de traitement au risque d'une dégradation des performances. Dans cette optique, il peut s'avérer plus avantageux de considérer une deuxième technique, très simple, qui consiste à décaler spatialement l'écriture interne du résultat de la convolution sur les données d'entrées. Ce décalage s'effectue dans la direction opposée au sens de balayage du masque comme détaillé à la Fi gure2-12. Il permet ainsi de faire coïncider de manière cohérente le patron d'entrée avec celui de sortie du nœud.

Figure 2-12. Décalage des résultats face au triple buffering



Cette technique s'applique également aux filtres IIR comme le suggère la Figure2-10(B). Dans le cas du nœud de convolution, nous soulignons que ce décalage ne concerne que l'organisation interne des données puisque, pour cet exemple, les buffers d'entrée et celui de sortie ne doivent pas nécessairement correspondre physiquement par opposition au cas général des filtres IIR.

Eviter le rechargement des données

Le triple buffering trouve également un intérêt si nous souhaitons contourner la solution du rechargement (notamment dans le cas où l'impact sur les performances serait trop important). Dans ce but, nous décrivons avec le tableau 2-5 l'utilisation suggérée des bancs pour le triple buffering. Ce tableau montre que les données produites avec la précédente requête DMA restent présentes en mémoire pendant toute la durée du traitement des nouvelles données. Tout en faisant tourner le rôle des bancs, nous pouvons envisager d'exploiter les données précédemment produites sans les recharger de la mémoire externe. Dans ce partitionnement, nous faisons coïncider le buffer d'entrée et de sortie du traitement, ce qui suggère, pour l'exemple de la convolution comme pour celui de la formulation générale des filtres IIR, de combiner la technique du triple buffering avec celle du décalage interne des données.

Table 2-5. Temporisation de la présence des données avec le triple buffering

Jeu de requêtes	**Action par banc de données (t:2..N-1)** • Voir légende de la Tabl e2-3 • "P || F": traitement et transfert DMA concurrent accédant au même banc • "P^{I+O}": traitement consommant et produisant des données dans le même banc		
	Banc 0	Banc 1	Banc 2
t % 3 = 0	$P^{O}_{t-2} \parallel F_{t-2}$	P^{I+O}_{t-1}	D_t
t % 3 = 1	D_t	$P^{O}_{t-2} \parallel F_{t-2}$	P^{I+O}_{t-1}
t % 3 = 2	P^{I+O}_{t-1}	D_t	$P^{O}_{t-2} \parallel F_{t-2}$

Réduire les effets de contention d'accès

Nous avons déjà évoqué le fait que le triple buffering revient à augmenter la granularité du patron de traitement qui doit s'impliquer davantage dans la gestion des transferts. Nous pouvons cependant relativiser cet impact négatif sur les performances (qui est avant tout fonction de la granularité moyenne de la chaîne) au regard de l'accélération que peut avoir la mise en place du triple buffering sur certains n œuds et qui permet de réduire les phénomènes de contention. Ils sont de deux natures.

La première concerne les contentions d'accès initiés depuis le coeur CPU. Dans ce sens, nous rappelons que l'architecture VLIW des PP autorise deux accès concurrents à la mémoire interne durant le même cycle dès lors que ces accès concernent des bancs de données distincts (sinon l'instruction VLIW correspondante s'exécute en deux cycles). A cet égard, il faut souligner que l'utilisation des techniques d'optimisation pour l'implantation des nœuds comme le déroulage de boucle ou le pipeline logiciel qui suscitent l'exécution d'un plus grand nombre d'opérations en parallèle, engendre une probabilité plus grande de rencontrer ce type de contention.

La deuxième source de contention est celle qui intervient lorsque le processeur et le DMA accèdent au même banc de données internes. La gestion des bancs du tableau 2-5 illustre un tel exemple de contention intervenant lors de l'accès par le nœud de traitement aux données précédemment produites (P^{O}_{t-2}). Cet accès coïncide en effet avec le banc utilisé de manière concurrente (||) par le DMA pour l'exportation des données (F_{t-2}).

Le triple buffering présenté dans le tableau 2-4 illustre un exemple d'optimisation que permet l'utilisation de trois buffers dans le contexte où l'implantation du nœud introduirait un accès concurrent de l'entrée (lecture) et de la sortie (écriture) du coeur de traitement. La présence de bancs distincts pour les opérations de lecture/écriture permet alors de gagner un cycle par itération de traitement. Cet exemple de partitionnement du rôle des bancs s'applique aussi bien à l'opérateur de convolution qu'aux filtres IIR et supprime toute

contention de type DMA/CPU. Si nous revenons à l'utilisation du triple buffering comme technique pour éviter l'approche du rechargement des patrons se recouvrant, nous constatons une possible contention de type CPU/CPU sur le tableau 2-5 en plus de celle existante avec le DMA. Afin de réduire l'impact de cette dernière, une idée simple consiste alors à favoriser le lancement de la requête de type D_t avant celle F_{t-2} afin que la ré-utilisation des données (accès à P^O_{t-2}) se termine avant le démarrage de l'exportation de ces mêmes informations.

Conclusions

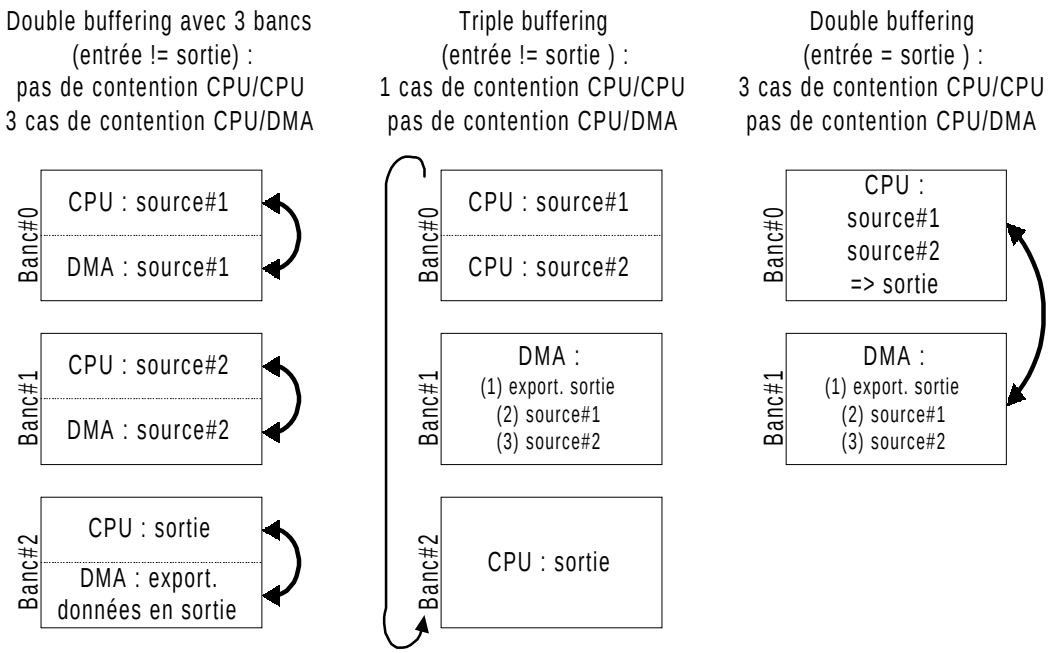
Pour conclure sur cet aperçu des avantages qu'offre le support du triple buffering, nous constatons que les cas de figure sont nombreux et que l'intérêt et la manière d'implanter l'approche dépendent véritablement du contexte. Nous avons montré plusieurs avantages à l'utilisation du triple buffering sur la base des principales structures élémentaires de filtres évoquées (FIR, IIR, et convolution 1D/2D). De même, notre approche adhère bien à l'objectif consistant à ne pas intégrer la gestion des transferts externes au niveau des nœuds. A l'inverse, nous avons déjà souligné que, dans le contexte du C80, le triple buffering requiert l'utilisation de l'ensemble des principales ressources de la mémoire interne par processeur (bien sûr, nous pourrions envisager un triple buffering sur un sous-espace des bancs internes, mais il en résulterait des phénomènes importants de contention) et correspond à une limitation importante de l'approche. Pour ces raisons, nous choisissons d'intégrer le double et le triple buffering au sein de la librairie de gestion du DMA sur C80 détaillée au point 2.3, l'utilisation de l'une ou l'autre des méthodes pour l'optimisation des transferts étant alors laissée au choix de l'utilisateur.

2.2.2.2 Gestion des bancs mémoire avec plusieurs flux

La gestion des bancs de mémoire pour les nœuds diadiques ou triadiques (deux/trois flux de données en entrée) augmente les sources de contention. Avec les trois bancs internes du C80, nous exposons ainsi nécessairement ce type de nœuds à une réduction des performances.

En revanche, nous sommes en mesure d'optimiser l'impact des contentions en modulant l'utilisation spatiale et temporelle des bancs. La Figure2-13 montre dans ce sens trois exemples de partitionnement interne allant d'une configuration n'impliquant pas de contention de type CPU/DMA à celle n'en favorisant aucune de type CPU/CPU.

Figure 2-13. Optimisation de la gestion des bancs pour le cas de nœuds diadiques



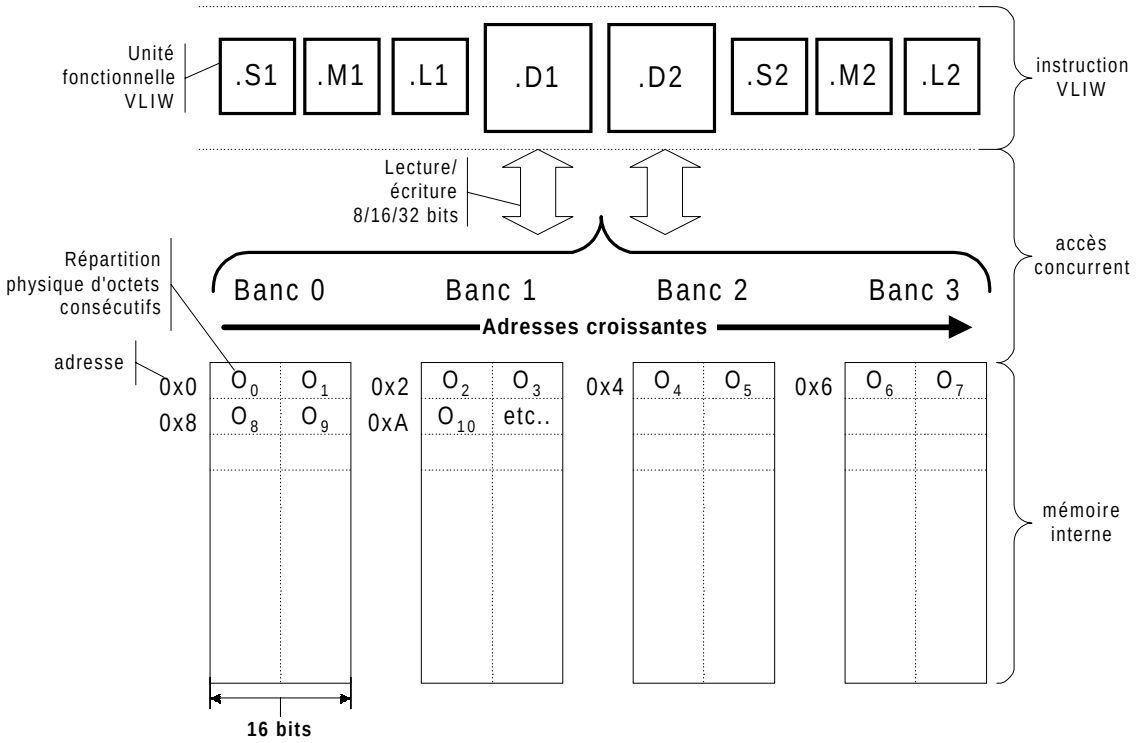
2.2.2.3 Portabilité de l'approche : le cas du C6X

Notre méthodologie se veut portable pour le cas général d'architectures combinant la présence de DMA et de zones de données internes à usage programmable. Si la plupart des processeurs DSP n'introduisent pas un moyen véritablement performant pour implanter le triple buffering, elles offrent en revanche très souvent une forme optimisée de support pour le double buffering. Dans cette partie, nous évoquons brièvement le cas du C6X et

montrons comment il convient d'implanter le double ou triple buffering pour cette architecture. Cette étude de portabilité souligne également la généricité des mécanismes que nous pouvons mettre en place pour gérer le couple DMA/mémoire interne avec quelques aménagements mineurs suivant les spécificités de la classe d'architectures visée par notre méthodologie.

Tout comme le C8X, le C6201 et le C6701 autorisent deux chargements simultanés dans la même instruction VLIW. Là encore, la durée des références est pénalisée d'un cycle lorsqu'il y a contention de l'accès à un même bloc, qu'il s'agisse d'une contention CPU/CPU ou CPU/DMA. Cependant, pour réduire cet impact, le constructeur s'appuie sur quatre bancs de mémoire interne d'une largeur de 16 bits sur lesquels les octets apparaissent entrelacés comme le présente la Figure2-14.

Figure 2-14. Entrelacement des bancs de mémoire interne pour le C6201 et C6701

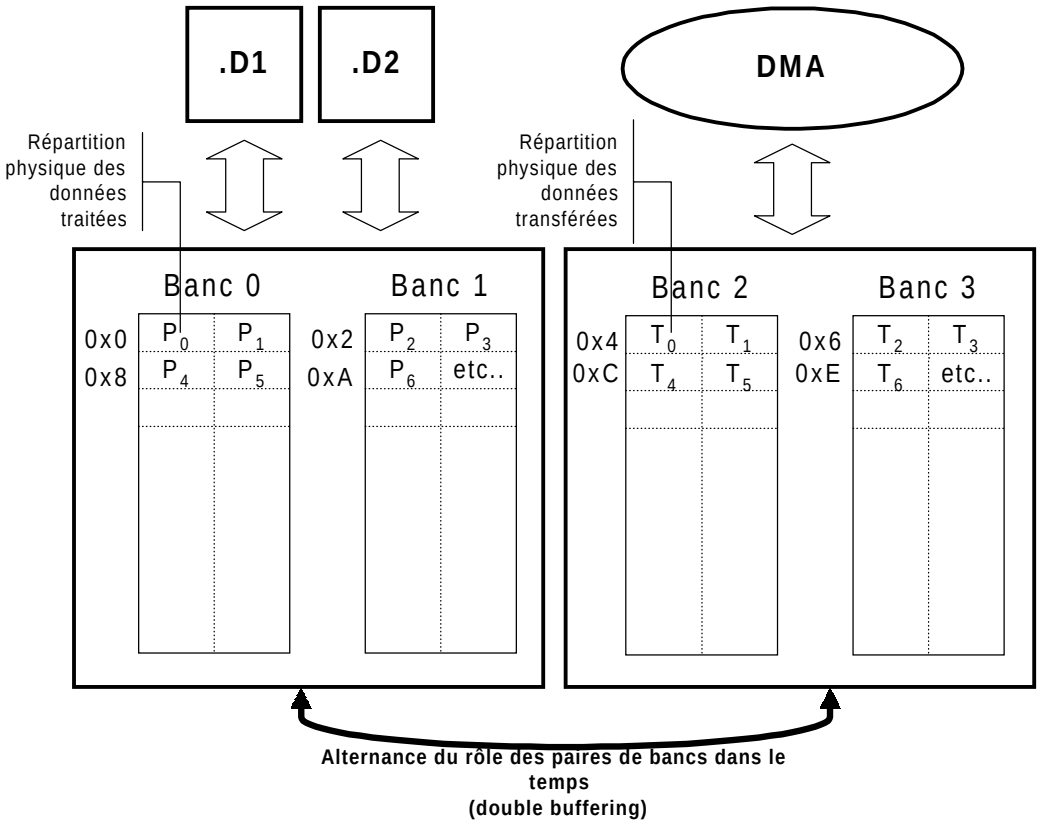


Au niveau de l'instruction VLIW, cette organisation autorise le chargement parallèle sans contention de données qui ne résident pas sur les mêmes bancs. Ainsi, le chargement concurrent de deux valeurs 16 ou 32 bits consécutives s'effectue en un cycle du pipeline (hors contention CPU/DMA) alors que le chargement de données 8 bits issues du même banc introduit un cycle de latence. Cette situation pénalise ainsi les algorithmes de

traitement d'images s'appuyant sur des pixels 8 bits. C'est notamment pour cette raison que le lisseur récursif implanté dans la suite de ce mémoire, s'appuie sur des données 16 bits.

La gestion du double ou triple buffering sans contention est rendue plus complexe par cette organisation des données. Si nous souhaitons implanté le double buffering sans qu'il y ait de contention entre le coeur et le DMA, nous pouvons choisir un partitionnement tel que celui présenté avec la Figure2-15.

Figure 2-15. Double buffering sur le C6201 et C6701



Au niveau du nœud de traitement comme au niveau du transfert DMA concurrent, cela revient à adresser une sous partie non-contiguë de la mémoire interne. Pour le DMA, cette gestion s'appuie sur l'utilisation d'une dimension de transfert (il en supporte trois comme pour le C8X) alors qu'au niveau du nœud, ce partitionnement nécessite l'intégration d'un mécanisme d'adressage des données plus sophistiqué (qui requiert l'introduction d'un pas entre deux lignes de données internes).

Pour permettre une gestion simplifiée (et plus performante) du double buffering, l'avénement récent du C6202 introduit deux jeux de quatre bancs de mémoire 16 bits distincts. En revanche, le support du triple buffering sur cette famille d'architectures nécessite, jusqu'à ce jour, un adressage comparable à celui évoqué avec la Figure2-15. Nous rencontrons le même type de restrictions avec le Sharc 21160 qui n'offre que deux bancs de mémoire (les données ne sont pas entrelacées). En revanche, les spécifications du TigerSharc montrent le support de trois bancs distincts de données offrant ainsi, comme pour le C80, un support performant et simplifié pour l'utilisation du triple buffering.

2.2.3 Partitionnement SPMD flexible des données

Dans cette partie, nous détaillons tout d'abord un modèle pour l'estimation de la quantité globale de données transférées à partir de la description générique d'une chaîne de traitement intégrant l'ensemble des entités que nous avons définies. Ensuite, en s'inspirant de ce modèle, nous proposons une procédure flexible pour le partitionnement SPMD des données entre les différents processeurs et suivant une approche synchrone. Le but de notre procédure est de permettre ainsi un découpage transparent des données des buffers qui vise, dans le même temps, à réduire le coût des transferts parallèles. Avec cette optique combinée aux techniques présentées de gestion performante des bancs internes, nous visons une amélioration des cadences de traitement.

Le raisonnement est progressif : nous commençons par le cas d'une chaîne mono-n œud avec le découpage d'un unique buffer, puis nous envisageons le cas multi-buffers. Ensuite, nous appréhendons le cas des chaînes multi-nœuds.

2.2.3.1 Modélisation du coût des transferts selon notre méthodologie

Nous cherchons tout d'abord à exprimer la quantité globale de données à transférer ainsi que le nombre associé de requêtes DMA. Le modèle analytique recherché a notamment pour but d'intégrer la flexibilité des contraintes sur la géométrie des patrons de traitement. Nous visons également une formulation générique permettant d'estimer le coût des transferts parallèles afin, par la suite, de proposer une approche pour l'optimisation de cette

durée. Cette analyse va nous fournir, dans le même temps, une partie importante du modèle de performance développé à la fin de ce chapitre.

Pour commencer, nous redéfinissons les paramètres des patrons de manière à faire apparaître la taille, en octets, des côtés de la surface rectangulaire correspondant au minimum des données requises pour l'exécution d'une itération du patron interne (a×b) et externe (A×B). Ces paramètres a,A,b,B permettent d'intégrer l'espace occupé par les données qui ne sont pas traitées dans la direction horizontale lorsque $D_p > D_s$ ou celles correspondant à une quantité d'espace interne réservé avec $d_p > d_s$. Nous spécifions ainsi un nouveau jeu de paramètres pour le patron interne et externe :

Equation 2-2. Re-définition de la taille des patrons internes/externes en octet

$$\begin{cases} a = d_s + (w_t - 1) \times d_p,\ b = h_t,\ c = w_s,\ d = h_s \\ A = D_s + (W_t - 1) \times D_p,\ B = H_t,\ C = W_s,\ D = H_s \end{cases}$$

Ici, nous rappelons imposer à la valeur du pas horizontal d'être multiple du pas de donnée autrement dit $C \bmod D_p = 0$. Pour le patron externe, nous redéfinissons également la quantité de données rechargées, par défaut, lorsque W_r et H_r ne sont pas explicitement spécifiés :

Equation 2-3. Quantité de données rechargées par défaut et par direction

$$\begin{cases} W_r = E = (A - C) \\ H_r = F = (B - D) \end{cases}$$

Par la suite, nous considérerons $E < A$ et $F < B$. La quantité de données rechargées apparaît également sous l'angle de la mémoire interne et nous définissons par conséquent les paramètres e et f par :

$$\begin{cases} e = (E / D_p) \cdot d_s \\ f = F \end{cases}$$

Avec l'ensemble de ces nouveaux paramètres, nous redéfinissons désormais l'aire $W \times H$ du buffer véritablement parcourue lors du traitement[1] :

Equation 2-4. Troncature de la taille d'image réellement traitée

$$\begin{cases} W = A + C \cdot \left\lfloor \dfrac{W_{ref} - A}{C} \right\rfloor \\ H = B + D \cdot \left\lfloor \dfrac{H_{ref} - B}{D} \right\rfloor \end{cases} \text{ avec } \begin{cases} W \geq A \\ H \geq B \end{cases}$$

A partir de l'aire $W \times H$, nous pouvons alors distinguer la quantité de données réellement utile au traitement et qui trouve un sens lorsque $D_p > D_s$. De cette quantité dépend directement le coût des transferts de données. Nous définissons alors la surface utile par $W^u \times H^u$ définie par :

Equation 2-5. Surface $W_u \times H_u$ des données utile

$$\begin{cases} W^u = D_s \cdot W_t \cdot \left(1 + \dfrac{W - A}{C}\right) \\ H^u = H \end{cases}$$

Dans le même sens, nous distinguons alors les données utiles associées au rechargement pour la direction horizontale avec E^u et verticale avec F^u :

Equation 2-6. Données utiles du rechargement

$$\begin{cases} E^u = (E / D_p) \cdot D_s \\ F^u = F \end{cases}$$

De là, la quantité globale de données transférées en parallèle du traitement peut se décomposer en deux termes. Nous avons d'une part la quantité de données relatives à la surface traitée avec $W^u \times H^u$. Ensuite, nous distinguons la quantité globale de données rechargées lorsque E^u ou F^u sont différents de zéro. Nous associons à ce deuxième terme caractérisant des transferts redondants de données, la quantité de données globalement rechargées pour la direction horizontale, R_W, et verticale avec R_R. La quantité globale Q de données transférées peut alors s'écrire :

Equation 2-7. Quantité Q globale de données transférées

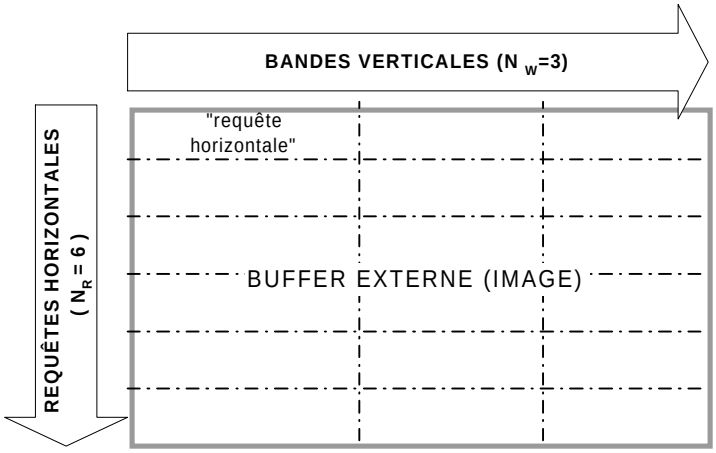
$$Q = \left(W^u + R^u_W\right) \times \left(H^u + R^u_R\right) \Rightarrow W^u \times H^u + R^u(R_W, R_R)$$

---

1. Dans ce mémoire, nous adopterons la notation "$\lfloor \mathbf{x} \rfloor$" pour spécifier la fonction partie-entière arrondissant un entier décimal à sa valeur entière inférieure (fonction *floor()* du langage C) et "$\lceil \mathbf{x} \rceil$" pour l'arrondi à la valeur entière supérieure (fonction *ceil*() du langage C). En particulier, si **x** est entier, nous prenons $\mathbf{x} = \lceil \mathbf{x} \rceil = \lfloor \mathbf{x} \rfloor$.

Afin d'estimer la fonction $R^u(R_W,R_R)$, nous définissons les paramètres N_W et N_R qui déterminent le nombre de bandes verticales (N_W) de N_R requêtes DMA horizontales nécessaires au parcours des données de l'image. Cette terminologie est illustrée avec la figure suivante :

Figure 2-16. Définitions des paramètres N_W et N_R



Ces paramètres dépendent du nombre de patrons que nous sommes capable de stocker temporairement dans le cache interne de taille *S*. Si β correspond au nombre de patrons que nous décidons de stocker horizontalement et γ à celui du nombre de lignes de β patrons stockés dans *S-a×b*, nous introduisons alors les égalités suivantes (avec $\beta \geq 0, \gamma \geq 0$) :

Equation 2-8.

$$N_W(\beta) = \max\left(1, \left\lceil \frac{W - A}{\beta \cdot C} \right\rceil\right)$$

$$N_R(\gamma) = \max\left(1, \left\lceil \frac{H - B}{\gamma \cdot D} \right\rceil\right)$$

Lorsque β et/ou γ sont égaux à zéro, nous définissons $N_W(\beta) = 1$ et/ou $N_R(\gamma) = 1$. La quantité de données rechargées pour les deux dimensions définies par $R^u(R_W,R_R)$, dépend des deux paramètres précédents et s'écrit alors :

Equation 2-9. Quantité R de données rechargées

$$R^u(N_W(\beta),N_R(\gamma)) = H^u \cdot R^u_W(N_W(\beta)) + \left(W^u + R^u_W(N_W(\beta))\right) \cdot R^u_R(N_R(\gamma))$$

avec

$$\begin{cases} R^u_W(N_W(\beta)) = E^u \cdot (N_W(\beta) - 1) \\ R^u_R(N_R(\gamma)) = F^u \cdot (N_R(\gamma) - 1) \end{cases}$$

Par ailleurs, nous établissons un système de contraintes afin de borner l'espace des valeurs possibles pour β et γ:

Equation 2-10. Contraintes sur le remplissage du cache interne

$$(\psi) = \begin{cases} (a + \beta \cdot c) \times (b + \gamma \cdot d) \leq S \\ A + \beta \cdot C \leq W \\ B + \gamma \cdot D \leq H \end{cases}$$

Avec l'ensemble de ces éléments, nous introduisons une fonction objectif pour le système de contraintes précédent qui intègre le coût du rechargement $R^u(N_W(\beta),N_R(\gamma))$ qu'il est intuitivement possible de minimiser en maximisant le remplissage du cache interne. D'une manière plus rigoureuse, la fonction intègre également le coût lié au lancement des requêtes horizontales et celui qu'implique le traitement d'une nouvelle bande verticale. Ces derniers coûts incluent le nombre de cycles nécessaires à l'initialisation par les différents patrons de traitement d'un nouveau jeu de requêtes DMA (ν) et celui de la gestion d'une nouvelle bande verticale lorsque $N_W > 1$ (μ). Ces durées traduisent notamment le coût indirect de la gestion des caches d'instructions après chaque exécution de la chaîne. La fonction objectif à minimiser pour le précédent système peut donc s'écrire :

Equation 2-11. Fonction objectif du système contraint

$$c'(\beta,\gamma) = \mu \cdot (N_W(\beta) - 1) + \nu \times N_W(\beta) \times N_R(\gamma) + R^u(N_W(\beta), N_R(\gamma))$$

Cette équation conclut l'introduction des grandes lignes de notre modèle pour l'estimation du coût des transferts, entièrement paramétrable et s'appuyant bien sur la notion générique du patron de données.

Nous soulignons que ce modèle n'implique, jusqu'alors, qu'un seul buffer et patron interne/externe associé. Nous étendons désormais ce modèle, en commençant par le cas d'une chaîne mono-n œud comportant un buffer en entrée et un autre en sortie. Le

partitionnement des données passe alors par l'estimation globale d'une solution pour le couple (β,γ) et que nous exprimons avec le système suivant :

Equation 2-12. Systèmes de contraintes pour une chaîne mono-nœud avec deux buffer

$$\underset{\beta \geq 0,\gamma \geq 0}{\text{Min}} \quad (C'(\beta,\gamma)), \text{ avec } \begin{cases} (a^I + \beta \cdot c^I) \times (b^I + \gamma \cdot d^I) \leq S^I \\ (a^O + \beta \cdot c^O) \times (b^O + \gamma \cdot d^O) \leq S^O \\ A^I + \beta \cdot C^I \leq W^I \\ A^O + \beta \cdot C^O \leq W^O \\ B^I + \gamma \cdot D^I \leq H^I \\ B^O + \gamma \cdot D^O \leq H^O \end{cases}$$

La valeur commune des paramètres β et γ pour le buffer d'entrée (*I*) comme pour celui de sortie (*O*) souligne le synchronisme recherché dans la mesure où, en effet, nous considérons qu'il y a autant de pas complémentaires en entrée qu'en sortie. Toutefois, nous rappelons que le synchronisme, à ce niveau, n'est pas strict. Nous pouvons, en effet, faire varier avec la valeur des pas, la fréquence du nombre de données consommées/produites entre l'entrée et la sortie du nœud (*x* données en entrée pour *y* en sortie).

Bien que ces variations soient possibles, elles doivent cependant rester synchrones comme le suggèrent les équations précédentes. A ce titre, nous pouvons tout de même souligner que ce synchronisme ne concerne que les données qualifiées de "complémentaires" (multiples des pas). Les données regroupées dans l'aire minimum paramétrable $A \times B$ permettent d'introduire une rupture dans l'équivalence stricte du n pour m. De ce point de vue, notre modèle étend la définition du flot de données synchrone (SDF). En revanche, afin de simplifier l'écriture du patron de traitement et de diminuer les coûts (μ,ν) de la gestion des requêtes DMA, nous préconisons une synchronisation stricte du nombre de requêtes en entrée et en sortie (une pour une). La contrepartie de ce raisonnement est qu'il ne permet pas un remplissage optimal de l'espace S pour le cache interne. L'hypothèse stricte de synchronisation des requêtes en entrée avec celles en sortie nous permet alors de définir la fonction objectif *C'* dans le cas mono nœud par :

Equation 2-13. Fonction objectif vers un partitionnement optimal

$$\alpha(\beta) = N'_W(\beta) = \max\left(N_W^i(\beta), N_W^o(\beta)\right)$$
$$\varepsilon(\gamma) = N'_R(\gamma) = \max\left(N_R^i(\gamma), N_R^o(\gamma)\right)$$
$$C'(\beta,\gamma) = \mu \cdot (\alpha - 1) + \nu \cdot \alpha\varepsilon + R^{u/i}(\alpha, \varepsilon) + R^{u/o}(\alpha, \varepsilon)$$

Cette nouvelle fonction montre que nous avons bien un nombre de requêtes horizontales et verticales identique pour les deux buffers de l'exemple. Par ailleurs, nous faisons l'hypothèse réaliste qui consiste à estimer le coût de l'initialisation des requêtes et bandes comme constant quel que soit le nombre de buffers.

Cet exemple nous permet alors de mieux appréhender la généralisation du système à résoudre pour l'estimation du nombre de requêtes DMA total à partitionner. En considérant une chaîne de traitement faisant intervenir une quantité K de buffers d'indice k (qu'il s'agisse de buffers en entrée ou en sortie), nous avons un système à résoudre et à optimiser qui peut prendre la forme suivante :

Equation 2-14. Modèle proposé pour la résolution du découpage de buffers reliés à un nœud

$$\underset{\beta \geq 0, \gamma \geq 0}{\text{Min}} \; [C'(\beta,\gamma) = \alpha(\beta)[\varepsilon(\gamma) \cdot (\nu + \pi) + \rho + \mu - \pi] + \varepsilon(\gamma) \cdot \sigma - (\mu + \sigma + \rho)]$$

$$\text{avec} \begin{cases} \alpha(\beta) = \max_k\left(\max\left(1, \left\lceil \frac{W_k - A_k}{\beta \cdot C_k} \right\rceil\right)\right), \quad \varepsilon(\gamma) = \max_k\left(\max\left(1, \left\lceil \frac{H_k - B_k}{\gamma \cdot D_k} \right\rceil\right)\right) \\ \pi = \sum_k F_k^u \cdot E_k^u, \rho = \sum_k H_k^u \cdot E_k^u, \sigma = \sum_k F_k^u \cdot \left(W_k^u - E_k^u\right) \end{cases}$$

$$\text{pour } (\beta,\gamma) \text{ vérifiant } (\psi) = \begin{cases} (a_k + \beta \cdot c_k) \times (b_k + \gamma \cdot d_k) \leq S_k \\ A_k + \beta \cdot C_k \leq W_k \\ B_k + \gamma \cdot D_k \leq H_k \end{cases}$$

Il s'agit d'un système de programmation entière non-linéaire puisque nous avons un couplage des inconnues entières β et γ au niveau de la première inégalité de (ψ) comme pour la fonction objectif. De plus, l'utilisation des fonctions *max* combinées à celle des parties entières pour la synchronisation des requêtes avec $\alpha(\beta)$ et $\varepsilon(\gamma)$, ajoute encore à la non-linéarité de ce système sous contraintes.

2.2.3.2 Partitionnement performant des données

La résolution optimale du système précédent est complexe et réputée NP complet. Dans ce contexte et compte tenu du fait que nous visons une détermination du couple (β,γ) qui intervienne lors de l'exécution du programme pour permettre une re-configuration dynamique et performante des traitements, nous proposons une méthode de résolution sous-optimale qui engendre un nombre restreint de calculs.

2.2.3.2.1 Découpage d'un buffer

En faisant abstraction des coûts (μ,ν), nous pouvons chercher à réduire la quantité globale de requêtes en maximisant simplement le remplissage du cache interne de chaque buffer. Dans ce cas, nous proposons une nouvelle fonction objectif qui conserve le système de contraintes (ψ) et s'écrit :

Equation 2-15. Fonction objectif et remplissage du cache

$$c(\beta,\gamma) = \min_{\beta,\gamma}(S - ab - \beta cb - \gamma ad - \beta\gamma cd)$$

Pour résoudre cette fonction d'une manière simplifiée, nous pouvons raisonner dans R^+ et prendre $\beta=\gamma$ pour introduire un polynôme du deuxième ordre et prendre la partie entière de la racine positive de cette nouvelle fonction. Pour améliorer cette solution, nous pouvons mettre en avant le résultat théorique avancé dans [40] (p. 181) où il est démontré que le partitionnement optimal d'une image $H\times W$ avec q processeurs (ou q requêtes DMA dans notre cas), correspond à des carrés dont le côté s'écrit :

Equation 2-16. Partitionnement optimal des données d'une image W×H par p processeurs

$$\sqrt{\frac{W \times H}{q}}$$

Ce partitionnement minimise la périphérie des blocs de données et réduit ainsi l'effet de bord qui se traduit, à notre niveau, par une optimisation de la quantité globale R rechargée. De là, nous pouvons utiliser ce résultat pour pondérer l'égalité $\beta=\gamma$ afin de réduire R. Dans cette optique, il nous suffit de trouver une valeur de $\beta=\gamma$ engendrant, au niveau du patron externe, une zone carrée de données cachées (aux arrondis près). Dans R, cela donne :

Equation 2-17. Pondération des inconnues pour la fonction objectif quadratique

$$\beta = \frac{B + (\gamma \cdot D) - A}{C}$$

Cette approche permet de trouver une "bonne" solution pour la fonction objectif mais fait abstraction des coûts μ et ν. Plus précisément, l'impact μ qu'engendre le traitement d'une nouvelle bande verticale lorsque $N_W > 1$ n'est pas négligeable (de l'ordre de plusieurs milliers de cycles), notamment lorsque la durée des transferts et des traitements est faible (comme pour le cas d'opérateurs simples appliqués à des images de petite taille). Pour tenir compte de ce contexte, nous proposons une solution très simple qui consiste à résoudre l'équation *c* par rapport à β tout en maximisant sa valeur afin de réduire l'impact de R_W et d'en déduire une valeur pour γ. Cette approche est cohérente dans la mesure où la surface associée à la fonction *c* (comme celle de *C'*) est monotone décroissante pour des valeurs croissantes de (β,γ). Nous obtenons ainsi également une solution optimale dans R^+ qui s'avère valable dans N^+. La procédure que nous suivons alors consiste tout d'abord à estimer le nombre maximum de patrons internes que nous pouvons stocker, en ligne, dans la mémoire interne et que nous écrivons par :

Equation 2-18.

$$\xi = \left\lfloor \frac{S/b - a}{c} \right\rfloor$$

Nous déduisons le nombre de bandes verticales nécessaires au traitement que nous écrivons :

Equation 2-19.

$$N_W = 1 + \left\lceil \frac{\max(0, W - (A + \xi \cdot C))}{\max\left(\xi \cdot C, \underbrace{E + o \cdot C}_{\Upsilon}\right)} \right\rceil, \; o = \left\lceil \frac{A - E}{C} \right\rceil \text{ si } E \neq 0, \; 0 \text{ sinon}$$

Lorsque nous ne sommes pas en mesure de cacher une ligne entière de patrons ($N_W > 1$) qui se traduit ici par un numérateur non nul au sein de la partie entière à droite de N_W, nous estimons alors le nombre de bandes verticales supplémentaires à l'aide du dénominateur de ce même ratio. Les termes *o* et Υ permettent de nous assurer qu'avec le rechargement des données, nous retrouvons bien une quantité minimale au moins égale à *A*.

Nous définissons les paramètres $N_{\beta 1}$ et $N_{\beta 0}$ qui correspondent, respectivement, au nombre de patrons externes complémentaires (ou "pas" complémentaires) pour les bandes verticales de largeur type W_1 et W_0 que nous définissons tout d'abord par :

Equation 2-20.

$$W = W_1 + (N_W - 1) \cdot W_0$$

Ce découpage nous permet d'intégrer le cas où W n'est pas multiple de N_W. De là, nous proposons d'obtenir $N_{\beta 1}$ et $N_{\beta 0}$ par :

Equation 2-21.

$$\begin{cases} N_{\beta 0} = \begin{cases} N_W = 1 \Rightarrow 0 \\ N_W > 1 \Rightarrow o + \left\lfloor \dfrac{W + (E - o \cdot C) \cdot (N_W - 1)}{C \cdot N_W} \right\rfloor \end{cases} \\ N_{\beta 1} = \dfrac{W - A}{C} - N_{\beta 0} \cdot (N_W - 1) \end{cases}$$

ce qui nous donne :

Equation 2-22.

$$\begin{cases} W_1 = A + N_{\beta 1} \cdot C \\ W_0 = N_{\beta 0} \cdot C \end{cases}$$

Nous notons que le terme W_0 ne fait pas apparaître la quantité E de données rechargées alors que le calcul du paramètre $N_{\beta 0}$ qui intervient lorsque $N_W > 1$ vise bien à répartir la quantité globale de données pour la direction horizontale (y compris les données rechargées). Ce découpage n'est autorisé que si l'inégalité suivante est vérifiée :

Equation 2-23.

$$W \geq A + (N_W - 1) \times C$$

Ensuite, nous évaluons les largeurs w_1 et w_0 des lignes de patrons auxquelles correspondrait, en mémoire interne, le stockage d'une ligne complète provenant de la mémoire externe pour chaque type de bande :

Equation 2-24.

$$w_1 = a + N_{\beta 1} \times c$$

$$w_0 = \begin{cases} N_W = 1 \Rightarrow 0 \\ N_W > 1 \Rightarrow e + N_{\beta 0} \times c \end{cases}$$

Nous pouvons désormais revenir à l'estimation du nombre de patrons complémentaires à ceux déjà contenus dans l'espace du cache a×b que nous écrivons $\beta \times \gamma$ et qui se définit par :

Equation 2-25.

$$\beta = \max(N_{\beta 1}, N_{\beta 0})$$

$$\gamma = \left\lfloor \frac{\underbrace{(a + c \cdot \xi)}_{\theta} \cdot \frac{b}{w} - b}{d} \right\rfloor, w = \max(w_1, w_0)$$

Ici, lorsque $N_W > 1$, nous simplifions le découpage et raisonnons par rapport à la plus grande largeur *w* de bande d'image verticale que nous allons traiter. De plus, pour permettre d'améliorer le remplissage face au problème d'arrondis intervenant au niveau de l'utilisation des parties entières, nous bornons la définition de γ par :

Equation 2-26.

$$\frac{S/b - a}{c} \leq \xi < 1 + \frac{S/b - a}{c}$$

$$\Rightarrow a + c \times \frac{S/b - a}{c} \leq \theta < a + c \times \left(1 + \frac{S/b - a}{c}\right)$$

$$\Rightarrow \left(\delta = \frac{S - b \times w}{d \times w}\right) \leq \gamma < 1 + \frac{b(c - w) + S}{d \times w}$$

La différence entre les deux dernières bornes s'écrit alors :

Equation 2-27.

$$\frac{b \times c}{d \times w} + 1$$

En considérant $w \times d >> b \times c$, il suffit d'évaluer la borne inférieure δ puis de vérifier :

Equation 2-28.

$$\gamma = \lfloor \delta \rfloor, \theta \times (b + \gamma \cdot d) \leq S$$

Lorsque l'inégalité est vérifiée, nous prenons $\gamma=\gamma+1$. Dans certains cas, cette optimisation permet de cacher une ligne de patrons supplémentaire et diminue alors le nombre de requêtes nécessaires ainsi que le coût de l'éventuel rechargement des données.

Une valeur de γ étant établie, nous sommes désormais en mesure de partitionner l'image en N_H bandes horizontales, chacune assignée à un PP. En définissant σ_{ref} comme le nombre initial de PP assigné au traitement, d'une manière comparable à l'équation 2-19, nous définissons σ comme le nombre effectif de PP utilisés par :

Equation 2-29.

$$\sigma = N_H = \min\left(\sigma_{ref}, 1 + \left\lceil \frac{\max(0, H - (B + \gamma \cdot D))}{\max(\gamma \cdot D, F + p \cdot D)} \right\rceil\right),\ p = \left\lceil \frac{B - F}{D} \right\rceil \text{ si } F \neq 0,\ 0 \text{ sinon}$$

Avec cette approche, nous visons à réduire l'utilisation des PP ainsi que le nombre de requêtes de transfert. Si la durée de traitement des données est telle qu'il devient avantageux d'utiliser le maximum de PP, nous pouvons alors écrire :

Equation 2-30.

$$\sigma = \min\left(\sigma_{ref}, 1 + \frac{\max(0, H - B)}{D}\right)$$

Comme avec l'équation 2-23, nous introduisons alors une contrainte sur H :

Equation 2-31.

$$H \geq B + (\sigma - 1) \times D$$

De là, nous pouvons en déduire le couple $N_{\gamma 0}$ et $N_{\gamma 1}$ qui apparaît comme le nombre de patrons verticaux complémentaires, symétriques de ceux définis par $N_{\beta 1}$ et $N_{\beta 0}$:

Equation 2-32.

$$\begin{cases} N_{\gamma 0} = \begin{cases} \sigma = 1 \Rightarrow 0 \\ \sigma > 1 \Rightarrow \left(p + \left\lfloor \dfrac{H + (F - p \cdot D) \cdot (\sigma - 1)}{D \cdot \sigma} \right\rfloor\right) \end{cases} \\ N_{\gamma 1} = \dfrac{H - B}{D} - N_{\gamma 0} \cdot (\sigma - 1) \end{cases}$$

Nous en déduisons les deux bandes horizontales types traitées qui s'écrivent H_1 et H_0 :

Equation 2-33.

$$H = H_1 + (\sigma - 1) \cdot H_0$$
$$H_1 = B + N_{\gamma 1} \cdot D$$
$$H_0 = N_{\gamma 0} \cdot D$$

Pour chaque catégorie de bande, nous pouvons alors déduire le nombre de requêtes DMA horizontales N_{R0} et N_{R1} correspondantes :

Equation 2-34.

$$N_{R0} = \left\lceil \frac{H_0}{D \cdot (\gamma + 1)} \right\rceil$$
$$N_{R1} = 1 + \left\lceil \frac{\max(0, H_1 - (B + \gamma \cdot D))}{D \cdot (\gamma + 1)} \right\rceil$$

Par ailleurs, la quantité maximale de données cachées s'écrit $w \times h$ avec h se définissant par :

Equation 2-35.

$$h = b + \gamma \cdot d$$

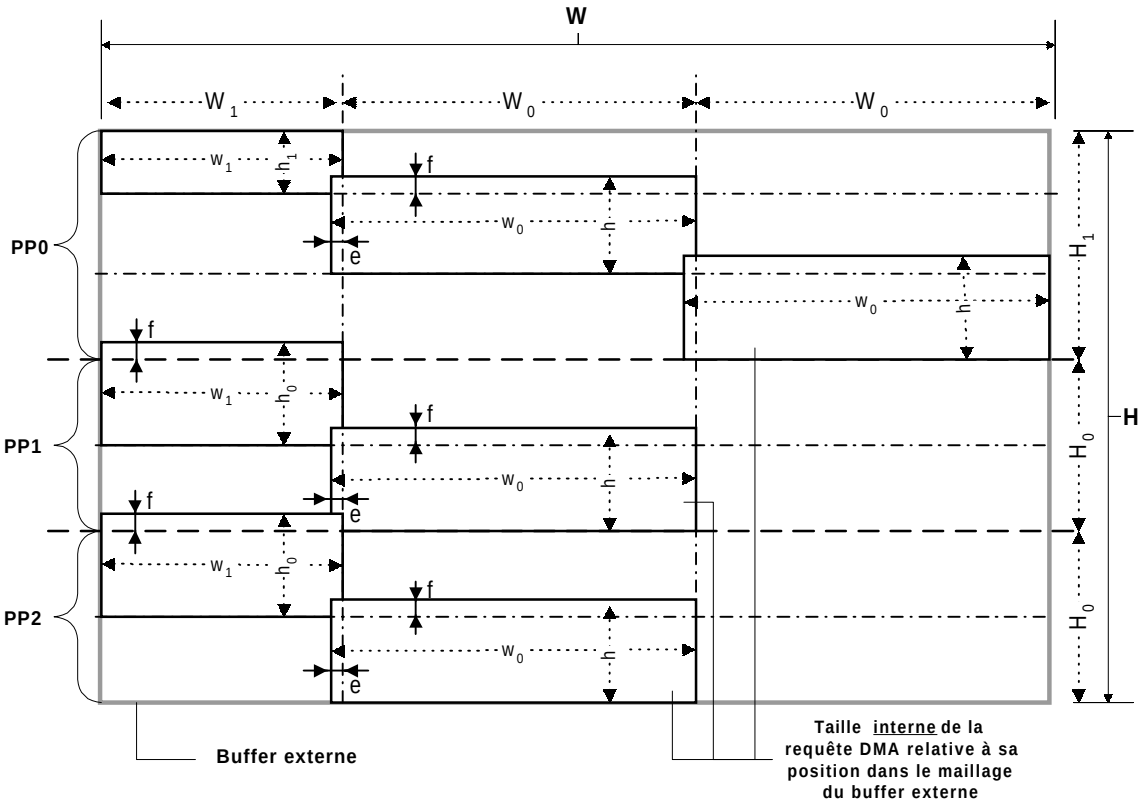
Lorsque les hauteurs H_0 ou H_1-(B+γ·D) ne sont pas multiples de D·(γ+1), nous distinguons alors une première requête ayant une hauteur interne spécifique pour chaque type de bande horizontale ($H_{1/0}$) :

Equation 2-36.

$$N_{R0} > 0 \Rightarrow h_0 = f + \frac{H_0 - (N_{R0} - 1) \cdot D \cdot (\gamma + 1)}{D}$$
$$h_1 = b + d \cdot \frac{H_1 - B - (N_{R1} - 1) \cdot D \cdot (\gamma + 1)}{D}$$

La Figure 2-17 illustre le long cheminement de la procédure de découpage que nous venons de détailler. Cet exemple montre la partitionnement d'un buffer avec σ=3 PP et N_W=*3*, N_{R1}=*3*, N_{R0}=*2*. La figure présente également les différentes tailles internes ($w_{1/0} \times h_{1/0/\_}$) de quelques requêtes de manière relative à leur position externe dans le maillage. Dans ce sens, à l'inverse de ce que suggère le graphique, nous pourrions avoir $W_0 \neq w_0$ et $W_1 \neq w_1$, de même que $E \neq e$ et $F \neq f$.

Figure 2-17. Partitionnement “horizontal” des données



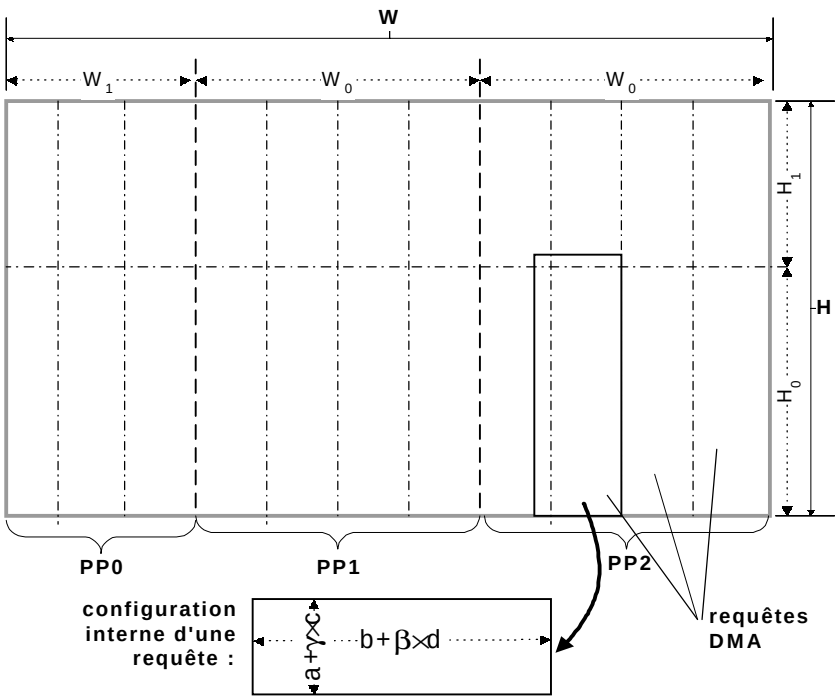
2.2.3.2.2 Ordre de parcours des données du maillage

L’ordre dans lequel les données sont parcourues au sein de chaque bande de PP n’est pas défini. Dans notre contexte, étant donné que le coût de rechargement des données est plus important lorsque nous traitons une nouvelle bande verticale, nous lançons tout d’abord l’ensemble des requêtes associées à la bande W_1 de chaque PP avant de considérer la première bande W_0. Le parcours des requêtes par bande verticale peut se faire de haut en bas ou inversement. De plus, l’ordre dans lequel les données sont lues ou écrites au sein même de chaque requête élémentaire est paramétrable et s’appuie sur le support multi-dimensionnel du DMA. Il peut s’agir de regrouper en mémoire interne, les données d’un bloc sous forme de vecteur et donc de balayer les données de chaque bloc avant de considérer le bloc suivant. Toutefois, pour homogénéiser l’organisation des données en mémoire interne afin de faciliter le chaînage des nœuds, nous choisissons dans la mesure

du possible, une organisation des données en mémoire interne linéaire (juxtaposition des lignes d'images les unes après les autres).

Pour permettre de parcourir l'intégralité de la hauteur de l'image sans qu'il y ait de dépendance de données inter-PP, nous définissons également un sens de parcours que nous qualifions de transposé et que nous illustrons avec la Figure2-18. Cette approche vise à s'affranchir des couplages processeurs avec l'exemple des filtres récursifs appliqués dans la direction verticale. Ce découpage s'appuie sur la même procédure que celle développée, nous transposons simplement les paramètres initiaux $W_{ref} \Leftrightarrow H_{ref}$ ainsi que les paramètres de la direction horizontale et verticale des patrons (avec une seule requête verticale de la Figure 2-18 ($H_0 = 0$), nous aurions par exemple $\beta = (H - B)/D$). Après le découpage, le nombre de bandes verticales correspond bien à la valeur de σ alors que celui pour la direction horizontale est défini par N_W (nous re-transposons les paramètres).

Figure 2-18. Découpage vertical des données



Ce découpage est mis en œuvre par la suite avec l'application de segmentation de contours optimale.

2.2.3.2.3 Le cas multi-buffers

Afin d'étendre l'approche proposée pour le cas d'un unique buffer au contexte d'une chaîne mono-nœud en impliquant plusieurs, nous proposons une procédure sous-optimale, mais simple de mise en œuvre au regard de la complexité qu'impliquerait la résolution optimale du système 2-14, qui se décompose en trois étapes. Dans un premier temps, nous déterminons la surface $W \times H$ que nous allons réellement traiter pour chaque buffer. Elle dépend du nombre de patrons complémentaires minima pour l'ensemble des buffers d'indice i que nous définissons, par direction, avec :

Equation 2-37.

$$\begin{cases} \beta_{MIN} = \min_i \left(\left\lfloor \frac{W_{ref_i} - A_i}{C_i} \right\rfloor \right) \\ \gamma_{MIN} = \min_i \left(\left\lfloor \frac{H_{ref_i} - B_i}{D_i} \right\rfloor \right) \end{cases}$$

De là, nous pouvons déduire, pour chaque buffer i, la surface $W_i \times H_i$ ainsi que le nombre de bandes verticales N_{Wi} nécessaires en fonction de la surface traitée. Nous mémorisons pour la suite, le nombre maximum de bandes horizontales par $N_{W_{MAX}}$. Nous écrivons donc :

Equation 2-38.

$$N_{W_{MAX}} = \max_i (N_{W_i}(W_i, H_i))$$

$$\text{avec} \begin{cases} W_i = A_i + \beta_{MIN} \cdot C_i \\ H_i = B_i + \gamma_{MIN} \cdot D_i \end{cases}$$

Dans une deuxième étape, nous calculons les paramètres β_1 et β_0 qui correspondent au nombre de patrons complémentaires pour la direction horizontale associée aux bandes de type W_1 et W_0 du buffer j pour lequel $N_{Wj}=N_{Wmax}$ (cf. équations 2-21 avec $N_W=N_{Wmax}$). Lorsque plusieurs buffers vérifient cette égalité, nous prenons la valeur minimum de β_1 et β_0 par type de bande. Cela donne :

Equation 2-39.

$$\begin{cases} \beta_1 = \min_j (N_{\beta 1_j}) \\ \beta_0 = \min_j (N_{\beta 0_j}) \end{cases}$$

Sur la base de (β_1,β_0), nous déduisons la valeur minimum de γ pour l'ensemble des buffers inspirée de l'équation 2–26 :

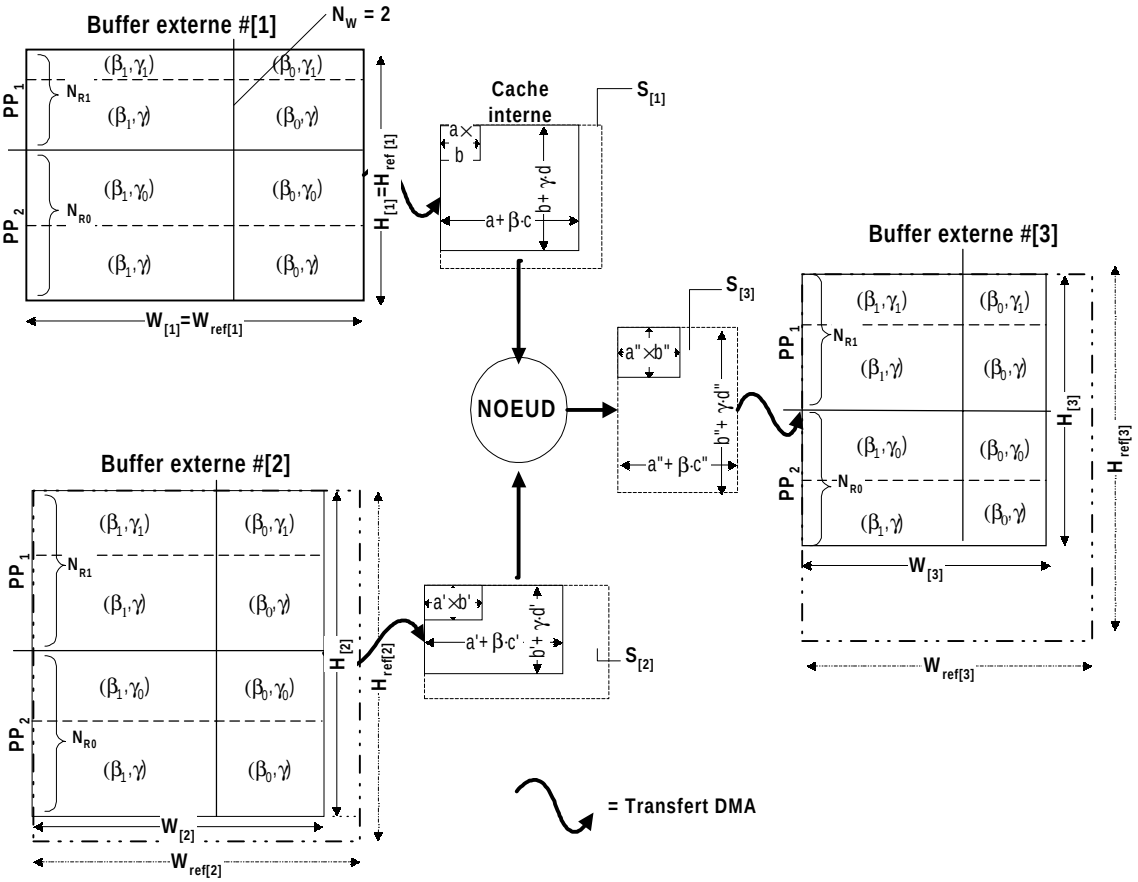
Equation 2-40.

$$\Omega_i = \max(a_i + \beta_1 \cdot b_i, a_i + \beta_0 \cdot b_i)$$

$$\gamma = \min_i\left(\gamma_i = \left\lfloor \frac{S_i - b_i \times \Omega_i}{b_i \times \Omega_i} \right\rfloor + 0/1\right)$$

Enfin, grâce aux paramètres (γ,β_1,β_0,σ_{ref}), nous dérivons les six types de requêtes nécessaires au partitionnement des données et basés sur les couples de paramètres (β_1,γ), (β_0,γ), (β_1,γ_1), (β_1,γ_1), (β_0,γ_0), (β_0,γ_0) qui sont décrit sur la Figure2-19 (où les valeurs de $h_{1/0}$ s'expriment en fonction de $\gamma_{1/0}$). Ici, nous employons les mêmes nombres N_{R1}/N_{R0} de requêtes pour tous les buffers, de même que le nombre σ de PP utilisé est commun à tous les buffers.

Figure 2-19. Découpage multi-buffers et exemple de nœud diadique



2.2.3.3 Cas des chaînes multi-nœuds

> *All well-structured object-oriented architectures are full of patterns. Indeed, one of the ways I measure the quality of an object-oriented system is to judge whether or not its developers have paid careful attention to the common collaborations among its objects. Focusing on such mechanisms during a system's development can yield an architecture that is smaller, simpler, and far more understable than if these patterns are ignored*. *Grady Booch* [65].

La procédure que nous venons de détailler pour le découpage des buffers s'entendait, jusque là, dans le cas d'une chaîne mono-nœud. Pour tirer profit de la localité des données et permettre ainsi l'amélioration des performances, nous visons le chaînage des nœuds en vue de leur exécution séquentielle après chaque opération d'entrée/sortie sur des sous-régions des buffers. Du point de vue de notre méthodologie, nous devons intégrer la combinaison des contraintes sur la taille des données que peut traiter ce type de chaîne. Dans la mesure où nous visons une modélisation synchrone des flots de données au niveau des requêtes DMA, nous devons faire correspondre la quantité de données produites par un nœud avec celle consommée par le nœud suivant dans la chaîne. L'objectif consiste notamment à lier la quantité de données en entrée d'une requête DMA avec celle de la requête synchrone en sortie de la chaîne (ou réciproquement). Pour les chaînes mono-nœud, cette relation est directement exprimée avec la différence de géométrie entre le patron en entrée et celui en sortie du nœud impliqué. Cette correspondance n'est pas directe pour les chaînes multi-nœuds dans la mesure où nous cherchons à lier les contraintes de taille entre deux nœuds. Vis à vis du partitionnement synchrone des données, nous cherchons ainsi à extraire le dénominateur commun des contraintes sur les tailles d'images qu'une chaîne est en mesure de traiter. Pour cela, nous introduisons la notion de patrons virtuels sur la base desquels, ensuite, nous appliquons la procédure du partitionnement des données évoquée au paragraphe 2.2.3.2.3.

2.2.3.3.1 Calcul du patron virtuel

Notre raisonnement débute avec la résolution de l'équation visant à équilibrer la quantité de données produites par un nœud avec celle consommée par le nœud suivant dans la chaîne. Dans cette optique, nous proposons une approche simplifiée conduisant à une résolution *indépendante* pour les directions horizontale et verticale. Dans ce but, nous introduisons les variables β^I et β^O pour la direction horizontale (ou verticale pour un parcours transposé), par analogie au paramètre β jusqu'alors évoqué. Ces paramètres permettent de faire varier les quantités minimales de données produites ou consommées compte tenu des paramètres de pas (patrons complémentaires). Ici, la synchronisation inter-nœuds des quantités consommées/produites s'éloigne des contraintes sur la taille d'image réellement traitée et de la contrainte d'espace alloué pour le cache interne (nous distinguons deux problématiques).

En prenant les couples (a^O,c^O) et (a^I,c^I) qui correspondent, respectivement, aux contraintes sur la taille des données produites et consommées telles que décrites par la première ligne des égalités de 2-2, nous cherchons alors, pour la direction horizontale, à trouver les valeurs minimales positives de β^I et β^O satisfaisant l'égalité :

Equation 2-41. Synchronisation des données d'entrée/sortie entre les nœuds

$$a^{o}_{k-1} + \beta^{o}_{k-1} \times c^{o}_{k-1} = a^{i}_{k} + \beta^{i}_{k} \times c^{i}_{k}$$

La résolution de cette égalité correspond à celle d'un problème diophantien dans la mesure où les racines de β^o_{k-1} et β^i_k sont recherchées dans N^+. Or, dans le cas général, il est démontré que la résolution d'équations diophantiennes est un problème insoluble [1]. Cependant, il existe des algorithmes de résolution pour les cas simples. Ainsi, dans notre

1. La résolution des équations diophantiennes soulevée par le dixième problème que posa le mathématicien Hilbert en 1900 [66][67], tire son origine d'un manuel d'algèbre écrit vers l'année 250 par Diophante d'Alexandrie dans lequel des solutions entières d'équations sont recherchées. En 1931, l'Autrichien Kurt Gödel introduit le théorème d'incomplétude et démontre que, dans le cas général, la résolution de telles équations est insoluble. Cette classe de problèmes est aussi appelée 'problèmes NP complet', traduisant, au sens de Turing (1936), qu'il n'existe pas de programme qui puisse, dans un nombre fini d'opérations, trouver une solution à une équation diophantienne quelconque [68].

contexte, nous trouvons dans la littérature mathématique [69], le théorème suivant (théorème dit “des congruences simultanées”) :

> *Soit, dans un anneau euclidien A, $S = q_1 \times q_2 \ldots q_n$ avec, pour $i \neq j$, $pgcd(q_i,q_j)=1$. L’ensemble des solutions x du système $x \equiv p_i(q_i)$, est une classe résiduelle modulo S.*

La démonstration de ce théorème qui nous permet de trouver le couple (β^i_k, β^o_{k-1}), est la suivante. Nous construisons tout d’abord les quotients $s_i = S / q_i$. L’égalité de Bézout ($pgcd(s_1,s_2) = s_1 \times x_1 + s_2 \times x_2$) appliquée aux s_i donne :

Equation 2-42. Démonstration du théorème des congruences simultanées

$$1 = x_1 \cdot s_1 + x_2 \cdot s_2 + \ldots + x_n \cdot s_n = e_1 + e_2 + \ldots + e_n$$

Nous multiplions les $e_i = x_i \cdot s_i$ par p_i, et pour obtenir $s = p_1 \cdot e_1 + p_2 \cdot e_2 + \ldots + p_i \cdot e_i$ qui est solution du système ainsi que tous les éléments qui lui sont congrus modulo *S*. Pour faire le lien avec l’équation 2-41, remarquons tout d’abord que cette expression peut également s’écrire sous forme d’une égalité de congruences (moyennant $pgcd(c^o_{k-1}, c^i_k) = 1$) :

Equation 2-43. Représentation du problème de synchronisation sous forme d’égalité de congruences

$$a^o_{k-1} + \beta^o_{k-1} \times c^o_{k-1} = a^i_k + \beta^i_k \times c^i_k$$

$$\Leftrightarrow a^o_{k-1}\left(c^o_{k-1}\right) \equiv a^i_k\left(c^i_k\right) \equiv s(S)$$

La démonstration du théorème précédent nous donne donc *s(S)* qui correspond à la forme générale des solutions recherchées. Pour chaque patron (o_{k-1}/i_k), il convient alors d’identifier les modifications à apporter sur la quantité minimale de données ainsi que sur le pas des données supplémentaires de manière à se synchroniser avec *s(S)*. Nous introduisons alors ϕ comme coefficient d’ajustement des quantités minimales nécessaires et Φ comme coefficient d’ajustement des pas. Si *u* représente le nombre de pas de la solution recherchée, nous avons :

Equation 2-44. Identification des solutions recherchées

$$(a + \phi \cdot c) + u \cdot (\Phi \cdot c) = s + u \cdot S$$

Par identification, nous pouvons ainsi déterminer les constantes ϕ et Φ :

Equation 2-45. Valeur des constantes d'ajustement des patrons

$$\begin{cases} \text{avec } u=0 & \phi = \dfrac{s-a}{c} \\ \text{avec } u=1 & \Phi = \dfrac{s+S-a}{c} - \phi = \dfrac{S}{c} \end{cases}$$

Lorsque des solutions à l'équation 2-41 existent, elles peuvent donc également s'écrire sous la forme de congruences simultanées :

Equation 2-46. Forme des solutions pour la synchronisation des données inter-nœuds

$$a^{o}_{k-1} + u \times c^{o}_{k-1} \times \left(\phi_{k-1} + \Phi_{k-1}\right) = a^{i}_{k} + u \times c^{i}_{k} \times \left(\phi'_{k} + \Phi'_{k}\right) = s + u \cdot S$$

$$\Rightarrow \qquad \beta^{o}_{k-1} \equiv \phi_{k-1}\left(\Phi_{k-1}\right), \ \beta^{i}_{k} \equiv \phi'_{k}\left(\Phi'_{k}\right)$$

Ici, les couples (ϕ_{k-1}, Φ_{k-1}) et (ϕ'_k, Φ'_k) s'obtiennent à l'aide des égalités 2-45 respectivement appliquées aux patrons *k* et *k-1*. Par ailleurs, étant donné le fait que nous cherchons des solutions de (β^{o}_{k-1}, β^{i}_{k}) qui rendent positif le premier jeu d'égalité de 2-46 et que nous pouvons avoir ϕ_{k-1} ou ϕ'_k inférieur à zéro, nous complétons la définition de ϕ_{k-1} et ϕ'_k par :

Equation 2-47. Recherche d'une solution positive de l'équation de congruences

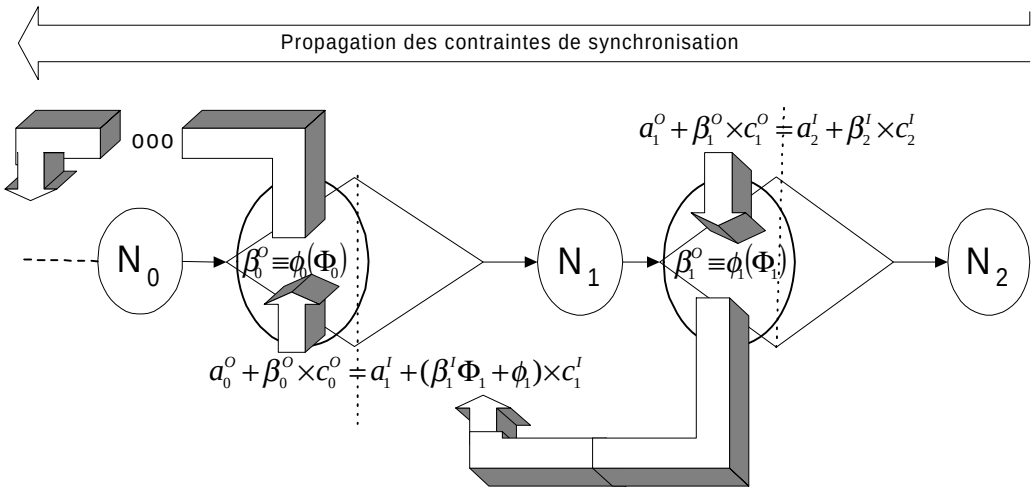
$$\begin{cases} \phi'_k = \phi'_k + v \cdot \Phi'_k \\ \phi_k = \phi_{k-1} + v \cdot \Phi_{k-1} \end{cases} \quad \text{avec } v \in \overset{+}{N} \text{ plus petit entier vérifiant} \quad \begin{cases} \phi'_k + v \cdot \Phi'_k \geq 0 \\ \phi_{k-1} + v \cdot \Phi_{k-1} \geq 0 \end{cases}$$

Ce raisonnement offre donc une procédure analytique pour établir (β^{o}_{k-1}, β^{i}_{k}). Ensuite, l'idée consiste à propager les contraintes de synchronisation de telle sorte qu'avec *k=k-1*, la nouvelle égalité à résoudre devienne :

Equation 2-48. Propagation des contraintes de synchronisation (1)

$$a^{o}_{k-1} + \beta^{o}_{k-1} \times c^{o}_{k-1} = a^{i}_{k} + \left(\beta^{i}_{k} \cdot \Phi_{k} + \phi_{k}\right) \times c^{i}_{k}$$

Figure 2-20. Propagation des contraintes de synchronisation



Le membre à droite de l'égalité correspond à ce que nous introduisons comme une synchronisation intra-nœud. Il s'agit de répercuter sur le patron d'entrée d'un nœud les contraintes de taille liées à la synchronisation du patron de sortie de ce même nœud (*k*) avec le nœud suivant dans la chaîne (*k+1*). Ce mécanisme est illustré par la Figure2-20. Nous maintenons ainsi, au niveau du nœud, la possible variation de quantités de données produites par rapport aux données consommées. Dans ce modèle, le nombre de patrons en entrée d'un nœud est toujours synchrone de celui en sortie. En propageant les contraintes de taille, l'égalité 2-48 peut se ré-écrire :

Equation 2-49. Propagation des contraintes de synchronisation (2)

$$a^{o}_{k-1} + \beta^{o}_{k-1} \times c^{o}_{k-1} = \underbrace{\left(a^{i}_{k} + \phi_k \cdot c^{i}_{k}\right)}_{\text{données min}} + \beta^{i}_{k} \times \underbrace{\left(\Phi_k \cdot c^{i}_{k}\right)}_{\text{nouveau pa}}$$

Nous voyons qu'il s'agit à nouveau d'une équation diophantienne avec $a^i_k + \phi_k \times c^i_k$ et $\Phi_k \times c^i_k$ qui apparaissent alors comme les nouvelles constantes du membre à droite de l'égalité. Pour les cas où cette équation diophantienne ne peut être résolue avec l'approche précédemment développée, ce qui, d'après le théorème des congruences simultanées, se produit lorsque les pas ne sont pas premiers entre eux ($pgcd(c^o_{k-1}, \Phi^i_k \times c^i_k) \neq 1$), nous pouvons alors distinguer deux cas de figure. Premièrement, si la différence des tailles minimum nécessaires pour l'entrée et la sortie ($a^o_{k-1} - (a^i_k + \phi_k \times c^i_k)$) est multiple du plus

grand dénominateur commun des pas, alors la résolution de $(\beta^{o}_{k-1}, \beta^{i}_{k})$ devient possible. Il suffit de diviser l'équation 2-49 par le *pgcd* des pas. Cela donne :

Equation 2-50. Résolution de $(\beta^{o}_{k-1}, \beta^{i}_{k})$ avec $(a^{o}_{k-1} - (a^{i}_{k} + \phi_k \times c^{i}_{k})) \bmod \text{pgcd}(c^{o}_{k-1}, c^{i}_{k} \times \Phi_k) = 0$

$$\begin{cases} \vartheta = \text{pgcd}\left(c^{o}_{k-1}, \Phi_k \cdot c^{i}_{k}\right), \ \eta = a^{o}_{k-1} - \left(a^{i}_{k} + \phi_k \cdot c^{i}_{k}\right) \\ \dfrac{\eta}{\vartheta} = \beta^{i}_{k} \times \dfrac{\left(\Phi_k \cdot c^{i}_{k}\right)}{\vartheta} - \beta^{o}_{k-1} \times \dfrac{c^{o}_{k-1}}{\vartheta} \end{cases}$$

Les solutions de β^{o}_{k-1} et β^{i}_{k} respectivement définies par (ϕ_{k-1}, Φ_{k-1}) et (ϕ'_{k}, Φ'_{k}) sont alors obtenues exactement de la même manière qu'avec les équations 2-45 et 2-47.

Le deuxième cas de figure concerne une valeur de η (la différence des tailles minimales inter-nœuds) qui n'est pas multiple du *pgcd* des pas. Dans ce cas, l'équation diophantienne ne peut être résolue et nous pouvons alors envisager deux approches. La première consiste simplement à s'assurer que la quantité de données produites par le nœud *k-1* est au moins supérieure à la quantité minimum consommée par le nœud *k*. Nous cherchons ainsi à garantir $a^{o}_{k-1} \geq a^{i}_{k} + \phi_k \times c^{i}_{k}$. Dans cette optique, la constante de propagation du pas Φ_{k-1} s'écrit :

Equation 2-51. Constante de propagation du pas lorsque $\text{pgcd}(c^{o}_{k-1}, \Phi_k \times c^{i}_{k}) \neq 1$

$$\Phi_{k-1} = \frac{\text{ppcm}\left(c^{o}_{k-1}, c^{i}_{k} \times \Phi_k\right)}{c^{o}_{k-1}}$$

alors que la constante de propagation de la quantité minimale ϕ_{k-1} devient :

Equation 2-52. Constante de propagation du minimum de données pour $\eta \bmod \text{pgcd}(c^{o}_{k-1}, \Phi_k \times c^{i}_{k}) = 0$ #1

$$\phi_{k-1} = \left\lceil \frac{\max\left(a^{o}_{k-1}, a^{i}_{k} + \phi_k \cdot c^{i}_{k}\right) - a^{o}_{k-1}}{c^{o}_{k-1}} \right\rceil$$

La deuxième technique vise également à vérifier $a^{o}_{k-1} \geq a^{i}_{k} + \phi_k \times c^{i}_{k}$ tout en minimisant $a^{o}_{k-1} - (a^{i}_{k} + \phi_k \times c^{i}_{k})$. Pour ce faire, il suffit de s'appuyer sur l'équation 2-50 et de tronquer la différence η. L'égalité diophantienne à résoudre devient alors :

Equation 2-53. Constante de propagation du minimum de données pour η mod pgcd(c^{o}_{k-1}, $\Phi_k \times c^{i}_{k}$) = 0 #2

$$\begin{cases} \vartheta = \mathrm{pgcd}\left(c^{o}_{k-1}, \Phi_k \cdot c^{i}_{k}\right), \ \eta = a^{o}_{k-1} - \left(a^{i}_{k} + \phi_k \cdot c^{i}_{k}\right) \\ \left\lfloor \frac{\eta}{\vartheta} \right\rfloor \left(\frac{c^{o}_{k-1}}{\vartheta} \right) \equiv 0 \left(\frac{\Phi_k \cdot c^{i}_{k}}{\vartheta} \right) \end{cases}$$

Dans ce cas, la différence maximale entre les quantités de données produites et celles consommées se quantifie de la manière suivante :

Equation 2-54. Non-résolution de la synchronisation et recherche d'une "bonne" solution

$$\left(\frac{\eta}{\vartheta} - \left\lfloor \frac{\eta}{\vartheta} \right\rfloor \right) \times \vartheta$$

Dans le cadre de la propagation des contraintes de taille pour la synchronisation de la chaîne, le jeu d'égalité précédent étend donc l'algorithme de résolution de (β^{o}_{k-1}, β^{i}_{k}) aux cas où les pas inter-nœuds ne sont pas premiers entre eux. Ce système vérifie, par ailleurs, le cas spécifique où les patrons sont déjà synchronisés, que nous traduisons par $a^{o}_{k-1} = a^{i}_{k}$ et $c^{o}_{k-1} = c^{i}_{k} \times \phi_{k-1}$. Dans ce cas de figure, nous vérifions bien $\phi_{k-1} = 0$ et $\Phi_{k-1} = 1$.

Dans l'optique où cet algorithme de résolution de (β^{o}_{k-1}, β^{i}_{k}) est appliqué dynamiquement (au moment de l'exécution du programme), nous pouvons souligner qu'il existe une implantation binaire rapide (sans division) de la fonction *pcgd*. Cet algorithme est décrit dans [70]-page 338 mais n'offre, bien sûr, qu'un intérêt relatif dans le cas de processeurs supportant des extensions matérielles performantes pour le calcul de la division (comme c'est le cas avec le mnémonique *divi* du MP/C80). Concernant la fonction *ppcm*, nous pouvons tirer parti d'une implantation performante du *pgcd* avec la relation bien connue *ppcm(x,y) = x × y / pgcd(x, y)*.

En revenant à l'équation 2-48, le système à résoudre pour obtenir la géométrie du patron virtuel de chaque chemin est défini de manière récursive avec *k = [K-1..L]* suivant l'égalité mathématique :

Equation 2-55. Système récursif pour la propagation indépendante horz./vert. des contraintes de synchronisation d'une chaîne

$$\begin{bmatrix} \left(-c^{o}_{k-1}\right) & \left(c^{i}_{k}\cdot\Phi_{k-1}\right) & \left(a^{i}_{k}+c^{i}_{k}\cdot\phi_{k-1}-a^{o}_{k-1}\right) \\ \left(-d^{o}_{k-1}\right) & \left(d^{i}_{k}\cdot\Gamma_{k-1}\right) & \left(b^{i}_{k}+d^{i}_{k}\cdot\tau_{k-1}-b^{o}_{k-1}\right) \end{bmatrix} \cdot \begin{bmatrix} \left(\beta^{o}_{k-1}\right) & \left(\gamma^{o}_{k-1}\right) \\ \left(\beta^{i}_{k}\right) & \left(\gamma^{i}_{k}\right) \\ (1) & (1) \end{bmatrix} = \vec{0}$$

Dans ce système, la propagation des contraintes de synchronisation est bien indépendante pour la direction horizontale et verticale. Pour la direction verticale, la solution des équations diophantiennes s'écrit de la même manière que pour l'équation 2-46 :

Equation 2-56. Congruences simultanées résolvant l'équation diophantienne verticale

$$\gamma^{o}_{k-1} \equiv \tau_{k-1}\left(\Gamma_{k-1}\right), \gamma^{i}_{k} \equiv \tau'_{k}\left(\Gamma'_{k}\right)$$

et, bien sûr, lorsque $pgcd(\Gamma^{i}_{k} \times d^{i}_{k}, d^{o}_{k-1}) \neq 1$, nous appliquons le même raisonnement que celui associé aux équations 2–42 à 2–45. Par ailleurs, intuitivement, la récursion du système 2-55 suppose les conditions initiales *(k=K-1)* suivantes :

Equation 2-57. Valeur initiale de la récursion du système d'équations 2-55

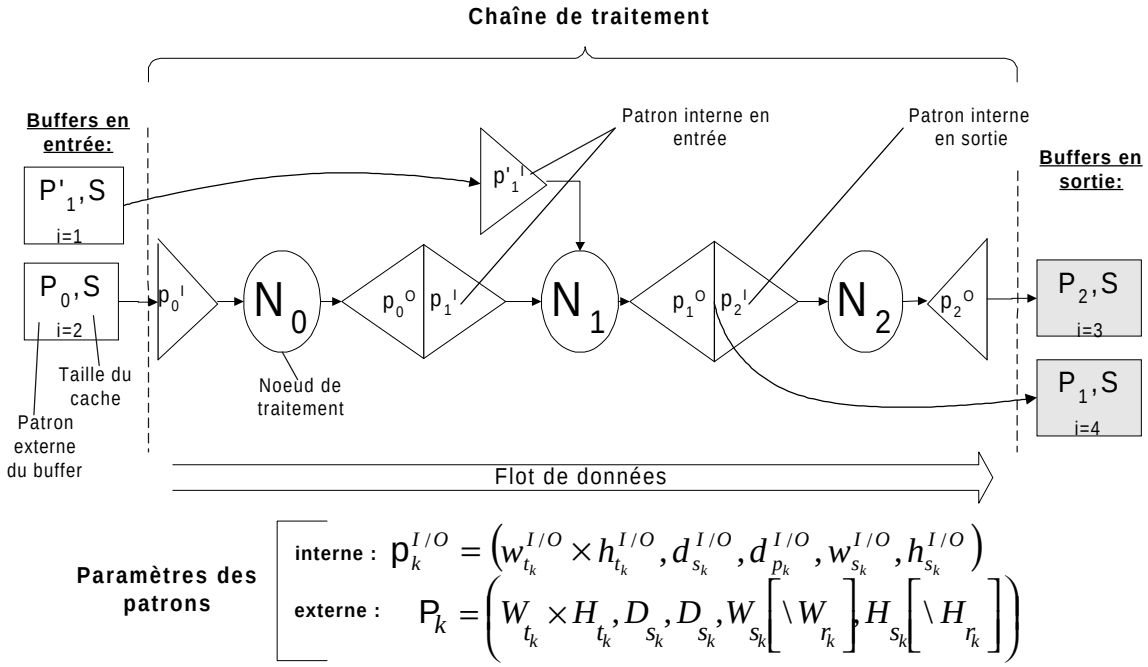
$$\Phi_{K-1} = \Gamma_{K-1} = 1, \ \phi_{K-1} = \tau_{K-1} = 0$$

2.2.3.3.2 Découpage multi-buffers et patrons virtuels

La résolution du système linéaire précédent nous permet alors de revenir sur la notion de patron virtuel que nous associons à chaque buffer. La séquence de nœuds impliquée dans la résolution récursive du système définit tout d'abord un chemin. A chaque chemin correspond un nœud de début (d'indice $L \geq 0$) et un autre de fin (indice *K-1*), où *K* correspond au nombre de nœuds dans la chaîne. Par convention, le nœud *L* à la tête du chemin est associé au buffer dont nous cherchons à établir le patron virtuel. La propagation des contraintes s'effectue donc en partant de la fin du chemin. Par ailleurs, la synchronisation des requêtes DMA doit à la fois tenir compte de la taille des buffers en entrée et en sortie de la chaîne. Dans cette optique, la notion de chemin concerne également

la séquence de nœuds suivant laquelle les contraintes sont propagées dans l'ordre logique d'exécution des nœuds jusqu'au buffer de sortie qui apparaît alors en tête du chemin. Si nous reprenons la Figure 2-20, la propagation s'effectue alors de gauche à droite, l'indice *L* du nœud en tête du chemin étant alors supérieur à *K-1*. Dans ce cas, à chaque nouvelle itération du système linéaire récursif, nous prenons *k=k+1*. Ainsi, si nous prenons l'exemple de chaîne de la Figure2-21, nous pouvons alors dénombrer quatre chemins qui se composent des nœuds (N_1,N_2) pour *i=1*, (N_0,N_1,N_2) pour *i=2*, (N_2,N_1,N_0) avec *i=3* et enfin, (N_1,N_0) avec *i=4*.

Figure 2-21. Un exemple de chaîne de traitement



Pour chaque chemin identifié dans la chaîne (donc pour chaque buffer*i*), nous évaluons les solutions ($\phi_L(\Phi_L)$) et ($\tau_L(\Gamma_L)$) de l'équation 2-55. Ensuite, sur la base des mêmes calculs que ceux impliqués dans le jeu d'égalités 2-1, nous définissons les paramètres de la géométrie du patron virtuel indicé par *v* selon :

Equation 2-58. Surface et pas du patron virtuel interne

$$\begin{cases} a_v = d_{s_L} + \left(w_{t_L} + \phi_L \cdot \frac{w_{s_L}}{d_{p_L}} - 1 \right) \times d_{p_L},\ c_v = \Phi_L \times w_{s_L} \\ b_v = h_{t_L} + \tau_L \times h_{s_L},\ d_v = \Gamma_L \times h_{s_L} \end{cases}$$

Similairement, en prenant les paramètres externes du patron du nœud de tête *L* ([$W_{t/L}$×$H_{t/L}$,$D_{s/L}$,$D_{p/L}$,$W_{s/L}$,$H_{s/L}$]), nous obtenons la géométrie du patron externe avec A_v×B_v pour la quantité de données minimum nécessaire et (C_v,D_v) pour, respectivement, le pas horizontal et vertical. Par ailleurs, nous conservons la valeur des rechargements *E* et *F* associés au patron externe du nœud directement relié au buffer considéré (celui en tête de chemin).

Pour partitionner les données de la chaîne, il nous suffit alors de substituer la géométrie du patron externe impliqué dans le découpage des buffers avec celle du patron virtuel.

Une fois le découpage définitif établi suivant notre procédure pour le découpage multi-buffers, nous pouvons alors passer à l'étape du paramétrage des dimensions des requêtes DMA en fonction de la géométrie externe des patrons de chaque buffer externe et suivant un nombre de pas complémentaires (β,γ) que nous déduisons simplement des valeurs de β_v et γ_v. Nous écrivons en effet :

Equation 2-59.

$$\begin{cases} \beta = \phi_L^o + \beta_v \times \Phi_L^o \\ \gamma = \tau_L^o + \gamma_v \times \Gamma_L^o \end{cases}$$

2.2.3.4 Conclusions

Ce dernier point conclut les grandes lignes de notre approche pour le partitionnement SPMD des données vers une gestion logicielle transparente des caches par le DMA. Nous soulignons la généricité de l'approche dans la mesure où elle supporte la définition de chaînes quelconques. La flexibilité recherchée est bien atteinte dans la mesure où notre modélisation est bien paramétrée en fonction de la géométrie des patrons de la chaîne, du nombre de processeurs et de la taille des buffers.

2.3 Estimation des performances sur C80

Dans cette partie, nous présentons un modèle de performance pour les chaînes de traitement définies suivant la méthodologie de gestion de flux introduite dans le chapitre précédent. La définition d'un modèle de performance nous permet d'anticiper les performances d'algorithmes, d'utiliser au mieux les ressources (dimensionner le nombre de processeurs nécessaire) et de fournir des informations pertinentes sur les optimisations à apporter pour satisfaire aux cadences de traitement. Ces optimisations peuvent porter sur les entrées/ sorties (avec la recherche du chaînage des nœuds ou l'optimisation de leur granularité), ou sur la vitesse de traitement des opérateurs grâce aux techniques développées dans la section 2.1.

2.3.1 Un modèle simple

Nous proposons un modèle simple pour l'estimation des performances qui s'applique aux chaînes de traitement partitionnées suivant le paradigme SPMD visé par notre méthodologie de découpage introduite au paragraphe 2.2. Pour cette approche, nous avons introduit un modèle analytique de l'estimation du coût des transferts que nous réévaluons ici, d'après l'équation 2-13, pour un ensemble de I (*i:1..I*) buffers en entrée/sortie partitionnés selon notre méthodologie, par :

Equation 2-60.

$$\varepsilon = N_{R1} + (\sigma - 1) \cdot N_{R0}$$

$$\tilde{T}(N_W, \varepsilon, \nu) = \underbrace{\mu \cdot (N_W - 1) + \nu \times N_W \times \varepsilon}_{\tilde{K}} + \sum_i \underbrace{\left(R_i^u(N_W, \varepsilon) + W_i^u \times H_i^u \right)}_{\Theta_i}$$

(*N_w* et ε étant commun pour tous les buffers afin de synchroniser l'ensemble des requêtes DMA). *W^u* et *H^u* sont définis par l'équation 2-5 et *R_u* par l'équation 2-9.

Nous revenons brièvement sur les paramètres μ et ν qui caractérisent, nous le rappelons, le coût de la gestion des caches pour le lancement d'une nouvelle itération de la chaîne après

avoir lancé les requêtes DMA d'entrée/sortie (ν) ou celui, plus lourd mais moins récurrent, lié à l'initialisation d'une nouvelle bande verticale (μ). Dans le détail, nous exprimerons ces coûts en cycles et montrerons expérimentalement qu'ils sont fonction du nombre de processeurs σ assignés au traitement. Ainsi, nous pouvons redéfinir le coût *K* associé à la gestion des transferts DMA par :

Equation 2-61.

$$K = \text{nscalable}(\sigma) \times \left(\underbrace{\sigma \cdot \underbrace{\mu' \cdot (N_W - 1)}_{K'} + \nu' \times N_W \times \varepsilon}_{K''} \right)$$

$$\text{avec} \begin{cases} \mu = \text{nscalable}(\sigma) \cdot \mu' \cdot \sigma \\ \nu = \text{nscalable}(\sigma) \cdot \nu' \end{cases}$$

Cette égalité montre que le coût du traitement d'une nouvelle bande verticale (lorsque $N_w > 1$) est logiquement proportionnel au nombre de processeurs et que le coût global de la gestion des transferts est, moins trivialement, fonction du nombre de processeurs (*nscalable*$(\sigma) > \sigma$, le préfix "n" de *nscalable* matérialisant le fait que l'architecture est non-scalable). En effet, d'une manière rigoureuse, l'architecture C8X n'est pas scalable malgré la présence du crossbar et des mémoires locales dédiées à chaque processeur. Des effets de congestion et des mécanismes de préemption au niveau du DMA interviennent et introduisent une non-linéarité pour l'ensemble des transferts (données+instructions), en fonction du nombre de processeurs, que nous exprimons globalement par :

Equation 2-62.

$$T(N_W, \sigma, \varepsilon, \nu') = \text{nscalable}(\sigma) \times \left(K''(N_W, \sigma, \varepsilon, \nu') + \sum_i \Theta_i \right)$$

Des expériences que nous avons menées montrent que la fonction *nscalable* est d'ordre $O(\sigma^2)$. Par souci de simplification, nous annulerons par la suite l'effet de cette fonction (*nscalable*$(\sigma) = 1.0$) et considérerons des durées de transferts indépendantes du nombre de processeurs hormis pour le terme σ×K' lorsque $N_w > 1$. Expérimentalement, nous estimons que cette approximation n'introduit pas plus de 15% d'erreur dans les prédictions et dépend

de la configuration du traitement (c'est notamment le cas pour la chaîne de détection de contours FGL #C1, cf 3.3.2.1.2).

Par ailleurs, nous soulignons que le coût μ' dépend du nombre de buffers de la chaîne et du mode de gestion des bancs internes (double/triple buffering). L'impact de la gestion du cache ν' pour chaque itération de la chaîne dépend, quant à lui, de la granularité des nœuds de cette dernière. Concernant la quantité explicite de données transférées qu'exprime le terme $\Sigma\Theta_\iota$, nous pouvons lui associer une durée en cycles en divisant cette quantité par 8 étant donné que les accès 64 bits s'effectuent en un cycle sur le C8X. Pour le cas spécifique où la première dimension de la source ou de la destination d'une requête DMA précise une largeur de données inférieure à 8 octets, autrement dit lorsque $(D_s \neq D_p$ et $D_s < 8)$ ou $(d_s \neq d_p$ et $d_s < 8)$, le nombre de cycles par élément transféré est augmenté d'un facteur $\varsigma = max(\varsigma_1, \varsigma_2)$ avec $\varsigma_1 = 8 / D_s$ et $\varsigma_2 = 8 / d_s$ ($\varsigma_1 = 1 / \varsigma_2 = 1$ lorsque $D_s = D_p$ / $d_s = d_p$). De là, nous réécrivons la durée globale des transferts avec $nscalable(\sigma) = 1.0$, par :

Equation 2-63.

$$\hat{T}(N_W, \sigma, \varepsilon, \nu') \approx \nu' \times N_W \times \varepsilon + \sigma \times K'(N_W, \mu')_{N_w > 1} + \sum_i \varsigma_i \cdot \frac{\Theta_i}{8}$$

Nous pouvons alors en déduire la durée globale du traitement en fonction de la durée p_j des nœuds J ($j:1..J$) qui composent la chaîne :

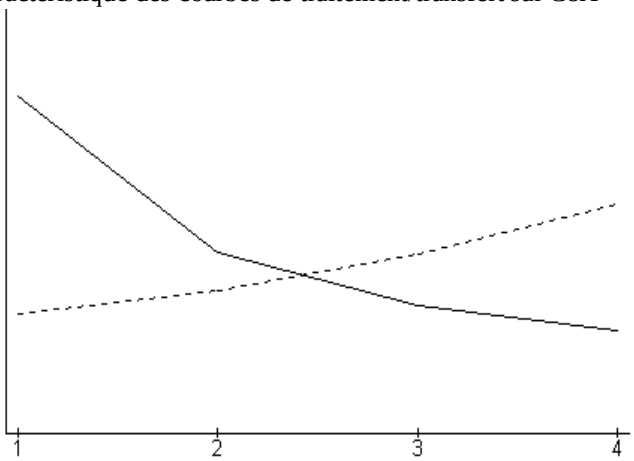
Equation 2-64.

$$D = H(\sigma) + \underbrace{\max\left(\hat{T}(N_W, \sigma, \varepsilon, \nu'), P = \frac{\sum_j p_j}{\sigma}\right)}$$

où $H(\sigma)$ modélise le coût de l'initialisation des traitements (démarrage des patrons de traitement) généralement négligeable face à C ($C >> H$). Ici, la fonction *max* traduit bien le recouvrement temporel des phases de calcul et de transferts. Par ailleurs, cette dernière expression souligne le paradoxe récurrent que nous rencontrons pour les architectures parallèles qui souligne qu'un trop grand nombre de processeurs assignés à un traitement ralentit très souvent la vitesse d'exécution avec l'accroissement du coût des transferts de données entre les processeurs. La figure 2-22 montre l'allure caractéristique de la courbe

de vitesse de traitement en trait plein, et celle du coût des transferts en pointillés (ici *nscalable*(σ)=1 et $N_w>1$) en fonction du nombre de processeurs assignés au traitement. Soulignons que lorsque $N_w=1$, la courbe du coût de transfert devient une droite.

Figure 2-22. Allure caractéristique des courbes de traitement/transfert sur C8X



Cette modélisation permet alors d'exprimer de manière analytique le nombre optimal de processeurs pour une configuration de traitement donnée moyennant une mesure fiable des paramètres μ' et ν'.

2.3.2 Evaluation de la performance des nœuds

Dans notre modèle, la performance des nœuds de traitement p_j n'intègre pas le coût de la gestion des caches d'instructions que nous renvoyons plus particulièrement aux paramètres μ' et ν'. Nous proposons une modélisation simple visant à approcher le nombre de cycles par pixel φ des coeurs de traitement de chaque nœud.

Afin d'affiner l'approximation et suivant l'emplacement des zones de cache paramétrées par l'utilisateur (qui peut être fonction du mode de gestion des bancs à savoir double ou triple buffering, ou du nombre de buffers impliqués dans le traitement), nous intégrons dans φ le coût des contentions de type CPU/CPU ainsi que celles DMA/CPU lorsqu'interviennent, dans le même cycle, deux accès concurrents à une même mémoire locale. Concernant ce dernier terme, nous simplifions l'évaluation en partant du principe

que le DMA est en contention permanente avec le processeur durant toute la durée du nœud (nous faisons l'hypothèse que les transferts durent aussi longtemps que les traitements) et ajustons en conséquence le coût des accès concernés (2 cycles plutôt qu'un). Bien sûr, dans certains cas, il s'agit là d'une simplification assez grossière mais qui s'avère généralement fiable.

Nous pouvons souligner que notre démarche revient à préciser un nombre de cycles par instruction, qui comme le souligne l'auteur de [71], reflète beaucoup mieux les capacités réelles de l'architecture que la mesure inverse du nombre d'instructions par cycle.

Une fois l'estimation faite de φ qui, dans notre contexte VLIW, se déduit directement du code assembleur et du contexte des E/S, nous pouvons estimer le nombre de cycles p_j de chaque nœud en fonction de la quantité globale de données traitées, qui dépend de la taille W×H de l'image traitée ainsi que de la nature de la chaîne. Ce deuxième critère est illustré par l'exemple d'un nœud d'échantillonnage en amont d'une chaîne multi-nœuds. Ici, très intuitivement, la quantité de données sur laquelle porte le deuxième nœud de la chaîne (et les suivants) dépend de la quantité de données restante après l'étape de l'échantillonnage.

Nous conclurons ce paragraphe en donnant un exemple de fiche descriptive détaillée associée aux différents nœuds de la librairie qui permet au programmeur d'apprécier la durée des traitements p_j en fonction de l'emplacement spécifié des zones internes des caches de données (**en gras**) :

Table 2-6. Exemple simple d'une fiche descriptive de nœuds

Noeuds arithmétiques: iEIA_PP_AddSat8_C8_8 iEIA_PP_SubSat_8_C8_8	
Description	Ajoute/Soustrait une constante 8 bits à des valeurs 8 bits non signés (u), résultats sur 8 bits non signés
Paramètres	ts.Arithmetic.ulVal: Constante à ajouter/soustraite
Patrons internes (w_sx-h_t,d_s,d_p,w_s[\$w_r$],$h_s$[\$h_r$]):	entrée i_1: 12X1,1u,1,4\0,1 sortie o_1: 12X1,1u,1,4\0,1
Gestion des bancs	i_1: wap_in_out[a], pong_pong[b], modulo[c] o_1: wap_in_out, pong_pong, modulo
Perf. : cycle/pixel(ds)	**CPU/CPU(i1 ≠ o1): 2/4** **CPU/CPU(i1 = o1): 2/4** **+DMA/CPU(i1): 1/4** **+DMA/CPU(o1): 1/4**
Organisation des données:	i_1: Q[d] o_1: TE
Valeur de sortie du noeud	_uiInputSizeInByte[e]
Note	Aucune

a. Double buffering avec écrasement des données d'entrée par celles en sortie

b. Double buffering avec buffer d'entrée différent de celui en sortie

c. Triple, quadruple, etc. buffering

d. Q = quelconque, TE=telle que l'entrée. Ce champ spécifie comment les patrons et les données intrinsèques des patrons doivent être agencées en entrée ou comment elles le sont en sortie des nœuds. Pour la plupart des nœuds, les données sont toujours linéaires et respectent l'ordre de lignes ou des colonnes de l'image.

e. Permet de spécifier un ratio entre la quantité d'octets consommée en entrée et celle produite. Ce champ permet notamment d'estimer la quantité de données traitées au fil des nœuds d'une chaîne pour une appréciation rapide des performances globales de la chaîne.

2.3.3 Les sources de divergence de l'estimation globale

Outre les simplifications apportées à l'équation 2-62 (*nscalable(σ)=1*), nous pouvons relever quelques sources de divergence de notre modélisation qui n'intègre pas, pour les aspects d'E/S :

- les coûts négligeables :
  - des cycles de rafraîchissement mémoire dynamique que gère le DMA
  - des fautes de page d'accès aux blocs de mémoire externes
  - des coûts de préemption des transferts avec l'ordonnancement round-robin de l'utilisation du DMA
- les transferts explicites qui sortent du cadre de notre méthodologie de découpage et notamment, les transferts de type "XPT" (*eXternal Packet Transfer*) initiés par les périphériques externes et qui font appel au DMA du C80 ainsi que ceux, similaires, émis par le périphérique VC (*Video Controller*) que possède le processeur [72]. Un grand nombre de systèmes à base de C80 intègrent en effet, pour la partie d'acquisition vidéo, ce mode de gestion pour le vidage des fifos de données numérisées. Ces transferts, lorsqu'ils existent, ont un impact direct sur la bande passante (données+instructions) et peuvent grossièrement se modéliser en intégrant un terme ω venant pondérer $\hat{T}(N_W,\sigma,\varepsilon,\nu')$ et qui rend compte du taux d'occupation de la bande passante par le périphérique de capture (en pourcentage par exemple) :

$$\tilde{T}_{sys}(N_W,\sigma,\varepsilon,\nu') = \underbrace{\omega}_{<1} \cdot \hat{T}(N_W,\sigma,\varepsilon,\nu')$$

Par ailleurs, concernant le coût *P* des nœuds, et outre les simplifications de type DMA/CPU suggérées, nous n'intégrons pas l'impact estimé comme négligeable de l'initialisation et de la terminaison des coeurs de traitement. De même, dans le cas de boucles imbriquées, nous n'intégrons pas les coûts associés aux boucles externes lorsque celles-ci sont (a priori) itérées moins souvent que la partie la plus interne. Nous excluons également de la modélisation les possibles contentions d'accès à la mémoire commune inter-processeurs qui peuvent intervenir dans l'implantation libre des nœuds mais qui sortent du cadre de la méthodologie de découpage SPMD sur laquelle s'appuie notre modèle de performance simplifié.

Enfin, il convient de préciser que concernant ν' qui modélise, rappelons-le, le coût en cycles du rechargement des blocs de cache d'instructions nécessaires à chaque exécution

de la chaîne sur de nouvelles données du cache, il apparaît que cette variable est difficile à estimer (tout comme μ') et nécessite des outils de simulation spécifiques et complexes tels que ceux développés dans [73] et [74]. Nous soulignerons avec la première application de référence que, sur certaines chaînes, comme avec l'exemple de la détection de contour optimale du paragraphe 3.3 (chaîne #C1), la mesure de cette variable permet à elle seule d'expliquer une partie importante de la divergence des durées de traitement rencontrés sans l'intégration de ce paramètre. En effet, sur cet exemple, nous avons pu globalement évaluer le terme $\nu' \times N_w \times \varepsilon$ par une technique de simulation fastidieuse reposant sur l'émulateur matériel/logiciel JTAG (*Joint Technical Assessment Group*, circuit dédié au débogage du processeur), le traçage de l'application instruction par instruction (induisant des durées de simulation très longues) pour aboutir à la post-analyse des registres de cache et la comptabilisation des mouvements de blocs. L'intérêt de cette démarche a permis de valider certains aspects du modèle de performance proposé sans pour autant offrir une solution véritablement satisfaisante pour l'anticipation (hors simulation) du terme ν'.

2.3.4 Vers une meilleure performance des chaînes de traitement

L'optimisation des performances d'application sur C80 necéssite à la fois de maximiser la vitesse de traitement des nœuds tout en réduisant le coût des transferts parallèle comme le suggère le terme *C* de l'équation 2-64.

L'amélioration de la vitesse des transferts peut s'appuyer sur une meilleure utilisation de la localité des données avec le chaînage des nœuds qu'intègre notre méthodologie de gestion de flux. Cependant, comme nous avons déjà pu le souligner, l'amélioration de la localité augmente l'impact du coût de la gestion des caches (notamment avec le terme ν'). Ainsi, tout comme pour le dimensionnement du nombre de processeurs, il existe bien un seuil du nombre de nœuds qu'il est souhaitable de chaîner (si l'application le permet) et au-delà duquel l'impact de la gestion des caches d'instruction devient trop pénalisant compte tenu du gain apporté dans la minimisation des transferts de données.

Nous pouvons également distinguer une borne "inférieure" qui précise le nombre de blocs du cache d'instructions disponibles avant que la taille du cache d'instructions ne soit saturée et qu'intervienne le rechargement d'un ou plusieurs blocs d'instructions à chaque

nouvelle exécution de la chaîne sur de nouvelles données cachées. Idéalement, nous souhaiterions, bien sûr, avoir toutes les instructions de la chaîne dans le cache pour l'ensemble des transferts des buffers.

Le seuil minimum de saturation peut être très grossièrement estimé sur la base du nombre total d'instructions que nécessite le patron de traitement pour l'encapsulation des requêtes DMA. Pour le balayage des bandes verticales, nous estimons qu'un bloc d'instructions est en moyenne nécessaire pour le patron de traitement dans le cas du double buffering. Ce bloc d'instructions correspond à la plus petite entité cachée, même si parmi les 64 instructions VLIW 64 bits cachées par bloc (soit 4 sous-blocs de 16 instructions), seule une partie des sous-blocs est réellement utilisée (nous l'estimons à deux sous-blocs). Cette situation laisse donc 3 blocs de 64 instructions VLIW pour les nœuds de l'ensemble de la chaîne avec un cache de 2Ko.

Afin de réduire le nombre de blocs utilisés, nous avons suggéré une programmation des nœuds visant à déporter certaines contraintes sur la taille des données traitées au niveau des patrons de données afin d'écarter certains cas de gestion spécifique (Cf 2.1).

Dans la même optique, notre librairie permet également d'optimiser plus globalement le nombre de blocs de cache utilisés par la chaîne en réorganisant très simplement le code des nœuds. Celui-ci se trouve en effet figé au stade de l'édition de liens où les nœuds de la librairie se voient affectés une adresse physique qui correspond naïvement à l'ordre dans lequel ils sont définis. En recopiant de manière contiguë le code des nœuds qui composent la chaîne dans une zone de la mémoire externe allouée dynamiquement nous favorisons les performances. Nous pouvons expliquer ce gain par le fait que les nœuds occupent alors des sous-blocs contigus induisant moins de perte d'espace et favorisant ainsi une réduction du nombre de cycles associés à ν'.

Enfin, pour réduire la durée des chaînes au niveau du terme P de l'équation 2-64, nous préconisons l'utilisation de techniques génériques d'optimisation qui permettent de mieux exploiter la puissance de calcul de l'architecture VLIW des PP.

2.4 Conclusions

Dans ce chapitre, nous avons tout d'abord présenté un ensemble de techniques génériques connues pour une implantation optimisée des nœuds de traitement. Dans un deuxième temps, nous avons développé une méthodologie originale pour la gestion des flux de données en adéquation avec les capacités de l'architecture C80 et, plus généralement, en accord avec les besoins du traitement d'images bas niveau en matière de gestion de flux optimisante pour les architectures DSP à base de DMA.

D'un point de vue scientifique, nous avons proposé une extension mathématique originale à la modélisation traditionnelle par flots de données synchrones. Ici, notre approche place véritablement au centre le paradigme matériel du DMA qui apparaît la plupart du temps mal exploité. Pratiquement, nous proposons deux niveaux de synchronisme : l'un, microscopique, qui intervient au niveau du chaînage des nœuds, l'autre, que nous qualifions de macroscopique, qui apparaît au niveau de la synchronisation des requêtes DMA. Notre approche étend également les méthodes traditionnelles de modélisation avec le support natif de chaînes de traitement bi-dimensionnelles et la gestion des architectures multi-processeurs qui transparaît dans les équations pour le partionnement des données du schéma de parallélisation SPMD.

Par ailleurs, au regard des objectifs d'aide à la programmation que nous avons détaillés à la section 1.1.4, notre résolution unifiée de la problématique permet bien d'appréhender des chaînes génériques pour le domaine du bas niveau. La notion de flexibilité intervient également concrètement dans les raisonnements mathématiques avec la taille des buffers et le nombre de processeurs qui apparaissent bien comme variables des modèles développés.

La recherche d'une amélioration des performances est également au centre de nos développements théoriques. Outre le chaînage des nœuds qui correspond au principal objectif de notre approche, notre méthodologie apparaît également comme originale puisque nous visons tant l'amélioration de la localité des données que la recherche d'une diminution de la granularité des nœuds de traitement chaînés. Cette diminution de la taille des nœuds vise à intégrer de manière performante les spécificités des techniques

d’optimisation développées dans la première partie du chapitre. C’est bien avec cette dualité des objectifs que nous afficherons des durées de traitement améliorées, notamment dans le cas du C8X. Le modèle de performance détaillé dans la dernière partie souligne cet aspect et propose des éléments pour anticiper les vitesses de traitement des algorithmes ainsi que des directions plus générales pour l’optimisation des chaînes respectant notre méthodologie de gestion des flux avec, par exemple, des pistes pour le dimensionnement optimal du nombre de processeurs.

Face à l’ensemble de ces développements, il nous reste à valider les méthodologies pratiques et théoriques développées. Le chapitre suivant y est consacré.

3 Validation des méthodologies de développement

Le but de ce chapitre est de valider expérimentalement les méthodologies originales de développement introduites dans le chapitre précédent sur le plan des techniques d'optimisation pour l'implantation des nœuds de traitement, de la gestion des flux et du modèle de performance associé.

Dans une première partie, nous présentons un résumé de l'infrastructure logicielle d'une librairie originale pour le traitement d'images développée spécifiquement pour le C80 qui repose sur les concepts mathématiques introduits au chapitre 2.

Sur la base de cet outil, nous présentons, dans une deuxième partie, les performances temporelles obtenues avec trois chaînes algorithmiques élémentaires qui combinent un sous-ensemble d'opérateurs issus de notre librairie. Ici, l'objectif de la démarche réside essentiellement dans la validation du modèle de performance que nous avons proposé ainsi que dans la démonstration de la généricité et flexibilité de notre méthodologie pour la gestion des flux.

Dans la troisième partie, nous validons plus précisément le modèle de performance et détaillons les principaux calculs impliqués dans le partitionnement et la synchronisation des requêtes de transfert avec une application plus complète de détection de contours optimale. Cette étude de cas nous permet également d'introduire la mise en œuvre des techniques d'optimisation pour l'implantation des nœuds à la fois sur le C80 et sur l'architecture du nouveau C62.

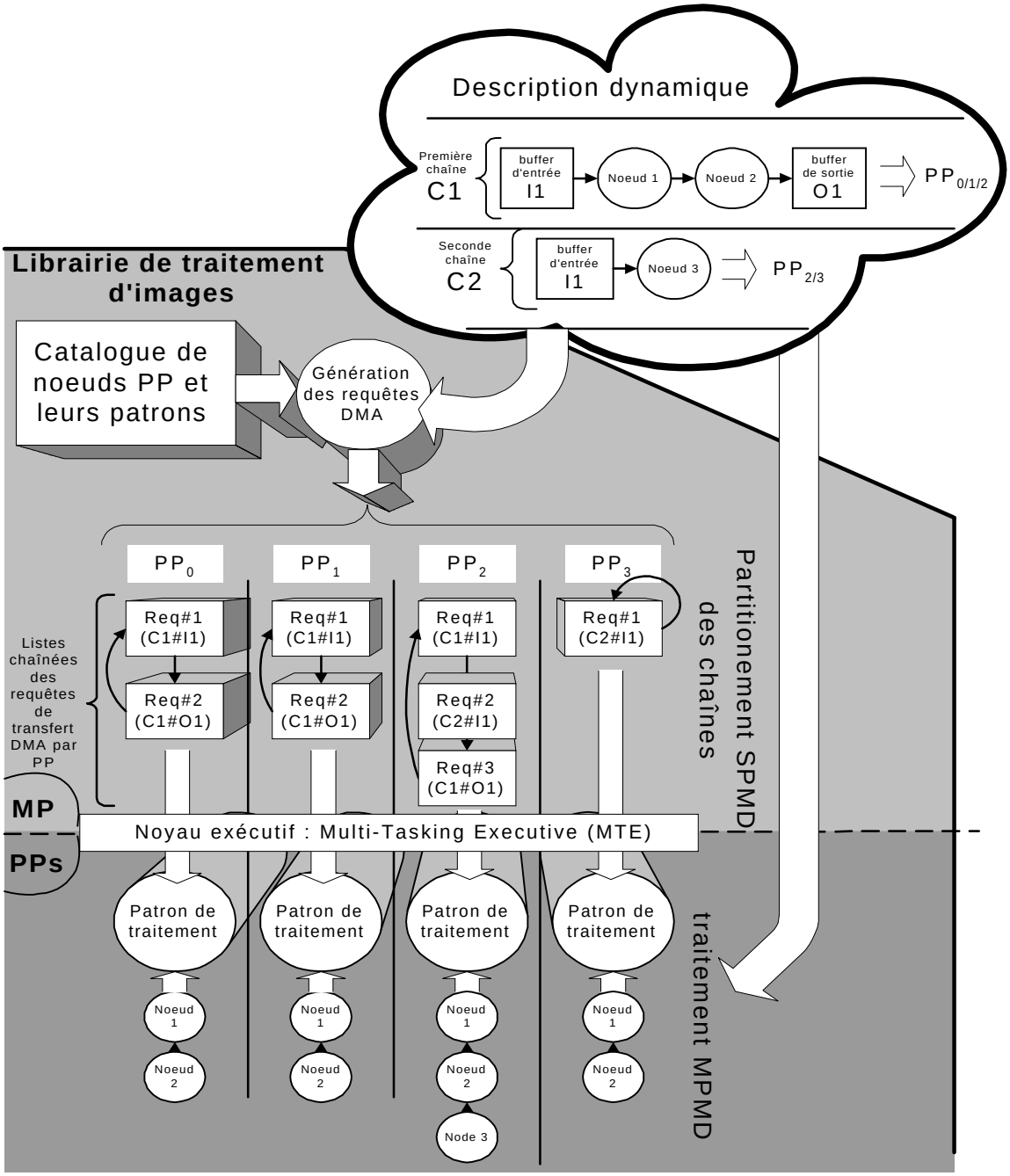
3.1 Librairie C80 de traitement d'images bas niveau

Pour illustrer la mise en œuvre de notre approche de gestion de flux par DMA, nous donnons un aperçu de l'infrastructure logicielle d'une librairie de traitement d'image bas niveau sur TMS320C80 qui compte près de 60 nœuds de traitement optimisés pour les PP et dont la liste est donnée en fin de section (Table3-1). La librairie est architecturée suivant différentes couches fonctionnelles, chacune accessible et paramétrable par l'utilisateur. Cette approche permet à l'utilisateur de bénéficier de l'ensemble des fonctions développées et d'orienter à un niveau assez fin les différentes étapes du partitionnement des données ainsi que celles liées à l'assignation des traitements.

Cette séparation en couches permet de modifier dynamiquement un certain nombre de paramètres sans relancer tout le processus de génération des requêtes DMA. A l'instar des mécanismes que nous rencontrons pour le C++, cette approche se base sur l'héritage d'un ensemble d'informations entre les différentes couches logicielles et permet de pré-calculer un maximum de paramètres pour les algorithmes les plus critiques (sachant que l'impact de l'exécution des couches est réduit grâce à l'optimisation des différentes fonctions écrites en C). A titre d'exemple, nous pouvons ainsi modifier le nombre ou l'identité des processeurs assignés à une chaîne de traitement sans pour autant relancer l'ensemble des mécanismes de découpage des données. En outre, l'emplacement physique des buffers externes peut être aisément modifié avec un minimum de redondances dans les calculs afin, par exemple, d'intégrer une gestion circulaire des buffers de capture vidéo externe d'une manière performante.

La définition de chaînes algorithmiques est réalisée par une description haut niveau de la séquence de nœuds qui composent la chaîne ainsi que des caractéristiques des différents buffers d'entrée ou de sortie impliqués. Nous précisons également le mode de gestion des caches internes (double ou triple buffering) et listons les différents bancs de mémoire à utiliser pour la mise en œuvre de l'optimisation des flux. La Figure3-1 donne une vue synoptique des principales étapes du partitionnement des données.

Figure 3-1. Synopsis de l'architecture de la librairie de traitement d'images sur C80



Par ailleurs, la description des chaînes est soit statique (figée à la compilation) soit dynamique. Le modèle MPMD pour la réalisation de chaînes distinctes s'exécutant de manière concurrente sur plusieurs processeurs est supporté. Ainsi, sur la base de cette description, un premier niveau fonctionnel analyse les chaînes de traitement et détermine,

pour chaque chaîne, le patron virtuel de référence pour le partitionnement SPMD des données suivant le nombre de PP attribués au traitement.

Dans un deuxième temps, l'identité exacte des PP assignés permet la génération effective des paramètres des requêtes DMA (ainsi que leur chaînage), d'une manière totalement transparente et indépendante pour chaque processeur. Dans ce contexte, l'étape de chaînage est spécifique à chaque processeur et correspond bien au regroupement des paramètres DMA en éventuelle provenance de multiples chaînes de traitement concurrentes. Cet aspect traduit la capacité de traitement MPMD qui est sous-jacente au partitionnement SPMD initial des données.

L'initialisation des traitements est réalisée au niveau du MP qui s'appuie par ailleurs sur un dictionnaire regroupant la géométrie des patrons de chacun des n œuds implantés dans la librairie. Ensuite, l'exécutif que propose Texas pour le C80 (le MTE) nous permet de lancer les différentes instances du patron de traitement générique et de se synchroniser avec chaque PP (de manière indépendante et par interruption matérielle). Dans le détail, cet exécutif permet la synchronisation par sémaphore des tâches logicielles concurrentes s'exécutant sur le MP avec les patrons de traitement qui se chargent d'encapsuler les transferts de données décrits par les listes de requêtes DMA, avec le lancement des nœuds spécifiques aux PP. Par ailleurs, selon un schéma client/serveur, chaque patron de traitement s'appuie sur la gestion d'une file de requêtes d'exécution séquentielles de chaînes algorithmiques issues des tâches du MP. Cette approche autorise une complète désynchronisation du MP avec les instances du patron de traitement et permet d'assigner au MP l'exécution parallèle de tâches d'imagerie de type moyen ou haut niveau qui tirent avantage de la présence du cache de données matériel pour ce processeur.

Pour finir, nous donnons la liste des principaux n œuds implantés au sein de cette librairie avec le tableau suivant :

Table 3-1. Nœuds bas-niveau de la librairie de traitement d'images sur C80

Nom	Définition
Conversion de dynamique	
iEIA_PP_Binarize_8_2	Troncature de pixels 8 bits vers des pixels binaires
iEIA_PP_Binarize_2_8	Expansion de pixels binaires vers des pixels 8 bits
iEIA_PP_Convert_8_16	Expansion de pixels 8 bits vers des pixels 16 bits
iEIA_PP_Convert_16_8	Troncature de pixels 16 bits vers des pixels 8 bits
iEIA_PP_Convert_8_32	Expansion de pixels 8 bits vers des pixels 32 bits
iEIA_PP_Convert_32_8	Troncature de données 32 bits vers des données 8 bits
Seuillage et LUT	
iEIA_PP_BinThreshold_8_C8_8	Binarisation de pixels 8 bits par rapport à un seuil 8 bits
iEIA_PP_Saturate_8_C8_8	Saturation de pixels 8 bits par rapport à un seuil 8 bits
iEIA_PP_LUT_8_8	Application d'une LUT de valeurs 8 bits à des pixels 8 bits
Opérations logiques (avec constante)	
iEIA_PP_Not_8_8	"Non" logique de pixels 8 bits
iEIA_PP_Or_8_C8_8	"Ou" logique de pixels 8 bits avec une constante 8 bits
iEIA_PP_And_8_C8_8	"Et" logique de pixels 8 bits avec une constante 8 bits
iEIA_PP_XOr_8_C8_8	"Ou" exclusif logique de pixels 8 bits avec une constante 8 bits
Opérations logiques diadiques (entre 2 buffers)	
iEIA_PP_Or_8_8_8	"Ou" logique entre 2 pixels 8 bits
iEIA_PP_XOr_8_8_8	"Ou" logique exclusif 2 pixels 8 bits
iEIA_PP_And_8_8_8	"Et" logique entre 2 pixels 8 bits
Opérations logiques triadiques (entre 3 buffers)	
iEIA_PP_Triadic_8_8_8	Opération pixel triadique de type (A&C) | (B&~C) entre 3 buffers A,B,C (incrustation de motifs vidéo avec un plan de masques)
Opérations arithmétiques avec constantes	
iEIA_PP_Add_8_C8_8	Addition d'une constante 8 bits à des pixels 8 bits
iEIA_PP_Sub_8_C8_8	Soustraction d'une constante 8 bits à des pixels 8 bits
iEIA_PP_AddSat_8_C8_8	Addition saturée d'une constante 8 bits à des pixels 8 bits
iEIA_PP_SubSat_8_C8_8	Soustraction saturée d'une constante 8 bits à des pixels 8 bits
iEIA_PP_Div_8_C8_8	Division par une constante 8 bits en précision fixe de pixels 8 bits
iEIA_PP_Mult_8_C16C8_8	Multiplication par une constante 8 bits en précision fixe de pixels 8 bits
iEIA_PP_RightShift_8_C8_8	Décalage à droite (mult. par 2) de pixels 8 bits
iEIA_PP_LeftShift_8_C8_8	Décalage à gauche (div. par 2) de pixels 8 bits
Opérations arithmétiques diadiques (entre 2 buffers)	
iEIA_PP_AddSat_8_8_8	Addition saturée de 2 pixels 8 bits

Table 3-1. Nœuds bas-niveau de la librairie de traitement d'images sur C80

Nom	Définition
iEIA_PP_SubSat_8_8_8	Soustraction saturée entre 2 pixels 8 bits
iEIA_PP_AbsDiff_8_8_8	Différence absolue entre 2 pixels 8 bits
iEIA_PP_AbsDiffSign_8_8_8	Différence absolue signée entre 2 pixels 8 bits
iEIA_PP_MultSat_8_8_8	Multiplication signée entre 2 pixels 8 bits
Opérations statistiques	
iEIA_PP_Histogram_8_C32s	Calcul d'un histogramme de valeurs 32 bits de pixels 8 bits
iEIA_PP_Sum_8_C32	Somme sur 32 bits de pixels 8 bits
iEIA_PP_MinMax_8	Minimum et maximum de pixels 8 bits
iEIA_PP_SumPow2_8_C32	Somme au carré sur 32 bits de pixels 8 bits
iEIA_PP_CountEvent_8_C32	Nombre de pixels 8 bits "<" ou "=" ou ">" à un seuil
Filtrage (les filtres 1D s'appliquent en ligne ou en colonne)	
iEIA_PP_IIR_8_8	Filtre récursif (IIR1D) sur des pixels 8 bits
iEIA_PP_IIRSat_8_8	Filtre récursif (IIR1D) saturé sur des pixels 8 bits
iEIA_PP_IIR_8_16	Filtre récursif (IIR1D) sur des pixels 8 bits, résultats sur 16 bits
iEIA_PP_FGL1_8_8	Filtre récursif (IIR1D) de FGL du premier ordre sur des pixels 8 bits (C.f. 3.1)
iEIA_PP_FGLGradient_8_8	Gradient (convolution) par noyau 2x2 de FGL sur des pixels 8 bits
iEIA_PP_Roberts_8_8	Gradient (convolution) par le noyau de Roberts sur des pixels 8 bits
iEIA_PP_Conv3x3_8_8	Convolution 3x3 générique optimisée pour des pixels 8 bits
iEIA_PP_GenConvNxN_8_8	Convolution NxN générique optimisée pour des pixels 8 bits (N<25)
iEIA_PP_FIR_8_8	Convolution par filtre FIR 1D sur des pixels 8 bits
iEIA_PP_FIRSat_8_8	Convolution saturée par filtre FIR 1D sur des pixels 8 bits
iEIA_PP_FIR_8_16	Convolution par filtre FIR 1D sur des pixels 8 bits, résultat 16 bits
Transformées	
iEIA_PP_RGB888_YUV888	Conversion de l'espace colorimétrique RGB vers YUV (4:2:2) pour des pixels 8 bits
iEIA_PP_YUV888_RGB888	Conversion de l'espace colorimétrique YUV à RGB pour des pixels 8 bits
Divers	
iEIA_PP_Fill_C32	Remplissage de la mémoire interne d'une valeur 32 bits
NŒUDS SPECIFIQUES : COMPRESSION "Markov/M-JPEG" (cf. chapitre 4)	
iEIA_PP_MarkovBinarizeSign_8_2	**Seuillage de l'étape de pré-traitement de la détection de mouvement Markovienne avec gestion du signe de la différence (pixels 8 bits, résultat binaire)**
iEIA_PP_MarkovICM_8	**Algorithme d'ICM pour la détection de mouvement Markovienne sur des pixels 8 bits**
iEIA_PP_MarkovMotionFilt_8_2_8	**Différence et filtrage du mouvement de l'application Markov/M-JPEG (chaîne T1#C3)**
iEIA_PP_JPEGAdd_8_8_8	**Reconstruction de l'image courante de l'application Markov/M-JPEG (chaîne T3#C1)**

3.2 Etude des performances de chaînes élémentaires

3.2.1 Contexte

Cette section présente les performances obtenues avec trois chaînes de traitement élémentaires qui permettent de valider les choix retenus lors du développement de notre méthodologie de gestion de flux ainsi que le modèle de performance associé. L'implantation des nœuds s'appuie par ailleurs sur les techniques d'optimisation que nous avons introduites dans la première partie du chapitre 2 ainsi que sur l'infrastructure de notre librairie de traitement d'images. Les durées de traitement que nous présentons ici s'appuient sur une configuration matérielle sous-optimale du C80. Sur notre carte d'évaluation (ou SDB pour *Software Development Board*), l'horloge du processeur est cadencée à 40 MHz (alors que les spécifications du processeur autorisent une cadence maximale de 60 MHz) et les mémoires DRAM et VRAM présentes nécessitent trois cycles par accès en lecture et deux en écriture. Ces caractéristiques ainsi que celles théoriquement offertes par le processeur sont présentées dans le tableau suivant :

Table 3-2. Performances de la carte SDB face aux performances maximales du C80

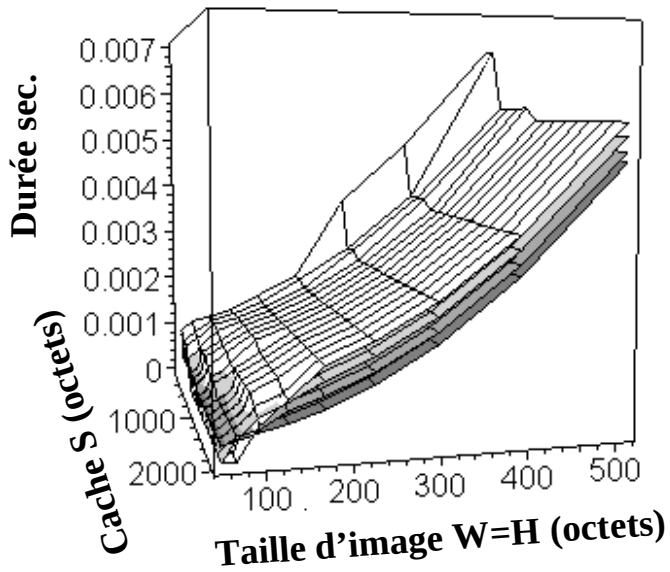
	Performances de la SDB	Configuration optimale
Fréquence du C80	40 MHz	60 MHz
Type de mémoire	VRAM/DRAM	SDRAM
Latence d'un accès externe en lecture	3/3 cycles	1 cycle
Latence d'un accès externe en écriture	2/2 cycles	1 cycle

L'utilisation d'une mémoire synchrone sans état d'attente (wait-state) réduirait ces coûts d'accès à un cycle. Pour chaque exemple de chaîne, la flexibilité de la librairie permet de faire varier le nombre de processeurs assignés au traitement ainsi que la surface d'image traitée et la taille de la zone des caches internes. Cette souplesse permet d'analyser l'impact de ces différents paramètres sur les performances et de mieux appréhender les capacités réelles de l'architecture. Nous soulignons que les chaînes présentées s'appuient uniquement sur le double buffering.

3.2.2 Résultats

La Figure 3-2 montre la durée de traitement d'une chaîne ne comportant qu'un n œud élémentaire : le nœud d'inversion logique des pixels (NON logique). Sur ce graphique apparaissent les durées de traitement suivant les différentes valeurs de la taille du cache interne (*S*) qui est la même pour le buffer d'entrée et de sortie, ainsi qu'en fonction de la taille du côté de la zone carrée traitée ($W = H$). Chaque surface de ce graphique tri-dimensionnel correspond au nombre de processeurs assignés au traitement. Plus la surface est claire, plus le nombre de PP assignés au traitement est important.

Figure 3-2. Performances du nœud d'inversion logique

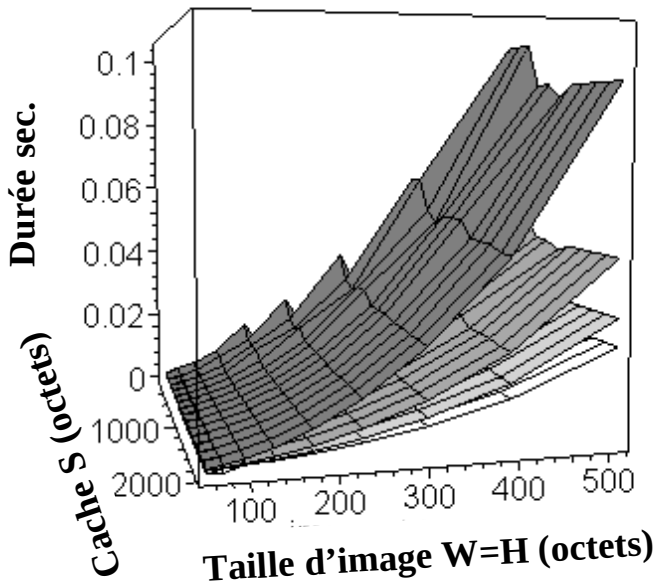


Dans cet exemple et pour une image 512^2, l'estimation de la durée du transfert nous donne 3,2 ms ($2\times512^2\times(2+3)/2/40$ MHz). Avec un seul PP assigné au traitement ($\sigma = 1$ qui correspond à la surface la plus sombre), nous mesurons une durée globale de traitement de 4,7 ms (avec $S = 2Ko$) qui s'accroît lorsque les données du traitement sont partitionnées sur plusieurs PP. Ici, la durée du traitement est inférieure à celle du transfert ce qui explique que paradoxalement, un nombre de PP élevé ne soit pas en faveur d'une durée du traitement réduite. Nous soulignons cependant que le crossbar de l'architecture remplit parfaitement son rôle dans la mesure où les contentions induites lorsque quatre PP fonctionnent en

parallèle sont limitées. Ainsi, pour les mêmes valeurs de *S* et *W*, nous mesurons 5,4 ms avec quatre PP, soit un accroissement de la durée de moins de 15% pour une image 512^2. Ici, le coût de la gestion des caches est bien sûr favorisé par la faible granularité de la chaîne qui ne comporte qu'un nœud. Par ailleurs, la durée de 4,3 ms correspond aux performances détaillées dans [36], le contexte matériel étant comparable.

Nous nous intéressons maintenant à une chaîne de traitement ne regroupant qu'un nœud de convolution 3×3 optimisé auquel nous associons le rechargement des données ($w_r=1$, $h_r=1$). La Figure 3-3 montre que l'approche du rechargement des zones de patron à cheval sur les requêtes DMA n'a quasiment pas d'impact sur cet exemple au regard de la complexité du nœud.

Figure 3-3. Performances du nœud de convolution 3x3 optimisé

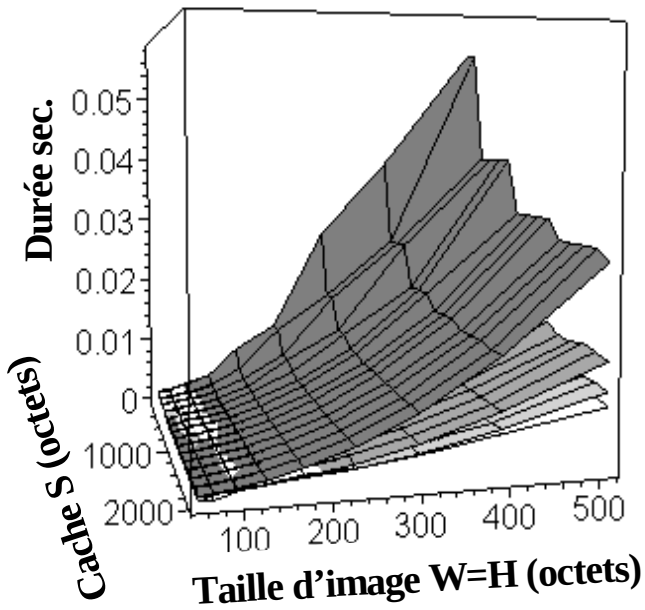


En effet, nous mesurons une durée de traitement de $\chi=96\ ms$ pour une image 512^2 avec $\sigma=1$ PP (et $S = 2Ko$). Le coeur de boucle du nœud requiert à lui seul 14 cycles par pixel (il s'appuie sur l'utilisation des capacités SIMD de L'ALU PP) que nous pouvons comparer aux 15 cycles mesurés si nous nous référons à la durée effective de traitement : $\chi \times freq \times \sigma / (W \times H)$. Malgré les rechargements, la performance mesurée est ainsi proche de l'optimum dans la mesure où, selon notre modèle de performance, $P > T$. Sur cet exemple, la durée des transferts parallèles (Q^i+Q^o) est estimée à *8 ms.*

Lorsque le nombre de PP est porté à quatre, nous mesurons une durée d'exécution de 26 ms que nous pouvons ramener à une latence effective du coeur de boucle de seize cycles pour deux pixels par processeur. Là encore, le rechargement des données combiné aux phénomènes de congestion des accès au DMA n'a qu'un impact réduit sur la scalabilité. Outre la présence du crossbar qui explique cette dernière remarque, soulignons que la bonne scalabilité est également possible grâce à l'optimisation du patron de traitement.

Nous considérons désormais une chaîne de traitement plus complexe qui s'appuie sur quatre nœuds ayant pour but de recueillir des informations d'ordre statistique sur l'image. Cette chaîne regroupe un nœud destiné à évaluer le minimum/maximum des pixels, un autre visant à sommer les valeurs des pixels pour calculer l'intensité moyenne de l'image, un troisième pour sommer le carré des pixels pour le calcul de la variance et, enfin, un dernier nœud déterminant le nombre de pixels en dessous d'un seuil.

Figure 3-4. Performances SPMD de la chaîne statistique



Pour une image 512^2 avec un cache d'entrées de 2 Ko (sur cet exemple, il n'y a pas de buffer de sortie), nous mesurons une vitesse de traitement SPMD de *8 ms* avec quatre PP contre *27 ms* lorsqu'un seul est utilisé (la scalabilité reste bonne). Sur la Figure3-4, nous notons que la taille du cache est un facteur déterminant dans l'estimation de la durée du traitement et qu'une taille de cache réduite peut quadrupler ces durées pour une même taille d'image. Avec 4 PP, les performances passent ainsi de *8* à *32 ms* lorsque la taille du cache est réduite, de 2 Ko à 256 octets (*W=H=512*). L'explication intuitive provient du fait que *S* influt directement sur le nombre total de requêtes DMA et modifie l'impact du terme $N_w \times \varepsilon \times \nu'$ de notre modèle de performance, ν' s'associant ici à une chaîne de granularité moyenne (nécessitant le rechargement de plusieurs sous-blocs correspondant à au moins trois blocs de cache d'instructions). Lorsque S est réduit, le coût de la gestion du cache vient alors peser sur la durée globale de traitement et réduit par ailleurs la scalabilité de l'algorithme[1].

Pour souligner l'intérêt du chaînage des nœuds, nous pratiquons l'expérience de diviser la chaîne en quatre chaînes mono-nœud distinctes, chacune de ces sous-chaînes correspondant à une étape du traitement statistique. Si chaque chaîne est assignée à un PP et porte sur l'intégralité de l'image d'entrée, nous obtenons un partitionnement de type MPSD (plusieurs flots d'instructions, un seul flot de données) selon la terminologie de Flynn [75]. Dans ce contexte, nous mesurons une durée de traitement de *15 ms* avec 4 PP (*W=H=512, S=2Ko*) ce qui correspond à un doublement de la durée mesurée avec le partitionnement SPMD initial (*8 ms*). Cette expérience illustre parfaitement l'intérêt de l'amélioration de la localité des données que vise alors notre méthodologie de gestion des flux

1. Un tel exemple souligne l'intérêt du C82 qui, rappelons-le, n'intègre que deux PP mais double la taille des caches d'instructions (4Ko) et propose deux bancs de données de 4 Ko par PP (contre trois de 2 Ko pour le C80).

.

3.2.3 Conclusions

La souplesse de notre librairie de traitement d'images nous a permis d'implanter, de manière très simple, les trois chaînes élémentaires présentées. La flexibilité qu'affiche notre méthodologie de gestion des flux associée à cette librarie, transparaît avec la possibilité de faire varier l'espace des paramètres du traitement sur le plan de la taille des images traitées, du dimensionnement des zones de cache et du nombre de processeurs assignés au traitement parallèle.

Ces caractéristiques nous ont permis d'avancer les premiers éléments en faveur de la cohérence du modèle de performance proposé au paragraphe 2.3. Plus précisément, nous avons mis l'accent sur la bonne scalabilité généralement mesurée (que nous avions envisagé avec l'approximation *nscalable(σ)=1*) ainsi que sur le coût qu'engendre un trop grand nombre de requêtes de transfert lorsque la taille des caches est limitée. Enfin, sur le plan de l'apport des performances, le dernier exemple nous a permis de souligner l'importance des gains que permet le chaînage des nœuds justifiant alors en grande partie l'intérêt de nos développements théoriques pour l'optimisation des flux.

Pour étendre et approfondir ces constatations et, dans l'optique de présenter une mise en œuvre des techniques d'optimisation pour l'implantation des nœuds évoquée au point 2.1, nous abordons désormais la présentation de l'implantation d'une application plus complète.

3.3 Implantation d’un algorithme de détection optimale de contours

“Il n’est pas, je le sais,
de route en ligne droite,
seul un grand labyrinthe
de carrefours multiples.

A tout moment nos pas
font naître en avançant
d’immenses éventails
de sentiers en germe.” **de Frederico Garcia Lorca, Poésies V,**
extrait de *“Les ponts suspendus”*.

Dans cette section, nous présentons les détails de l’implantation de l’algorithme pour la détection optimale de contours mise au point par Frederico Garcia Lorca [76] (qui ne se trouve pas être le célèbre écrivain !). Cette étude de cas fouillée nous permettra de valider le modèle de performance et mettra en évidence la mise en œuvre des méthodologies de développement introduites.

La détection des contours des objets d'une image apparaît souvent comme une des composantes clé des systèmes de vision. Il s'agit d'un traitement bas niveau qui vise à isoler les traits caractéristiques d'une image. Généralement, cette étape est suivie par des algorithmes moyen/haut niveau comme la transformée de Hough ou la vectorisation des segments qui, entre autres choses, constituent des algorithmes importants vers la reconnaissance ou la classification de formes.

Nous proposons un bref aperçu de l’état de l’art des méthodes de détection de contours et présentons l’approche de filtrage optimal retenue. Ensuite, nous détaillons les étapes de l’implantation de l’algorithme sur C80 en soulignant la mise en œuvre de notre méthodologie de gestion de flux. Pour illustrer plus avant l’utilisation des techniques d’optimisation introduites au chapitre 2, nous présentons également une implantation simulée de l’algorithme sur l’architecture C62 et concluons par une analyse qualitative et quantitative de l’implantation sur ces deux architectures.

3.3.1 Etat de l’art des méthodes pour la détection de contours

La segmentation des contours se différencie de la segmentation en régions par le fait qu'elle repose sur la détection de transitions entre zones homogènes (non texturées). Ainsi, la segmentation des contours s'appuie généralement sur la détection des maxima locaux du gradient ou sur le passage par zéro de la dérivée seconde (le laplacien). Le seuillage binaire des variations localisées du gradient ou laplacien produit l’image des contours. Généralement, ce seuillage s'appuie sur les variations du module mais il intègre parfois l'information de direction du gradient ou du laplacien de manière à améliorer la qualité de la détection en présence de bruit ou lorsque les images sont faiblement contrastées. Au niveau même de l'estimation du module du gradient, nous pouvons d'ailleurs citer l'exemple des masques multi-directionnels de Kirsh qui permettent de retenir la valeur du module correspondant à la direction parmi les 8 testées qui présente la meilleure réponse (caractérisant la transition la plus franche des voisinages considérés).

Nous trouvons dans la littérature [77][78], une pléthore de techniques plus avancées que l’approche à base de gradient/laplacien pour la segmentation des contours comme celles qui combinent l'information du laplacien et du gradient, celles basées sur une modélisation par champ de Markov, ou bien encore l’approche des contours dynamiques (snakes). La charge de calcul qu'engendrent ces techniques dans une optique de traitement d'images à cadence vidéo est telle que le seul seuillage du module du gradient constitue encore bien souvent une approche privilégiée en terme de rapport complexité/performance.

Pour le calcul du gradient ou laplacien, nous trouvons dans la littérature, l'article de Canny [79] qui précède la publication bien connue de Rachid Deriche [80]. Au sens de la théorie du traitement de signal, Canny établit 3 critères pour la détection pour une segmentation optimale et en déduit une fonction de transfert pour le gradient et le laplacien. Les critères de détection optimale pour un signal d'entrée en forme d'échelon (Heaviside) sont les suivants :

- Une bonne détection en présence de bruit
- Une bonne localisation des transitions
- La non-multiplicité de la réponse à une transition

Sur la base de ces critères, Canny introduit un filtre à base d'une exponentielle ayant comme fonction de transfert un filtre FIR. L'idée consiste à convoluer le signal avec la dérivée de cette fonction exponentielle initialement choisie pour ses propriétés passe-bas de manière à favoriser la détection en présence de bruit. Plus tard, Deriche reprend ces travaux et suivant ces mêmes critères, il introduit un filtre à réponse impulsionnelle infinie (qu'il modélise toujours à base d'une exponentielle) pour le gradient et le laplacien. Il obtient la même équation différentielle que celle de Canny mais change les conditions aux limites de l'équation. Ce nouveau filtre récursif présente l'avantage de necéssiter moins de calculs que le filtre FIR de Canny. Un coefficient (α) est associé aux coefficients du filtre récursif et permet de régler l'étalement de la réponse. Pour son filtre, Deriche démontre une amélioration de la qualité de détection de l'ordre de 25% par rapport à l'approximation du filtre de Canny par une Gaussienne. Plus récemment, nous trouvons dans la littérature [81] un filtre basé sur une fonction hyperbolique qui reprend la structure générale du filtre de Deriche et qui offre une qualité de localisation des contours analytiquement identique. L'introduction d'un paramètre supplémentaire permet de régler de manière indépendante les critères de détection et de réponses multiples et permet ainsi une détection plus fine des contours pour une complexité calculatoire équivalente à celle de Deriche.

D'après [77], le filtrage optimal selon Deriche s'avère plus performant que les filtres séparables de type Sobel ou Roberts. Cette performance se traduit par une complexité de calcul accrue mais aussi par une quantité de données transférées plus grande. Dans le cas 2D, si *SS(i,j)* représente le filtre de lissage de Deriche suivant les directions horizontale et verticale, la règle de la dérivation pour la convolution nous incite à convoluer le signal d'entrée avec la dérivée du filtre initial et ce indépendamment pour la direction horizontale (indice i) et verticale (indice j). Si GSS_x représente la dérivée du lisseur *SS(i,j)* suivant la direction horizontale, nous obtenons ainsi la composante du gradient G_x alors que le filtre

GSS_y nous permet d'obtenir la composante verticale du gradient G_y. Avec k se définissant comme une constante de normalisation, nous avons les fonctions de transfert suivantes :

Equation 3-1. Structure des filtres de Deriche en 2D

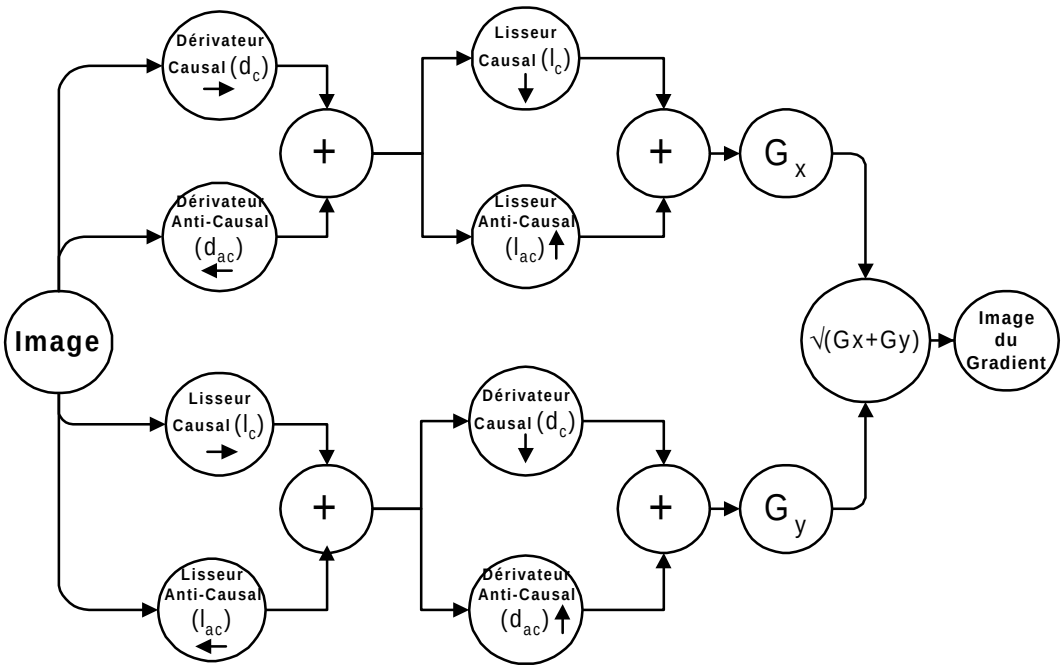
$$SS(i,j) = k \cdot (\alpha|i| + 1) \cdot e^{-\alpha|i|} \times k \cdot (\alpha|j| + 1) \cdot e^{-\alpha|j|}$$

$$GSS_x(i,j) = \frac{\partial SS(i,j)}{\partial i} = \underbrace{k' \cdot i \cdot e^{-\alpha|i|}}_{\text{(D)érivateur horz.}} \times \underbrace{k \cdot (\alpha|j| + 1) \cdot e^{-\alpha|j|}}_{\text{(L)isseur vert.}}$$

$$GSS_y(i,j) = \frac{\partial SS(i,j)}{\partial j} = \underbrace{k \cdot (\alpha|i| + 1) \cdot e^{-\alpha|i|}}_{\text{(L)isseur horz.}} \times \underbrace{k' \cdot j \cdot e^{-\alpha|j|}}_{\text{(D)érivateur vert.}}$$

La structure des filtres GSS_x et GSS_y montre que nous appliquons en cascade et pour chaque direction le filtre de dérivation ainsi que le lisseur suivant la direction orthogonale. A chacune de ces étapes, ces deux filtres se décomposent comme le produit d'un filtre causal et anti-causal. L'application du gradient de Deriche peut alors globalement se schématiser avec la figure suivante :

Figure 3-5. Filtrage de Deriche en 2D



Avec *x(i)* représentant le signal d'entrée, les équations des filtres sont par ailleurs données par :

Equation 3-2. Lisseur causal

$$l_c(i) = x(i) + (\alpha - 1)e^{-\alpha} \cdot x(i-1) + 2e^{-\alpha} \cdot l_c(i-1) - e^{-2\alpha} \cdot l_c(i-2)$$

Equation 3-3. Lisseur anti-causal

$$l_{ac}(i) = (\alpha + 1)e^{-\alpha} \cdot x(i+1) - e^{-2\alpha} \cdot x(i+2) + 2e^{-\alpha} \cdot l_{ac}(i+1) - e^{-2\alpha} \cdot l_{ac}(i+2)$$

Equation 3-4. Lisseur 1D de Deriche

$$L(i) = \frac{(1 - e^{-\alpha})^2}{1 + 2\alpha e^{-\alpha} - e^{-2\alpha}} \cdot (l_c(i) + l_{ac}(i))$$

Equation 3-5. Dérivateur causal

$$d_c(i) = x(i-1) + 2e^{-\alpha} \cdot d_c(i-1) - e^{-2\alpha} \cdot d_c(i-2)$$

Equation 3-6. Dérivateur anti-causal

$$d_{ac}(i) = x(i+1) + 2e^{-\alpha} \cdot d_{ac}(i+1) - e^{-2\alpha} \cdot d_{ac}(i+2)$$

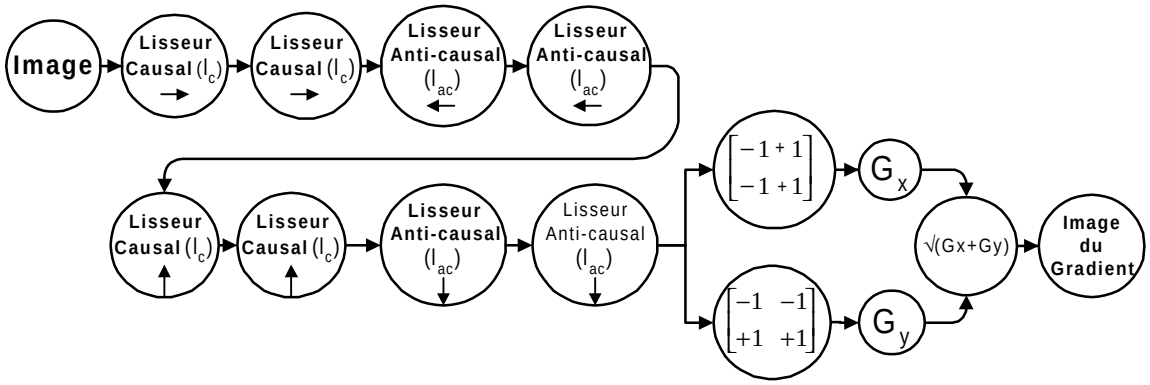
Equation 3-7. Gradient 1D de Deriche

$$D(i) = (1 - e^{-\alpha})^2 \cdot (d_{ac}(i) - d_c(i))$$

D'après la Figure 3-5, nous remarquons que l'application du filtre de Deriche nécessite de parcourir l'image à plusieurs reprises dans les quatre directions. Cette contrainte rend difficile l'exploitation de la localité de l'information et nécessite soit de transférer une quantité importante de données soit de disposer de mémoires caches de grande capacité afin de réduire les transferts.

Cette constatation, combinée à la complexité calculatoire intrinsèque de ce filtre, a conduit plusieurs équipes de recherche à réaliser des implantations FPGA ou ASIC (*Application Specific Integrated Circuit*) dans le cadre d'une mise en œuvre temps réel de ce type d'opérateurs [82][83][84][85]. L'importante quantité de mémoire nécessaire à la réalisation du filtre a initialement motivé l'équipe de D. Demigny à en simplifier la structure. Dans sa thèse, Frederico Garcia Lorca (FGL) décompose ainsi le filtre de Deriche en une cascade de filtres causaux et anti-causaux pour chaque direction (Figure3-6).

Figure 3-6. Filtres cascades récursifs du 1er ordre de FGL



Ces filtres sont des filtres récursifs d'ordre un ou deux et ont pour but de lisser les données. Pour chaque direction, le filtre d'ordre un est appliqué deux fois dans le sens causal puis anti-causal, alors qu'une seule passe suffit pour le filtre du deuxième ordre. Ce dernier filtre correspond au carré du filtre d'ordre un et permet de gagner quatre multiplications. Nous donnons les équations de ces filtres passe-bas :

Equation 3-8. Lisseur causal du premier ordre

$$l_c(i) = (1 - e^{-\alpha}) \cdot x(i) + e^{-\alpha} \cdot l_c(i-1)$$

Equation 3-9. Lisseur anti-causal du premier ordre

$$l_{ac}(i) = (1 - e^{-\alpha}) \cdot x(i+1) + e^{-\alpha} \cdot l_{ac}(i+1)$$

Equation 3-10. Lisseur causal du deuxième ordre

$$L_c(i) = (1 - e^{-\alpha})^2 \cdot x(i) + 2e^{-\alpha} \cdot L_c(i-1) + e^{-2\alpha} \cdot L_c(i-2)$$

Equation 3-11. Lisseur anti-causal du deuxième ordre

$$L_{ac}(i) = (1 - e^{-\alpha})^2 \cdot x(i) + 2e^{-\alpha} \cdot L_{ac}(i+1) + e^{-2\alpha} \cdot L_{ac}(i+2)$$

Le calcul du gradient est remplacé par un noyau de convolution 2×2 traditionnel (du type Roberts) qui se décompose lui-même en un noyau 1-D pour l'estimation du gradient horizontal et un autre pour le gradient vertical. Le calcul du module s'appuie, par ailleurs, sur la norme *L1*[1] que nous employons plus facilement compte-tenu de l'existence généralisée de mécanismes matériels qui permettent le calcul de la valeur absolue très

1. A la page 56 de [76], l'auteur reprend une idée de Mr. Jean Devars qui suggère d'approximer le module du gradient par $max(|G_x|,|G_y|,¾|G_x+G_y|)$, le facteur ¾ ayant pour but d'arrondir le module pour une orientation à 45° ($\cos(45°)=½\sqrt{2}\approx¾$). Afin d'optimiser les calculs, nous retiendrons cependant la norm *L1* du gradient lors de l'implantation tout comme le fait F.G. Lorca pour ses travaux sur FPGA et ASIC.

rapidement. Cette remarque qui sera illustrée par la suite, s'applique également à l'estimation du gradient de Deriche.

L'approche de FGL affiche les mêmes performances en terme de qualité de détection que le filtre de Deriche alors que le nombre d'opérations par point du lisseur du gradient combiné est presque diminué par deux. Dans le même temps, la structure cascadée des filtres permet une implantation plus aisée et nécessite moins de mémoire. Le tableau suivant résume la complexité calculatoire des deux filtres :

Table 3-3. Complexité comparée des filtres de Deriche et de FGL

Deriche			**FGL 1er ordre & Gradient**			**FGL 2ème ordre & Gradient**		
MUL	**ADD**	**ABS**[a]	**MUL**	**ADD**	**ABS**	**MUL**	**ADD**	**ABS**
26	**25**	**2**	**16**	**15**	**2**	**12**	**15**	**2**

a. Opération "valeur absolue" associée au calcul de la norme L1 du gradient

3.3.2 Implantations matérielles sur DSP

Cette section est consacrée à l'implantation matérielle sur les architectures C62 et C80 de l'opérateur de détection de contours optimale introduit par Frederico Garcia Lorca. Pour chaque architecture, nous détaillons les techniques d'optimisation employées qui mènent à une implantation temps réel du détecteur en précision fixe. Par ailleurs, nous reprenons certains aspects architecturaux des processeurs cibles pour une meilleure compréhension de la projection du graphe algorithmique sur les ressources du processeur.

3.3.2.1 Implantation sur le C80

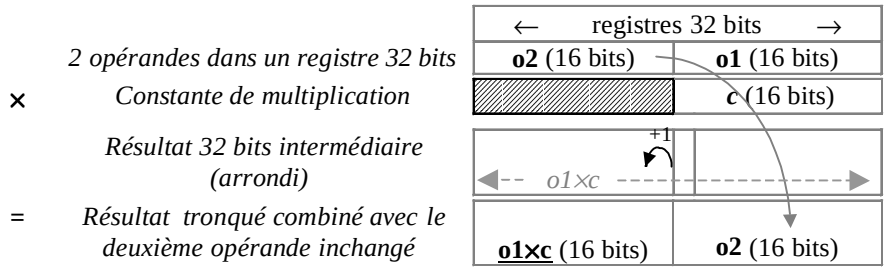
Ici, l'implantation de la chaîne algorithmique s'articule autour de l'infrastructure logicielle C80 qui met en œuvre notre approche pour la gestion des flux de données par le biais DMA. Le partitionnement est de type SPMD avec les processeurs PP. Nous présentons tout d'abord l'exploitation des ressources VLIW pour l'implantation du nœud de lissage ainsi que les performances des nœuds de calcul de gradient ainsi que celui du seuillage. De là, nous introduisons la description de la chaîne et validons les performances estimées.

3.3.2.1.1 Codage des nœuds

Dans un premier temps, nous détaillons l'implantation du lisseur récursif du premier ordre. Nous soulignons que paradoxalement, l'implantation du filtre du deuxième ordre qui compte moins d'opérations que le premier (cf. tableau 3-3) ne s'avère pas plus efficace dans le cas du C80 du fait des restrictions imposées sur la structure du code opératoire VLIW et de la faible quantité de registres de données à disposition (8).

Pour le premier ordre, nous obtenons deux cycles par pixel et par PP en mettant en œuvre le pipeline logiciel. A cette fin, nous exploitons une spécificité du multiplieur qui est propre aux capacités de traitement SIMD de l'ALU. Ainsi, nous utilisons un mode de fonctionnement qui permet d'effectuer deux multiplications 16 bits en parallèle en deux cycles. Ce mécanisme est notamment utilisé pour le calcul de la transformée en cosinus discret et se détaille selon le schéma suivant :

Figure 3-7. Multiplication SIMD 16 bits des PP



La première étape consiste à multiplier l'opérande qui se trouve dans les 16 bits de poids faible du registre d'opérande (*o1*) par une constante *c*. Le résultat intermédiaire s'écrit sur 32 bits, mais les 16 bits de poids fort sont arrondis en propageant l'éventuelle retenue de l'addition du $15^{ème}$ bit de poids faible avec un bit à 1. A la fin de ce traitement, l'opérande de poids fort initiale (*o2*) est recopiée dans la partie basse du registre résultat. L'exécution de l'ensemble de ces opérations intervient en un cycle d'horloge. Si nous relançons ces calculs à l'aide des paramètres o1×c|o2 et de la constante *c'*, nous obtenons comme résultat final le produit des deux opérandes initiaux par les constantes *c* et *c'* (o2×c'|o1×c) en deux cycles d'horloge.

Pour le lisseur, nous n'utilisons que le premier cycle du schéma de double-multiplication précédent afin de bénéficier de la bonne précision des calculs qui repose sur la propagation du 15ème bit. La réponse du filtre de FGL étant normalisée, nous utilisons l'ensemble des 16 bits pour coder la dynamique des valeurs des coefficients A et $B \in$ [0,1.0] avec, pour le premier ordre causal :

$$l_c(i) = A \cdot x(i) + B \cdot l_c(i-1)$$
$$A = (1 - e^{-\alpha}),\ B = e^{-\alpha}$$

Ainsi, si C représente la valeur continue de A ou B, nous avons $c = (2^{16}-1) \times C$. Nous utilisons ensuite les capacités de masquage et de rotation de bits de l'ALU pour manipuler la partie haute des résultats de la multiplication.

Nous donnons tout d'abord le pseudo code séquentiel des opérations du coeur de boucle où X et Y correspondent respectivement aux pixels de l'image d'entrée et de sortie alors que x, y, θ, τ représentent les résultats intermédiaires :

1.x ← X[i ← i + 1]
2.θ ← x×B
3.τ ← y×A
4.y ← θ + τ
5.Y[j ← j + 1] ← y

Nous notons que le pseudo code précédent met en avant les capacités de post-incrémentation des pointeurs pour le chargement et stockage des données (via les indices i et j).

En introduisant les opérations parallèles et en repliant le graphe de dépendance de ce coeur, le pseudo code s'écrit alors (avec '||' représentant les opérations exécutées en parallèle) :

•Initialisation :	x ← X[0] || τ ← 0
	θ ← x×B || x ← X[1]
•Coeur : 1er cycle :	θ ← x×B || y ← θ + τ || x ← X[i ← i + 1]
•Coeur : 2^{ième} cycle :	τ ← y×A || Y[j ← j + 1] ← y
•Fin :	θ ← x×B || y ← θ + τ
	τ ← y×A || Y[j ← j + 1] ← y
	Y[j] ← y

Ce code s'appuie notamment sur le fait que les registres ont plusieurs ports d'accès ce qui justifie que certaines variables apparaissent à la fois comme opérande et destination des opérations. En outre, l'utilisation d'un contrôleur de boucle matériel permet d'occulter les instructions nécessaires à la gestion des itérations successives du coeur.

Si nous mettons en avant les ressources de l'ALU utilisées pour isoler les résultats du schéma de double-multiplication précédemment détaillé, nous avons :

- Coeur : 1^{er} cycle : $\theta \leftarrow x \times B \parallel y \leftarrow (\theta \,\&\, \%16) + (\tau \setminus\setminus 16 \,\&\, \%16) \parallel x \leftarrow X[i \leftarrow i + 1]$
- Coeur : $2^{ième}$ cycle : $\tau \leftarrow y \times A \parallel \theta \leftarrow \theta \setminus\setminus 16 \parallel Y[j \leftarrow j + 1] \leftarrow y$

Nous notons que ce coeur est facilement adaptable à une dynamique de pixels de huit ou seize bits ($d_s = 1/2$). Par ailleurs, nous imposons un minimum de données à évaluer qui correspond au nombre d'itérations logiques de l'algorithme traité avec la phase d'initialisation (le prologue), une occurrence du coeur de boucle, et la phase de fin (l'épilogue). Ici, nous choisissons de conserver le prologue et l'épilogue de ce coeur (par souci de simplification, nous n'introduisons pas d'exécution conditionnée ou spéculative tel que présentée au point 2.1.4). Nous optimisons la granularié du nœud en n'intégrant pas une version du coeur non-pipeliné qui s'appliquerait dans le cas où le nombre de pixels à traiter serait inférieur à 3. En effet, avec une seule exécution de la boucle, nous énumérons 3 pixels traités dans le coeur et l'épilogue (nous comptons 3 opérations de la forme $Y[j] \leftarrow y$) ce qui nous renseigne directement sur la géométrie interne du patron de sortie : $w_t \times h_t = 3 \times 1$. Le pas horizontal/vertical de ce patron se définit comme étant d'un pixel ($w_s = 1$, $h_s = 1$). Pour ce qui concerne le patron en entrée, 4 pixels sont nécessaires pour l'obtention de 3 pixels en sortie, soit 3 nouvelles valeurs du tableau X complétées par la valeur rechargée de $l_c(i-1)$ lorsque $N_w > 1$[1]. L'approche du rechargement que nous visons nécessite de faire coïncider le buffer d'entrée avec celui en sortie (cas similaire à ce que nous décrivons dans la Figure 2-9 (B), le filtre étant ici appliqué horizontalement et la dépendance $y(n-1)=l_c(i-1)$ intervenant entre deux bandes verticales. Pour le patron d'entrée, le rechargement horizontal est de 1 pixel ($w_r = 1$) et nul pour la direction verticale ($h_r = 0$).

1. Pour le cas où $N_w > 1$, le rechargement des valeurs se trouvant sur la précédente bande verticale du buffer annule l'initialisation "$\tau \leftarrow 0$".

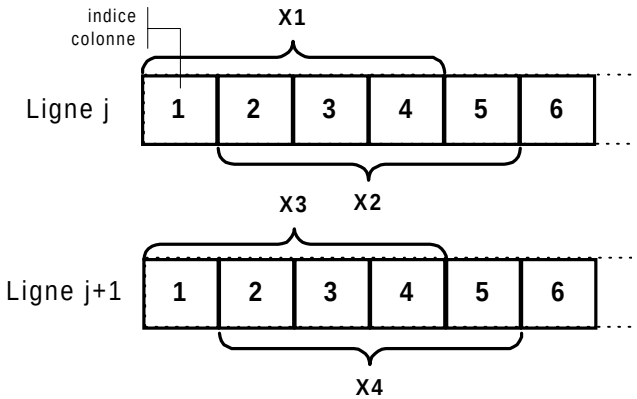
Concernant le gradient, l'implantation s'appuie sur les possibilités SIMD de l'architecture. Nous obtenons quatorze cycles pour quatre pixels (par PP) grâce à l'utilisation de l'expansion de masques. Ce mécanisme détaillé avec l'équation 1-2 de la page 36, nous permet d'obtenir le calcul de quatre valeurs absolues de pixels 8 bits en seulement deux cycles. Avec '~' représentant l'inversion binaire, nous calculons :

1.Dif $\leftarrow \theta - \tau$

2.AbsDiff $\leftarrow$ (Diff & *@mf*) | (-Diff & ~*@mf*)

Ici, le calcul de la différence du premier cycle s'entend comme une opération SIMD qui met à jour le registre de statut multiple *mf*. Comme déjà évoqué, ce registre nous permet notamment de sélectionner les sous-opérations de la dernière addition/soustraction SIMD ayant engendré un résultat négatif. Ainsi, au cycle suivant, l'expression logique de l'ALU qui est utilisée, permet de s'appuyer sur *mf* pour masquer les octets ayant engendré une différence négative et de prendre, dans ce cas, l'opposé. Ce mécanisme nous procure bien les quatre valeurs absolues d'octets dans un registre 32 bits.

Pour l'application des masques, l'organisation des pixels est donnée par $\begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix}$ et sa version SIMD par $\begin{bmatrix} X_1 & X_2 \\ X_3 & X_4 \end{bmatrix}$ que nous pouvons détailler à l'aide du graphique suivant :



Pour le gradient horizontal, nous avons le masque $\begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix}$ qui permet de définir les variables θ et τ introduites avec le pseudo code précédent pour le calcul de la valeur

absolue. En convoluant ce noyau avec la représentation SIMD de la matrice de voisinage pixels, nous avons :

$$\theta = X2 + X4$$
$$\tau = X1 + X3$$

Le gradient vertical se calculant suivant les mêmes étapes, nous comprenons qu'un minimum de 2×4 cycles est nécessaire pour le calcul de la valeur absolue des deux gradients ainsi qu'un cycle supplémentaire pour leur combinaison. Nous expliquons la différence entre ces neuf cycles et les douze cycles de l'implantation effective par l'adjonction d'opérations visant à diminuer la dynamique des variables intermédiaires θ et τ pour effectivement autoriser l'arithmétique SIMD sans qu'il y ait dépassement dans les calculs (la saturation n'étant pas directement supportée). Cette réduction de la dynamique s'effectue par simple décalage des registres. Par souci de clarté, nous ne détaillons pas ici l'ensemble des douze instructions qui sont par ailleurs pipelinées.

La quantité minimum de données "horizontales" (w_t) en entrée est de dix. Tout comme pour le lisseur, nous ne cherchons pas à optimiser le prologue ou l'épilogue, mais plutôt à supprimer le besoin d'une version du coeur non-pipelinée. Ici, w_t se décompose selon la quantité de données traitées pour une itération du coeur de traitement (soit un ensemble d'opérations SIMD portant sur cinq pixels) ainsi que les pixels supplémentaires traités dans le prologue. Le coeur chargeant quatre des cinq pixels de l'itération de boucle suivante, un prologue d'une itération vient traiter le dernier lot de cinq pixels. Pour ce patron d'entrée, le pas horizontal (w_s) est de cinq et pour ce qui concerne la direction verticale, nous avons $h_t=2$, $h_s=1$, le rechargement correspondant ici à la valeur par défaut, c-à-d $h_r=h_t-h_s=1$. La quantité minimum de données produites (patron de sortie) est de neuf et le pas associé est de cinq. Pour finir, nous introduisons un dernier nœud très simple qui permet de seuiller le module du gradient. Ce nœud reprend le même mécanisme que pour le calcul SIMD de la valeur absolue et s'écrit, avec *X* regroupant quatre pixels d'entrée consécutifs :

1. Diff ← seuil - X
2. SeuilGradient ← (0x00000000 & *@mf*) | (0xFFFFFFFF & ~*@mf*)

Avec ce principe, seuls deux cycles pour quatre pixels sont nécessaires pour produire l'image binaire des contours (avec des pixels sur 8 bits en sortie). La géométrie d'entrée et de sortie de ce nœud s'écrit alors simplement $(w_t \times h_t, w_s, h_s) = (4 \times 1, 4, 1)$.

Le tableau suivant récapitule les performances théoriques hors impact de la gestion des flux de cette chaîne algorithmique :

Table 3-4. Performance théorique sans l'impact DMA des traitements sur C80

	Par PP		**Performance avec 4 PP**
	Cycles par pixel	**Nbr. de passes**	
FGL 1er ordre	**2/1**	**8**	**4×512^2**
Gradient	**12/4**	**1**	**$12/16 \times 512^2$**
Seuillage	**2/4**	**1**	**$2/8 \times 512^2$**
		Nbr. total de cycles	**1310720**
		Durée de traitement théorique à 60 MHz (sans les transferts)	**22 ms**

3.3.2.1.2 Partitionnement et gestion des flux

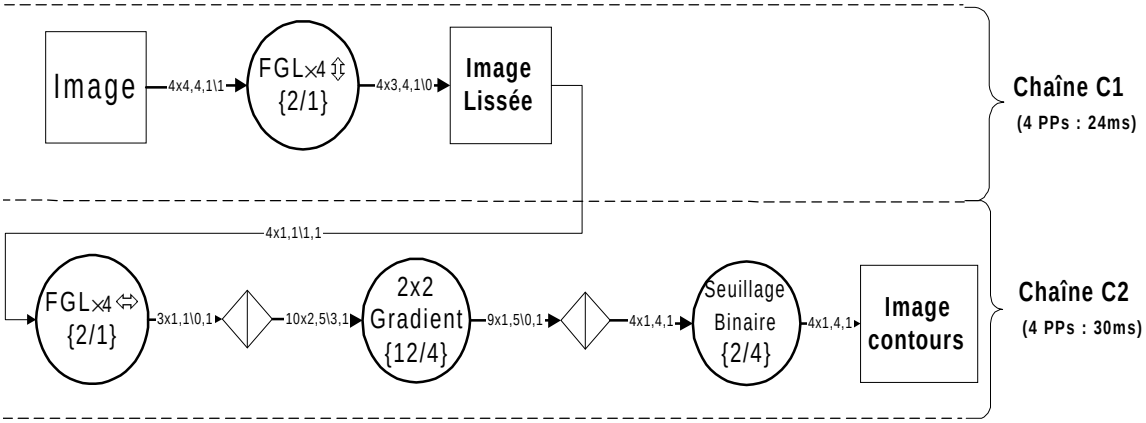
Le partitionnement est de type SPMD et s'appuie sur la librairie de gestion des flux de données détaillée au chapitre 2. Les patrons d'entrée/sortie internes des différents nœuds de traitement de la chaîne sont résumés dans le tableau suivant :

Table 3-5. Patron de données des nœuds de la chaîne FGL

Nœud	**Géométrie du patron ($w_t \times h_t, w_s[\backslash w_r], h_s[\backslash h_r]$) et $d_s = d_p = 1$**	
	en entrée (I)	**en sortie (O)**
FGL 1er ordre	**4×1,1\1,1\0**	**3×1,1\0,1\0**
Gradient selon FGL	**10×2,5\3,1\1**	**9×1,5\0,1\0**
Seuillage	**4×1,4,1**	**4×1,4,1**

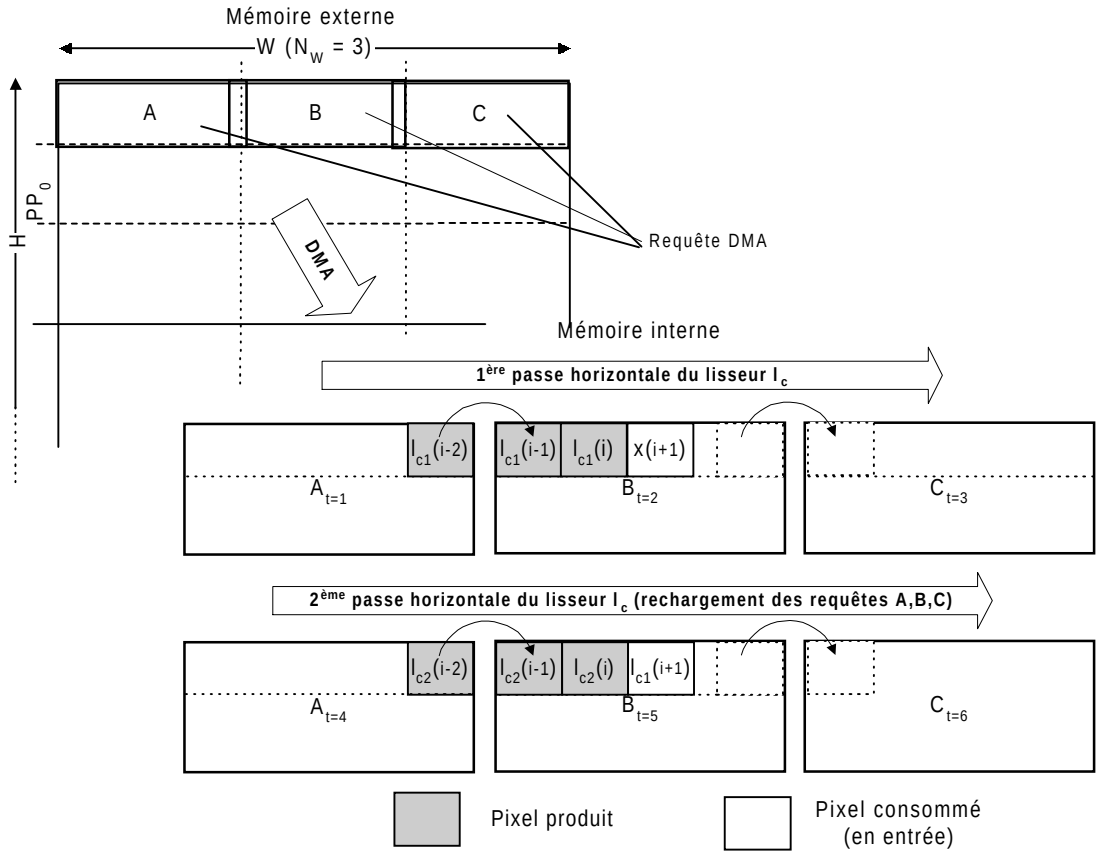
Le traitement se déroule en deux temps. Une première étape applique les quatre passes du lisseur du premier ordre suivant la direction verticale (2 passes causales et 2 passes anti-causales). Ensuite, nous exécutons une deuxième chaîne qui prend comme entrée la sortie de la première étape et qui regroupe les 4 passes du lisseur horizontal, le calcul du module du gradient et le seuillage. La Figure3-8 résume ce partitionnement qui, pour chacune des étapes, met en œuvre le double buffering des transferts parallèles.

Figure 3-8. Chaînes de détection de contours sur le C80



Nous soulignons, par ailleurs, que la combinaison de la nature récursive du filtre avec les quatre itérations locales de traitement qui regroupent donc les deux passes causales et les deux anti-causales, pose un problème lorsque le nombre de bandes horizontales (N_W) (ou verticales (N_H) pour le parcours transposé) est supérieur à 1. La Fi gure3-9 souligne en effet que la dépendance de données $l_c(i-1)/l_c(i)$ (cas causal) concerne bien des données issues d'une même passe de traitement.

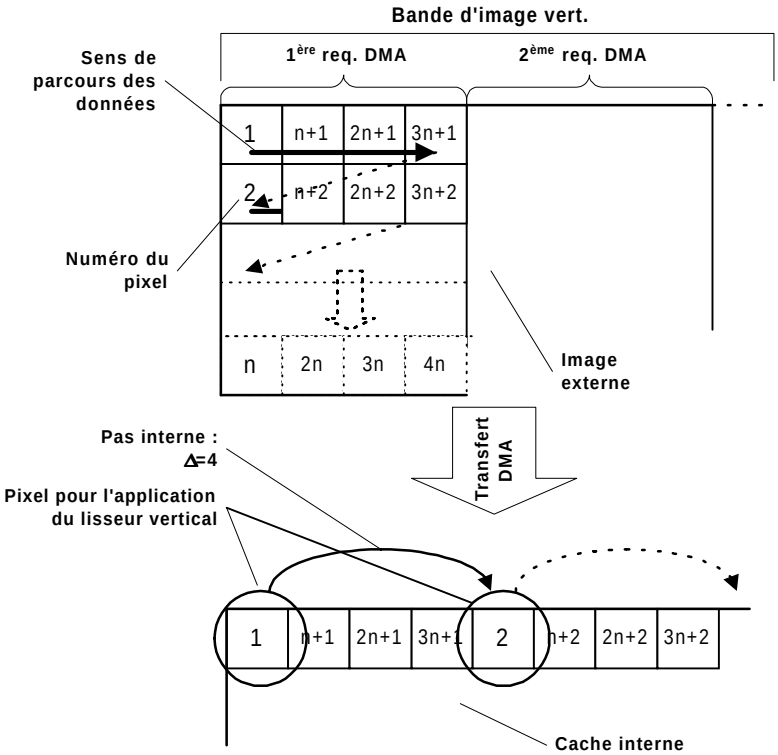
Figure 3-9. $N_W > 1$ et multiples itérations du lisseur de FGL



Ainsi, en toute rigueur, lorsque $N_W > 1$, nous devons recharger l'ensemble des requêtes horizontales entre chaque passe du traitement. Par opposition, lorsque nous avons $N_W = 1$, les quatre passes du lisseur de chaque chaîne s'exécutent sur les données issues des mêmes requêtes DMA ce qui permet d'atteindre des performances accrues. Pour favoriser la localité des données, nous préférons alors limiter la taille des lignes traitées (ou colonnes pour le lisseur vertical). Dans ce sens, comme le patron interne du nœud gradient de la deuxième chaîne de traitement nécessite un minimum de deux lignes d'image, nous limitons donc la portée des deux chaînes à des buffers 1024^2 (pixels 8 bits) pour $S=2048$.

Par ailleurs, pour réduire l'impact des transferts de la passe verticale pour laquelle la récupération des octets contigus ligne par ligne engendre un taux de transfert divisé par huit pour des pixels 8 bits (paramètre ς_1 du modèle de performance présenté au paragraphe 2.3), nous modifions la structure du patron de manière à rapatrier le maximum d'octets contigus d'une même ligne comme l'illustre la Figure3-10. Ainsi, pour une image 512^2, nous pouvons charger les pixels 8 bits quatre par quatre avec un cache de 2 Ko (512×4 = 2 Ko). Cette modification s'associe à une modification mineure du nœud et le support d'un "pas Δ" interne spécifique pour la gestion des données verticales.

Figure 3-10. Application verticale du lisseur de FGL



Si la taille d'image que nous cherchons à traiter est de 512^2, les patrons du lisseur se re-définissent alors selon le tableau suivant :

Table 3-6. Géométrie des patrons du lisseur vertical

Patrons du nœud FGL 1er ordre vertical : $W_t \times H_t, D_S, D_p, W_S[\backslash W_r], H_S[\backslash H_r]$ (externe) et $w_t \times h_t, d_s, d_p, w_S, h_S$ (interne)			
Entrée		**Sortie**	
Externe	**Interne**	**Interne**	**Externe**
4×4,1,1,4\0,1\1	**4×4,1,1,1,1,4**	**3×4,1,1,1,4**	**4×3,1,1,4\0,1\0**

Nous notons que le pas horizontal externe ne s'écrit pas W_S=512 car le parcours se fait bien suivant la direction verticale et par ligne de quatre pixels (octets). En outre, le pas d'entrelacement interne des pixels n'intervient pas au niveau de la géométrie du patron interne (nous n'avons pas $d_p = 4$ car nous ne souhaitons par "réserver de l'espace interne"). Techniquement, Δ qui apparaît comme un argument spécifique au n œud, se calcule indépendemment de notre librairie d'automatisation de la gestion des flux suivant la taille d'images traitées (pour des images 512^2, Δ=S/512=4 avec S=2Ko). Dans la mesure où, du point de vue du traitement, deux cycles par pixel sont nécessaires avec quatre PP pour l'application de l'ensemble des directions et formes du lisseur, l'optimisation du transfert de quatre pixels consécutifs permet de passer d'une bande passante que nous arrondissons à 2×3 cycles pour l'entrée et la sortie d'un octet sur notre carte d'évaluation, à une durée de 1,5 cycle par pixel (2×3/4). Cette approche permet donc de mieux équilibrer la charge de la durée du traitement face à celle des transferts.

Pour illustrer la mise en œuvre de la méthodologie de gestion des flux, nous détaillons les principaux calculs nécessaires à l'obtention du patron virtuel de référence pour les deux chaînes de traitement :

Table 3-7. Calcul du patron virtuel de référence pour la deuxième passe de FGL

Chemins	**Patron virtuel : <(patron de sortie nœud A),(patron d'entrée nœud B)> : constantes de syncho.**	
	(W', W'_S)	**(H'_t, H'_S)**
C1: FGL(2×⇑⇓) : entrée	**(4,4)**	**(4,1)**
C1: FGL(2×⇑⇓) : sortie	**(4,4)**	**(3,1)**

Table 3-7. Calcul du patron virtuel de référence pour la deuxième passe de FGL

Chemins	**Patron virtuel :** **<(patron de sortie nœud A),(patron d'entrée nœud B)> : constantes de syncho.**	
	(W'_r,W'_s)	**(H'_r,H'_s)**
C2: Seuillage → Gradient → FGL(2×⇒⇐)	$<(9,5),(4,4)> : \phi_1=3,\Phi_1=4$ $<(3,1),(10+\phi_1\times5{,}5\times\Phi_1)> : \phi_0=22,\Phi_0=20$ $(4+\phi_0\times1{,}1\times\Phi_0) = \underline{(25,20)}$	$<(1,1),(1,1)> : \tau_1=0,\Gamma_1=1$ $<(1+\tau_1\times1{,}1\times\Gamma_1),(2,1)> : \tau_0=1,\Gamma_0=1$ $(1+\tau_0\times1{,}1\times\Gamma_0) = \underline{(2,1)}$
C2: FGL(2×⇒⇐) → Gradient → Seuillage	$<(3,1),(10,5)> : \phi_0=0,\Phi_0=1$ $<(9+\phi_0\times5{,}5\times\Phi_0),(4,4)> : \phi_1=5,\Phi_1=5$ $(4+\phi_1\times4{,}4\times\Phi_1) = \underline{(24,20)}$	$<(1,1),(2,1)> : \tau_0=0,\Gamma_0=1$ $<(1+\tau_0\times1{,}1\times\Gamma_0),(1,1)> : \tau_1=0,\Gamma_1=1$ $(1+\tau_1\times1{,}1\times\Gamma_1) = \underline{(1,1)}$

Avec ces résultats, nous apprenons que, pour la chaîne C1, la taille minimum d'images traitées doit être de 4 pixels en largeur pour un pas de 4 pixels et de 4 pixels en hauteur pour un pas de 1 dans la direction verticale. De même, la largeur minimum traitée avec la chaîne C2 doit correspondre à 25 pixels pour un pas de 20, avec un minimum de 2 lignes pour la direction verticale et un pas de 1. Pour l'ensemble des buffers, nous avons $W_{ref}=H_{ref}=512$ et $S=2048$, ce qui nous permet d'évaluer le paramètre N_W pour chaque chaîne :

Table 3-8. Paramètres N_W des deux chaînes de FGL

Chaîne	$\xi = \left\lfloor \frac{S/b_v - a_v}{c_v} \right\rfloor$	$N_W = 1 + \left\lceil \frac{\max(0, W-(A+\xi\cdot C))}{\max(\xi\cdot C, E+m\cdot C)} \right\rceil$
C1[a]	$\xi = \left\lfloor \frac{2048/4-4}{1} \right\rfloor = 508$	$N_W = 1 + \left\lceil \frac{\max(0,512-(4+508\cdot 1))}{\max(\ldots)} \right\rceil = 1$
C2	$\xi = \left\lfloor \frac{2048/2-25}{20} \right\rfloor = 49$	$_W = 1 + \left\lceil \frac{\max(0,512-(25+49\cdot 20))}{\max(\ldots)} \right\rceil = 1$

a. Cas du parcours transposé.

Ce tableau nous informe que pour chaque chaîne, $N_{W/Max}=1$. De là, nous pouvons estimer les paramètres β et γ :

Table 3-9. Paramètres beta et gamma des deux chaînes de FGL

Chaîne	$\beta_1 = \beta = \left\lfloor \frac{W_{ref} - A}{C} \right\rfloor$ **(a)**	$\Omega = a + \beta \cdot b$ $\gamma = \left\lfloor \frac{S - b \times \Omega}{d \times \Omega} \right\rfloor + 0/1$
C1	$\beta = \left\lfloor \frac{512-4}{1} \right\rfloor = 508$	$\gamma = \left\lfloor \frac{2048 - 4 \times 512}{1 \times 512} \right\rfloor + 0 = 0$
C2	$\beta = \left\lfloor \frac{512-25}{20} \right\rfloor = 24$	$\gamma = \left\lfloor \frac{2048 - 2 \times 505}{1 \times 505} \right\rfloor + 0 = 2$

a. Sur cet exemple multi-buffers, nous avons $\beta_{MIN}=\beta$ et $\gamma_{MIN}=\gamma$.

Ces paramètres nous permettent finalement d'obtenir le nombre de requêtes DMA synchrones $\mathbf{N_{R1}}$ et $\mathbf{N_{R0}}$ (cf paragraphe 2.2.3.2.1) pour l'unique bande verticale des deux chaînes avec $\boldsymbol{\sigma=\sigma_{ref}=4}$:

Table 3-10. Nombre de requêtes DMA des deux chaînes de FGL

Chaîne	$N_{\gamma 0} = n + \left\lfloor \frac{H + (F - n \cdot D) \cdot (\sigma - 1)}{D \cdot \sigma} \right\rfloor$ $N_{\gamma 1} = \frac{H - B}{D} - N_{\gamma 0} \cdot (\sigma - 1)$	$N_{R0} = \left\lceil \frac{H_0}{D \cdot (\gamma + 1)} \right\rceil$ $N_{R1} = 1 + \left\lceil \frac{\max(0, H_1 - (B + \gamma \cdot D))}{D \cdot (\gamma + 1)} \right\rceil$ **(a)**
C1	$N_{\gamma 0} = 1 + \left\lfloor \frac{512 + (4-0) \cdot (4-1)}{4 \cdot 4} \right\rfloor = 32$ $N_{\gamma 1} = \frac{512-4}{4} - 32 \cdot (4-1) = 31$	$N_{R0} = \left\lceil \frac{128}{4 \cdot (0+1)} \right\rceil = 32$ $N_{R1} = 1 + \left\lceil \frac{\max(0, 128 - (4 + 0 \cdot 4))}{4 \cdot (0+1)} \right\rceil = 32$
C2	$N_{\gamma 0} = 1 + \left\lfloor \frac{512 + (1 - 1 \cdot 1) \cdot (4-1)}{1 \cdot 4} \right\rfloor = 129$ $N_{\gamma 1} = \frac{512-2}{1} - 129 \cdot (4-1) = 123$	$N_{R0} = \left\lceil \frac{129}{1 \cdot (2+1)} \right\rceil = 43$ $N_{R1} = 1 + \left\lceil \frac{\max(0, 125 - (2 + 2 \cdot 1))}{1 \cdot (2+1)} \right\rceil = 42$

a. $H_1=B+N_{\gamma 1} \times D$ et $H_0=N_{\gamma 0} \times D$.

Enfin, le tableau 3-11 regroupe les durées de traitement estimées et celles mesurées pour chaque chaîne de traitement. Sans l'intégration de la mesure du nombre de requêtes de cache d'instructions nécessaires, nous constatons une divergence importante de l'estimation. Lorsque nous simulons le programme (cf paragraphe 2.3.3) et comptabilisons a posteriori le nombre de blocs de cache d'instructions mis à jour pour chaque chaîne (terme $\nu' \times N_w \times \varepsilon$ de notre modèle de performance), "l'estimation" des durées de traitement devient alors relativement bonne avec moins de 15% d'erreur. Cette constatation permet de valider plus précisément notre modèle de performance et illustre l'importance de l'optimisation des requêtes de cache à travers l'optimisation de la granularité des nœuds.

Table 3-11. Résumé des performances des chaînes FGL pour une image 512^2 à 40 Mhz

Chaîne	Estimation de la durée = max(transferts, traitement)		Durée mesurée	Nbr. MAJ de blocs cache d'instructions	Estimation avec impact cache d'instr.[a]	Ecart mesure - nouvelle estimation
	Transfert de données	Traitement				
C1	**8 ms**	**13 ms**	**24 ms**	**6123**	**20 ms**	**15**
C2	**4 ms**	**19 ms**	**30 ms**	**8764**	**31 ms**	**< 1 %**

a. Basée sur nbr. de requêtes × coût / fréquence. Le coût s'exprime suivant le nombre d'accès pour mettre à jour un bloc du cache (128/8) et le nombre de cycles par accès (3), ce qui nous donne 48 cycles par requête de bloc sur notre carte d'évaluation SDB.

Enfin, pour justifier l'écart de précision qu'offre notre modèle de performance entre la chaîne C2 pour laquelle une très bonne précision est atteinte (< 1%) au regard des 15% d'imprécisions de la chaîne C1, nous estimons en particulier que parmi les raisons évoquées à la section 2.3.3, le coût cumulé des fautes de pages d'accès à la mémoire externe lors d'un parcours verticale a un impact significatif que nous ne modélisons pas.

3.3.2.1.3 Conclusions

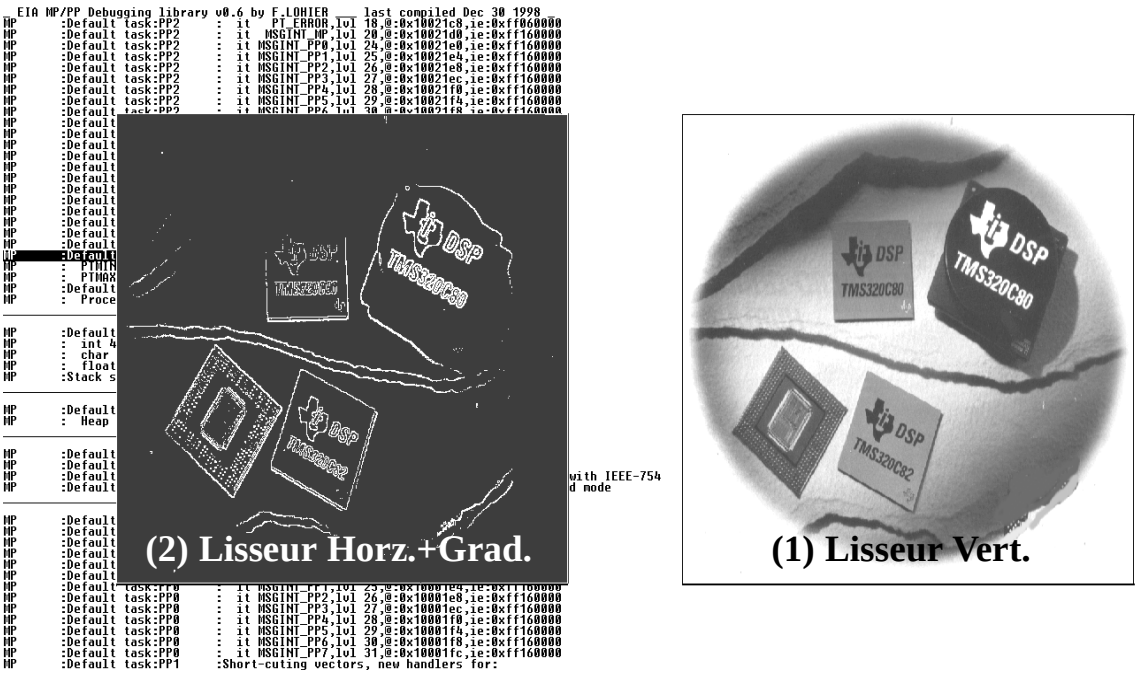
Sur la carte d'évaluation SDB dont les caractéristiques sont résumées par le tableau 3-2, l'algorithme affiche une performance de 54ms pour une image 512^2 à 40 MHz. Nous pouvons tout d'abord comparer cette durée aux 152 ms détaillées dans [36] pour l'implantation du filtre de Canny (sans le seuillage), à configuration matérielle et taille d'images équivalentes.

Ensuite, en reprenant les performances brutes estimées du tableau 3-4 ramenées à 40 MHz, nous constatons que la gestion des flux (données + instructions) représente environ les 3/8ème de la durée globale de traitement. Nous soulignons que cet impact peut être réduit par l'utilisation de mémoire synchrone plus performante alors que d'autre part, nous pouvons atteindre des performances temps-réel pour des images 512^2 avec une version du C80 cadencée à 60 MHz.

Pour finir, dans [86][87][88], nous montrons que cette performance rivalise avec une implantation simplifiée de l'algorithme de FGL compilée sur un éventail de processeurs RISC (les calculs sont flottants et décrits en C optimisé). La performance du C80 est ainsi légèrement meilleure que celle offerte par les meilleurs RISC disponibles en 1998, sans toutefois tirer partie des extensions SIMD de ces derniers. Cette constatation souligne la puissance du C80 pourtant apparu trois ans plus tôt (en 1995).

La Figure 3-11 présente une copie d'écran d'un exemple de détection de contours obtenu avec les deux chaînes sur le système SDB. A droite figure le résultat du lissage vertical et à gauche, nous présentons le résultat de la binarisation du gradient de la précédente image lissée dans la direction horizontale.

Figure 3-11. Copie d'écran du résultat des 2 chaînes FGL sur C80



3.3.2.2 Implantation sur C62

Cette partie du mémoire détaille l'implantation de la détection de contours proposée par FGL sur l'architecture C62. Nous présentons la programmation assembleur des différents nœuds et estimons les performances en dehors du coût de la gestion des flux (qui n'est pas implantée). L'intérêt est de décrire l'utilisation de méthodes de mise en œuvre génériques qui permettent le traitement à cadence vidéo de l'algorithme. Par là, nous soulignons les différences architecturales avec le C80 et notamment, l'exploitation de la structure VLIW des instructions dans le contexte d'un pipeline profond (douze étages contre trois pour les PP).

3.3.2.2.1 Implantation du nœud gradient

L'architecture du C62 s'appuie sur la présence d'un pipeline d'instructions unique et profond dans lequel les opérations des huit unités fonctionnelles concurrentes s'exécutent selon des durées variables (c.f. 1.1.1.3). Les latences des principales opérations qui nous servent pour l'implantation du filtre de FGL ainsi que les différentes unités fonctionnelles (*.L, .S, .M, .D*) dans lesquelles elles s'exécutent, sont détaillées dans le tableau suivant :

Table 3-12. Principales opérations C62 : latence et unité fonctionnelle d'exécution

Mnémonique opératoire	Description	Latence dans la phase d'exécution du pipeline	Unités fonctionnelles (×2)			
			.L	.S	.M	.D
ADD/SUB	Addition/soustractio	1/1	√	√		√
SHR	Décalage à droite signé	1		√		
B	Branchement	6		√		
ADD2/SUB2	Addition/soustractio SIMD de 2×16 bits	1/1		√		
ABS	Valeur absolu	1	√			
MPY	Multiplication	2			√	
LD/ST	Chargement/mémorisation	5/1				√

Ce tableau rend compte du fait que certains mnémoniques assembleur (*ADD/SUB*) peuvent s'exécuter dans plusieurs unités fonctionnelles ce qui offre une grande souplesse dans la mise en œuvre du pipeline logiciel. Par ailleurs, la latence des différentes opérations

détaillées correspond au nombre d'étages de la phase d'exécution du pipeline qui sont utilisés pour le traitement. Une latence de 1 signifie que le résultat de l'opération concernée peut être utilisé au cycle suivant c'est-à-dire, par une unité fonctionnelle de l'instruction VLIW suivante. Pour l'opération de branchement par exemple, l'exécution de l'instruction ne prend qu'un cycle mais cinq cycles sont nécessaires pour ré-initialiser le pipeline d'instructions et les étages qui permettent de récupérer le code opératoire VLIW pointé par le branchement, d'aiguiller les opérations de l'instruction VLIW dans les différentes unités fonctionnelles pour qu'enfin, elles puissent être décodées. La latence effective pour débuter les opérations pointées par l'instruction de branchement est alors de six cycles.

Nous détaillons tout d'abord l'implantation du gradient qui met en œuvre l'utilisation des capacités de traitement SIMD du coeur ainsi que le pipeline logiciel. Pour cela, soulignons la redondance du calcul intermédiaire *k* entre deux itérations successives horizontales du masque de convolution.

En effet, si la configuration des pixels entre crochets donnée par $\begin{bmatrix} x_1 & x_2 \\ x_3 & x_4 \end{bmatrix} \begin{matrix} x_5 \\ x_6 \end{matrix}$ correspond

au voisinage à la colonne *i*, la norme *L1* du gradient proposé par FGL s'écrit alors :

$$\nabla_i \approx |G_x| + |G_y|$$
$$\nabla_i \approx |x_2 + x_4 - (x_1 + x_3)| + |x_3 + x_4 - (x_1 + x_2)|$$

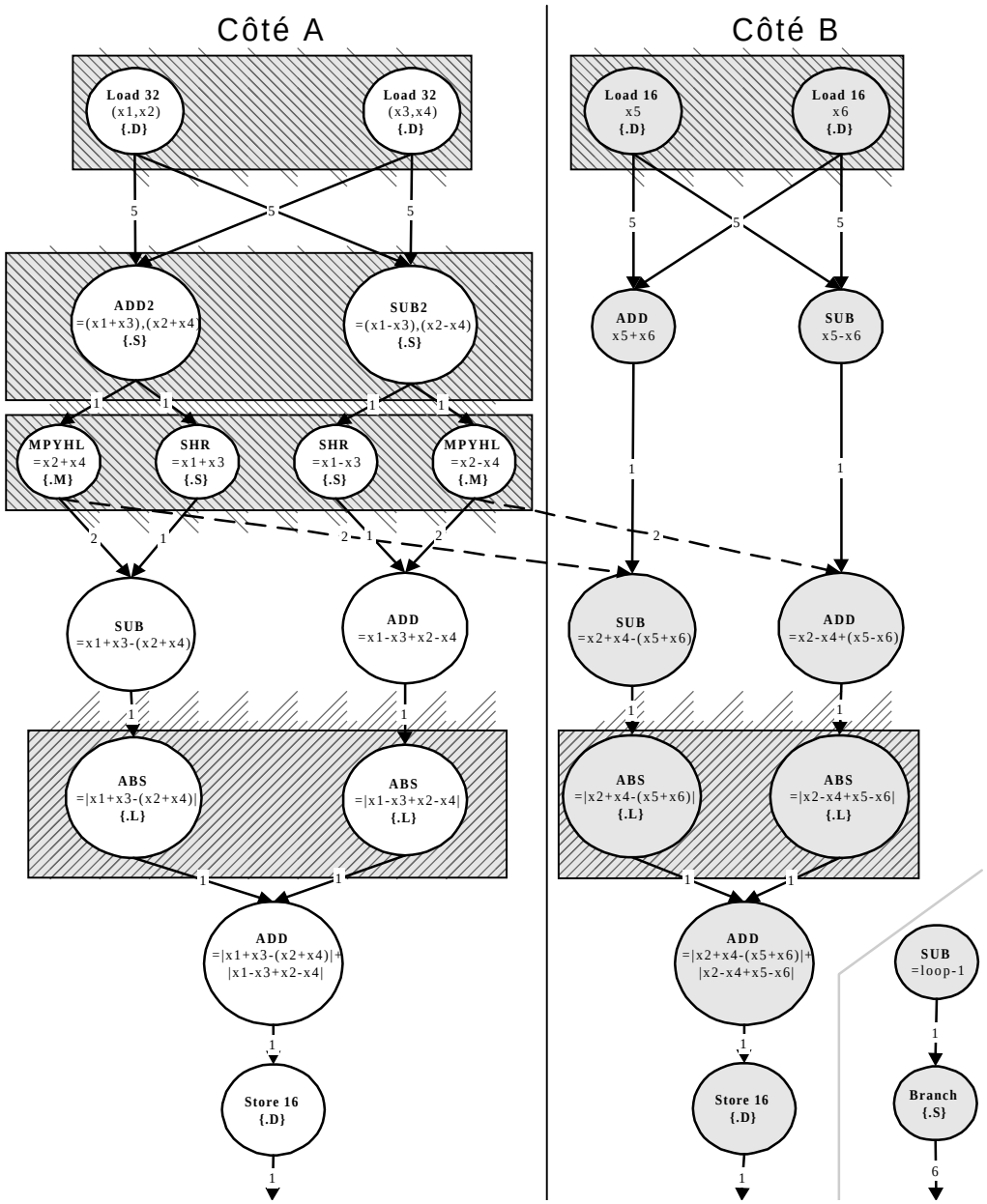
Avec $k=x1 + x4$, le gradient suivant ($i+1$) s'écrit alors :

$$\nabla_{i+1} \approx |x_5 + x_6 - k| + |x_4 + x_6 - (x_2 + x_5)|$$

En travaillant sur des pixels 16 bits, et en chargeant les couples de pixels consécutifs (x_1,x_2) et (x_3,x_4) dans deux registres distincts, nous pouvons nous appuyer sur l'opération SIMD *SUB2* pour l'estimation du gradient vertical. D'une manière similaire, l'opération *ADD2* nous donne une partie des calculs pour le gradient horizontal à savoir, le couple (x_1+x_3, x_2+x_4). Pour chacune de ces opérations, il convient alors de combiner les parties hautes et basses des résultats SIMD. Pour ce faire, nous utilisons le mnémonique *SHR* du décalage logique (vers la droite) ainsi que l'opération spécifique *MPYHL* de la classe des opérations de l'unité *.M* qui permet de multiplier la partie haute d'un registre avec la partie basse d'un

autre (High-Low). En prenant le coefficient 1 comme opérande de ce mécanisme, nous parvenons ainsi à isoler les 16 bits de poids fort d'un registre sans utiliser l'unité *.S* du décalage. A partir de cette description générale, nous pouvons suivre la procédure mise en avant au paragraphe 2.1.4 pour le repliement du graphe de dépendance vers le pipeline logiciel. La Figure 3-12 donne le graphe de dépendance du coeur de boucle qui met en œuvre les opérations SIMD précédemment détaillées.

Figure 3-12. Graphe de dépendance du gradient sur C6X



Par souci de clarté, la figure reprend les opérations assembleurs effectuées ainsi que les principaux résultats obtenus. Sur ce graphique (et les suivants), les nœuds qui s'exécutent sur le côté B apparaissent en gris. Les arcs de dépendance sont pondérés avec la latence des différentes instructions. En accolade, nous reprenons également, les unités fonctionnelles qui, à ce stade, se présentent comme l'unique alternative à l'exécution de certaines opérations (d'après la table 3-12). Pour les autres opérations, nous n'assignons pas encore d'unité fonctionnelle dans la mesure où le repliement du graphe va nous imposer certaines contraintes. Afin d'exploiter le résultat intermédiaire *k,* le graphe de dépendance montre que 2 valeurs du gradient sont calculées par itération de boucle (une par côté A/B). Par ailleurs, en l'absence de contrôleur de boucle, nous intégrons 2 instructions qui permettent de gérer les itérations de la boucle (en bas à droite). Ici, le test de la condition d'arrêt n'apparaît pas dans la mesure où le branchement est conditionné au niveau même de l'opération.

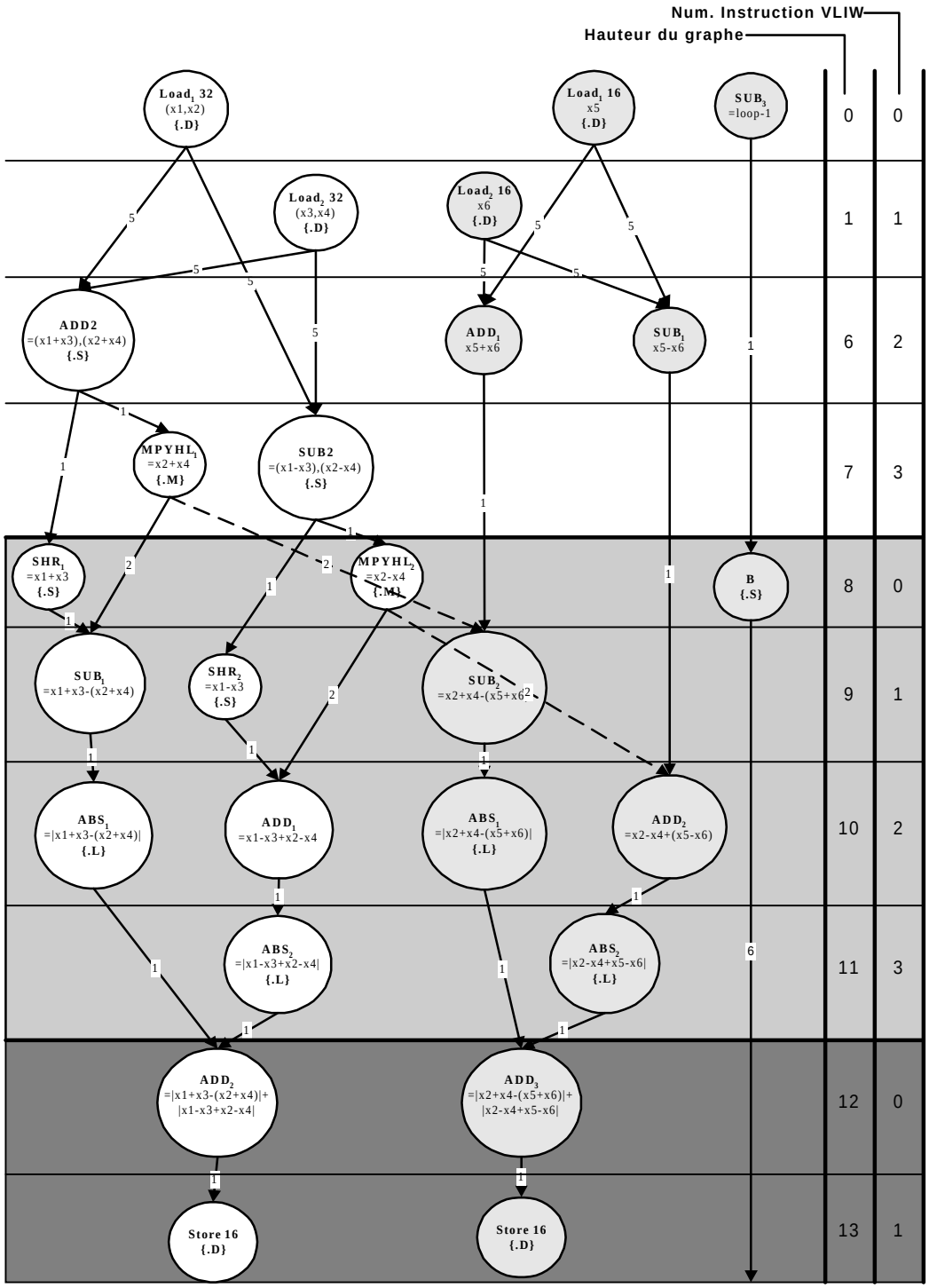
Le choix que nous avons fait de partitionner sur le même côté de l'architecture l'ensemble des opérations portant sur une même valeur de gradient vise à réduire le nombre de références croisées. En effet, comme nous l'avons déjà évoqué au paragraphe 1.1.1.3.1, une seule opération par côté de l'architecture et par instruction VLIW ne peut posséder d'opérande registre provenant du banc de registre associé au côté opposé du coeur CPU (c'est la notion de "crosspath")[1]. Ici, le graphe de dépendance se scinde globalement en deux sous-graphes distincts : à gauche, l'estimation du gradient à la position *i*, à droite à la position *i+1.* Seulement 2 dépendances croisées sont ainsi induites (en pointillés). De plus, pour chaque côté, nous essayons ensuite de diversifier la nature des opérations pour lesquelles l'unité fonctionnelle d'exécution est fixée (en accolade) comme avec l'exemple des opérations *SHR/MPYHL*.

Après cette étape très largement guidée par l'utilisateur, nous pouvons appliquer la procédure d'optimisation dite du "pipeline logiciel". Etant donné que 26 nœuds composent le graphe de dépendance, avec un maximum de 8 instructions par cycle (4 par côté), nous pouvons d'ores et déjà borner notre coeur de boucle qui ne peut compter moins de 4

1. Dans le détail, cette restriction s'applique au groupe d'opérations exécutées en parallèle, le format VLIW autorisant l'ordonnancement séquentiel, partiellement-séquentiel ou totalement parallèle des 8 opérations qui le composent.

instructions VLIW. Nous commençons par “étirer” le graphe de dépendance de manière à supprimer les zones hachurées pour lesquelles nous avons des conflits de ressources (une occurrence d’unité fonctionnelle par côté de l’architecture et par cycle ou hauteur de sous-graphe). Nous obtenons ainsi le graphe de la Figure3-13.

Figure 3-13. Résolution des conflits d’unités du graphe de dépendance

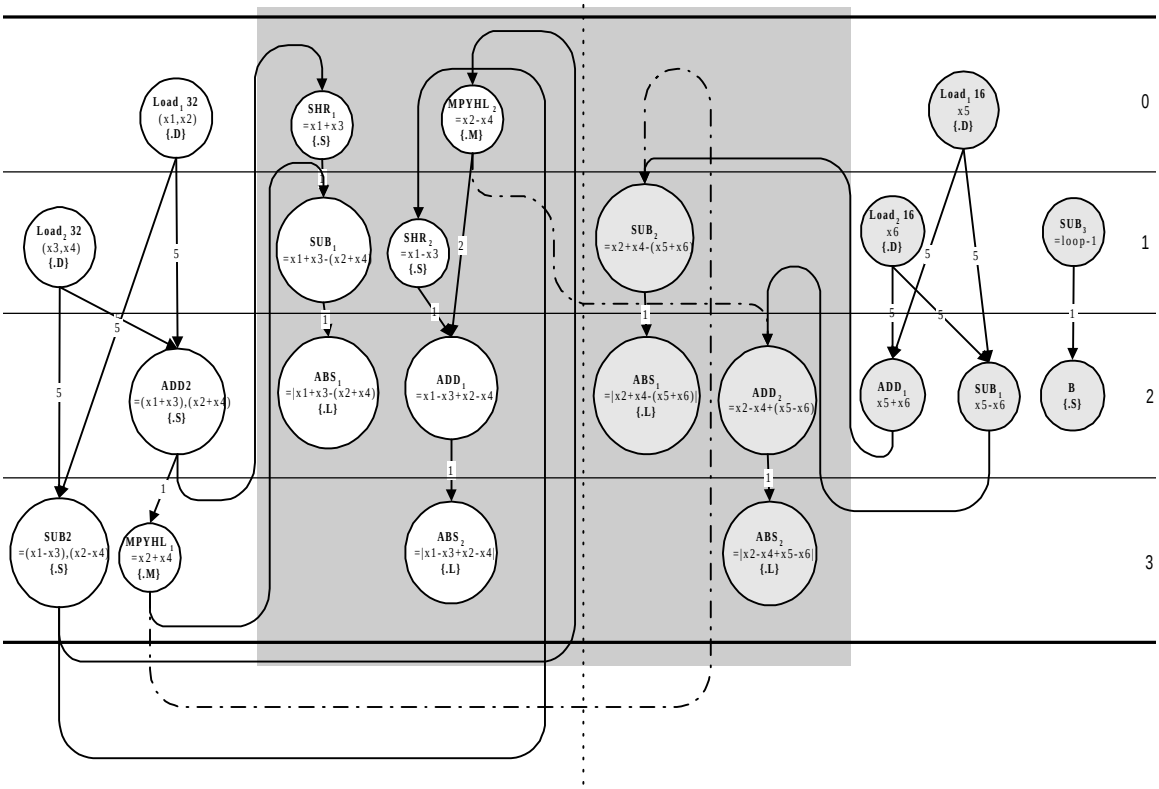


Bien que l'ensemble des nœuds de ce graphe n'ait pas d'unités assignées, il correspond à une première projection fonctionnelle du graphe de dépendance sur l'architecture. Partant du principe que les ressources registres sont en nombre suffisant et en assignant les unités fonctionnelles disponibles aux différents nœuds *ADD* et *SUB* (*.D, .S* ou *.L*) par niveau du graphe, nous obtenons bien un code valide tant du point de vue des ressources utilisées (registres/crosspaths) que de l'ordonnancement qui nécessite 14 cycles. Soulignons par ailleurs que cette projection correspond à un premier niveau d'optimisation dans la mesure où nous désynchronisons l'exécution des nœuds en ordonnançant certaines opérations pendant les cycles de latence des opérations précédentes. C'est le cas pour le nœud SHR_1 que nous exécutons pendant le cycle de latence de l'opération $MPYHL_1$ (qui s'exécute en 2 cycles d'après le tableau 3-12).

L'ensemble des nœuds que nous recontrons à chaque hauteur du graphe définit les opérations parallèles qui constituent chaque instruction VLIW de la boucle (elles se "lisent" horizontalement). A ce stade de l'optimisation, ces instructions montrent une sous-utilisation des unités fonctionnelles avec une moyenne de 2,6 opérations par instruction VLIW.

A partir de ce dernier graphe fonctionnel, nous pouvons replier les nœuds tout en respectant les dépendances de données. Nous prenons la borne théorique des 4 cycles (ou instructions VLIW) comme nombre d'instructions VLIW à atteindre pour optimiser au mieux le coeur de boucle. La projection des opérations sur ces 4 instructions débute avec le choix que nous faisons d'assigner les opérations de chargement figurant aux hauteurs 0 et 1 du graphe sur les deux premières instructions VLIW. Avec une latence de cinq cycles, nous récupérons l'ensemble des données chargées au troisième cycle (instruction VLIW numéro deux) de l'itération suivante du coeur de boucle ((1+5) mod 4 = 2). De là, nous pouvons généraliser l'indice de l'instruction VLIW sur lequel sera projeté chaque opération du graphe. Si *n* représente la hauteur de l'opération dans ce graphe, sa position dans le coeur est donnée par *n* mod 4. En appliquant ce mécanisme du cycle zéro à onze, nous obtenons le graphe de la Figure 3-14.

Figure 3-14. Première étape du repliement du graphe de dépendance “Gradient” sur C62



Concernant les unités du côté B, soulignons que le graphe de dépendance nous offre la possibilité d’exécuter les opérations *ADD*$_1$/*SUB*$_1$ à différents instants. Ainsi, le nœud *ADD*$_1$ peut s’exécuter avec l’instruction VLIW numéro 2 (hauteur 6), 3 (hauteur 7) ou bien 0 (hauteur 8).

Ces alternatives d’ordonnancement combinées aux choix encore laissés concernant l’assignation des unités fonctionnelles pour les opérations *ADD* et *SUB* sont détaillées dans la colonne la plus à droite du tableau suivant. Pour chaque côté de l’architecture, ce tableau montre la projection des nœuds (colonnes de gauche) au fur et à mesure que les choix d’unités fonctionnelles ou d’instructions VLIW s’estompent avec le processus de repliement :

Table 3-13. Première étape du repliement du calcul du Gradient sur C62

Num. Instr. LIW	Unité	Côté A		Côté B		
		Nœuds projetés	Unités possibles pour les nœuds[8-11]	Nœuds projetés	Unités possibles pour les nœuds[8-11]	Unités possibles pour les nœuds[0-11]
0	.L					←ADD_1/SUB_1
	.S	**SHR_1**	⇐ SHR_1			←ADD_1/SUB_1
	.M	**$MPYHL_2$**	⇐ $MPYHL_2$			
	.D	$Load_1$ 32		$Load_1$ 16		
1	.L	**SUB_1**	⇐ SUB_1		←SUB_2	←SUB_1
	.S	SHR_2			←SUB_2	
	.M					
	.D	$Load_2$ 32		$Load_2$ 16		
2	.L	**ABS_1**	⇐ABS_1	**ABS_1**	⇐ ABS_1	←ADD_1/SUB_1
	.S	ADD2		B		
	.M					
	.D	**ADD_1**	⇐ ADD_1	**ADD_2**	⇐ADD_2	←ADD_1/SUB_1
3	.L	**ABS_2**	⇐ ABS_2	**ABS_2**	⇐ ABS_2	
	.S	SUB2				←ADD_1/SUB_1
	.M	$MPYHL_1$				
	.D					←ADD_1/SUB_1

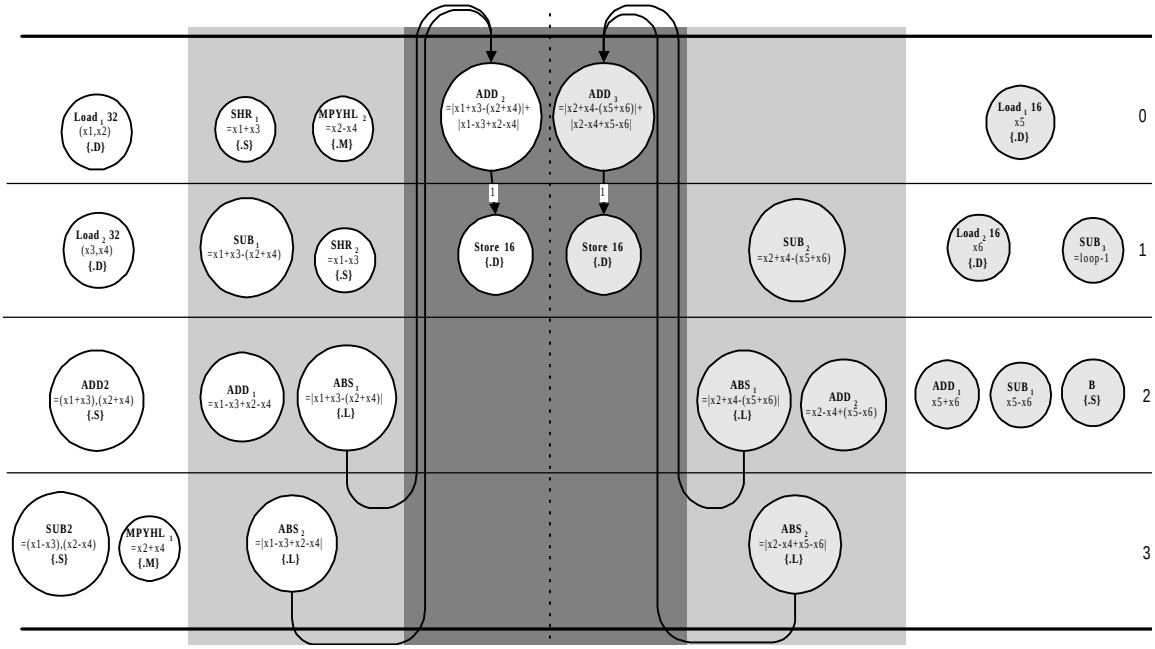
Si nous prenons l'exemple de la deuxième instruction VLIW (#1), nous remarquons que l'instruction SUB_1 du côté A doit être exécutée dans l'unité *.L* étant donné que les unités alternatives *.S* et *.D* sont respectivement utilisées pour l'exécution de l'opération de décalage (*SHR*) et celle de chargement (*Load*).

A l'inverse, les instructions ADD_1/SUB_1 du côté B ne sont pas encore assignées dans la mesure où elles peuvent s'exécuter avec différentes instructions VLIW et dans différentes unités. Pour ce côté, le choix des unités existe (colonne la plus à droite) avant même le repliement des opérations de hauteur 8 à 11 (colonne "[8-11]" du côté B). Après cette première étape, ces possibilités initiales subsistent comme pour les instructions 0 et 3 mais les choix sont restreints avec l'exemple de l'instruction #2 qui ne permet plus d'exécuter l'opération ADD_1 ou SUB_1 avec l'apparition de ABS_1 et ADD_2 occupant les unités *.L* et *.D*.

Par ailleurs, soulignons que l'instruction de branchement est positionnée au niveau de l'instruction VLIW numéro 2, ce qui, compte tenu de la latence de 6 cycles de ce nœud, a pour but de valider le branchement à la fin de l'itération suivante du coeur de boucle.

La Figure 3-15 montre la dernière étape du repliement pour les hauteurs 12-13 du graphe :

Figure 3-15. Deuxième étape du repliement du graphe “Gradient” sur C62



Cette étape met en avant des conflits de ressources au niveau des opérations de mémorisation des résultats. En effet, le tableau suivant souligne que l’unité *.D* de la deuxième instruction VLIW se trouve déjà employée. Pour remédier à ce problème, il suffit de retarder les opérations de mémorisation au niveau des premières unités *.D* que nous rencontrons libres (ce qui signifie que nous augmentons la durée de vie des registres résultats).

Table 3-14. Deuxième étape du repliement du calcul du Gradient sur C62

Num. Instr. LIW	Unité	Côté A		Côté B		
		Nœuds projetés	Unités possibles pour les nœuds [12-13]	Nœuds projetés	Unités possibles pour les nœuds[8-13]	Unités possibles pour les nœuds [0-11]
0	.L	**ADD_2**	⇐ ADD_2	**ADD_3**	←*ADD_3*	←*ADD_1/SUB_1*
	.S	SHR_1		**ADD_1**	←*ADD_3*	←*ADD_1/SUB_1*
	.M	$MPYHL_2$				
	.D	$Load_1$ 32		$Load_1$ 16		
1	.L	SUB_1		**SUB_2**	← *SUB_2/SUB_3*	←*SUB_1*
	.S	SHR_2		**SUB_3**	← *SUB_2/SUB_3*	
	.M					
	.D	$Load_2$ 32	⇐ Store 16	$Load_2$ 16	⇐ Store 16	
2	.L	ABS_1		ABS_1		
	.S	ADD2		B		
	.M					
	.D	ADD_1		ADD_2		
3	.L	ABS_2		ABS_2		
	.S	SUB2		**SUB_1**		←*ADD_1/SUB_1*
	.M	$MPYHL_1$				
	.D	**Store 16**		**Store 16**		

Le tableau 3-14 montre que nous pouvons déporter les 2 opérations de sauvegarde des valeurs du gradient au niveau de la dernière instruction VLIW qui possède ses deux unités *.D* libres.

Nous assignons ensuite les dernières unités fonctionnelles aux opérations non encore affectées en vérifiant la cohérence de l'utilisation des ressources. Cette étape nous donne une configuration possible de l'ordonnancement des opérations pour les côtés A et B. Pour terminer l'implantation, il convient enfin d'assigner les registres aux différentes variables et de vérifier qu'ils sont en nombre suffisant pour notre coeur de boucle optimisé. Nous ne détaillerons pas cette étape et, par souci de clarté, ne développerons pas le prologue et l'épilogue servant à initialiser le lancement des différentes opérations du coeur.

Pour conclure sur cette implantation, nous pouvons remarquer que le pipeline de ce coeur traite trois "parties" de la boucle initiale en parallèle et permet de passer d'un graphe ayant une durée d'exécution de 14 cycles à un coeur n'en nécessitant plus que 4 (le taux d'utilisation moyen des unités passe de 2,6/8 à plus de 6/8). Sur cet exemple, la technique du pipeline logiciel offre donc un facteur d'accélération de 4 sur le code que nous avions déjà optimisé avec l'utilisation des capacités de traitement SIMD.

3.3.2.2.2 Implantation du lisseur

Ce paragraphe s'intéresse à l'implantation du lisseur du deuxième ordre à l'aide de la combinaison de la méthode du pipeline logiciel et de celle du déroulage de boucle. A l'inverse du C80, nous nous concentrons sur l'implantation du filtre de deuxième ordre en précision fixe dans la mesure où les ressources en présence nous permettent une implantation efficace de cette version du lisseur.

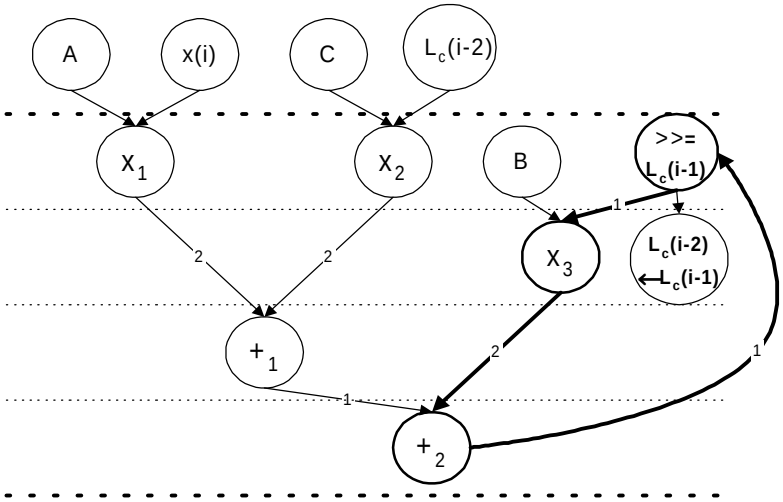
La structure récursive de ce filtre fait apparaître un graphe de dépendance cyclique qui se traduit par l'existence d'un intervalle minimum d'itérations (IMI) élevé. Ainsi, en reprenant la structure du filtre :

$$L_c(i) = A \cdot x(i) + B \cdot L_c(i-1) + C \cdot L_c(i-2)$$

$$A = (1 - e^{-\alpha})^2, B = 2e^{-\alpha}, C = e^{-2\alpha}$$

nous obtenons le graphe de dépendance suivant (où ">>" correspond au décalage à droite permettant la mise à l'échelle de la multiplication en précision fixe) :

Figure 3-16. Graphe de dépendance du lisseur de 2ème ordre sur C62

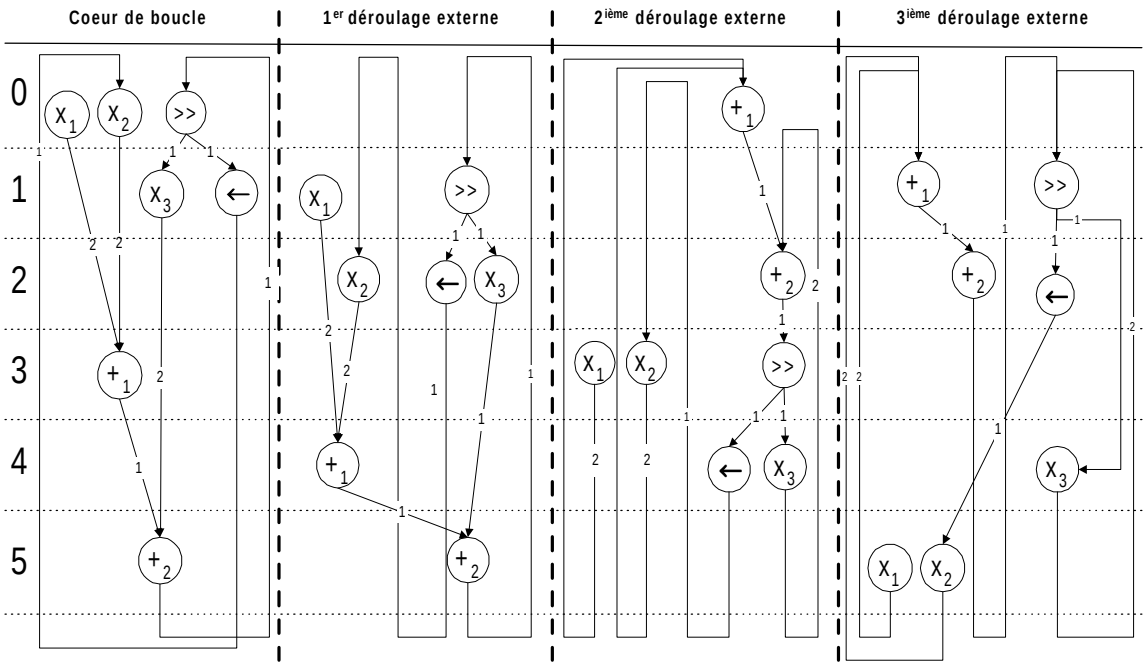


Sur cette figure, nous nous apercevons que la dépendance cyclique à droite du graphe empêche a priori de lancer une autre itération de calcul en parallèle. En repliant le graphe de dépendance, au mieux, nous avons ainsi *IMI = 4* avec une nette sous-utilisation des unités. Pour pouvoir augmenter les performances dans ce cas de figure, nous choisissons d'introduire le déroulage de boucle.

Dans [87], nous avions proposé d'augmenter la durée de l'IMI en modifiant la structure de l'équation récursive de manière à repousser la dépendance *L_c(i-1)* et de permettre l'exécution de plusieurs itérations de boucles en parallèle suivant un déroulage de boucle interne de quatre itérations "pipelinées" (approche qui est notamment intéressante dans le cas 1D). Cette combinaison de techniques permet alors d'effectuer 4 itérations de lisseur en seulement 8 cycles soit deux cycles par pixel.

Ici, nous proposons une technique encore plus performante qui s'appuie sur le déroulage externe de la boucle de traitement. Ce déroulage repose sur l'application du lisseur horizontal sur plusieurs lignes d'image en parallèle ce qui supprime ainsi toute dépendance entre les itérations parallèles de la boucle et permet l'entrelacement efficace des différentes opérations pipelinées. Nous proposons d'exécuter quatre itérations de boucle en parallèle en 6 cycles, soit 1,5 cycle par pixel. Cet objectif correspond à l'utilisation maximale théorique du nombre de multiplications disponibles. En effet, étant donné que trois multiplications sont nécessaires par itération de boucle, en traitant 4 itérations externes en parallèle, nous avons au mieux besoin de 4×3/2 cycles. Paradoxalement, nous augmentons la durée de l'IMI tout en rationalisant au mieux l'utilisation des ressources. La figure suivante montre un ordonnancement possible des opérations de ce coeur de boucle pipeliné. Pour alléger le graphique, seuls les nœuds des principales opérations sont ici détaillés :

Figure 3-17. Déroulage de boucle et pipeline logiciel du lisseur sur C6X



Sur la base de ce schéma qui montre bien deux multiplications par cycle, nous proposons une solution pour le partitionnement des différentes opérations sur les unités fonctionnelles. Nous devons intégrer les opérations d'accès à la mémoire interne

(chargement (LD) et mémorisation (ST)) et énumérer le nombre de "crosspath" par coté de l'architecture qui sont utilisés pour chacune des six instructions VLIW. Pour chaque instruction VLIW, le tableau suivant résume l'affectation des opérations parallèles ('||') aux différentes unités :

Table 3-15. Projection du graphe sur les ressources processeur

	Ressources VLIW par instruction →				Cross path
	Coeur	**1er déroulage**	**2ème déroulage**	**3ème déroulage**	
0	**$\times_1$.MA || $\times_2$.MB || >>.SA**		**$+_1$.LB***	**LD.DA || B.SB**	**0+1**
1	**LD.DA || $\times_3$.MA || ←.LB***	**$\times_2$.MB || >>.SB**	**$+_2$.DB**	**$+_1$.LA* || >>.SA**	**1+1**
2	**ST.DA**	**LD.DB || $\times_2$.MB || $\times_3$.MA* || ←.LB**		**←.SB***	**1+1**
3	**$+_1$.SA***	**ST.DB**	**$\times_1$.MA || $\times_2$.MB || >>.SB**	**$+_2$.LA || ST.DA**	**1+0**
4		**$+_1$.SB**	**LD.DA || $\times_3$.MB || ←.LB**	**$\times_3$.MA**	**0**
5	**$+_2$.SA**	**$+_2$.SB***	**ST.DB**	**$\times_1$.MA || $\times_2$.MB**	**0+1**

Chacune des opérations mettant en œuvre un "*crosspath*" (c-à-d, un opérande mis à jour par le côté opposé de l'architecture de celui dans lequel s'exécute l'opération) se trouve marquée d'une étoile. Ce tableau nous permet de vérifier que les nombres d'unités fonctionnelles et de "*crosspath*" employés sont conformes aux ressources disponibles (1 "*crosspath*" par côté par instruction VLIW) et nous permet ainsi de valider l'ordonnancement proposé.

Nous vérifions également, sans le détailler, que le nombre de ressources registres (32 au total) est suffisant pour implanter ce coeur de six cycles. 38 cycles composent alors le coeur de boucle, ce qui porte l'utilisation des unités fonctionnelles a près de 60% contre 35% sans le déroulage du coeur initial pipeliné.

Pour le nœud de seuillage, nous appliquons la même technique du déroulage externe avant l'étape du pipeline logiciel. Avec un déroulage externe de quatre itérations, nous réduisons l'impact du seuillage à un cycle par pixel 16 bits (contre deux sans déroulage). Le coeur compte alors cinq cycles pour une utilisation de près de 70% des unités.

3.3.2.2.3 Perspectives de performances avec le C62

Le tableau suivant résume les très bonnes performances de la chaîne sans l'impact de la gestion des transferts :

Figure 3-18. Estimation des performances brutes sur le C6202

	Cycles par pixel	Nbr. de passes	Performance
FGL 2^{ième} ordre	**6/4**	**4**	**6×512^2**
Gradient	**4/2**	**1**	**2×512^2**
Seuillage	**5/5**	**1**	**512^2**
		Nbr. total de cycles	**2359296**
Durée de traitement théorique à 250 MHz (sans les transferts)			**9,5 ms**

Ici, bien que nous n'ayons pas programmé les transferts DMA et donc mesuré précisément le coût des entrées/sorties à l'aide du simulateur ou d'une solution matérielle, nous pouvons estimer que cet impact est réduit au regard de la durée des calculs. En effet, si nous partitionnons les traitements de la même manière que pour le C80, nous avons la première passe qui lisse verticalement l'image à une cadence de 3 cycles par pixel 16 bits alors que la deuxième passe (lisseur horizontal→gradient→seuillage) affiche une performance de 6 cycles par pixel 16 bits. Avec une bande passante de 32 bits par cycle (si la mémoire interfacée est de type statique ou dynamique synchrone), la capacité d'entrée et de sortie du DMA est d'un pixel 16 bits par cycle. Ce calcul montre donc qu'il existe respectivement un facteur 3 et 6 entre la vitesse de transfert parallèle et celle du traitement de la première et deuxième chaîne de l'algorithme.

Par ailleurs, disposant d'un cache d'instructions d'une grande capacité (8K instructions VLIW), l'impact de la gestion du cache d'instructions est également négligeable. Dans le même sens, la quantité de requêtes DMA nécessaires est considérablement réduite avec la possibilité de cacher 64K données 16 bits, c'est-à-dire, 32 Ko dans le cadre du double buffering et 21 Ko pour le triple. Avec le double buffering, le traitement d'une image 512^2 n'implique ainsi qu'une dizaine de requêtes DMA.

L'ensemble de ces éléments nous incite à penser que les performances brutes proposées pour l'algorithme de FGL sur C6202 doivent s'avérer précises par opposition à ce que nous rencontrons pour le C80.

3.3.3 Conclusions

Cette implantation de la détection de contours nous permet de tirer deux types de conclusions. Elles sont d'ordre qualitatif et quantitatif.

Qualitativement, nous avons montré l'intérêt des techniques d'optimisation pour l'implantation des nœuds. Leur mise en œuvre sur l'architecture du C80 et celle du C62 souligne en effet que, malgré des différences architecturales importantes, ces méthodes génériques permettent d'augmenter significativement les performances. Dans le cas du C80, nous avons également montré l'adéquation de notre méthodologie en faveur d'une gestion des flux optimisée. La mesure de l'impact des caches d'instructions sur les performances globales nous a permis de souligner que la recherche d'une diminution de la granularité des nœuds apparaît comme un critère important.

Si nous comparons le C80 et le C62 sur le plan logiciel, nous pouvons souligner la difficulté de mise en œuvre de l'assembleur VLIW PP qui repose largement sur des mécanismes complexes d'opérations pré-câblées mixant des calculs d'ordre arithmétique et logique (exemple du calcul de la valeur absolue). Par opposition, l'architecture du C62 s'appuie sur des opérations élémentaires qui ont la particularité de s'exécuter dans un pipeline profond. Ici, le pipeline logiciel apparaît comme une technique plus facilement automatisable grâce à la simplicité des opérations de base même si sa mise en œuvre manuelle reste une étape complexe.

Quantitativement parlant, nous montrons l'adéquation de notre modèle de performance pour le C80 lorsque le coût de la gestion des caches d'instructions est mesuré. Par ailleurs, les performances obtenues sur C80 montrent des durées de traitement comparables à celles obtenues sur les processeurs RISC comme nous le présentons dans [88]. En outre, nous pouvons remarquer qu'au niveau des performances brutes (hors transfert), la capacité de traitement SIMD du C80 pour le nœud gradient (0.75 cycle par pixel (c/p)) et le nœud seuillage (0,25 c/p) permet d'égaler les performances de ces mêmes nœuds sur C6202 (2 c/p et 1 c/p) et ce d'après le rapport de fréquences 60MHz/250MHz des processeurs. Pour le lisseur, les performances tournent en faveur du C62 (pour lequel nous avons

implanté l'équation moins complexe du deuxième ordre) et permet d'atteindre des durées de traitement comparables à celle des meilleures solutions FPGA du moment. Par comparaison, dans [84] l'équipe de D. Demigny atteint une cadence de traitement de 40 ms pour une image 1024^2 sur la solution "Ardoise" à base de FPGA.

3.4 Conclusions

Dans ce chapitre, nous avons tout d'abord présenté la structure d'une librairie originale de traitement d'images sur C80 qui met en œuvre les principes théoriques pour la gestion des flux développés au chapitre 2. La généricité de cette librairie qui repose sur la notion unifiée de patrons de données nous a permis d'implanter un triplet de chaînes élémentaires constituées de nœuds extraits d'un ensemble de 60 opérateurs réalisés en moins de 6 mois (l'ensemble de la librairie compte environ 25.000 lignes de programme C et d'assembleur). Ce niveau de productivité s'explique grâce au formalisme unique de modélisation ainsi qu'à notre infrastructure logicielle qui s'avère véritablement fonctionnelle et ouverte.

De plus, nous avons montré la grande flexibilité qu'offre notre librairie capable de paralléliser le traitement d'images de taille variable avec un nombre de processeurs paramétrables. Nous avons aussi atteint l'objectif visant à permettre la reconfiguration dynamique des traitements avec une implantation optimisée des mécanismes de partitionnement et de génération des requêtes DMA synchrones intervenant pendant l'exécution des programmes incorporés dans le modèle de performance.

A l'aide des trois chaînes élémentaires évoquées, nous avons ensuite donné une première validation de notre modèle de performance pour le C80 et montré l'intérêt qu'offre le chaînage des opérateurs. Nous avons poursuivi l'étude de la validité de notre modèle avec l'application de la détection de contours en analysant plus particulièrement l'impact de la gestion des caches d'instructions.

Avec cette dernière application, nous avons également présenté une mise en œuvre des techniques d'optimisation introduites au point 2.1 pour l'implantation des opérateurs et poursuivi l'étude sur l'application de ces méthodes à la nouvelle architecture du C62. Sur le plan des résultats, ce travail nous a permis de montrer l'importance de ces techniques et nous a conduit à souligner les très bonnes performances obtenues pour cet algorithme de référence dans le monde académique.

4 Implantation sur C80 d'une technique de compression vidéo originale basée sur la détection de mouvement

Après avoir validé les outils et les méthodologies de développement dans le précédent chapitre, nous présentons ici l'implantation d'un algorithme plus complet qui vise la compression de séquences vidéo à l'aide d'une technique récente de détection de mouvement.

Après une brève analyse de l'état de l'art des méthodes de segmentation de mouvement, nous rappellerons, dans un premier temps, l'algorithme de détection de mouvement retenu et présenterons son implantation sur C80 grâce aux méthodologies de développement exposées au chapitre 2. Ensuite, nous montrerons comment cet algorithme permet d'améliorer une librairie de compression/décompression JPEG existante pour le TMS320C80 vers le codage des séquences vidéo. Plus précisément, deux approches seront exposées et comparées grâce à un démonstrateur fonctionnel intégrant les deux méthodes et pour lequel nous détaillerons l'implantation de notre technique de compression novatrice.

4.1 Etat de l'art des techniques pour la détection/ compression d'images en mouvement

La problématique de la détection du mouvement s'associe à une classe d'algorithmes aux applications très variées. Nous pouvons citer des débouchés pour la vidéo surveillance [89], le suivi de cibles [90] ou l'indexation vidéo [91]. Plus récemment, l'avènement de la norme MPEG-4 a accru l'intérêt de cette classe d'algorithmes pour la segmentation des objets en

mouvement afin de permettre une définition hiérarchique des éléments qui composent la scène et par là-même, de rendre possible la re-composition interactive (spatiale et contextuelle) de la séquence par le décodeur [92]. L'objectif est double puisque cette approche permet également d'obtenir des taux de compression accrus. D'une manière moins ambitieuse et plus traditionnelle, dans le domaine de la compression, la détection qui sera mise en œuvre vise à réduire l'entropie globale du codage de séquences. Cet objectif s'associe initialement à la réalisation d'un système de compression à très bas débit (de 8 à 16 Ko/s) pour une architecture temps réel embarquée de vidéo-surveillance.

Nous trouvons dans la littérature un vaste éventail de techniques pour la segmentation et l'estimation des paramètres du mouvement qui s'analysent comme des problématiques complémentaires mais distinctes. Dans [93], l'auteur présente une classification par niveaux d'abstraction (du bas au haut niveau) de diverses approches pour la segmentation et l'estimation des paramètres du mouvement. Dans [94], nous trouvons également un bon aperçu de quelques techniques du bas niveau pour le calcul des paramètres (plus particulièrement pour le domaine de la compression vidéo) avec des approches par points (calcul des flots optiques), par blocs (comme pour la corrélation 2D des normes MPEG-1/2) ou plus récemment par régions ou contours actifs (autour de la norme MPEG-4). Ici, nous nous intéressons plus spécifiquement à une technique récente reposant sur la segmentation robuste des pixels en mouvement pour des séquences à fond fixe, méthode mise au point par Alice Caplier [95] et capable de supporter des variations d'éclairage et d'atténuer les effets de traînée. Par ailleurs, cet algorithme correspond à l'approche privilégiée par Lionel Lacassagne dans sa thèse sur la détection et le suivi de cibles [1]. Les recherches de ce mémoire étant complémentaires aux travaux de L. Lacassagne, nous renvoyons à sa thèse pour une comparaison plus précise des méthodes constituant l'état de l'art de la segmentation du mouvement. Par ailleurs, nous pouvons citer les articles [96][97][98][99][100][101] comme références récentes sur l'implantation d'algorithmes pour l'analyse d'images sur le processeur C80. La multiplicité des publications souligne ici la bonne adéquation du C80 pour ce type d'applications.

4.2 La détection de mouvement par modélisation Markovienne

Dans cette partie, nous rappelons la terminologie (tirée de [95]) ainsi que les fondements théoriques de la modélisation par champ de Markov qui repose, d'un point de vue statistique, sur l'estimation bayesienne d'un ensemble d'informations contextuelles vers une prise de décision localisée a priori. Dans ce formalisme, le processus de décision dépend non seulement des données disponibles en un point, mais également des informations issues d'un modèle de voisinage qui peut être spatial ou temporel. La localité des données nécessaires à la prise de décision offre ainsi l'avantage d'être parallélisable pour des traitements en temps-réel et cadre donc parfaitement avec les objectifs de nos travaux. Par ailleurs, nous soulignons que si ce formalisme s'appuie sur des théorèmes statistiques consistants, il se définit néanmoins comme une méthodologie très générale pour la résolution de problèmes divers. Dans ce contexte, une modélisation locale des solutions doit être mise en œuvre de manière spécifique à chaque domaine d'applications. Nous trouvons, dans [95][77], les références de problèmes relevant du traitement d'images et dont la résolution est modélisée avec cette approche. Par la suite, nous employons les notations suivantes afin de modéliser la problématique de la détection de mouvement :

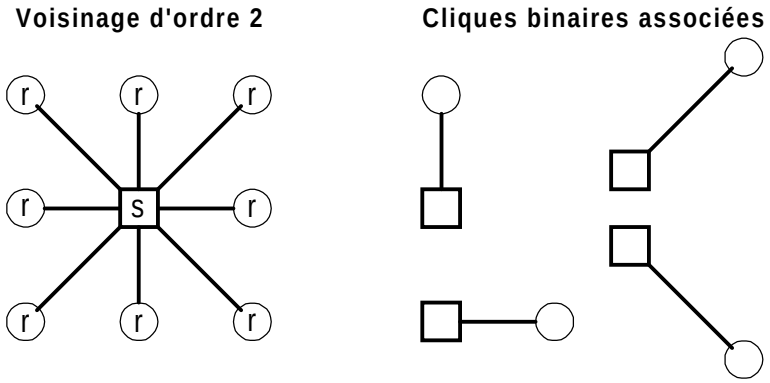
Table 4-1. Notations pour la détection de mouvement par modélisation markovienne

Nom du symbole	Signification
S	Ensemble des sites d'une image
s=(x,y)	Un site de l'image aux coordonnées (x,y)
$E = \{e(s), s \in S\}$	Le champ d'étiquette
R	L'ensemble des réalisations possibles du champ E
r	Un site appartenant au voisinage d'un point
e	Une réalisation particulière du champ E
$O = \{o(s), s \in S\}$	Le champ des observation
o	Une réalisation particulière du champ O
e(s), o(s)	Valeurs des champs E et O au point s(x,y)

4.2.1 Champs de Markov

Un champ de Markov se définit comme un champ d'étiquettes continues ou discrètes qui permettent de décrire certains paramètres d'une image ou d'une séquence. Ce champ se caractérise en fonction d'un système de voisinage η qui se décompose en cliques correspondant aux configurations spatiales élémentaires qui définissent l'ensemble des relations du voisinage[1]. Par la suite, nous nous intéressons au voisinage d'ordre 2 et à ses cliques binaires que nous décrivons dans la Figure4-1 :

Figure 4-1. Voisinage d'ordre 2 et cliques associées



De là, nous donnons la définition d'un champ de Markov :

E est un champ de Markov relativement au voisinage η s'il vérifie les propriétés suivantes avec Pr[X] désignant la probabilité de l'événement X :

1. $\forall e \in R, Pr[E = e] > 0$
2. $Pr[E(s) = e(s) / E(r) = e(r), r \neq s, r \in S] = Pr[E(s) = e(s) / E(r) = e(r), r \in \eta]$

Nous constatons que l'égalité de probabilités conditionnelles de la deuxième expression traduit bien le fait qu'ici, le choix d'une étiquette dépend seulement du voisinage plutôt que de l'ensemble des points d'une image. Dans ce contexte, le théorème de Hammersley-Clifford permet d'établir explicitement la probabilité à priori $Pr[E = e]$:

1. Nous trouvons dans [95][77] une définition formelle de la clique

Equation 4-1. Probabilité de réalisation d'une configuration d'un champ markovien

$$Pr[E = e] = \frac{1}{Z} \cdot \exp(-U_m(e))$$

où *Z* est une constante de normalisation qui s'écrit :

Equation 4-2. Constante de normalisation du théorème de Hammersley-Clifford

$$Z = \sum_{e \in R} \exp(-U_m(e))$$

et $U_m(e)$, que nous appelons fonction d'énergie du modèle, prend la forme suivante, avec C désignant l'ensemble des cliques de l'image :

Equation 4-3. Définition de la fonction d'énergie du champ markovien

$$U_m(e) = \sum_{c \in C} V_c(e)$$

Ici, $V_c(e)$ qui apparaît comme une fonction potentiel associée au système de cliques, est spécifique au problème à résoudre. Nous pouvons citer l'exemple du critère d'homogénéité spatiale du champ des étiquettes dont une traduction, au niveau de la fonction potentiel, peut sécrire :

Equation 4-4. Exemple de fonction potentiel

$$V_c(e(s),e(r)) = \begin{cases} -\beta \text{ si } e(s) = e(r) \\ \beta \text{ si } e(s) \neq e(r) \end{cases}, \text{ avec } \beta > 0$$

4.2.2 Champs de Markov en traitement d'images

Dans le domaine du traitement d'images, la modélisation markovienne permet de traiter des problèmes locaux (bas niveau). Nous définissons *o(s)* comme une observation de l'image au site *s* et *e(s)* une étiquette associée au problème à résoudre. A partir des observations locales *o(s)*, nous utilisons généralement un estimateur du maximum a posteriori (MAP) pour déterminer une configuration d'étiquettes qui corresponde à une solution robuste du problème. Statistiquement, nous écrivons la probabilité conditionnelle suivante :

Equation 4-5. Estimateur MAP

$$\max_{e \in R} (Pr[E = e / O = o])$$

Le théorème de Bayes nous permet d'inverser la dépendance conditionnelle et ainsi d'écrire, à une constante de normalisation près (dans la mesure où *Pr[O = o]* est une constante du problème) :

Equation 4-6. Estimateur MAP et théorème de Bayes

$$\max_{e \in R} (Pr[E = e / O = o]) \Leftrightarrow \max_{e \in R} (Pr[E = e] \cdot Pr[O = o / E = e])$$

Ici, le terme *Pr[E = e]* se définit grâce à la modélisation markovienne du champ E et la probabilité conditionnelle *Pr[O = o / E = e]* est issue de la modélisation statistique entre les observations et les étiquettes. Si *n* définit un bruit blanc gaussien de variance σ^2, *o* s'écrit alors, suivant une hypothèse supplémentaire par rapport au théorème de Bayes, par :

Equation 4-7. Modèle statistique entre l'observation et le champ d'étiquettes

$$o = \psi(e) + n$$

La distribution de Gauss associée à cette modélisation nous permet d'écrire :

Equation 4-8. Probabilité de l'estimateur MAP

$$Pr[O = o / E = e] = \prod_{s \in S} \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left(-\frac{[o(s) - \psi(e(s))]^2}{2\sigma^2}\right)$$

En reprenant la probabilité conditionnelle initiale, nous écrivons finalement l'égalité proportionnelle suivante :

Equation 4-9. Probabilité d'un champ markovien suivant l'estimateur MAP

$$Pr[E = e / O = o] \propto \frac{1}{Z} \cdot \exp(-U_m(e)) \cdot \prod_{s \in S} \frac{1}{\sqrt{2\pi\sigma^2}} \cdot \exp\left(-\frac{[o(s) - \psi(e(s))]^2}{2\sigma^2}\right)$$

En appliquant la fonction logarithme aux membres de l'égalité précédente, apparaissent intuitivement les deux termes $U_m(e)$ et $U_a(o/e)$:

Equation 4-10. Fonction d'énergie de l'estimateur MAP d'un champ Markovien

$$\ln(Pr[E = e / O = o]) \propto U = -U_m(e) - U_a(o/e)$$

où $U_a(o/e)$ se définit par :

Equation 4-11. Terme d'attache aux données

$$U_a(o/e) = \frac{1}{2\sigma^2} \cdot \sum_{s \in S} [o(s) - \psi(e(s))]^2$$

Ainsi, la recherche du maximum de la probabilité a posteriori du champ d'étiquettes *E,* étant donné un champ d'observation *O,* revient à minimiser une fonction d'énergie constituée de deux termes :

- $U_m(e)$ qui définit l'énergie du modèle a priori et permet de spécifier un modèle de solution du problème par le biais des fonctions potentiel ;
- $U_a(o/e)$ que nous définissons comme l'énergie d'adéquation aux données et qui procure une mesure (distance) de l'adéquation entre l'observation et la solution du problème (il s'agit du terme d'attache aux données).

Cette fonction d'énergie n'étant généralement pas convexe, la recherche de son minimum avec des techniques classiques de descente de gradient ne s'avère pas applicable. Nous rencontrons, en revanche, un éventail de techniques déterministes ou stochastiques plus ou moins lourdes en charge de calculs dont un aperçu est donné dans [77].

Les méthodes déterministes qui convergent vers un minimum local semblent particulièrement attrayantes pour diminuer la masse des calculs, mais elles nécessitent une bonne initialisation du champ des étiquettes de manière à "guider" la solution pour éviter une mauvaise convergence du processus de relaxation. L'algorithme déterministe de l'ICM (*Iterated Conditional Modes*) proposé dans [102] minimise, de manière locale, l'énergie du modèle et permet ainsi de réduire considérablement la charge de calcul. Par la suite, nous utiliserons donc cette méthode de relaxation qui affiche, avec le modèle d'énergie développé dans [95], une bonne qualité de détection.

4.2.3 Un Modèle de fonction potentiel pour la détection de mouvement

Alice Caplier [95], s'inspirant des travaux de Lalande [102], développe un modèle de fonction potentiel pour la détection du mouvement qui combine l'information spatiale et temporelle. Elle propose un unique paramètre d'observation associé au champ O_t qui caractérise les variations temporelles de luminance :

Equation 4-12. Paramètre du champ d'observation pour l'estimateur du mouvement

$$o_t(s) = \left| I_t(s) - I_{t-1}(s) \right|$$

où I_t/I_{t-1} sont deux images prises à différents instants d'une séquence. Pour éliminer la zone d'écho du mouvement, Lalande intègre l'information des données sur le passé, le présent et le futur. Avec le même objectif, Caplier modifie le modèle d'énergie de Lalande qui propose d'utiliser, comme étiquettes initiales du processus de relaxation, une première estimation des étiquettes calculées lors de la relaxation du précédent couple d'étiquettes conjointes (E_{t-1} (passé), E_t (présent)) suivant le principe d'une fenêtre temporelle glissante. Par opposition, l'originalité du modèle Caplier réside dans ce que l'initialisation des étiquettes présentes et futures se déduit directement de l'observation ce qui diminue la portée des paramètres estimés lors de la relaxation (qui ne concerne plus que les données présentes), et permet ainsi de réduire d'une manière significative la charge de calcul. L'initialisation des étiquettes se traduit alors par un simple seuillage (binarisation des observations) ou par un test de vraisemblance. Dans ce contexte, le critère d'énergie globale U à optimiser s'écrit alors, selon Caplier :

$$\max_{E(t)}(Pr[E_{t-1}, \hat{E}_t, \hat{E}_{t+1}, O_t]) \Leftrightarrow \min_{E(t)}(U(E_{t-1}, \hat{E}_t, \hat{E}_{t+1}, O_t))$$

où l'utilisation de la notation *"$\hat{E}$"* caractérise le fait que l'initialisation des étiquettes se déduit directement de l'observation plutôt que des précédentes relaxations des champs (ce qui n'est pas le cas pour le champ *E(t-1)*). En reprenant l'équation 4-10, cette énergie se décompose suivant 2 termes distincts :

Equation 4-13. Energie globale de la modélisation markovienne de détection de mouvement selon Caplier

$$U(E_{t-1}, \hat{E}_t, \hat{E}_{t+1}, O_t) = U_m(E_{t-1}, \hat{E}_t, \hat{E}_{t+1}) + U_a(O_t / \hat{E}_t)$$

avec la fonction U_m que nous subdivisons en 3 fonctions potentiel distinctes suivant la nature des cliques spatio-temporelles considérées :

Equation 4-14. Fonctions potentiels selon Caplier

$$U_m(E_{t-1}, \hat{E}_t, \hat{E}_{t+1}) = W_p(E_{t-1}, \hat{E}_t) + W_s(\hat{E}_t) + W_f(\hat{E}_t, \hat{E}_{t+1})$$

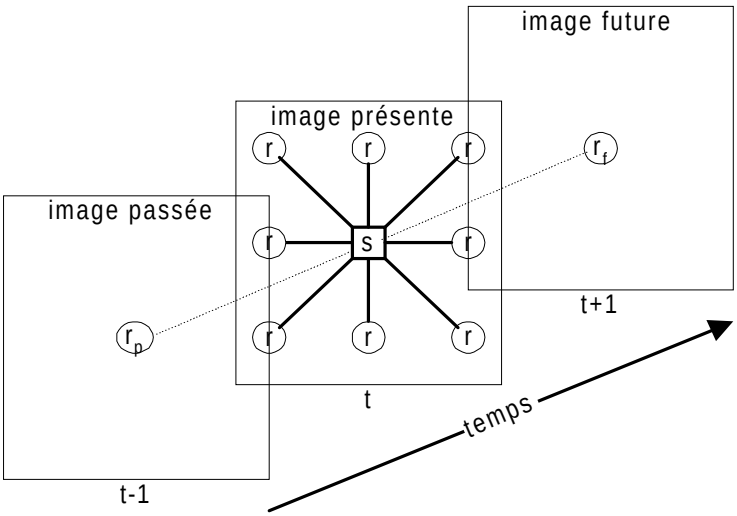
En reprenant le critère d'homogénéité, Caplier propose la définition suivante des termes W :

Table 4-2. Les fonctions potentiel du voisinage spatio-temporel

Nature des cliques	Terme de U_m associé	Définition de V_c
Temporelle (passé)	W_p	$V_c(e(s),e(r_p)) = \begin{cases} -\beta_p & \text{si } e(s) = e(r_p) \\ \beta_p & \text{sinon} \end{cases}$
Spatiale	W_s	$V_c(e(s),e(r)) = \begin{cases} -\beta_s & \text{si } e(s) = e(r) \\ \beta_s & \text{sinon} \end{cases}$
Temporelle (futur)	W_f	$V_c(e(s),e(r_f)) = \begin{cases} -\beta_f & \text{si } e(s) = e(r_f) \\ \beta_f & \text{sinon} \end{cases}$

Pour ce modèle, Caplier suggère l'utilisation du voisinage spatio-temporel détaillé dans la Figure 4-2 qui introduit un total de 10 cliques. Par ailleurs, pour prendre en compte les discontinuités temporelles des données du flux et permettre des changements de la carte du mouvement, Caplier propose de prendre $\beta_f > \beta_p$, ce qui autorise également une bonne élimination de la traînée du mouvement.

Figure 4-2. Modèle de voisinage et cliques spatio temporelles



En ce qui concerne le terme d'attache aux données $U_a(O_t/\hat{E}_t)$, nous reprenons l'équation 4-11 pour laquelle la modélisation du lien statistique entre les étiquettes et l'observation s'écrit :

Equation 4-15. Modèle d'attache aux données pour la détection de mouvement

$$o(s) = \psi(e(s)) + n \quad \text{avec} \quad \psi(e(s)) = \begin{cases} 0 & \text{si } e(\ ) = b \\ \alpha > 0 & \text{si } e(\ ) = a \end{cases}$$

où *n* représente un bruit blanc gaussien centré de variance σ^2. L'égalité *e(s) = b* est vraie lorsque l'étiquette appartient au fond fixe de l'image ayant *b=0* comme étiquette. Dans ce cas de figure, il n'y a pas de modification temporelle de l'intensité et l'observation est quasi nulle. A l'inverse, la valeur de α que nous intégrons lorsque du mouvement est détecté (*e(s) = a*), correspond à la valeur moyenne de l'observation pour les zones en mouvement d'étiquette *a=1*. Caplier propose de fixer cette dernière valeur ainsi que celle des coefficients β de U_m quelle que soit la nature des séquences traitées. A l'inverse, elle suggère une mise à jour régulière de la variance. Le tableau 4-3 résume les valeurs typiquement recommandées pour les différents paramètres de l'algorithme.

Table 4-3. Paramétrage de l'algorithme de détection markovienne du mouvement

	Valeurs des paramètres recommandées dans [95]
Seuillage du champ d'observation O_t	Valeur du Seuil : Fonction de la séquence (C.f. thèse Dumontier)
Nombre de passes d'ICM	4-5
β_p	10
β_s	20
β_f	30
α	10

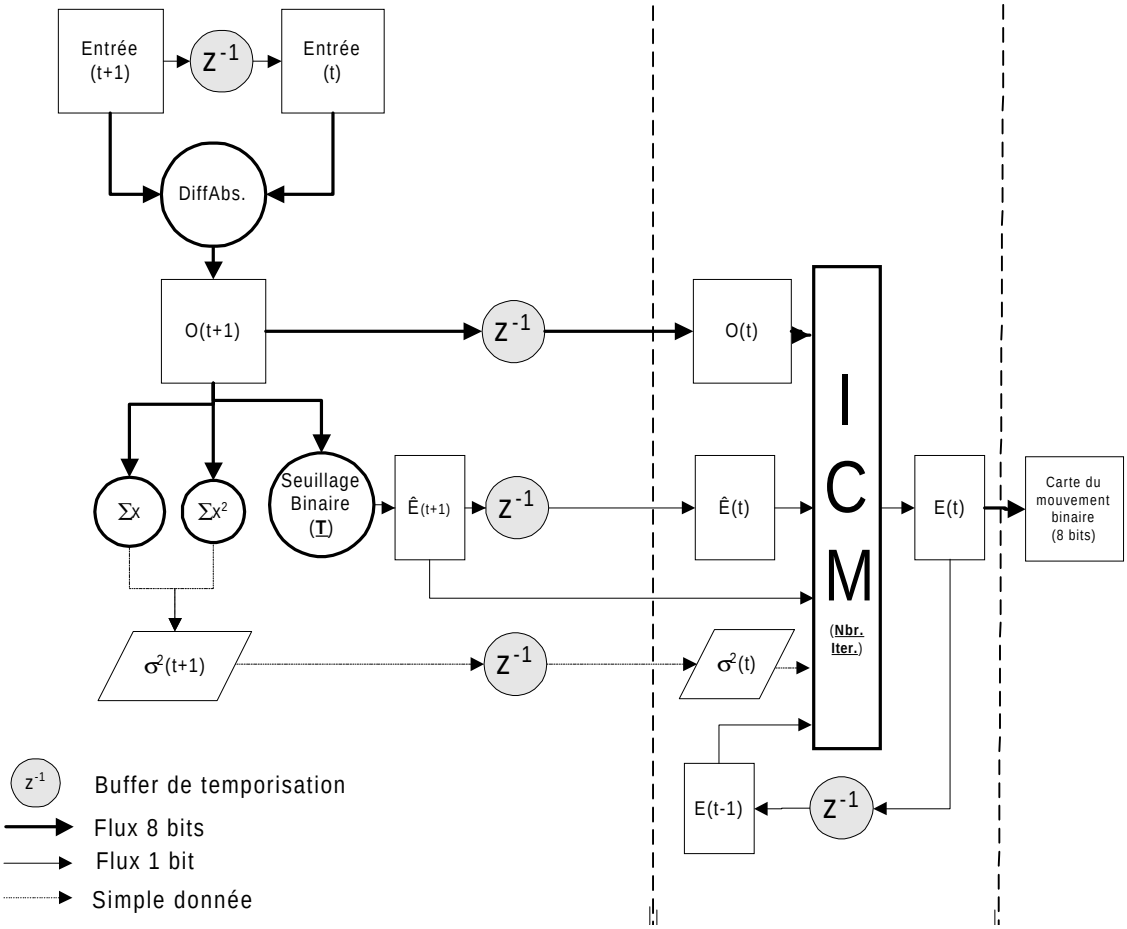
4.2.4 Conclusions

Plus qu'une simple soustraction d'images, l'algorithme que nous venons de détailler permet une détection de mouvement robuste dans la mesure où d'une part, il autorise des variations d'éclairage et d'autre part, il atténue la traînée du mouvement.

Cet algorithme se décompose en deux étapes distinctes. La première constitue la phase de pré-traitement avec l'initialisation du champ d'étiquettes à relaxer et le calcul de la variance. La deuxième étape regroupe l'algorithme de l'ICM pour lequel nous employons une mise à jour de l'état binaire des étiquettes (en mouvement ou pas) qui s'effectue site par site dans le sens où toute modification d'état est immédiatement prise en compte dans la

relaxation du site voisin, dans la direction de parcours de l'image. Nous pourrions envisager une modification ligne par ligne ou image par image mais cette mise à jour par site nécessite moins de buffers et simplifie l'implantation. Le critère d'arrêt pour la relaxation peut être défini soit par rapport à un nombre d'itérations de l'ICM (nombre de passes complètes de traitement portant sur l'ensemble de l'image), soit suivant le critère strict que plus aucun changement d'état n'intervient. Ce dernier test correspond à l'obtention d'un état stable qui traduit la convergence vers le premier minimum local ou global de la fonction d'énergie. Cette convergence est rapide et intervient essentiellement lors des toutes premières itérations (4 ou 5 passes suffisent généralement pour atteindre la stabilité). Par la suite, nous utiliserons le critère simple d'arrêt de la relaxation suivant un nombre figé d'itérations. Pour conclure, la Figure 4-3 schématise les nœuds de traitement impliqués pour chacune des étapes de la détection de mouvement ainsi que les dépendances de flux et les temporisations intermédiaires (schématisé par z^{-1}).

Figure 4-3. Approche de Caplier et al. pour la détection markovienne du mouvement



4.3 Implantation de la détection markovienne sur C80

L'implantation des deux phases de la détection markovienne du mouvement se définit suivant deux chaînes de traitement. Nous choisissons de partitionner ces deux chaînes selon le modèle SPMD et, à chaque étape, nous mobilisons l'ensemble des PP. Ces deux chaînes sont exécutées séquentiellement ce qui implique de reconfigurer dynamiquement le traitement.

4.3.1 Le pré-traitement

L'étape du pré-traitement se décompose en une chaîne de 4 nœuds distincts. Le premier nœud *DiffAbs* permet d'obtenir la valeur absolue de la différence d'images *t* et *t+1* et les nœuds Σx et Σx^2 nous permettent d'en calculer la variance. Enfin, le nœud *Seuillage binaire* permet l'initialisation des étiquettes du champ $\hat{E}_t$. Le tableau ci-dessous détaille la géométrie des patrons des nœuds et le nombre de cycles par pixel nécessaires à une itération du coeur de boucle :

Table 4-4. Géométrie et complexité des patrons pour la phase de pré-traitement

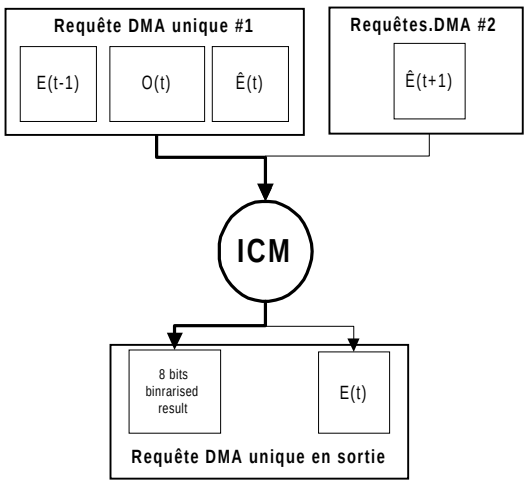
Nom du nœud	Géométrie des patrons (d_s=d_p=1)		Cycles/pixel (optimisations)
	Entrée $w_t \times h_t, w_s[\backslash w_r], h_s[\backslash h_r]$	Sortie $w_t \times h_t, w_s, h_s$	
DiffAbs	Image t & t+1: 16×1,8\0,1\0	16×1,8\0,1\0	7/8 (déroul. & pip. log. & SIMD)
Σx	2×1,1\0,1\0	-	1/1 (pip. logiciel)
Σx^2	6×1,2\0,1\0	-	3/2 (pip. logiciel)
Seuillage binaire	16×2,8\0,1\0	2×2,1\0,1\0	7/8 (déroul. & pip. log. & SIMD)

Pour cette étape de l'algorithme qui met en œuvre le double buffering interne, l'optimisation des différents nœuds de traitement permet d'atteindre une durée de traitement de 6 ms pour une image 256^2 avec quatre PP fonctionnant à 40 MHz.

4.3.2 L'ICM

L'implantation de la phase de relaxation s'appuie sur une chaîne constituée de l'unique nœud *ICM*. Ce nœud nécessite quatre buffers en entrée (*E(t-1), O(t), Ê(t), Ê(t+1)*) et produit le champ d'étiquettes relaxées *E(t)*. La carte du mouvement binaire calculée est en réalité constituée de deux buffers : le buffer d'étiquettes binaire et sa représentation 8 bits en noir et blanc afin de permettre la visualisation des zones de mouvement détectées. Afin de réduire le nombre de buffers nécessaires (ce qui se traduit par une quantité de requêtes DMA chaînées qui devient prohibitive sur cet exemple au regard de la quantité de mémoire interne limitée et du coût qu'engendrerait la gestion d'une telle liste), nous considérons le nœud ICM comme diadique. Dans la pratique, nous regroupons les buffers de manière à transférer différentes natures de données avec la même requête DMA. Ce principe que nous illustrons par la Figure 4-4, est alors associé à une juxtaposition physique des buffers externes.

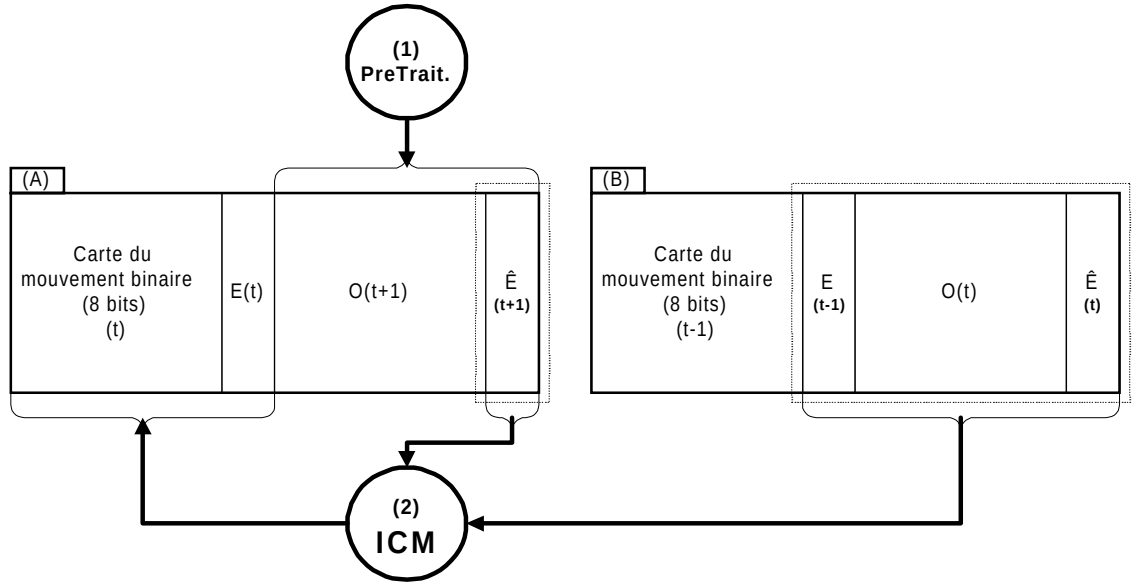
Figure 4-4. Regroupement des données au sein des transferts du nœud ICM



Cette organisation est décrite à l'aide de la Figure4-5 qui montre les buffers impliqués par les deux étapes de la détection. Par ailleurs, nous intégrons les temporisations z^{-1} de la Figure 4-3 en appliquant la technique du double buffering sur les buffers externes. Ainsi, nous alternons temporellement l'emplacement des différents buffers entre le jeu de buffers *A* et *B*. Notre librairie de gestion de flux est alors paramétrée suivant la définition de deux

buffers d'entrée et d'un en sortie. Dans la pratique, le découpage du buffer combinant *E(t-1)*, *O(t)* et *Ê(t)* est réalisé par rapport à la géométrie du buffer *O(t)* et du patron interne associé. Nous modifions ensuite manuellement le paramétrage du transfert pour qu'il englobe la combinaison des buffers. Cette approche est cohérente du point de vue de notre méthodologie de gestion de flux dans la mesure où le patron associé au buffer *O(t)* intègre par nature toutes les contraintes de taille des buffers des champs *E(t-1)* et *Ê(t)* qui ont une largeur diminuée d'un facteur 8 par rapport à *O(t)* (les étiquettes sont binaires).

Figure 4-5. Organisation externe des buffers



Le tableau suivant présente la géométrie des patrons du nœud ICM et illustre ainsi cette dernière remarque :

Table 4-5. Géométrie et complexité des patrons pour la phase ICM

Géométrie des patrons (d_s=d_p=1)		Cycles/pixel (optimisations)
Entrée $w_t \times h_t, w_s[\backslash w_r], h_s[\backslash h_r]$	Sortie $w_t \times h_t, w_s, h_s$	
O(t) : 32×3,32\8,1\2	E(t) : 4×1,4\0,1\0	40 (pipeline logiciel)
E(t-1) : 4×3,4\1,1\2		
Ê(t) : 4×3,4\1,1\2	E(t) 8 bits : 32×1,32\0,1\0	
Ê(t+1) : 4×1,4\1,1\0		

Nous constatons, par ailleurs, que la géométrie du patron associée au buffer *E(t-1)* implique le chargement d'un minimum de trois lignes alors que le modèle de voisinage spatio-temporel de la Figure 4-2 ne suggère qu'une ligne pour les données du passé (tel que pour le patron du nœud $\hat{E}(t+1)$). Cette géométrie découle du regroupement des buffers et induit une quantité de données transférées qui est légèrement abondée (la complexité calculatoire du nœud ICM occultant ce surcoût). Dans la pratique, la mise en œuvre du nœud ICM qui ne perçoit qu'un flux unique pour les données des buffers regroupés, se charge alors d'adresser les données d'intérêt (nous intégrons, au niveau du traitement, une notion de pas entre les lignes des données d'un même buffer). Le transfert des données est optimisé suivant le double buffering. En utilisant un banc de 2Ko, nous parvenons, pour une image 256^2 qui correspond au format d'image que nous ciblons, à appliquer l'algorithme en cachant quatre lignes de sites. Le nombre de lignes (L) que nous imposons pour le paramétrage du transfert par notre librairie, se calcule suivant la formule :

Equation 4-16. Nombre de lignes cachées pour le nœud ICM

$$\left(L \times \underbrace{\lfloor W/8 \rfloor \times 8}_{(a)} + (3 + L) \times \underbrace{\left\lfloor \frac{W}{(8+1+1)} \right\rfloor \times (8+1+1)}_{(b)} \right) \leq 2048$$

$$\Rightarrow L = 1 + \left\lfloor \frac{2048 - (30 \cdot \lfloor W/10 \rfloor)}{10 \cdot \lfloor W/10 \rfloor + 8 \cdot \lfloor W/8 \rfloor} \right\rfloor$$

où W spécifie la largeur de l'image, (a) la largeur de l'image en octet d'une ligne du buffer $\hat{E}(t+1)$ et (b) la largeur de la ligne du buffer combiné. L'expérience montre que pour une valeur minimum de L (que nous estimons à 3), les itérations successives de l'ICM peuvent s'appliquer sur le même jeu de données présentes en mémoire interne sans que nous constations une dégradation notable de la qualité de détection. Dès lors, plutôt que de re-parcourir l'ensemble de l'image par passe de l'ICM, nous favorisons la localité vers une amélioration des performances (nous évitons notamment le coût de l'initialisation du nœud de traitement). Nous constatons ainsi que pour $W = 256$ et $L = 4$, la détection affiche toujours de très bonnes performances. Par ailleurs, la mise à jour récursive des sites que nous visons introduit une dépendance de données entre deux mises à jour des étiquettes. Dans le cadre d'un partitionnement SPMD des données, le strict respect de cette dépendance nécessite une synchronisation des processeurs au niveau des zones de données

partagées. Ainsi, si le calcul de l'énergie s'opère par ligne et de haut en bas, l'estimation de l'énergie de la première ligne de site du second PP ne doit s'exécuter, en toute rigueur, qu'à l'issue de la mise à jour de la dernière ligne de site de la bande d'image assignée au premier PP. Cette approche perdrait tout l'intérêt du partitionnement SPMD et induirait une durée de traitement équivalente à l'utilisation d'un seul processeur (un traitement pipeliné serait alors plus indiqué). Pour simplifier le développement, nous choisissons de ne pas introduire de couplage entre les processeurs ce qui induit une très légère dégradation de la qualité de détection sur les trois lignes "frontières" de l'image.

L'implantation en assembleur PP du nœud ICM nécessitant un nombre élevé d'opérations, nous nous appuyons sur le compacteur (*ppca*) de code déjà évoqué et qui, rappelons-le, permet l'optimisation du code source par la mise en parallèle des opérations assembleur suivant le graphe de dépendance des données. Notre version dite "linéaire" du programme assembleur qui ne comporte que le séquencement "logique" des opérations est réduite, grâce au pré-processeur *ppca*, de 313 opérations algébriques à seulement 199 instructions VLIW. Cependant, nous soulignons que, dans la mesure où les ressources registres sont limitées au regard de la complexité du nœud (chaque PP dispose de seulement 8 registres de données et 2 jeux de 5 registres d'adresse associés à 3 registres d'index), la compaction de code ne s'avère véritablement intéressante (notamment pour le coeur de boucle) que lorsque la durée de vie des différentes variables qui sont assignées par *ppca* aux différentes catégories de registres est manuellement optimisée. Cette procédure consiste à augmenter la disponibilité des registres (en réduisant la durée de vie de certaines variables) dans les phases de traitement les plus critiques en empilant temporairement certaines données. Si *ppca* ne pratique pas ce "*spillin* " des registres, il nous procure cependant une analyse des différentes durées de vie des registres suivant l'assignation des variables automatiquement effectuée. Ce retour d'informations permet de comprendre les raisons pour lesquelles la compaction n'est parfois pas opérée et permet ainsi de compléter l'assembleur linéaire avec les opérations nécessaires à une gestion de pile. En suivant cette approche, nous parvenons ainsi à un coeur de boucle qui compte 40 cycles par site (par itération de traitement). Ce niveau de performance est atteint par l'utilisation manuelle du pipeline logiciel (deux

itérations partielles sont exécutées en parallèle) mais également par l'utilisation de LUT pour le calcul de l'énergie. Il s'agit d'une idée reprise de [103] qui propose l'utilisation d'une LUT pour le calcul du terme U_m. Dans le même sens, l'auteur étend cette optimisation au calcul du terme U_a pour lequel nous associons deux LUT de 256 valeurs (en rapport avec la dynamique des pixels 8 bits). Ces deux LUT $U_{a0}(o)$ et $U_{a1}(o)$ sont complémentaires et occupent la totalité du troisième banc de mémoire interne. Elles correspondent au calcul de l'équation 4-11 suivant la définition des deux valeurs de $\psi(e(s))$ données par l'équation 4-15 :

Equation 4-17. Pré-calcul des termes de l'énergie d'attache

$$\begin{cases} U_{a0}(o) = \dfrac{1}{2\sigma^2} \times o^2 \\ U_{a1}(o) = \dfrac{1}{2\sigma^2} \times (o - \alpha)^2 \end{cases}$$

Dans la pratique, nous calculons au moyen de ces trois LUT, deux valeurs de l'énergie globale U (sur la base de U_{a0} et U_{a1}) qui correspondent aux deux états possibles de l'étiquette $e(s)$ (soit a ou b pour reprendre l'équation 4-15). Nous retenons, ensuite, la valeur de l'étiquette qui minimise l'énergie locale.

Avec l'ensemble de ces optimisations, nous atteignons ainsi, pour cette étape du traitement configuré suivant 4 passes d'ICM, une durée de 74ms à 40 MHz pour une image 256^2 avec 4 PP. Ce chiffre est à rapprocher des 65ms que nous estimons à cette fréquence, en n'intégrant que les 40 cycles du coeur de boucle. Avec un écart proche de 15%, l'estimation est alors satisfaisante. Par ailleurs, nous pouvons noter que cette durée est quatre fois plus grande pour une version en C de ce même nœud PP (avec l'utilisation des LUT).

4.3.3 Conclusions

Pour conclure, nous pouvons d'abord souligner la bonne performance de notre implantation au regard des durées de traitement publiées de ce même algorithme (tableau 4-6). Nous pouvons, en effet, mettre en avant les quinze images par seconde pour une image 128^2 avec le système "PC-Markov" de [103][104] qui se base sur un FPGA pour le pré-traitement et sur un DSP Motorola 96002 pour la phase d'ICM (qui s'appuie également sur

quatre passes). Dans [103], nous trouvons également une cadence de dix images par seconde pour une image de taille identique sur une architecture CNAPS (SIMD) comptant 256 processeurs élémentaires. Par comparaison, à 60 MHz, notre implantation permet d'atteindre 18 images par seconde sur des images 4 fois plus grandes que sur les réalisations antérieures.

Qualitativement, nous montrons l'adéquation de notre méthodologie de gestion de flux sur cet exemple complexe qui relève toujours du bas niveau. Pour aller plus loin dans cette direction, nous nous intéressons maintenant à la réalisation d'une application complète autour de notre librairie de traitement d'images C80 qui met en œuvre cette implantation pour la détection de mouvement.

Table 4-6. Performances de la détection de mouvement MRF sur quelques architecture

<table>
<tr><td rowspan="2">Taille d'image</td><td colspan="2">Durée en images par seconde (i/s) avec 4 passes d'ICM</td></tr>
<tr><td>LIS UPMC/EIA</td><td>LIS Grenoble (INPG)</td></tr>
<tr><td rowspan="3">128×128</td><td rowspan="3">-</td><td>Sparc-10 : 0.5 i/s</td></tr>
<tr><td>CNAPS 256 PE@20MHz : 10 i/s</td></tr>
<tr><td>FPGA + Motorola 96002@40MHz : 15 i/s</td></tr>
<tr><td rowspan="3">256×256</td><td rowspan="2">C80 : 40 MHz, 12 i/s
18 i/s à 60 MHz</td><td>Autre laboratoire</td></tr>
<tr><td>Transvision (12 T800) : 7,5 i/s</td></tr>
<tr><td>Pentium II 400 MHz, C optimisé (sans MMX) : 40 i/s [105]</td><td rowspan="2"></td></tr>
<tr><td>512×512</td><td>C6202 @ 250 MHz (<u>simulé, hors E/S</u>) : 47 i/s [105]</td></tr>
</table>

4.4 Amélioration des performances d'une librairie de compression M-JPEG sur C80

4.4.1 Intérêt de l'encodage JPEG

Si le standard de compression JPEG et la transformée en cosinus discret (DCT) apparaissent aujourd'hui comme obsolètes avec l'avènement de la norme JPEG2000, les auteurs de [106] mettent en avant que bien que la compression par ondelettes ait reçu une vive attention cette dernière décennie (et notamment après la publication de l'article de Shapiro [107]), la suprématie des nouvelles techniques d'encodage sur la base de cette transformée espace/temps repose essentiellement sur la bonne structuration et quantification du flux et moins directement sur les performances pures de cette nouvelle transformée réversible. Ainsi, dans [106], les auteurs proposent une technique d'encodage progressive originale qui s'appuie sur la quantification par plan de bits des données issues de la DCT. A taux de compression égal, cette approche montre une qualité de restitution des images supérieure à la norme JPEG et comparable à l'encodeur par ondelettes proposé dans [108] qui s'impose comme une référence.

Parallèlement, les auteurs de [109] proposent la structure d'un encodeur progressif où une priorité est affectée à chaque bit de donnée DCT de manière à mesurer et à optimiser l'impact de la réduction du bruit de l'image reconstruite. En effet, les bits qui favorisent une réduction importante du bruit et qui sont associés à un flux comprimé de faible taille sont ainsi regroupés et transmis en priorité. Cette approche montre que pour les taux de compression importants, la qualité de reconstruction dépasse celle de l'encodeur SPIHT pour la transformée en ondelettes proposée dans [108].

L'amélioration du standard JPEG passe également par la réduction des artefacts de blocs qu'introduit la norme JPEG pour les taux de compression élevés. Il se trouve, en effet, que cette déformation n'apparaît pas avec les encodeurs par ondelettes pour lesquels, seul un effet de flou homogène apparaît (taux de 100:1 et au-delà). Face à cela, nous trouvons plusieurs techniques qui permettent de réduire les effets de blocs introduits par la norme JPEG et dont certaines sont détaillées dans [110] et p.261-264 de [111].

Pour conclure, nous résumerons cette discussion en soulignant que moyennant quelques extensions apportées aux principes de compression JPEG (parfois même en accord avec les spécifications initiales du standard), cette norme est toujours capable d'offrir des performances véritablement compétitives. Cette importante constatation permet d'augmenter la durée de vie des systèmes à base de JPEG ou de M-JPEG (*"Motion"*-JPEG, pour la compression de séquences). En outre, il s'avère que ces techniques sont aujourd'hui très présentes au sein des appareils photo et caméscopes numériques. L'explication tient à la faible complexité calculatoire qu'elles nécessitent face à des principes de compression plus lourds tels que le MPEG et sa coûteuse compensation de mouvement (c.f. p. 132 de [112]). Dans ce contexte, nous proposons une extension originale au format M-JPEG dont nous pensons qu'il affiche un intérêt industriel certain dans le contexte des systèmes embarqués que nous venons d'évoquer.

4.4.2 Du format M-JPEG à la structure d'un encodeur original

Le codage M-JPEG ne faisant pas référence à un standard comme il est souligné p.403 dans [112], nous rappelons le principe de fonctionnement généralement admis. Ce codage se base sur l'alternance de la compression JPEG d'une image de référence que nous pouvons qualifier d' "intra" par analogie au format MPEG, et d'une image de différence traduisant les variations d'intensité lumineuse entre l'image courante et la dernière image "intra". La mise à jour de l'image "intra" qui sert de référence s'effectue généralement de manière périodique. Par opposition aux standards de type MPEG (ou ceux pour la vidéoconférence comme le H263/H324), cette approche offre des taux de compression inférieurs mais au prix d'une puissance calculatoire réduite (il n'y a pas, entre autres, de compensation de mouvement). Cette approche permet alors d'envisager plus facilement l'utilisation de composants embarquables tels que ceux mis en œuvre dans les appareils photo numériques qui, pour la plupart, s'appuient sur la norme JPEG voire, pour certains, sur le M-JPEG (nous trouvons une liste de composants matériels pour le JPEG dans [113], p.267-293 et p.324-327 de [112]).

Sur la base d'une idée issue d'un brevet industriel [114] de la société Télédiffusion de France (TDF), nous nous proposons à la fois de réduire l'entropie du signal *différence* et d'améliorer la qualité du signal reconstruit en annulant les variations d'intensité lumineuse lorsqu'aucun mouvement n'est détecté dans la scène. Cette approche s'appuie bien entendu sur la mise en œuvre de la détection markovienne du mouvement et vise à ne coder que les parties de l'image pour lesquelles nous détectons du mouvement ce qui permet, notamment, de réduire le bruit qu'engendrent généralement les capteurs vidéo. Cette technique se destine plus particulièrement aux scènes ayant un fond fixe (cas des images de plateau de télévision). Elle est indépendante de la technique de codage employée en aval (nous pouvons parfaitement imaginer un codage à base d'ondelettes). Par ailleurs, si aucun codage entropique n'est employé et que, par conséquent, le signal n'est pas compressé, cette technique permet de réduire la distorsion entre deux images du capteur. Par la suite, nous comparons notre approche à celle proposée dans [114] de manière à mettre en évidence les différences entre les deux techniques. Cette comparaison sera quantitative et qualitative. Elle repose sur la réalisation d'un encodeur original sur C80 qui met en œuvre l'implantation de la détection de mouvement ainsi qu'une librairie de compression JPEG parallèle pour C80. Le schéma synoptique de ce codeur est décrit par la Figure4-6 et se compose de 7 blocs fonctionnels :

1. Nous trouvons tout d'abord un nœud pour le calcul du signal *différence* que nous obtenons en soustrayant l'image courante avec la dernière image prise comme référence ;

2. L'information de différence est ensuite masquée suivant la carte binaire des pixels en mouvement (nous forçons l'entropie à zéro) ;

3. Ce troisième nœud regroupe les deux chaînes de traitement de la détection de mouvement et produit la carte du mouvement utilisée par le nœud précédent ;

4. Ce nœud représente la fonction de compression du signal. Ici, il s'agit d'un encodage de type JPEG "baseline" (les exigences minimales de la norme sont satisfaites) avec perte, et qui s'appuie sur un codage entropique de type Huffman. Par ailleurs, les tables de base pour la quantification des coefficients DCT ainsi que celles des codes d'Huffman sont statiques et correspondent à celles préconisées par la norme [113]. Par ailleurs, dans la mesure où nous souhaitons mesurer la dégradation du signal, ce nœud

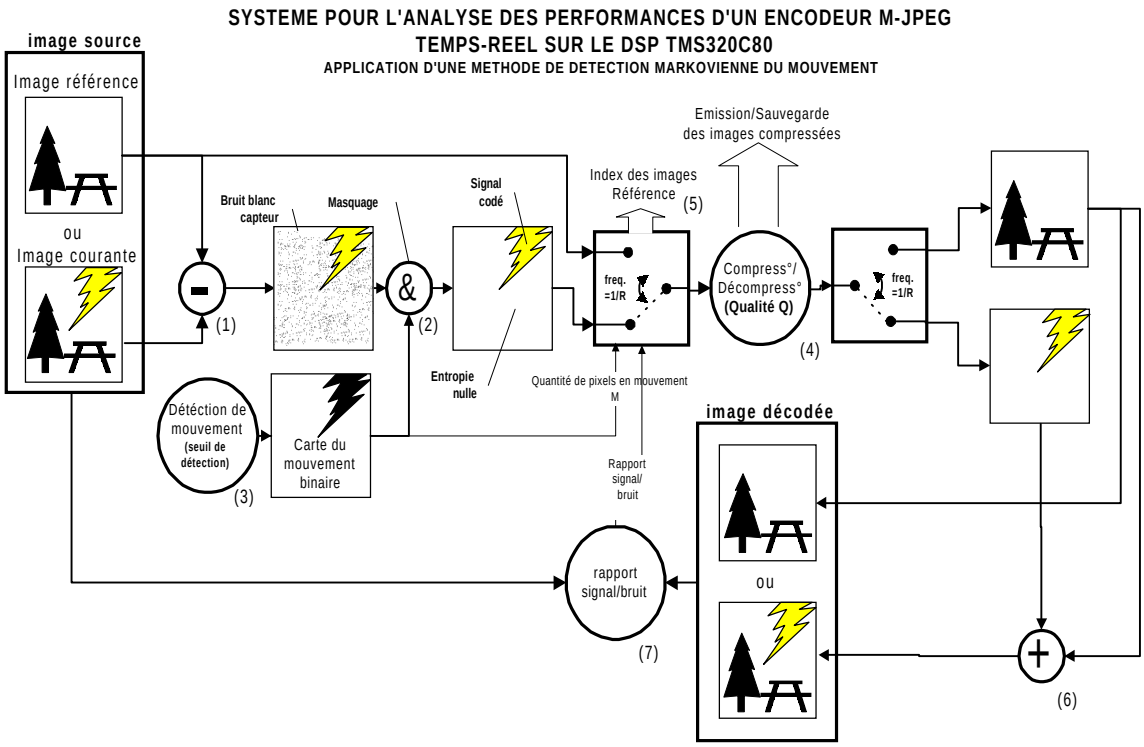
comporte également la fonction de reconstruction du signal compressé (décompression JPEG) ;

5. Ce bloc correspond à un système d'aiguillage pour décider ou non l'encodage et la mise à jour de l'image de référence. Ce système s'appuie soit sur une mise à jour périodique de cette image (de fréquence *1/R*), soit suivant la pondération linéaire de deux critères qui sont, d'une part, la quantité de données en mouvement (*M*) et d'autre part, la mesure de la dégradation de l'image pour le signal *différence*. Pour ce dernier cas de figure, il convient alors de maintenir une table de l'indice des images qui sont prises comme référence pour la reconstruction ultérieure des images de la séquence ;

6. Le nœud "addition" permet de reconstruire l'image courante par rapport à une image de référence et le signal de différence décompressé. Bien sûr, ce nœud n'est pas utilisé quand l'image de référence est remise à jour ;

7. En fin de chaîne, nous trouvons un nœud pour le calcul de la dégradation de l'image reconstruite par rapport à l'image courante initiale. Ici, nous nous appuyons sur la mesure du PSNR (*Peak Signal to Noise Ratio*), des alternatives étant la mesure du SNR ou RMSE (*Root Mean Square Error*). Nous notons cependant que ces métriques ne donnent qu'une mesure stricte de la différence existant entre deux signaux, le confort visuel de la dégradation pour l'oeil humain n'étant pas nécessairement quantifiable. Cette remarque est illustrée par la suite.

Figure 4-6. Principe d'un encodeur M-JPEG original



4.4.3 Une autre approche à base de seuillage spatio-temporel fréquentiel

Le brevet TDF [114] précédemment évoqué met avant tout l'accent sur l'optimisation de la taille d'un flux vidéo avec, comme contrainte, une bande passante réduite de l'ordre de 64 Kb/s (qui correspond à un canal du protocole Numéris). Le principe de base s'appuie sur le format M-JPEG et l'introduction du signal *différence*. Cette différence est par ailleurs masquée, par bloc, d'après une mesure de "mobilité". Cette mesure s'effectue dans le domaine fréquentiel et correspond à l'élévation au carré de certaines des données résultant de la transformée en cosinus discret du signal *différence*. Il s'agit bien d'une approche spatio-temporelle dans la mesure où on intégre les variations d'intensité lumineuse entre l'image courante et celle de référence et dans la mesure où la transformée DCT s'effectue par blocs de taille 8×8 (comme pour la norme JPEG). Dans le cas 2D, si x est le signal de différence, la transformée en cosinus discret s'écrit
alors :

Equation 4-18. Définition de la DCT 8×8 pour le JPEG

$$dct(i,j) = k \cdot \sum_{u=0}^{7} \left(\sum_{v=0}^{7} x(u,v) \cdot \cos\left(\frac{(2 \cdot u + 1) \cdot i \cdot \pi}{16} \right) \cdot \cos\left(\frac{(2 \cdot v + 1) \cdot j \cdot \pi}{16} \right) \right)$$

avec :

$$\begin{cases} i = j = 0 \Rightarrow k = 1/8 \\ i = 0 \text{ ou } j = 0 \Rightarrow k = 1/(4 \cdot \sqrt{2}) \\ \text{sinon } k = 1/4 \end{cases}$$

Le calcul de la mobilité correspond à la somme des énergies comprises dans un intervalle du spectre. Ici, on s'intéresse notamment aux énergies les plus significatives associées aux fréquences basses. Ainsi, on éleve au carré les coefficients DCT de la sous-matrice 4×4 de chaque bloc afin de déterminer si des variations suffisantes sont apparues entre le bloc 8×8 de l'image courante et son homologue de l'image de référence. Comme le suggère les documents [114] et [115], on intégre cependant pas la composante continue de la transformée (d'indice (0,0) dans la matrice des coefficients DCT) qui correspond à la moyenne de l'ensemble des pixels différence du bloc que l'on ne juge pas significative. La définition de la mobilité a s'écrit alors :

Equation 4-19. Gradient spatio-temporel fréquentiel

$$a = \left(\sum_{i=0}^{3} \left(\sum_{j=0}^{3} dct(i,j)^2 \right) \right) - dct(0,0)^2$$

L'idée consiste ensuite à forcer l'ensemble des pixels d'un bloc à zéro dans l'hypothèse où la mobilité est inférieure à un seuil (*A*) :

Equation 4-20. Seuillage du gradient fréquentiel

$$a < A \Rightarrow i = 0 \ldots 7, (j = 0 \ldots 7), dct(i,j) = 0$$

Le brevet propose alors de comprimer l'image de différence après que les blocs dont l'activité n'est pas suffisante ont été masqués. Dans ce sens, le rafraîchissement de l'image de référence est décidé manuellement (par le biais d'un opérateur externe), une mise à jour progressive des blocs de référence est suggérée. Cette gradation est en effet contrainte par la bande passante qui doit, en plus, permettre la transmission du flux "différence" à cadence suffisante.

De même, nous trouvons dans ce document des propositions pour une quantification "intelligente" des données afin de réduire et de mieux réguler la taille du flux. Par la suite, nous comparons les résultats de cette approche suivant une remise à jour en une passe de l'image de référence (nous intégrons la mise à jour graduelle ainsi que la quantification intelligente). Nous nous intéressons plus particulièrement aux taux de compression élevés.

4.4.4 Implantation du codeur M-JPEG expérimental

La structure du codeur expérimental est détaillée sur la Figure4-7. Ce schéma fait apparaître les différents nœuds de traitement ainsi que la géométrie des patrons associés à chaque flux. Nous notons, en bas du schéma, que le partitionnement des traitements est réalisé au moyen de 5 chaînes SPMD distinctes (*C*) et du nœud de compression et décompression JPEG. Par souci de clarté, ces chaînes sont elles-mêmes regroupées au sein de tâches logicielles (*T*) qui s'exécutent en séquence.

Les deux premières chaînes reprennent les nœuds déjà présentés. Nous notons, cependant, une modification qui concerne l'extraction du signe (*Si*) de la différence entre l'image courante *I(t+1)* et celle de référence. Ce signe est nécessaire à la troisième chaîne de la première tâche (T1#C3). En effet, l'unique nœud de cette chaîne effectue le masquage des

pixels avec la carte du mouvement *E(t)* et met à jour l'information de différence et non de différence absolue d'où la nécessité de préserver le signe initial de la différence. La dynamique des pixels 8 bits est réduite à 7 bits pour intégrer le signe de la différence *Diff*. Par ailleurs, la dynamique du signal *Diff* est centrée autour de la valeur 128 de manière à homogénéiser son entropie et à ne pas introduire de pics de hautes fréquences dans la DCT qui fausseraient la reconstruction du signal. Avec l'opérateur logique '&' qui matérialise l'opération de masquage (si du mouvement apparaît), et *Si* qui se définit égal à 1 si le signe de la différence est négatif (0 sinon), nous obtenons :

Equation 4-21. Ajustement de la dynamique du signal différence

$$\mathrm{Diff} = 128 + (1 - 2 \times \mathrm{Si}(t)) \times \frac{(\mathrm{O}(t)\ \&\ \mathrm{E}(t))}{2}$$

Le signal *Diff* est ensuite transmis à la deuxième tâche qui réalise la compression (ainsi que la décompression dans notre contexte). L'opération inverse de modification de la dynamique est ensuite prise en charge par le nœud de la troisième tâche (T3#C1) qui intègre alors les données de l'image de référence. Par ailleurs, les données *Diff* étant quantifiées lors de la compression, il convient de saturer les résultats de la reconstruction au risque d'introduire des dégradations visuelles importantes (apparition ded carrés noirs ou blancs dans l'image). La troisième et dernière tâche ne compte également qu'une chaîne de trois nœuds pour l'estimation de la distorsion. Nous mesurons le SNR, PSNR et RMSE qui sont des distances communément employées dans la communauté de la compression pour quantifier la distorsion. La suite de ce chapitre utilise plus particulièrement le PSNR.

Le partitionnement de la librairie de compression et de décompression est de type MPMD et repose sur l'utilisation des 4 PP. Trois de ces PP sont consacrés au calcul de la DCT, de la quantification et de l'étape permettant de compter le nombre d'occurrences des coefficients DCT (*Run Length Coding*). Ces 3 étapes constituent la charge de calcul la plus importante et justifient l'utilisation de plusieurs PP, chacun prenant en charge une partie de l'image. Chacun de ces PP se synchronise avec le PP chargé d'appliquer le codage d'Huffman sur la base des données produites par les trois autres.

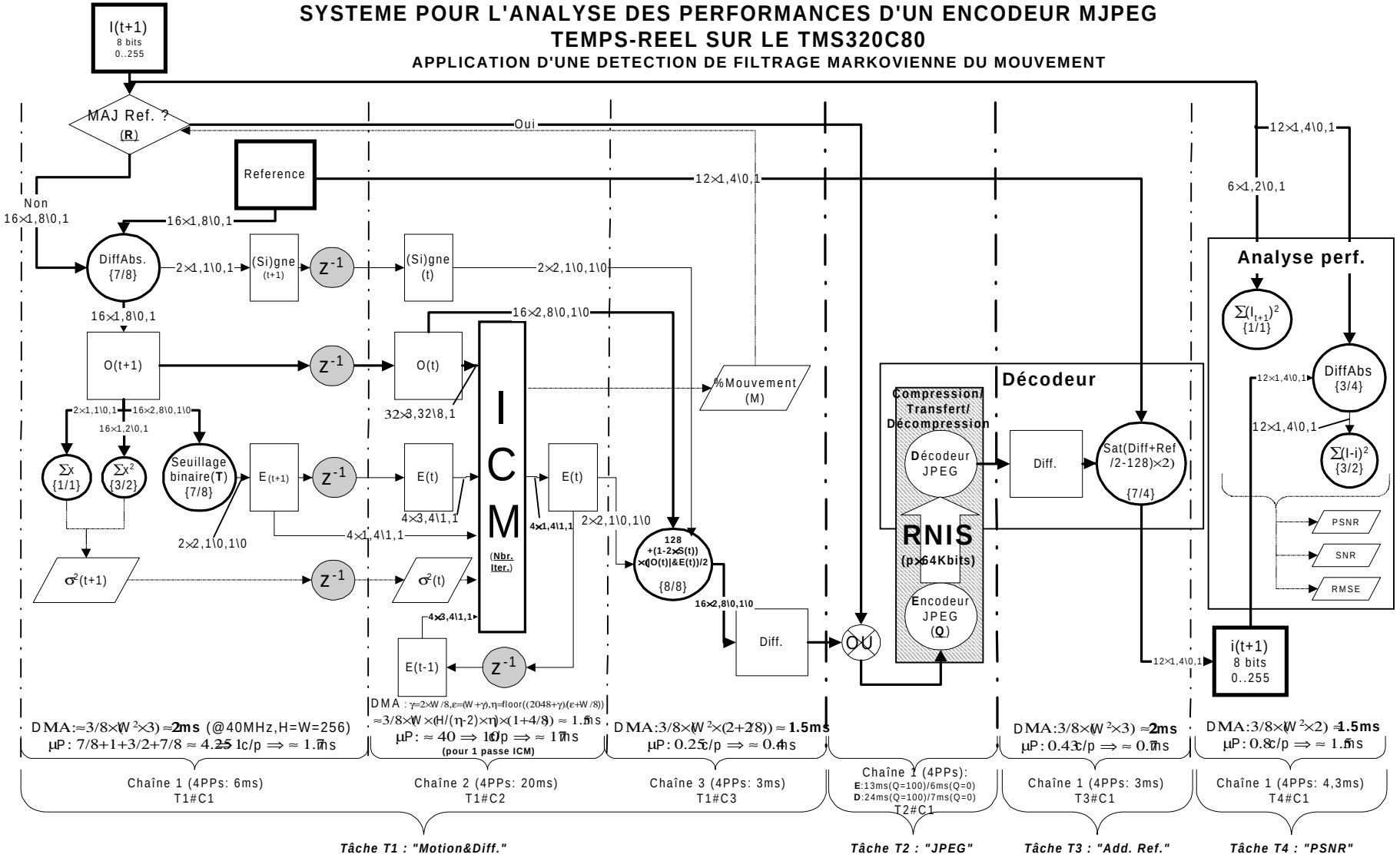
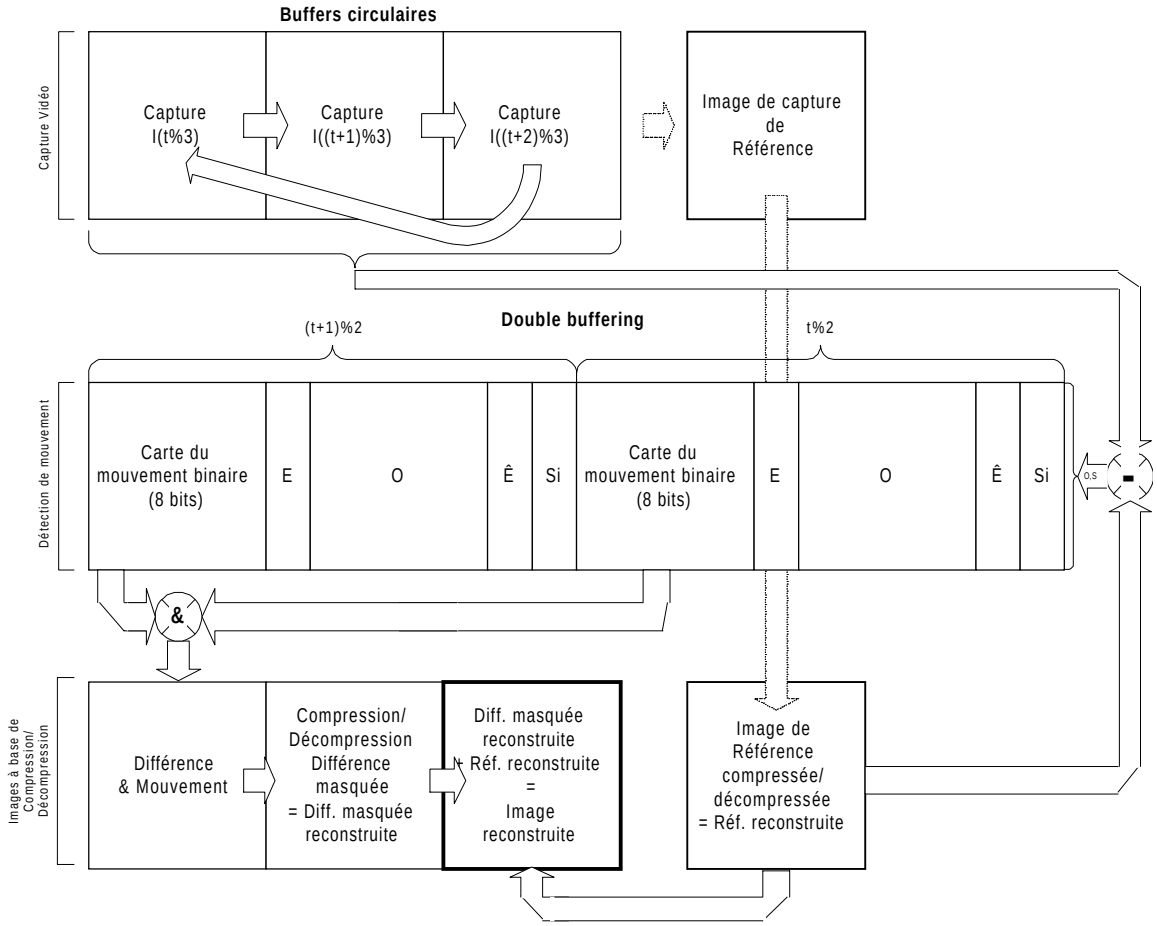


Figure 4-7. Structure du prototype d'encodeur

Pour l'ensemble des PP, le double ou triple buffering n'est pas mis en œuvre (l'espace de données internes nécessaire aux différentes étapes du traitement est en effet incompatible avec cette optimisation). Par ailleurs, le module d'encodage de cette librairie est modifié pour permettre d'évaluer le principe du brevet [114]. Dans ce mode, nous forçons la quantification JPEG à zéro lorsque le seuil de mobilité n'est pas atteint.

L'organisation externe des buffers du prototype est calquée sur la Figure4-5 pour la partie détection de mouvement. Pour notre application, nous distinguons trois lignes de buffers comme le suggère la Figure4-8 qui permet d'appréhender la copie d'écran du système que nous découvrons avec la Figure 4-9. La première ligne correspond aux buffers nécessaires à la capture vidéo. Nous nous appuyons sur une suite de trois buffers circulaires pour la capture d'images à la volée. Nous y extrayons de manière sporadique l'image de référence que nous plaçons à droite de cette ligne. Cette image que nous décidons de compresser possède sa reconstruction sur la ligne du bas, à droite.

Figure 4-8. Aperçu de l'organisation des buffers externes

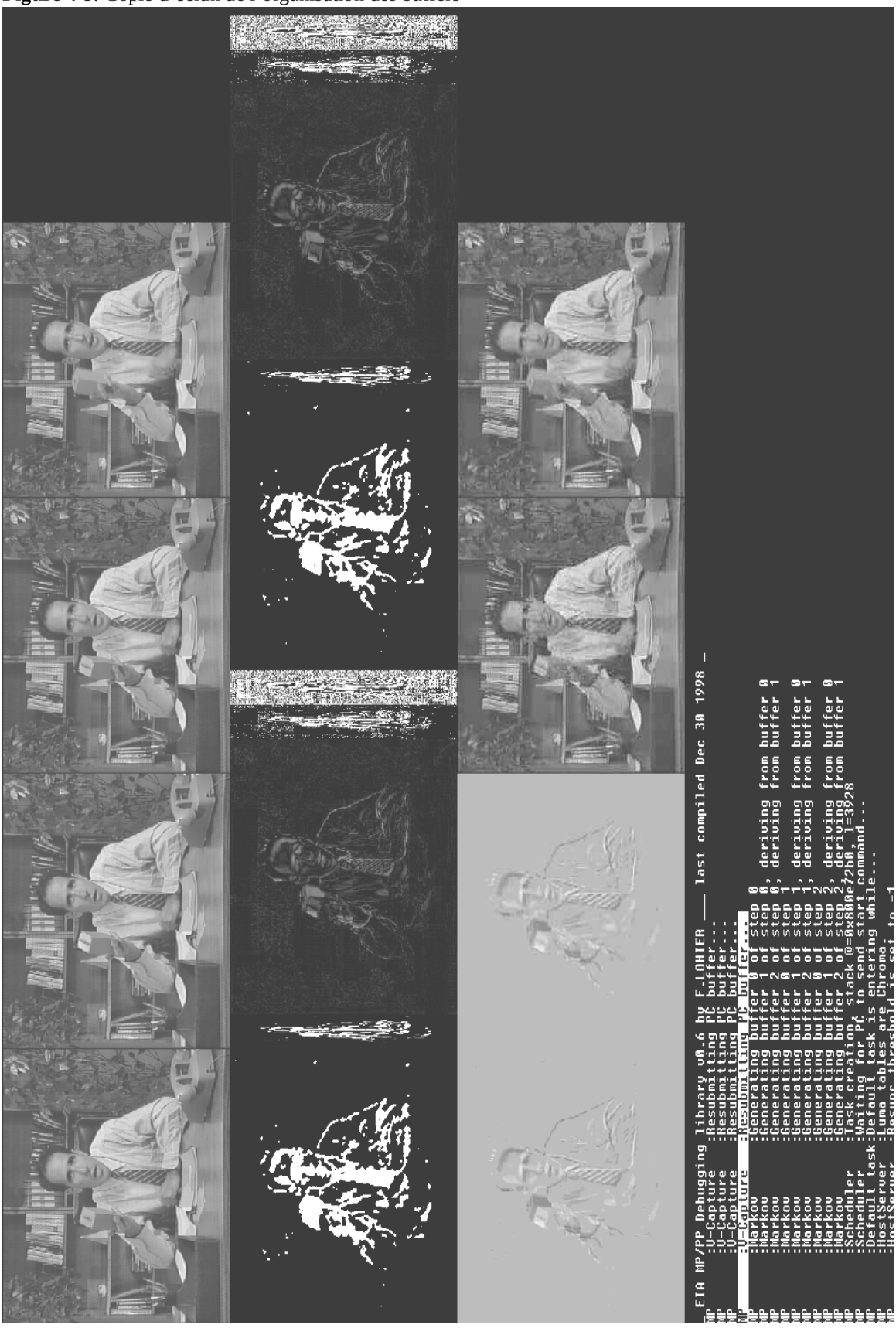


La deuxième ligne reprend les buffers externes introduits pour la détection de mouvement que nous gérons par double buffering (cf Figure4-5). Nous pouvons noter l'adjonction des plans binaires *Si* pour l'intégration du signe. Enfin, sur la dernière ligne, nous trouvons un groupe de trois buffers sur la gauche qui correspondent respectivement aux résultats :

- du masquage de la différence par l'information du mouvement,
- de la compression et décompression de ce dernier signal,
- de la reconstruction de l'image avec l'ajout de l'image de référence compressée.

La Figure 4-9 illustre une copie d'écran produit par le système sur la séquence de test dite du *Salesman*. Sur cette exemple, les images ne sont pas capturées à la volée mais transmises de manière synchrone depuis le PC à la carte C80. Cette approche permet ainsi de comparer les performances de notre codeur par rapport à la technique suggérée dans le brevet et ce sur la base des mêmes données (nous supprimons le bruit qu'engendrerait la numérisation d'un signal analogique).

Figure 4-9. Copié d'écran de l'organisation des buffers



4.5 Performances du système de compression proposé

Dans cette section, nous analysons les performances du système tout d'abord en terme de rapidité de traitement. A ce titre, nous pouvons rappeler que la qualité de la détection de mouvement pour les images numérisées à la volée impose une cadence de traitement suffisamment importante. Cette remarque souligne alors l'importance de la mesure de la complexité calculatoire afin de borner l'espace des paramètres associés au système. Ensuite, nous poursuivons l'analyse sur le plan qualitatif en opposant essentiellement les deux procédés concurrents évoqués. Les performances sont alors mesurées en terme d'écart par rapport à l'image originale et de dégradation du confort visuel.

4.5.1 Cadence du traitement

Les données en accolade des nœuds de la Figure 4-7 reprennent les nombres de cycles par pixel nécessaires à une itération de traitement. Ces données sont regroupées sur le tableau ci-dessous avec une estimation de la durée des transferts et du traitement par opposition à la durée globale réellement mesurée pour chaque chaîne :

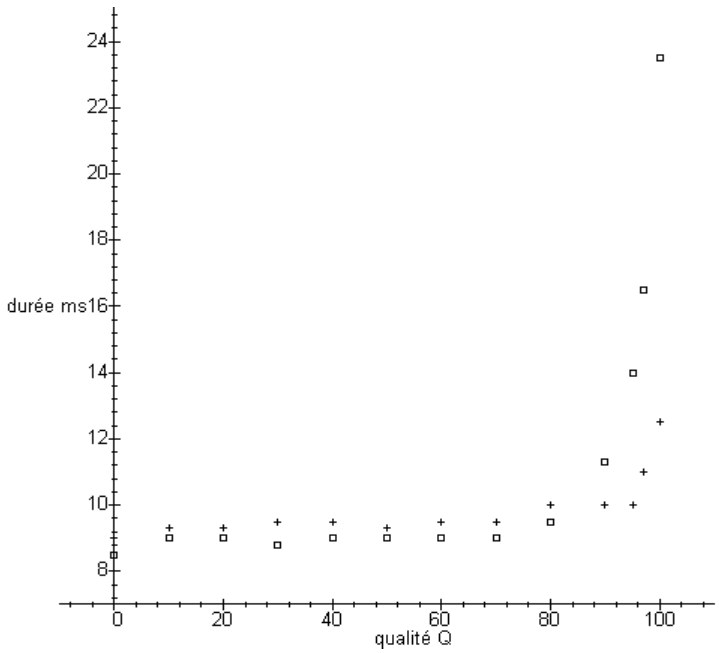
Table 4-7. Performances de l'encodeur Markov-M-JPEG C80

Chaîne	nb. cycles par pixel des noyaux (par PP)	Est. 4 PP à 40 MHz	Est. DMA à 40 MHz	Durée mesurée
T1#C1	7/8+1+3/2+7/8 ≈ 4.25 c/p	1.7 ms	2 ms	6 ms
T1#C2	40 c/p	17 ms	1.5 ms	20 m
T1#C3	1 c/p	0.4 ms	1.5 ms	3 ms
T3#C1	7/4 ≈ 1.75 c/p	0.7 ms	2 ms	3 ms
T4#C1	1+3/4+3/2 ≈ 1.5 c/p	1.5 ms	1.5 ms	4.3 ms

Ces durées se basent sur l'utilisation des quatre PP pour le traitement d'images 256^2 à 40 MHz et sur le paramétrage d'une seule itération d'ICM. Précisons que les durées estimées n'intègrent pas le coût de la mise à jour des caches d'instructions que nous n'avons pas mesuré (sur cet exemple, les durées de simulations sont véritablement prohibitives) d'où une imprécision importante. Pour le noeud ICM, soulignons que cette estimation partielle offre pourtant une précision de 15%.

La convergence rapide de l'algorithme nous permet, en effet, de ne mettre en place qu'une seule itération de traitement pour la chaîne *T1#C2* et, ainsi, de favoriser des performances proches du temps réel avec notre prototype à 40 MHz. Cette remarque intègre le coût de la compression/décompression JPEG. Ici, bien que la réduction de la complexité calculatoire diminue l'efficacité de la détection du mouvement, les expériences montrent que les résultats produits sont assez satisfaisants. Nous illustrons cette constatation par la suite. Par ailleurs, la durée de la compression/décompression JPEG est fonction du facteur de qualité et de la nature des données. Elle oscille approximativement de 6ms/7ms pour l'encodage/ le décodage avec une quantification maximale (qualité *Q* égale à zéro) à 13ms/24 ms pour la quantification minimale associée au facteur de qualité maximale (*Q=100*). La Figure 4-10 montre ces variations pour une image 256^2 à 40 MHz où les rectangles montrent la durée du décodage JPEG alors que les croix présentent les durées de l'encodage et ce pour différents facteurs de qualité :

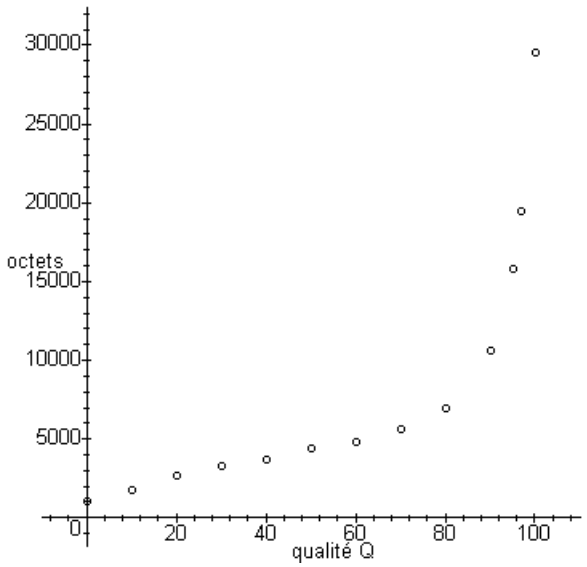
Figure 4-10. Durée de l'encodage/décodage pour différents facteurs de qualité JPEG



Nous notons que les durées affichées sur ce graphique intègrent le coût des entrées/sorties de la capture vidéo et se basent sur la compression d'images de référence de type "scènes d'intérieur". Les courbes nous apprennent alors qu'à 60 MHz, nous pouvons envisager un système intégrant la compression et la décompression JPEG avec l'adjonction des chaînes

qui font l'originalité de notre codeur, tout en autorisant une cadence de traitement en temps réel. En effet, à 60 MHz, les chaînes du tableau 4-7 représentent une durée de traitement d'environ 25ms alors que la durée cumulée de la compression et décompression ne représente qu'environ 13ms (à 60MHz) jusqu'à un facteur de qualité 80. Pour cet intervalle de valeur de *Q*, la taille du flux de l'image de référence oscille de 1Ko (*Q=0*) à près de 8Ko (*Q=100*) qui représente la limite de la bande passante que nous nous octroyons a priori. Ces chiffres concernent toujours le traitement d'images 256^2 et traduisent un taux de compression de l'image de référence variant de 65:1 à 9:1. La Figure4-11 montre un exemple d'évolution de la courbe de la taille du flux comprimé pour différents facteurs de qualité. Par ailleurs, nous soulignons que ce taux est bien sûr accru pour le signal*différence*.

Figure 4-11. Courbe d'évolution de la taille du flux



4.5.2 Comparaisons quantitatives et qualitatives des deux algorithmes

La mesure des performances des deux procédés qu'implante notre système pose deux difficultés. La première concerne l'espace des paramètres dans lequel doit se faire la comparaison si nous voulons être rigoureux. Ainsi, plus encore que la nature même des données dont dépend la taille du flux comprimé, nous distinguons quatre paramètres associés à notre système qui font varier sensiblement la taille du flux issu de la compression. Ces paramètres sont :

1. Le nombre de passes d'ICM pour la détection markovienne,
2. Le seuil de détection de mouvement, qui pour notre approche, se définit comme le seuil *T* de binarisation du champ d'étiquettes initiales $\hat{E}(t)$. Pour l'approche proposée dans le brevet, il s'agit du seuil de mobilité *A*,
3. La valeur du facteur de qualité JPEG, oscillant entre 0 et 100,
4. La fréquence ou le critère de re-synchronisation de l'image de référence (*1/R*).

Pour réduire la dimension de l'espace des possibilités nous fixons par la suite le nombre d'itérations de la relaxation à un (ce qui permet d'envisager une implantation temps réel). De même, le critère de mise à jour de l'image de référence est ici basé sur une simple remise à jour périodique. Nous rappelons que dans le cadre d'un fonctionnement normal du système, nous privilégions une mise à jour qui vise à réduire dynamiquement la distorsion du signal reconstruit par pondération linéaire de la quantité de pixels en mouvement et de la distorsion en terme de PSNR avec l'introduction d'un seuil.

Ce premier niveau de simplification laisse ouvert le second problème que constitue le choix d'un critère permettant de comparer les deux méthodes dans le contexte où les seuils de mouvement ont une nature différente. Pour cette raison, nous proposons de comparer ces approches soit par rapport à un seuil *T* et son homologue de mobilité *A* engendrant une quantité de pixels en mouvement identique (ou, à défaut, voisine) soit suivant des valeurs de ces mêmes seuils engendrant une taille de flux compressé identique (ou la plus proche possible). Dans la pratique, étant donné une fréquence de re-synchronisation de l'image de référence (*1/R*), un facteur de qualité *Q* et un seuil de mouvement *T*, nous déterminons, de manière dichotomique, la valeur du seuil de mobilité *A* qui engendre un critère de comparaison cohérent soit en terme de pourcentage d'image en mouvement *M* soit en terme de taille de flux *S*.

Pour comparer précisément les performances quantitatives des méthodes, nous diversifions la nature des données en prenant notamment des séquences comportant plus ou moins de mouvement ainsi qu'en faisant varier les valeurs de *Q*, *R* et *T*. Pour chacune de ces configurations de traitement, nous automatisons alors la procédure dichotomique consistant à déterminer le seuil de mobilité *A* autorisant une analyse pertinente. En échantillonnant le facteur de qualité avec les valeurs *Q*=[10,30,50,70,100], celui de re-

synchronisation avec R=[5,7,10,15] et celui du seuil de mouvement par T=[0,5,7,10,11,12,13,15,17,20,25,30,40], nous atteignons, pour un jeu d'une dizaine de séquences allant de 40 images à plus de 500, une masse de calcul considérable nécessitant la réalisation de tests sur plusieurs semaines avec notre prototype.

La Figure 4-12 présente la synthèse détaillée des résultats pour 7 séquences dont les 5 premières apparaissent comme des références dans la communauté scientifique alors que les deux dernières ont été réalisées pour les besoins des travaux présentés. Celles-ci comptent un nombre d'images important et affichent un mouvement lent pour une qualité de numérisation médiocre. Pour chaque séquence, cette figure montre les courbes interpolées du rapport signal/bruit en fonction de différentes valeurs de Q (pour l'abscisse) et R (par colonne). En noir apparaissent les courbes associées à l'approche markovienne, en gris, les courbes selon le procédé du brevet, et enfin, les courbes en pointillés montrent les performances de la compression JPEG seule (sans différence d'image et sans masquage, c'est-à-dire $R=1$). Ici, les deux procédés sont comparés sur le critère d'égalité du pourcentage de mouvement M détecté, en moyenne, pour toute la séquence et sont synchronisés par exploration dichotomique. L'interpolation des courbes est réalisée avec le logiciel Maple et s'appuie sur les travaux initiaux de Neal et Natarajan (résumés dans [112], p.61-68, p.96-97 et p.105-116) qui introduisent l'utilisation d'un modèle isotropique de fonction covariance pour la modélisation statistique des données. Cette covariance décroît de manière exponentielle lorsque nous nous éloignons de la ligne (i) ou colonne (j) de référence. Avec σ^2 représentant la variance des données de l'image, l'équation s'écrit :

Equation 4-22. Modèle isotropique de covariance

$$\mathrm{Cov}(i, j) = \sigma^2 \cdot e^{-\alpha\sqrt{i^2 + j^2}}$$

Par ailleurs, pour ce modèle, nous considérons comme fixe la valeur de la corrélation horizontale et verticale entre deux lignes ou colonnes d'image avec ρ=0.95[1]. Sur la base de ces paramètres et des travaux théoriques de Berger sur la compression optimale [116], plusieurs auteurs ont, par la suite, développé des modèles analytiques pour l'estimation de la courbe optimale du taux de compression R par rapport à la dégradation du signal D

1. D'après [112], nous avons la relation $\rho = e^{-\alpha}$ pour l'équation 4-22.

(courbes *R(D)*) dans le cas d'images 2D. Ici, le terme R est bien sûr à différencier de la fréquence 1/R de mise à jour de l'image de référence précemment évoqué.

Dans [112], nous trouvons un résumé de différentes fonctions de transferts associées à cette modélisation analytique, résumé qui s'attache à décrire les performances théoriques que nous pouvons attendre des principales approches d'encodage comme pour le cas de la compression d'images différence ou ceux avec et sans différence mais avec compensation de mouvement. Pour la compression par différence, le modèle proposé et que nous retiendrons comme une approximation du modèle théorique pour le masquage de la différence avec la carte du mouvement, est décrit par l'équation suivante :

Equation 4-23. Modèle de courbe *R(D)* théorique optimale pour le codage de différence

$$\begin{cases} R(D)_{Diff} = \frac{1}{2} \cdot \log_2\left(\frac{\sigma^2}{D} + 1\right) \\ \sigma = 2\sigma_I^2 \cdot \left(1 - \rho e^{-2\pi f_0 \sqrt{u^2 + v^2}}\right) \end{cases}$$

Dans ce modèle, la variance du signal différence, σ, s'exprime par rapport à la variance de l'image de référence décrite par σ_I et la distance *(u,v)* qui sépare la distance moyenne des pixels de l'image courante avec l'image de référence (nous supposons que l'image n'est pas déformée mais simplement translatée). Plus cette distance est grande, plus la taille *R* du flux ou le nombre de bits par pixel augmente pour une même valeur de *D*. F_0 représente la fréquence de coupure de la fonction de transfert. Nous proposons, alors, d'interpoler la courbe *R(D)* avec la fonction linéaire suivante, où *k* et *k'* sont les constantes caractéristiques du polynôme à définir :

Equation 4-24. Modèle de fonction linéaire pour l'interpolation des courbes *R(D)*

$$e^{2\log(2)R(D)} = 2^{2R(D)} = k \cdot D^{-1} + k'$$

Par la suite, *R* représente la taille du flux moyen pour des images 256^2 et s'exprime en Ko. L'échelle logarithmique du PSNR ne pouvant convenir à l'interpolation d'une fonction en 1/D, nous spécifions la relation permettant de passer de la mesure du PSNR *d* en décibels (dB) à celle de la distorsion *D* par :

Equation 4-25.

$$D = \sigma = \sqrt{\frac{255^2}{e^{\frac{d \cdot \log(10)}{10}}}}$$

4.5.2.1 Analyses quantitatives comparées

Les courbes de la Figure4-12 sont données pour une valeur du seuil de mouvement égale à 15. Les trois premières séquences montrent une supériorité sensible de notre procédé, notamment pour les qualités importantes associées à l'intervalle de PSNR compris entre 30 et 40 dB. Pour cet intervalle, à PSNR égal, le taux de compression est divisé par un facteur supérieur à deux comparé au brevet et lorsque nous considérons des valeurs de re-synchronisation *R* égales à 10 ou 15. Pour ces trois séquences, notamment celles de *salesman* et *trevor*, nous constatons également que les courbes *R(D)* des deux approches sont meilleures que celle de la seule compression JPEG (en pointillés) jusqu'au seuil de *R*=5. Après, avec *R=10* et *R=15*, et à partir de 35 db, la remarque s'inverse.

Les séquences suivantes *train* et *ping-pong* introduisent plus de mouvement et montrent que les performances des deux procédés sont, cette fois-ci, meilleures que pour le JPEG avec *R=5* mais pas avec *R=10* et *R=15*. En outre, pour ces deux séquences, notre approche engendre des courbes *R(D)* qui sont moins performantes par rapport à la méthode du brevet d'un facteur s'étalant entre 1.5 et 2 pour l'intervalle de PSNR compris entre 25 et 35dB avec *R=5*. Avec *R=10* et 15, les deux procédés présentés affichent des performances proches avec un avantage pour le principe du brevet.

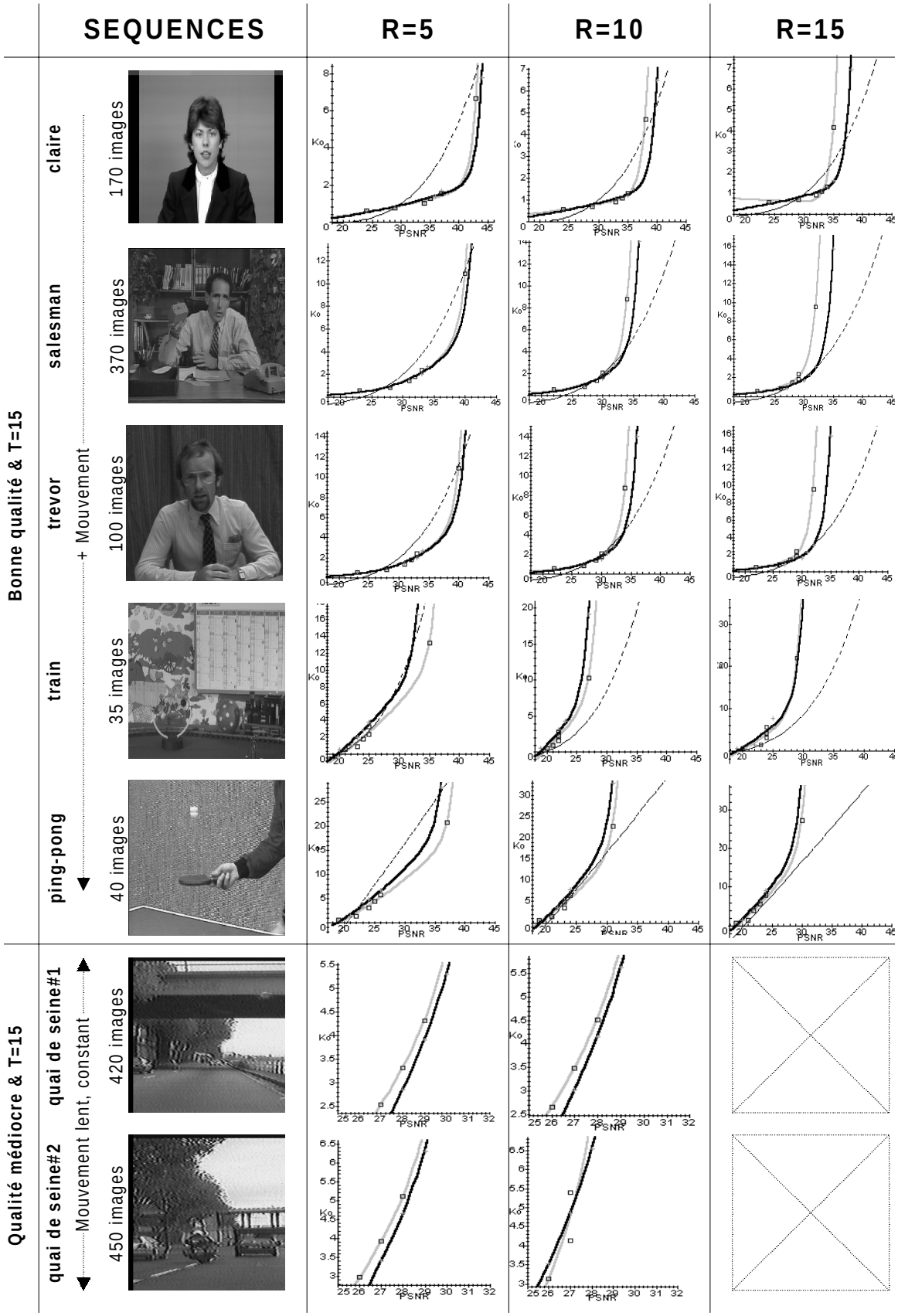
Les deux dernières séquences possèdent un contenu moins texturé avec un mouvement lent et régulier. Pour ces exemples de plus de 400 images, notre procédé montre des performances globalement constantes et meilleures d'un facteur 1/6 pour *R=5* et *R=10* et l'intervalle *D* compris entre 27 et 29 dB.

Pour permettre d'analyser ces résultats d'une manière plus fine, nous proposons d'introduire la mesure *G* de la performance globale entre deux courbes *R(D)* que nous exprimons dans un intervalle de distorsion donné et pour une valeur du seuil d'activité/de mobilité fixée. Cette mesure s'exprime avec le rapport d'intégrales suivant :

Equation 4-26. Mesure comparée de la performance de courbe *R(D)*

$$G(psnr_1,psnr_2) = \frac{\int_{psnr_1}^{psnr_2} R_{Grad/JPEG}(x)\cdot dx - \int_{psnr_1}^{psnr_2} R_{Markov}(x)\cdot dx}{\int_{psnr_1}^{psnr_2} R_{Grad/JPEG}(x)\cdot dx}$$

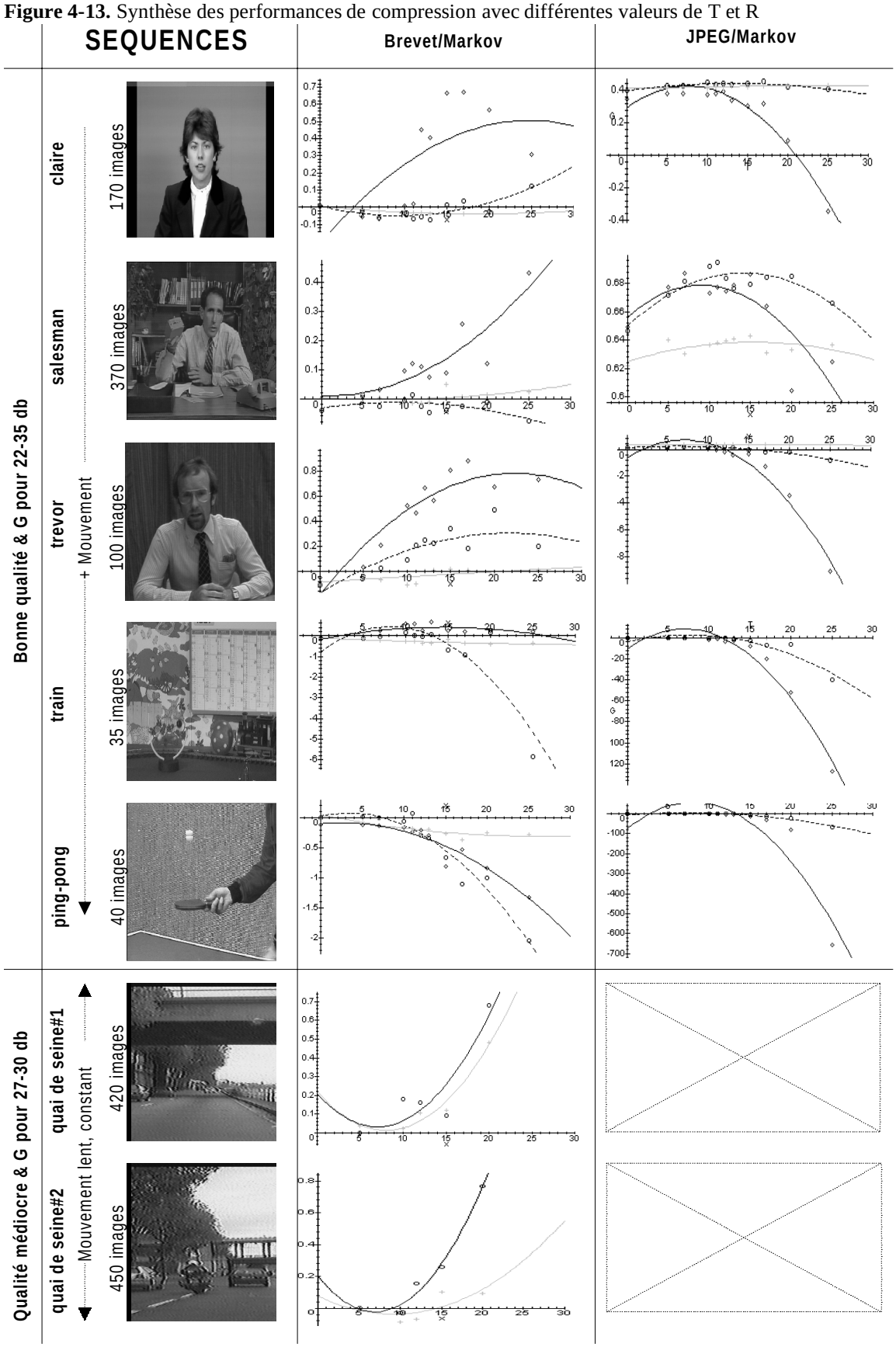
Figure 4-12. Performance des procédés de compression



Pour l'intervalle [*psnr1...psnr2*] associé à certaines valeurs de *Q*, ce ratio nous permet de mesurer le gain *R(D)* entre la méthode du gradient fréquentiel ou celle du JPEG seul par rapport à notre procédé. Cette mesure nous permet de réduire d'une dimension l'espace des paramètres et d'introduire alors de nouvelles courbes qui mettent en œuvre la mesure *G* du gain de notre approche pour les différentes valeurs de R et pour l'ensemble des valeurs données au seuil de mouvement. Ces courbes sont regroupées dans la Figure 4-13 et introduisent l'interpolation des courbes de la figure précédente pour l'intervalle de qualité compris entre 22 et 35dB (qui correspond à des facteurs de qualité faible, généralement *Q<50*). Ce calcul s'appuie bien entendu sur les équations des courbes interpolées. Une valeur de *G* positive nous indique que notre procédé affiche un meilleur taux *R(D)* pour l'intervalle de qualité donné que la technique du brevet ou celle que permettrait la seule utilisation du JPEG. A l'inverse, des valeurs négatives correspondent à une aire de la surface *R(D)* considérée plus petite pour le premier terme de la différence au numérateur, ce qui traduit le fait que notre procédé n'apporte pas de gain par rapport à la courbe *R(D)* comparée. Pour les 5 premières séquences, nous distinguons alors 3 nuages de points associés à chaque graphique. La colonne de gauche montre le gain *G* en ordonnée qu'engendre notre procédé par rapport à l'approche TDF pour différentes valeurs *T* de l'abscisse. A droite, nous montrons le gain affiché par notre approche en fonction de la seule compression JPEG. Pour chaque graphique, nous comptons 3 nuages de points de symboles différents que nous associons aux valeurs de *R* (5/10/15). Ces nuages de points sont interpolés par une fonction linéaire cubique permettant ainsi de mieux détacher l'allure de l'évolution de *G*. En gris clair, nous montrons la répartition du gain pour *R=5* (points en croix) ; en pointillés, celle pour *R=10* (points circulaires) et en traits noirs pleins, la courbe de la répartition pour *R=15* (losanges).

Ces courbes montrent tout d'abord que le gain *G* évolue suivant les valeurs données au seuil de mouvement qui ne transparaît pas avec la précédente figure où les courbes sont uniquement données pour *T=15*. Ensuite, nous pouvons mettre en avant que vis-à-vis de la comparaison brevet/markov, pour les 3 premières séquences, les valeurs positives de *G* en notre faveur, sont maximales pour *R=15* avec *T* proche de 15.

Figure 4-13. Synthèse des performances de compression avec différentes valeurs de T et R



Par ailleurs, l'évolution des nuages diverge selon les séquences mais montre un gain pouvant dépasser 50% dans le contexte que nous venons d'évoquer. Ces données semblent ainsi confirmer une amélioration des performances pour les séquences ayant un fond fixe.

Pour la comparaison JPEG/Markov, les meilleures valeurs de *G* positives sont mesurées pour *R=10* et pour *T* variant autour de 15 pour les deux premières séquences et avec *T<10* pour la troisième. Pour cette séquence, nous mesurons que notre approche offre des performances moins bonnes que pour le JPEG à partir de *T>12* pour les trois valeurs de *R*, ce que nous vérifions avec la Figure4-12. Les autres courbes de ce jeu de 5 séquences montrent des allures comparables à ce dernier graphique et confirment notamment que pour la comparaison brevet/Markov, notre approche est moins efficace pour la plupart des valeurs de *T* et *R*. Pour la comparaison JPEG/Markov, nous constatons également que pour *R=15*, notre approche dégrade sensiblement les performances du JPEG avec un gain *G* affichant des performances négatives d'un facteur inférieur à -100 si nous respectons le taux de croissance de la courbe *R(D)* initialement interpolée. En revanche, pour les deux dernières séquences où seule apparaît la comparaison brevet/markov (les simulations éffectuées lors de la synchronisation dichotomique des paramètres *T* et *A* sont très longues), nous constatons que pour *R=10* et *R=15,* notre procédé affiche des gains allant jusqu'à 60% pour *T=20*.

Nous pouvons alors conclure qu'à dégradation égale, notre procédé est également capable d'améliorer les taux de compression de ce type de séquence suivant la nature du mouvement (dans le deuxième cas, il est lent et régulier alors qu'il est rapide pour les séquences *train* et *ping-pong* qui comptent, de surcroît, plus de dix fois moins d'images) ou suivant la nature même des données en mouvement (les séquences *train* et *ping-pong* sont plus texturées).

4.5.2.2 .Analyses qualitatives comparées

Si l'ensemble des résultats qualitatifs montrent que les deux méthodes affichent globalement des performances comparables, la qualité visuelle des images reconstruites par notre procédé est, en revanche, supérieure et ce, toujours dans le contexte où les images de référence sont identiques pour les deux procédés. Soulignons également que notre principe de remise à jour intelligente de la référence vient encore améliorer les taux de compression en minimisant la dégradation.

Pour illustrer l'amélioration visuelle, la Figure4-14 montre la reconstruction du flux selon les techniques proposées et entre les deux images de référence 1 et 2. Ces deux images JPEG (comprimées/décomprimées) sont respectivement prises à l'instant t et $t'=t+R$. Les images *a*, *b* et *c* correspondent à trois images intermédiaires issues de la reconstruction selon la méthode TDF. Ici, nous voyons nettement apparaître des effets de blocs très gênants qui traduisent le manque de robustesse du calcul de mobilité (que nous pouvons résumer à la simple estimation de la variation d'intensité lumineuse). Cette constatation explique alors que des blocs de l'image de référence ne soient pas correctement modifiés par l'information de différence. Par opposition, les images *A, B* et *C* reconstruites selon notre approche n'affichent pas de tels défauts. Ici, le seuil d'activité (T=12) et celui de mobilité (*A=4687*) engendrent une quantité globale de zones en mouvement de près de 37% pour les deux procédés. Par ailleurs, nous constatons que la mesure du PSNR est proche pour les deux méthodes, la taille du flux étant, en revanche, supérieure de 15% pour notre procédé. La Figure 4-15 met en avant les différences binaires seuillées entre l'image reconstruite par les deux techniques et ce que produit la compression et décompression de l'image #106 de la séquence (juste avant la mise à jour de l'image de référence). Cette analyse fait principalement ressortir les artefacts de blocs mais montre également la répartition spatiale des zones masquées (en blanc). Si les densités sont comparables (hormis pour la ligne des blocs erronés), nous remarquons une répartition spatiale divergente du masquage de mouvement avec, notamment, une dégradation plus grande pour la zone de la chaussée selon l'approche TDF. Nous pouvons justifier cette constatation par le fait que la détection du mouvement de cette région est délicate de part l'homogénéité des données et souligne encore par là même la plus grande robustesse de notre approche.

Figure 4-14. Exemple de reconstruction : pourcentage de zone en mouvement identique

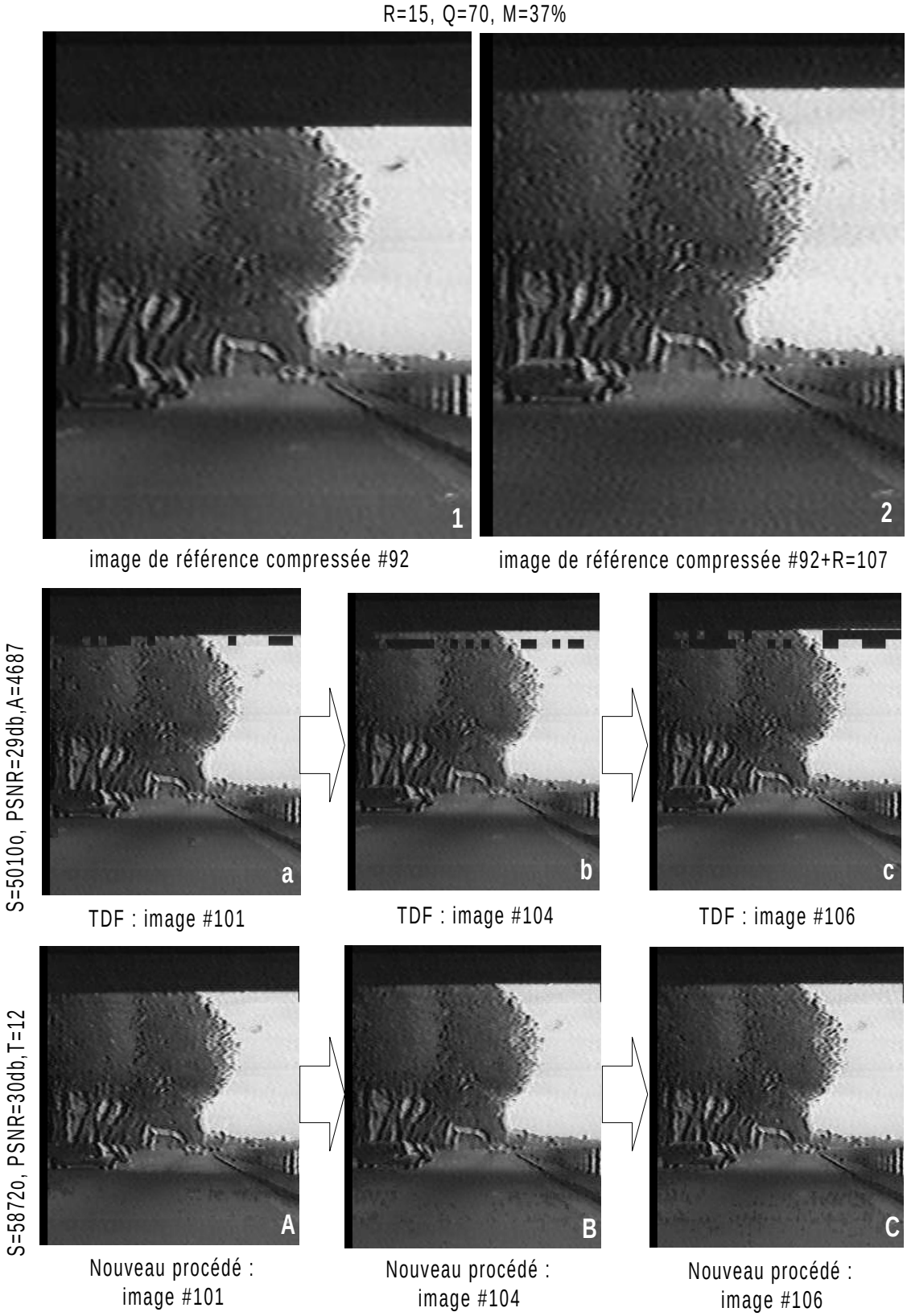
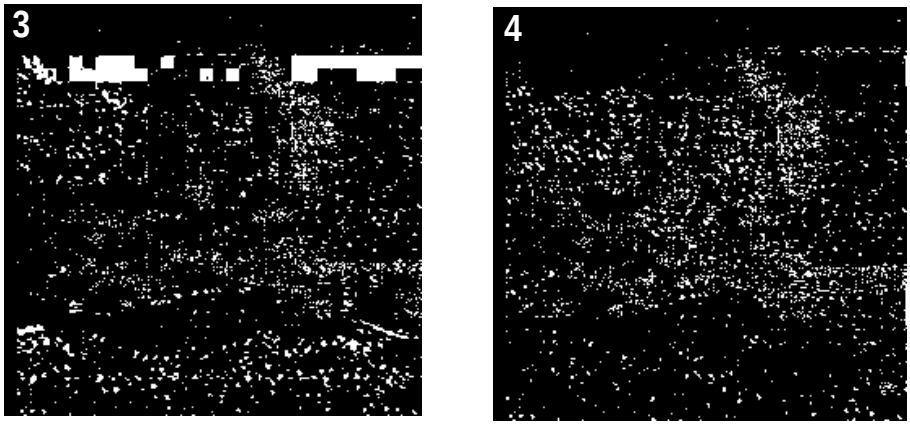


Figure 4-15. Différences seuillées des zones de dégradation

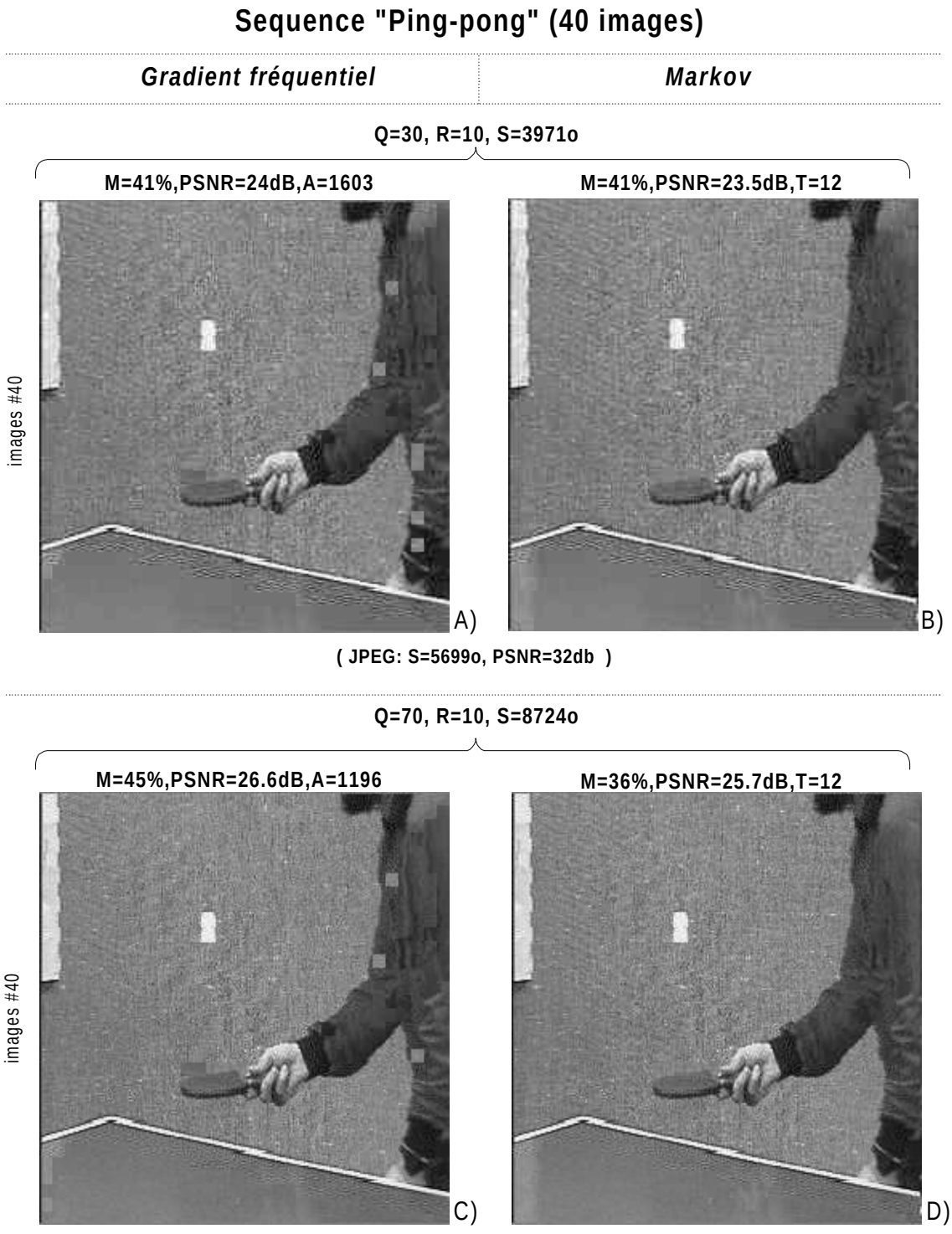


De plus, l'expérience montre que nous pouvons faire les mêmes remarques sur cette séquence lorsque les données sont comparées avec des tailles de flux identiques. Ce critère de comparaison s'avère, en effet, être naturellement plus pertinent dans un contexte d'évaluation de la qualité visuelle de reconstruction. Dans ce contexte là encore, la mesure du PSNR moyen est très proche et la même robustesse de masquage de la différence caractérise notre approche (il n'y a pas d'artefacts de blocs). Ces résultats sont illustrés sur d'autres séquences avec les figures qui suivent. Ils traduisent le très bon comportement de notre algorithme ce malgré, nous le rappelons, l'utilisation d'une seule passe d'ICM.

Pour illustrer cette remarque, la Figure4-16 montre donc un autre exemple de reconstruction avec la séquence *ping-pong* pour laquelle les deux méthodes de compression engendrent une taille de flux de 4Ko avec un facteur de qualité faible (*Q=30*) et de près de 9 Ko avec une quantification moins importante (*Q=70*). Pour les deux mesures de *Q*, les procédés affichent un PSNR moyen très proche alors que notre approche efface totalement les artefacts de blocs liés à la trop simple estimation de mobilité (résultats de la colonne de droite par opposition à ceux de la colonne de gauche). Sur cet exemple de 40 images, nous prenons *R=10* et remarquons que l'approche de compression par masquage de différences permet d'améliorer le taux de compression par rapport à une compression JPEG de respectivement 30% et 20% pour *Q=30* et *Q=70*. A titre de comparaison, la compression sans masquage mais avec le signal différence suivant *R=10* augmente la taille du flux de

près de 10% (S=4382) pour un même rapport signal/bruit (23.7dB) avec *Q=30* et par rapport à *T=12*. Ce chiffre est réduit à 7% pour le facteur de qualité 70.

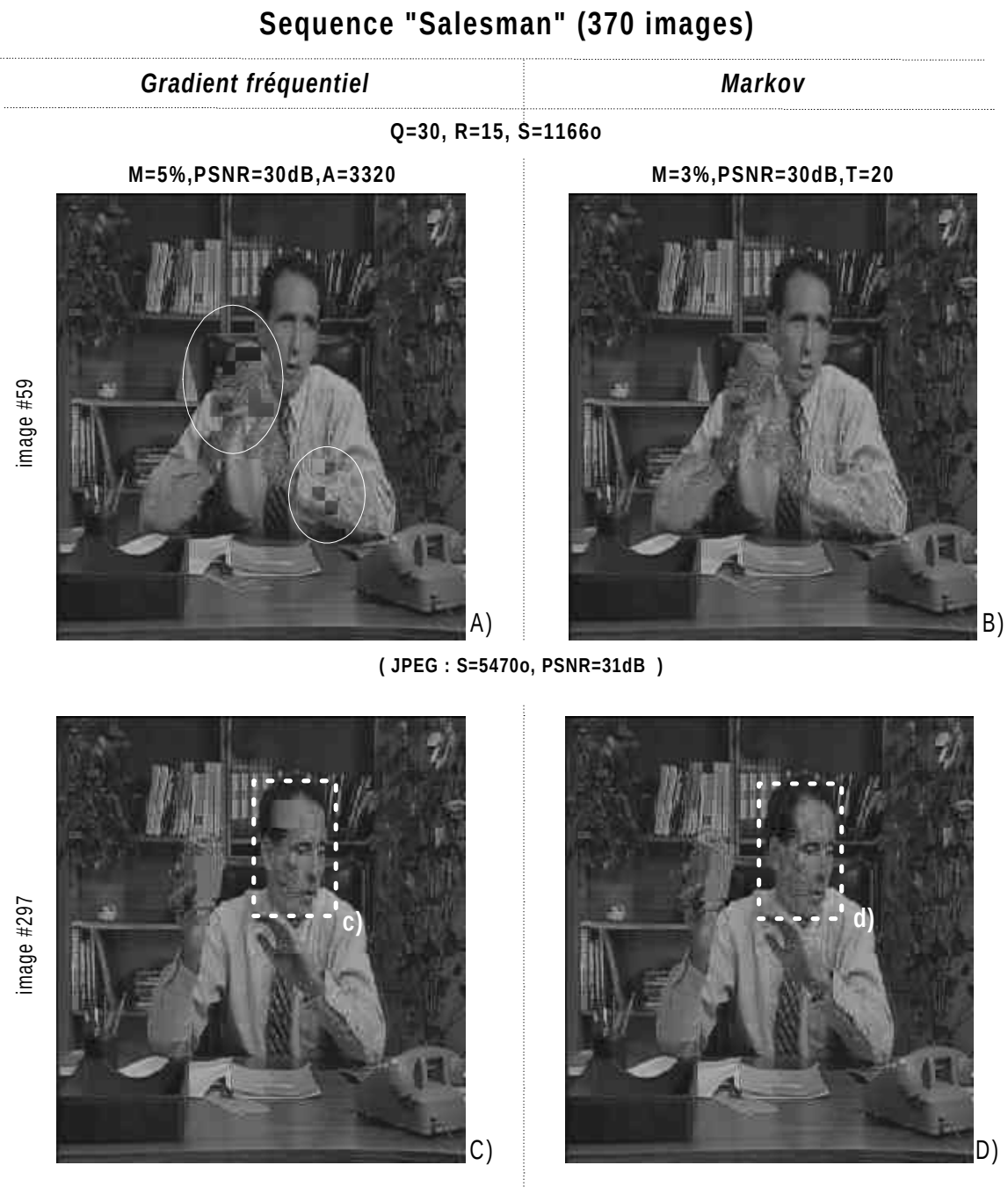
Figure 4-16. Qualité de reconstruction à taille de flux égale : séquence “ping-pong”



La figure suivante montre un autre exemple d’amélioration de la dégradation à taille de flux égale pour la séquence dite du *salesman*, plus longue. Cet exemple montre, pour un même

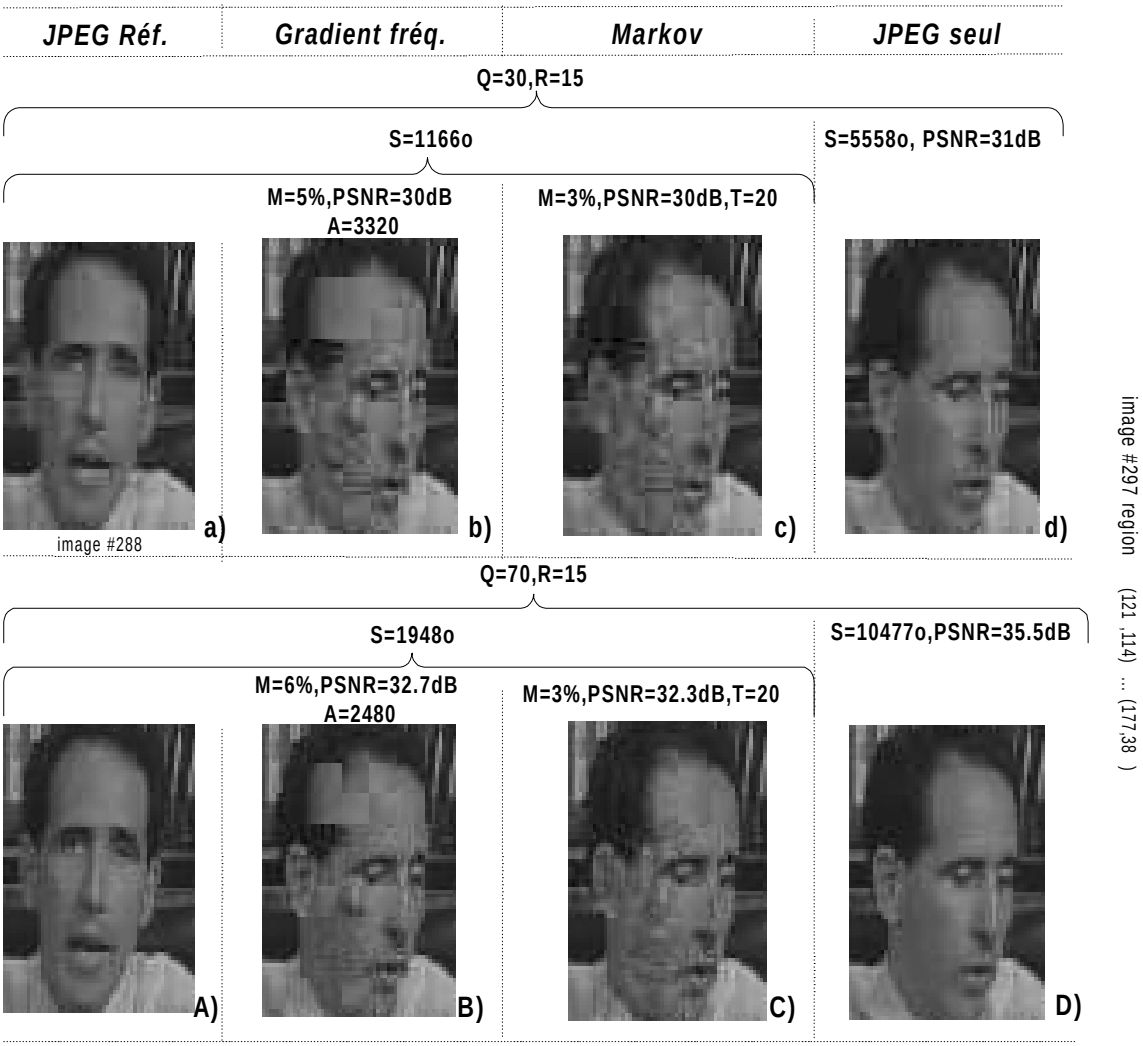
facteur de qualité *Q* de 30 et une fréquence de rafraîchissement *R* = 15, deux images traitées par les deux approches. Ici, le PSNR moyen est identique (30dB) pour les deux procédés et nous rencontrons à nouveau des effets de blocs sur l'image *A* (cerclés) introduits par le procédé TDF et inexistants avec la détection markovienne du mouvement (*B*).

Figure 4-17. Qualité de reconstruction à taille de flux égale : séquence "Salesman"



Les images C et D permettent de mettre en évidence le même type de déformation. Nous proposons avec la Figure 4-18 d'étudier l'agrandissement des régions *c* et *d* respectives de ces images, afin d'analyser dans le détail la qualité de reconstruction du visage en mouvement. Pour cette expérience, nous fixons volontairement une valeur élevée du seuil de mouvement *T* (20) pour favoriser la distorsion. Les vignettes *a, b, c* et *d* de la figure suivante montrent les résultats obtenus avec les diverses techniques de compression et ceci pour deux valeurs du facteur de qualité (*Q = 30* et *Q = 70* avec les imagettes *A, B, C, D*). L'image *a* correspond à l'image de référence comprimée à un instant *t* et *d* montre le résultat de l'image courante (à $t+R > t' > t$) comprimée selon la norme JPEG (sans masquage de différence).

Figure 4-18. Gros plan sur quelques défauts de reconstruction



Nous pouvons alors comparer les résultats *b* et *c* correspondant, respectivement, à l'image courante (*t'*) reconstruite suivant le procédé TDF et suivant notre nouvelle méthodologie, le résultat idéal correspondant alors à l'image *d.* Nous soulignons, cependant, que l'image JPEG idéale engendre une taille moyenne de flux 5 fois plus grande qu'avec les deux procédés proposés pour les deux facteurs de qualité.

La comparaison des images *b* et *c* montre que de nombreux blocs de l'image de référence n'ont pas été mis à jour avec l'approche TDF et ceux, notamment, au niveau du front, du nez et de la bouche. Si nous nous intéressons maintenant aux images *A, B, C* et *D* pour le facteur de qualité supérieure, nous constatons que les artefacts de bloc pour la méthode TDF sont moins nombreux mais subsistent.

Pour tous ces exemples, notre approche montre une amélioration sensible de la qualité visuelle de reconstruction avec le paradoxe qu'elle ne transparaît pas nécessairement sur la mesure du PSNR. Nous soulignons, également, l'intérêt du masquage de l'information de différence. Sur ce dernier exemple, nous mesurons ainsi, pour *R=15*, une amélioration du taux de compression de la seule différence JPEG de près de 15% pour *Q=30* et de près de 35% pour *Q=70* avec le masquage des zones jugées statiques (suivant *T=20*). Nous pouvons également souligner que le niveau pixélique auquel agit notre approche, permet de réduire les artefacts de bloc qui apparaissent même pour les blocs dont la mobilité est correctement détectée avec l'approche TDF. Ces artefacts apparaissent notamment pour les forts taux de compression et correspondent à la mise à jour des parties du plan fixe qui appartiennent à des blocs dont seul un sous-ensemble de données apparaît effectivement suffisamment modifié (ou mobile). Si nous considérons le principe de quantification des coefficients fréquentiels issus de la transformée DCT, la nature de la répartition statistique des données en mouvement peut, en effet, fausser l'intensité reconstruite des zones du plan fixe qui appartiennent à ces blocs. Nous pouvons alors constater un effet de "blocs" rehaussé par une intensité des zones du fond qui n'est pas homogène avec la couleur de l'arrière plan issue de l'image de référence. Ce phénomène se constate notamment au niveau du pourtour (de la silhouette) des objets introduits de manière sporadique dans le champ visuel. Notre procédé efface également ce type de déformation.

4.6 Conclusions

Dans cette partie, nous avons tout d'abord détaillé la mise en œuvre d'une technique de détection de mouvement robuste sur la base de notre méthodologie de gestion de flux. Deux résultats significatifs se dégagent. Tout d'abord, nous montrons l'adéquation de notre approche pour la gestion du DMA dans le cadre d'une application complexe relevant du bas niveau. Ensuite, nous mentionnons les bonnes performances en terme de vitesse de traitement de notre implantation que nous devons à une utilisation optimisée des ressources de l'architecture du C80 sur plusieurs plans.

Fort de ces résultats, nous proposons une application plus lourde mettant en œuvre un principe de compression original s'appuyant sur l'implantation de la détection de mouvement. Nous montrons la faisabilité d'un tel système embarquable à base de C80 et montrons sa capacité à traiter une séquence d'images numérisées à la volée en temps réel. Ensuite, sur la base d'un prototype fonctionnel, nous détaillons les performances quantitatives et qualitatives de l'approche proposée et ce, notamment, face à une technique concurrente issue d'un brevet industriel. Les résultats de cette comparaison montrent notamment l'apport de la qualité visuelle de la reconstruction des images compressées qu'offre la solution proposée et souligne par conséquent tout l'intérêt de la démarche de mise en œuvre. Ces travaux font l'objet d'un dépot de brevet.

Conclusions

> *The performance increases over the past 15 years have been truly amazing, but it will be hard to continue these trends by sticking to the basically evolutionary path we are on. We need to rethink the types of problems we should be addressing in terms of how we exploit parallelism within processors and how we build memory hierarchies. We need to move towards a more integrated approach that doesn't treat hardware and software as separate entities. And we need to look for new focuses for research.* John Hennessy (Université de Standford), août 1999, [117].

Notre conclusion générale s'articule en trois points. Tout d'abord, nous établirons une synthèse des apports scientifiques de nos travaux. Ensuite, nous résumerons l'intérêt de nos recherches théoriques sur le plan de l'ingénierie des applications développées dans ce mémoire. Enfin, nous présenterons les perspectives à court, moyen et long terme de nos recherches.

5.1 L'apport scientifique

Dans ce mémoire, nous avons souligné l'intérêt des processeurs de traitement de signal parallèle avec l'exemple du C80 et moins directement, celui du C62, comme cibles de nos travaux pour le traitement d'images. Nous avons évoqué la difficulté de mise en œuvre des architectures récentes et proposé des pistes pour faciliter l'étape de programmation.

Dans cette optique, nous avons mis l'accent sur la nécessité de faire évoluer les techniques et outils pour la gestion des flux avec le paradigme des co-processeurs DMA associés aux mémoires de données internes qui correspondent à des caractéristiques architecturales récurrentes sur les DSP. Plus précisément, nous avons mis l'accent sur le besoin de méthodes d'aide à la programmation pour ce paradigme matériel. Ces méthodes permettent

d'appréhender de manière générique la réalisation de chaînes algorithmiques aussi variées que possible. Nous avons également souligné le besoin de flexibilité en terme de taille d'images traitées, de nombre de processeurs assignés de manière parallèle au traitement et de reconfiguration dynamique de l'exécution des algorithmes. Enfin, nous avons mis en avant le besoin d'une approche visant à optimiser le coût des entrées/sorties afin de favoriser des performances temps réel.

De là, nous avons confronté ces trois grands objectifs à l'état de l'art pour constater qu'à notre connaissance, aucune approche ne répondait de manière satisfaisante à ces pistes pratiques compte tenu, notamment, des spécificités du multi-processeur C80 et du domaine applicatif visé : la vision temps-réel embarquée.

Partant de ce constat, nous avons choisi de développer une méthodologie originale de mise en œuvre des DMA visant à automatiser et à optimiser la gestion des flux pour des applications relevant du traitement d'images bas niveau. L'apport scientifique de ce travail réside dans l'extension mathématique que nous avons donnée à la modélisation traditionnelle par flots de données synchrones avec le support de flots bi-dimensionnels dans un contexte multi-processeurs. L'originalité de notre approche réside essentiellement dans le lien que nous avons établi entre la description synchrone d'algorithmes 2D et le partitionnement des données qu'induit la recherche d'un parallélisme de type SPMD. L'objectif final de cette démarche consiste à mettre en œuvre une programmation transparente de l'ensemble des requêtes de transfert DMA en fonction de la géométrie du partitionnement qui se trouve contrainte par le synchronisme des flux.

La démarche mathématique pour l'encapsulation des flux que nous avons unifiée autour de la notion de patron de données a permis d'apporter une réponse favorable à l'objectif de généricité. Au niveau de la flexibilité, nous avons également atteint nos objectifs avec le développement de modèles ayant comme variables la taille des images et le nombre de processeurs assignés au traitement.

Sur le plan de l'amélioration des performances, nous avons initialement fondé notre approche sur l'idée de chaîner l'exécution des opérateurs de traitement afin d'améliorer la localité des données. Pour que le gain soit effectif, l'originalité de notre travail s'explique aussi par l'intégration, dans la modélisation, des techniques visant à diminuer la granularité

des opérateurs pour réduire le surcoût de la gestion des caches induit par le chaînage. Cet aspect d'importance transparaît avec le modèle de performance théorique que nous proposons en accord avec notre méthodologie de gestion des flux.

Sur le plan académique, l'ensemble de ce travail nous a permis de dépasser plusieurs limitations actuelles dans l'exploitation des processeurs de traitement de signal parallèle, ce qui répond à la problématique scientifique que posait notre cahier des charges initial.

5.2 L'apport des applications

L'autre volet de notre étude relève davantage du domaine des applications. La réalisation d'une librairie de traitement d'images bas niveau reposant sur près de 25.000 lignes de programme nous a permis de montrer l'intérêt de notre travail théorique. A partir de celle-ci, nous avons développé l'implantation de deux applications : l'une adressant la problématique de la détection optimale de contours et l'autre celle d'une nouvelle technique de compression vidéo. En atteignant ces trois objectifs pratiques, nous avons, là aussi, répondu aux spécifications du cahier des charges.

Mais plus encore, nous avons souligné que notre librairie comparée à l'état de l'art apparaît unique dans la grande productivité et flexibilité qu'elle offre au programmeur avec, notamment, le support de la reconfiguration dynamique, la gestion des tailles d'images variables et du nombre paramétrable de processeurs assignés de manière très souple à la parallélisation des traitements. Au troisième chapitre, nous avons montré que le support de ces caractéristiques s'est révélé d'un grand intérêt à la fois pour la validation des concepts entourant la mise en œuvre transparente du DMA et pour l'effort de vérification de notre modèle de performance plus particulièrement développé autour de l'application de détection de contours.

Dans le dernier chapitre, nous avons également pu mettre en avant le rôle de cette librairie et des méthodes sous-jacentes dans l'implantation de notre deuxième application de référence pour la compression vidéo. C'est bien grâce à cette infrastructure logicielle réfléchie et opérationelle que nous avons pu nous concentrer davantage sur les aspects

algorithmiques de l'application ciblée et que nous avons proposé, dans le temps imparti, une technique originale pour la compression de séquences.

Enfin, notre travail précurseur sur la comparaison des performances sur RISC, C80 et C62 de l'algorithme de détection de contours de FGL ([87][88]) a, nous semble-t-il, stimulé une véritable prise de conscience auprès de la communauté scientifique et industrielle des nouvelles capacités des processeurs généralistes actuels. Nous avons pu, en effet, montrer que l'implantation d'algorithmes de traitement de signal affiche des performances comparables aux résultats obtenus sur FPGA et d'un ordre de grandeur inférieur à 10 par rapport aux ASIC spécialisés très lourds à réaliser. Ce niveau de performances est atteint avec l'utilisation de techniques d'optimisation que nous avons appliquées à différents niveaux de granularité (en assembleur sur DSP et en C sur RISC) mais qui sont bien génériques pour les deux niveaux d'approches et encore insuffisamment intégrées aux chaînes de compilation. Ces conclusions sur le plan matériel et logiciel nous apparaissent comme très importantes pour les orientations de recherche actuelles alors qu'elles ne sont, paradoxalement, que très rarement publiées, sans doute du fait de l'ampleur du travail qu'elles nécessitent ou des enjeux qu'elles représentent.

5.3 Les perspectives de nos recherches

Nos recherches sur l'optimisation des flux ont tout d'abord permis d'apporter une réponse favorable au défi indirectement posé de l'ingénierie d'une librairie pour l'imagerie sur le C80 qui se devait d'améliorer les réalisations existantes avec des moyens humains réduits. Les retombées à court terme pourraient se situer chez les industriels qui l'exploitent encore le C80 comme les sociétés Matrox, Pinnacle, TDF ou Matra.

Par ailleurs, la mise au point de notre technique brevetée de compression d'images a tout d'abord permis de montrer l'adéquation de notre méthodologie avec un exemple complexe d'application de traitement d'images. Sur le plan algorithmique, et grâce au prototype industriel réalisé avec notre librairie C80, notre travail a présenté des améliorations quantifiées à un problème pratique posé. Ces améliorations sont suffisamment intéressantes

pour que nous puissions envisager, à court terme, l'application de notre approche à d'autres techniques de codage (nous pensons, notamment, à la norme JPEG2000) avec des débouchés dans les domaines de la post-production vidéo, des magnétoscopes et appareils photos numériques ou des systèmes de vidéo surveillance/conférence qui, pour l'instant, reposent essentiellement sur le format M-JPEG. En analysant les performances actuelles des processeurs embarquables et la complexité (voire lourdeur) de la norme MPEG-4 pour la compression du mouvement, il semble que l'intérêt à court terme des techniques M-JPEG puisse être établi avec des retombées industrielles clairement définies.

Devant la graduelle obsolescence du C80, nous avons également très tôt anticipé l'architecture montante du C60 et publié les premiers résultats de l'utilisation de techniques d'optimisation génériques sur des exemples précis vers la programmation véritablement performante des nouvelles générations d'architectures ([86][87][88]). Le pari que nous avons initialement relevé en 1996 d'étudier le parallélisme de type VLIW avec les PP du C80 puis à l'aide du C62 semble aujourd'hui totalement tenu avec l'éventail d'architectures de cette nature actuellement offert :

Table 5-1. Les architectures de type VLIW en 2000

Contructeur	Architecture {date d'introduction commerciale}
Motorola	StarCore140 (anciennement StarCore 440) {début 2000}
Fujitsu	FR500 {2000}
Siemens	Carmel {1998}
Analog Device	TigerSharc {début 2000}
Texas Instruments	TMS320C6X {1997}, TMS320C8X {1995}
Intel	Merced {2000}
Philips	Trimedia {1996}, RD1602X {1995}

Aussi, nous pensons que notre réflexion sur l'adéquation de notre méthodologie de gestion de flux pour le C6X ainsi que nos remarques apportées au constructeur (Texas Instruments) sur les plans logiciel et matériel ont permis de faire évoluer cette nouvelle famille de DSP vers des systèmes plus souples et mieux adaptés au domaine du traitement d'images visé par nos travaux (exemple du double buffering avec le C6202 et ses 2 jeux de 2×4 bancs de

mémoire entrelacée). L'exploitation de ces nouvelles caractéristiques offre des perspectives très attrayantes à moyen terme.

Pour clore ce bilan très positif, nous souhaitons enfin souligner l'intérêt plus général à moyen/long terme de notre travail qui a contribué à montrer la viabilité des architectures mono-composant multi-processeurs. Ainsi, alors que le C80 apparaissait comme un DSP vieillissant il y a 2 ans, les résultats de nos recherches pour ce composant nous permettent d'anticiper le futur des systèmes intégrant plusieurs coeurs de traitement. A ce titre, certaines approches actuelles autorisent déjà la composition de composant multi-processeurs parallèles avec l'exemple de [118]. De même, nous assistons à un regain d'intérêt pour ce type d'architectures comme le montre la récente offre d'Analog Devices concernant une technologie de DSP "à la carte" capable de regrouper plusieurs coeurs du nouveau DSP 219X [119] et celle concurrente de Texas Instruments - le TMS320VC5441- capable de regrouper 4 coeurs de type C54X (annonce faite au début 2000). Un autre exemple apparaît également avec l'architecture "MAJC" de Sun intègrant 2 coeurs VLIW connectés à la mémoire interne via un crossbar. L'essor actuel de la notion plus générale de systèmes SOC (*System On-a-Chip*), en grande partie justifiée par le formidable essor de la téléphonie mobile, semble également marquer le retour du besoin de composants embarquables intégrant plusieurs blocs de traitement interconnectés. Très récemment, dans une présentation intitulée "The Rebirth of DSP: Communications Applications of DSP-based SOC Technologies" [120], le Dr. Pinto de la société Motorola a souligné qu'il s'agirait bien là d'un axe d'évolution majeur pour la technologie des DSP.

Par opposition, la difficulté de programmation de ces composants est en mesure d'inhiber l'expansion de ce type de systèmes et l'échec commercial du C80 dû à une complexité de mise en œuvre trop grande apparaît être un exemple significatif. C'est pourquoi une réflexion de fond sur les méthodes et techniques de développement est véritablement nécessaire pour faire de ces architectures un succès.

Ainsi, des pistes très intéressantes émergent grâce à des ateliers de génie logiciel pour le traitement de signal tel que SIMULINK et bien d'autres, qui introduisent les notions très prisées de programmation graphique et de génération de code automatique. Cependant, ce type d'approche est encore très souvant restreint aux signaux mono-dimensionnels dans le cas d'architecture mono-processeur même si des initiatives d'extensions aux signaux multi-dimensionnels apparaîssent progressivement (exemple du projet Mustig [121]). A partir de ce constat, il nous semble que les travaux présentés dans ce manuscript offrent de nouvelles perspectives pour une génération de code véritablement performante d'algorithmes 2D et vers des systèmes multi-processeurs complexes.

Bibliographie par Chapitre

Introduction

[1] L. Lacassagne, *Détection de mouvement et suivi d'objets en temps-réel sur RISC et DSP* (titre provisoire), Thèse de Doctorat de l'Université Paris 6 (Pierre et Marie Curie) à paraître (courant 2000)

Chapitre 1

[2] J. Fridman, W.C. Anderson, *A New Parallel DSP with Short-Vector Memory Architecture*, ICASSP99, 24th IEEE International Conference on Acoustics, Speech and Signal Processing, Phoenix, Arizona,15-19 Mars 1999, Vol. 4, article N°2317

[3] J.L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach, Second edition*, International Thomson Publishing France, ISBN 2-84180-022-9

[4] H. Shi, R. Arnold, K. Westerholz, *C/C++ Compiler Support for Siemens Tricore DSP Instruction Set*, ICASSP99, 24th IEEE International Conference on Acoustics, Speech and Signal Processing, Phoenix, Arizona,15-19 Mars 1999, Vol. 4, article N°1108

[5] M. Lam, E. Rothberg, M. Wolf, *The Cache Performance and Optimizations of Blocked Algorithms*, ASPLOS91, Proceedings of the 4th ACM Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, Californie, 8-11 Avril 1991, p. 63-75

[6] KAP, Kuck & Associates, Inc (KAI), www.kai.com/kap

[7] A.D. Robison, *Method of replacing Ivalues by variables in programs containing nested aggregates in an optimizing compiler*, Brevet Industriel Américain N° 5790866, société KAI, Août 1998

[8] B. Case, *Philips hopes to displace DSPs with VLIW*, Microprocessor Report, 5 Décembre 1994, Vol. 8, Num. 16, p. 12-15

[9] L. Gwennap, *Alpha 21364 to Ease Memory Bottleneck*, Microprocessor Report, 26 Octobre 1998, Vol. 12, Num. 14, p. 12-15

[10] Michael R. Smith, *How RISCy is DSP ?*, IEEE Micro, Décembre 1992, Vol. 12, Num. 6, p. 10-23

[11] M. Schlett, *A New Way of Integrating RISC and DSP*, ICSPAT96, 7th International Conference on Signal Processing Applications and Technologies, Boston, MA, 8-10 Octobre 1996

[12] J. Fridman, W.C. Anderson, *A New Parallel DSP with Short-Vector Memory Architecture*, ICASSP99, 24th IEEE International Conference on Acoustics, Speech and Signal Processing, Phoenix, Arizona,15-19 Mars 1999, Vol. 4, article N°2317

[13] P.N. Glaskowsky, *Philips Advances TriMedia Architecture*, Microprocessor Report, 26 Octobre 1998, Vol. 12, Num. 14, p. 33-35

[14] André Seznec, Thierry Lafage, *Evolution des gammes de processeurs MIPS, DEC ALPHA, POWERPC, SPARC, X86 et PA-RISC*, Rapport de Recherche de l'Institut National de Recherche en Informatique et Automatique (INRIA), Rennes, France, Num. RR-3188, Juin 1997, 154 pages

[15] L. Gwennap, *G4 Is First PowerPC With AltiVec*, Microprocessor Report, 16 Novembre 1998, Vol. 12, Num. 15, p. 17-19

[16] S.M. Joshi, *Some Fast speech processing algorithms using altivec technology* , ICASSP99, 24th IEEE International Conference on Acoustics, Speech and Signal Processing, Phoenix, Arizona,15-19 Mars 1999, Vol. 4, article N°1477

[17] André Seznec, Fabien Lloansi, *Etudes des architectures des microprocesseurs MIPS R10000, Ultrasparc et Pentium Pro*, Rapport de Recherche de l'Institut National de Recherche en Informatique et Automatique (INRIA), Rennes, France, Num. RR-2893, Mai 1996, 123 pages

[18] www.intel.com, *Intel Architecture Optimization Manual* , Documentation constructeur num. 242816-003

[19] M. Lam, E. Rothberg, M. Wolf, *The Cache Performance and Optimizations of Blocked Algorithms*, ASPLOS91, Proceedings of the 4th ACM Conference on Architectural Support for Programming Languages and Operating Systems, Santa Clara, Californie, 8-11 Avril 1991, p. 63-75

[20] M. Wolf, M. Lam, *A Data locality Optimizing Algorithm* , SIGPLAN91, ACM Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, 26-28 Juin 1991, Vol. 26, Num. 6, p. 30-40

[21] O. Temam, E. Granston, W. Jalby, *To Copy or Not to Copy : A compile Time Technique for Assessing When Data Copying Should be Used to Eliminate Cache Conflicts*, ICS'93, Proceedings of IEEE Supercomputing'93, Portland, Oregon, 15-19 Novembre 1993, p. 410-419

[22] D. Truong, F. Bodin, *A. Seznec, Improving cache behavior of dynamically allocated data structures*, PACT'98, International Conference on Parallel Architectures and Compilation Techniques (IEEE & IFIP), Paris, 12-18 Octobre 1998

[23] F. Bodin, C. Eisenbeis, W. Jalby et D. Windheiser, *A quantitative algorithm for data locality optimization*, Code Generation-Concepts, Tools, Techniques, Stringer Verlag 1992

[24] S.A. McKee, A. Aluwihare, K.L. Wright, Wm.A. Wulf, J.H. Aylor, *Design and Evaluation of Dynamic Access Ordering Hardware*, ICS'96, 10th ACM International Conference on Supercomputing, Philadelphie, USA, Mai 1996

[25] F. Bodin, A. Seznec, *Skewed Associativity Improves Program Performance and Enhances Predictability*, IEEE Transactions on Computers, Mai 1997, Vol. 46, Num. 5

[26] A. Agarwal, M. Horowitz, J. Hennessy, *An Analytical Cache Model,* ACM Trans. on Computer Systems, Vol. 7, Num. 2, Mai 19989, p. 184-215

[27] C. Fricker, Ph. Robert, *An Analytical cache model* , Rapport de Recherche de l'Institut National de Recherche en Informatique et Automatique (INRIA), Rocquencourt, France, Num. RR-1496, Septembre 1991, 27 pages

[28] Peter Soderquist, *Optimizing the Data Cache Performance of a Software MPEG-2 Video Coder*, ACM Multimedia Conference, Seattle, Novembre 1997, p. 291-301

[29] T. Takizawa, J. Tajine, H. Harasaki, *High Performance and Cost Effective Memory Architecture For an HDTV Decoder LSI* , ICASSP99, 24th IEEE International Conference on Acoustics, Speech and Signal Processing, Phoenix, Arizona,15-19 Mars 1999, Vol. 4, article N°1842

[30] R.M. Stasinski, *Efficiency of Radix-K transforms on computers with cache* , ICASSP99, 24th IEEE International Conference on Acoustics, Speech and Signal Processing, Phoenix, Arizona,15-19 Mars 1999, Vol. 3, article N°1224

[31] *www.intel.com/vtune*

[32] Eric Biscondi, *Wait-states on the TMS3206201 CPU Data accesses versus various memory types*, Documentation interne Texas Instruments : "TMS320 DSP Designer's NoteBook"

[33] J.Kim, G. Short, *Performance evaluation of register allocator for the advanced DSP of TMS320C80*, ICASSP98, 23th IEEE International Conference on Acoustics, Speech and Signal Processing, Seattle, Washington, 12-15 Mai 1998, Vol. 5, article N°1379, p. 3077-3080

[34] *Implementation of an Image Processing Library for the TMS320C8x (MVP)* , Documentation Texas Instruments N° BPRA059

[35] *TMS320C80(MVP): Parallel Processor User's Guide* , Documentation Texas Instruments N° SPRU110A, 1995

[36] University of Washington Image Computing Library (UWICL), *http:// icsl.ee.washington.edu/projects/iclib*

[37] W. Lee, Y. Kim, R.J. Gove, C.J. Read, *MediaStation 5000: Integrating Video and Audio*, IEEE Multimedia, 1994, Vol. 1, Num. 2, p. 50-61

[38] W. Lee, A.P. Alleman, D.M Parsons, Y. Kim, *UWGSP5: A multimedia workstation for medical applications*, SPIE Medical Imaging VIII, 1994, Vol. 2164, p. 344-351

[39] Luc Bougé, *The Data-Parallel Programming Model: a Semantic Perspective*, Rapport N°96-27 du Laboratoire de l'Informatique du Parallélisme (LIP), Ecole Normale Supérieure (ENS) de Lyon, France

[40] G. Authié, A. Ferreira, JL. Roch, G. Villard, J. Roman, C. Roucairol, B. Virot, *Algorithmes parallèles : analyse et conception*, Editions Hermès, Paris, 1994, ISBN 2-86601-414-6

[41] D.B. Skillicorn, D. Talia, *Programming Languages for Parallel Processing*, IEEE Computer Society Press, Los Alamitos, Californie, 1995, ISBN 0-8186-6502-5

[42] D.B. Loveman, *High Performance Fortran*, IEEE Parallel & Distributed Technology, Fevrier 1993, Vol.1, p. 25-42

[43] D. Dzierzgowski, *Quatre exemples de langages ou environnements pour le développement de programmes où le temps intervient, Revue T.S.I. - Technique et Science Informatiques*, N°0752-4082, édition AFCET-Bordas, 1990, Vol. 9, Num. 4, p. 289-312

[44] *Hypersignal Software Products - An Overview of Hyperception Software Products for Windows 98/95/NT* , Documentation commerciale N°HSMK1075, Société Hyperception, Janvier 1999

[45] A. Clouard, A. Pool, P. Tessier, O. Debon, J. Kulp, *CapCASE: A Graphical Development Tool Supporting Scalable, Heterogeneous Multicomputers*, ICSPAT96, 7th International Conference on Signal Processing Applications and Technologies, Boston, MA, 8-10 Octobre 1996

[46] Y. Sorel, *Massively Parallel Computing Systems with Real Time Constraints: The "Algorithm Architecture Adequation" Methodology* , Proc. Massively Parallel Computing Systems, the Challenges of General-Purpose and Special-Purpose Computing, Ischia, Italie, Mai 1994

[47] C. Aiglon, C. Lavarenne, Y. Sorel, A. Vicard, *Utilisation de SynDEx pour le traitement d'images temps-réel*, Rapport de Recherche de l'Institut National de Recherche en Informatique et Automatique (INRIA), Rocquencourt, France, Num. RR-2968, Septembre 1996, 80 pages

[48] C. Lavarenne, Y. Sorel, *Modèle unifié pour la conception conjointe logiciel-matériel*, revue Traitement du Signal Vol. 14, Num. 6, Numéro spécial "Adéquation Algorithme Architecture", 1997

[49] E.A. Lee, *Overview of the Ptolemy Project*, Université de Californie à Berkeley, Electronic Research Laboratory (ERL), Rapport de Recherche Num. M98/71

[50] *Ptolemy 0.7 almagest: User Manual,* Université de Californie à Berkeley, Département d'Ingénierie Electronique et de Sciences Informatiques (EECS), en ligne à l'adresse http://ptolemy.eecs.berkeley.edu

[51] S.S. Bhattacharyya, P.K. Murthy, E.A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Press, 1996, ISBN 0-79239-722-3

[52] E.A. Lee, *Multidimensional Streams Rooted in Dataflow*, IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism, Orlando, Floride, 20-22 Janvier 1993

[53] M.J. Chen, E.A. Lee, *Design and Implementation of a Multidimensional Synchronous DataFlow Environment*, IEEE Asimolar Conference on Signals, Systems and Computers, Pacific Grove, Californie, Octobre 1994

[54] P.K. Murthy, *Two cycle-related problems of Regular Data Flow Graphs: Complexity and Heuristics* , Eletronics Research Laboratory (ERL), Rapport de Recherche No. M97/76, 19 Novembre 1997

[55] W. Baetens, M. Adé, R. Lauwereins, *Porting GRAPE to the TMS320C80* , ICSPAT97, 8th International Conference on Signal Processing Applications and Technology, San Diego, Californie, 14-17 Septembre 1997, p. 1589-1593

[56] T.M. Parks, J.L. Pino, E.A. Lee, *A Comparison of Synchronous and Cyclo-Static Dataflow*, IEEE Asimolar Conference on Signals, Systems and Computers, Pacific Grove, Californie, 1 Novembre 1995

[57] Greet Bilsen, *Assignment and Scheduling of DSP Applications on Heterogeneous Multiprocessors*, Thèse de Doctorat de l'Université Catholique de Leuven, Belgique, Laboratoire ESAT/ACCA, 30 Mai 1996

[58] P.K. Murthy, *Multiprocessor DSP Code Synthesis in Prolemy* , Université de Californie à Berkeley, Eletronics Research Laboratory (ERL), Rapport de Recherche Num. M96/66, Août 1993

Chapitre 2

[59] *TMS320C62x/C67x Programmer's Guide*, Documentation Texas Instruments N° SPRU198, 1998

[60] A. Darte, G.-A. Silber, F. Vivien, *Combining retiming and scheduling techniques for loop parallelization and loop tiling*, Parallel Processing Letter, 1997, Num. 7, p. 379-392

[61] L.-F. Chao, E. H.-M. Sha, *Scheduling data-flowgraphs via retiming and unfolding* , IEEE Transcations on Parallel and Distributed Systems,1997, Vol. 8, p. 1259-1267

[62] J.C. Huang, T. Leng, *Generalized Loop-Unrolling: A Method for Program Speedup*, ASSET99, IEEE Symposium on Application-Specific Systems and Software Engineering & Technology, 24-27 Mars 1999, Richardson (Texas), p. 244-248

[63] M. Lam, *Software Pipelining*, ACM SIGPLAN, Conference on Programming Language Design and Implementation, Atlanta, Juin 1988, p. 318-328

[64] J. Wang, B. Su, *Software pipelining of nested loops for real-time DSP applications*, ICASSP98, 23th IEEE International Conference on Acoustics, Speech and Signal Processing, Seattle, Washington, 12-15 Mai 1998, Vol. 5, article n°1219, p. 3065-3068

[65] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns : Elements of Reusable Object-Oriented Software*, Addison-Wesley, USA, 1995, ISBN 0-201-63361-2

[66] Jean-Michel Kantor, *Problèmes de Hilbert*, Encyclopédie Universalis, 1985, Vol. 9, p. 300

[67] Hors serie de La Rercherche, *L'univers des Nombres*, N°2, Société d'Editions Scientifiques, France, Août 1999

[68] G.J. Chaitin, *Information, Randomness and Incompleteness*, Series in Computer Science, Vol. 8, Seconde Edition, World Scientific, Singapour 1990, ISBN 9-810-20171-0

[69] F. Reinhardt, H. Soeder, *Atlas des mathématiques*, livre de poche, Edition La Pochotèque, Librairie Générale Française, 1997, (égalité de Bézout p.119)

[70] Donald E. Knuth, *The Art Of Computer Programming*, Vol. 2, 3ième édition, Addison Wesley, ISBN 0-201-89684-2

[71] P.G. Emme, *Understanding some simple processor-performance limits*, IBM Journal of Research & Development, Vol. 41, Num. 3

[72] *TMS320C80 (MVP): Transfer Controller User's Guide* , Documentation Texas Instruments N° SPRU105A, 1995

[73] J. Kim, Y. Kim, *Performance Analysis and Tuning for a Single-Chip Multiprocessor DSP*, IEEE Concurrency, Janvier-Mars 1997, Vol. 5, No. 1, p. 68-79

[74] J. Kim, Y. Kim, *Simulating Multimedia Systems with MVPSIM*, IEEE Design & Test of Computers, hivers 1995, Vol. 12, No. 4, p. 18-27

[75] M. Flynn, Some Computer Organizations and Their Effectiveness, IEEE Transactions on Computers, 1972, Vol. C-21, p. 94

Chapitre 3

[76] F. Garcia Lorca, *Filtres récursifs temps réel pour la détection de contours : optimisations algorithmiques et architecturales*, Thèse de Doctorat de l'Université d'Orsay, Paris XI, 27 Novembre 1996

[77] J.-P. Cocquerez, S. Philipp, *Analyse d'images : filtrage et segmentation*, Masson Paris 1995, ISBN 2-22584-923-4

[78] R. Klette, P. Zamperoni, *Handbook of Image Processing Operators*, John Wiley & Sons, Angleterre 1996, ISBN 0-47195-642-2

[79] J.F. Canny, *A Computational Approach to Edge Detection* , IEEE Transactions on Pattern Analysis And Machine Intelligence (PAMI), 1986, Vol. 8, p. 679-714

[80] R. Deriche, *Fast Algorithms for Low-Level Vision* , IEEE Transactions on Pattern Analysis And Machine Intelligence (PAMI), 1990, Vol. 12, Num. 1, p. 78-87

[81] Y.F. Wan, F. Cabestaing, J.G. Postaire, *Un opérateur hyperbolique pour la détection des contours*, GRETSI95, Colloque du Groupe de Recherche en Traitement du Signal et des Images, Juan les Pins, France 1995, p.633-637

[82] Sarifuddin, *Implémentation sous forme de circuit spécialisé d'un algorithme de détection de contours multi-échelles*, Thèse de Doctorat de l'Université de Bourgogne, 1995

[83] L. Torres, *Intégration de filtres numériques pour le traitement d'images : du silicum au système reconfigurable*, Thèse de Doctorat de l'Université de Montpellier II, 1996

[84] R. Bourguiba, D. Demigny, L. Kessal, *Architecture reconfigurable dynamiquement, application au traitement d'images*, AAA99, Journées Thématiques Universités/ Industries sur l'Adéquation Algorithme-Architectures pour les Applications Temps Réel Industrielles Complexes, Lille, 23-24 Mars Septemble 1999, Vol. 1, p. 80-87

[85] F. Lohier, L. Lacassagne, P. Garda, *A DSP Implementation of Optimal Edge Detector*, Multimedia Systems Design Magazine (www.msdmag.com), Septembre 1998, Vol. 2, Num. 9, p. 26-35

[86] F. Lohier, L. Lacassagne, P. Garda, *A DSP Implementation of Optimal Edge Detector*, Multimedia Systems Design Magazine (www.msdmag.com), Septembre 1998, Vol. 2, Num. 9, p. 26-35

[87] F. Lohier, L. Lacassagne, P. Garda, *Programming techniques for the real-time software implementation of optimal edge detectors : A comparison between state of the art DSP and RISC architectures*, DSP World Spring Design Conference, Santa Clara, 21-23 Avril 1998, p. 343-359

[88] L. Lacassagne, F.Lohier, P. Garda, *Real time execution of optimal edge detectors on RISC and DSP processors*, ICASSP98, 23th IEEE International Conference on Acoustics, Speech and Signal Processing, Seattle, Washington, 12-15 Mai 1998, Vol. 5, article n°1735, p. 3101-3104

Chapitre 4

[89] S. Bouchafa, *Détection du mouvement insensible aux variations du contraste : application à la détection de comportement anormaux dans le métro par traitement d'images*, Thèse de Doctorat de l'Université Paris 6 (Pierre et Marie Curie), 14 Décembre 1998

[90] N. Paragios, R. Deriche**,** *Geodesic Active Regions for Motion Estimation and Tracking*, Rapport de Recherche de l'Institut National de Recherche en Informatique et Automatique (INRIA), Sophia Antipolis, France, Projet ROBOVIS, Num. RR-3631, Mars 1999, 30 pages

[91] P. Lalande, P. Bouthemy, *Détection de mouvement dans les séquences d'image selon une approche markovienne. Application à la robotique sous-marine*, Thèse de Doctorat de l'université de Rennes I, France 1990

[92] R. Koenen, *Overview of the MPEG-4 Standard*, Document de travail intermédiaire, Organisation ISO, Mars 1999, ISO/IEC JTC1/SC29/WG11 N°2725

[93] F. Cabestaing, *La détection et l'analyse du mouvement dans les séquences d'images*, Colloque International sur la Vision Artificielle, Fès, Maroc, 15-17 Juin 1995

[94] C. Stiller, J. Konrad, *Estimating Motion in Image Sequences* , IEEE Signal Processing Magazine, Juillet 1999, Vol. 16, Num. 4, p. 70-91 (Errata p. 116-117 Vol. 16, Num. 5)

[95] A. Caplier, *Modèle Markovien de détection de Mouvement dans les sequences d'images : Approche spatio-temporelle et mises en œuvre temps réel*, Thèse de Doctorat de l'Institut National Polytechnique de Grenoble (INPG), France, Décembre 1995

[96] R.J. Fergusson, J.J. Soraghan, *Motion Segmentation on the TMS320C80 Multimedia Video Processor*, ICASSP98, 23th IEEE International Conference on Acoustics, Speech and Signal Processing, Seattle, Washington, 12-15 Mai 1998, Vol. 5, article N°1592, p. 2805-2808

[97] Y.S. Sohn, S.B. Pan, R.-H. Park, *Implementation of a two-stage block matching algorithm using integral projections on the TMS320C80 DSP Board*, ICSPAT97, 8th International Conference on Signal Processing Applications and Technology, San Diego, Californie, 14-17 Septembre 1997, p. 1213-1217

[98] J.-H. Park, K.-D. Hwang, S.B. Pan, R.-C. Kim, R.-H. Park, S.U. Lee, I.C. Kim, *Implementation of the Navigation Parameter Extraction from Aerial Image sequence on the TMS320C80 DSP Board*, ICSPAT97, 8th International Conference on Signal Processing Applications and Technology, San Diego, Californie, Septembre 1997, p. 1562-1566

[99] C.G.-Artal, J.C-Gomez, M.C. Santana, J.D. H-Gonzalez, *A C80 DSP-Based Active vision System For Real-Time Tracking*, ICSPAT98, 9th International Conference on Signal Processing Applications and Technology, Toronto, Canada, Septembre 1998, p. 1184-1188

[100] S.B. Pan, Y.S. Oh, Y.S. Sohn, R.H. Park, *Implementation of a Fast Hierarchical Motion Vector Estimation Algorithm Using Mean Pyramid on TMS320C80 DSP* , International Conference on Circuits, Systems, Computers and communication, Korée, Juillet 1998, p.171-174

[101] D.E. Becker, B. Roysam, H.J. Tanenbaum, J.N. Turner, *Real-Time Image Processing Algorithms for an Automated Retinal Laser Surgery System* , ICIP95, IEEE International Conference on Image Processing, Washington DC, 1995

[102] P. Lalande, P. Bouthemy, *Détection de mouvement dans les séquences d'image selon une approche markovienne. Application à la robotique sous-marine*, Thèse de Doctorat de l'université de Rennes I, France 1990

[103] C. Dumontier, *Etude et mise en œuvre temps réel d'un algorithme de détection de mouvement par approche markovienne*, Thèse de Doctorat de l'Institut National Polytechnique de Grenoble (INPG), France, Novembre 1996

[104] C. Dumontier, F. Luthon, J. P. Charras, *Real Time Implementation of an MRF-based Motion Detection Algorithm on a DSP Board*, Journal of Real Time Imaging, Academic Press, Février 1998, Vol. 4, Num. 1, p. 41-54

[105] L. Lacassagne, F. Lohier, M. Milgram, P. Garda, *Implémentation temps réel d'un algorithme de détection de mouvement par champs de Markov sur RISC et DSP C6X*, GRETSI99, 17ème Colloque sur le traitement du Signal et des Images, Vannes, France, 13-17 Septembre 1999, Vol. 2, p. 315-317

[106] D. Nister, C. Christopoulos, *An Embedded DCT-Based still image coding algorithm*, ICASSP98, 23th IEEE International Conference on Acoustics, Speech and Signal Processing, Seattle, Washington, 12-15 Mai 1998,Vol. 5, article n°1048, p. 2617-2620

[107] J.M. Shapiro, *Embedded Image Coding using zerotrees of wavelet coefficients*, IEEE Trans. on Signal Processing, Décembre 1993, Vol. 42, No.12, p 3445-3462

[108] A. Sais, W.A. Pearlman, *A new, fast and efficient image codec based on set partitioning in hierarchical trees*, IEEE Trans. on Circuits and Systems for Video Technology, Juin 1996, Vol. 6, Num. 3, p. 243-250

[109] J. In, S. Shirani, F. Kossentini, *JPEG Compliant Efficient Progressive Image Coding*, ICASSP98, 23th IEEE International Conference on Acoustics, Speech and Signal Processing, Seattle, Washington, 12-15 Mai 1998, Vol. 5, article N°2414, p. 2633-2636

[110] S.-A. Martucci, *A new Approach for reducing blockiness in DCT image coders*, ICASSP98, 23th IEEE International Conference on Acoustics, Speech and Signal Processing, Seattle, Washington, 12-15 Mai 1998, Vol. 5, article N°1307, p. 2549-2552

[111] W.B. Pennebaker, J.L. Mitchell, *JPEG : Still image data compression standard*, Van Nostrand Reinhold, New York, New York, 1993, ISBN 0-442-01272-1

[112] V. Bhaskaran, K. Konstantinides, *Image and Video Compression Standards : Algorithms and Architectures*, Kluwer Academic Publishers, Norwell, Massachusetts (USA), 1997, ISBN 0-79239-952-8

[113] W.B. Pennebaker, J.L. Mitchell, *JPEG : Still image data compression standard*, Van Nostrand Reinhold, New York, New York, 1993, ISBN 0-442-01272-1

[114] D. Le Goff, P. Pignon, *Procédé de transmission d'images vidéo numérisées*, Brevet Industriel Européen, Société TDF, Num. d'enregistrement National 9713999, 1999

[115] B. Jamal, D. Jacques, *Road traffic qualification control method and system by analyzing digital images* , 1996, Brevet Industriel Européen N° EP960400670 19960328, Société TDF

[116] T. Berger, *Rate distortion theory; a mathematical basis for data compression*, Prentice-Hall, 1971

Conclusions

[117] J. Hennessy, *The Future of Systems Research*, IEEE Computer Magazine, Août 1999, Vol. 32, Num. 8, p. 27-33

[118] D. Kirovski, M. Potkonjak, *Synthesis of DSP Soft Real-time multiprocessor systems-on-silicon*, ICASSP99, 24th IEEE International Conference on Acoustics, Speech and Signal Processing, Phoenix, Arizona,15-19 Mars 1999, Vol. 4, article N°1832

[119] Microprocessor Forum, San Jose, Californie, 5-6 Octobre 1999

[120] Dr. Pinto, *The Rebirth of DSP: Communications Applications of DSP-based SOC Technologies*, Keynote address, ICSPAT99, 10th International Conference on Signal Processing Applications and Technologies, Orlando, Floride, 1-4 Novembre 1999. Enregistrement audio disponible en ligne sur www.dspworld.com

[121] G. Lejeune, J. Liénard, *MUSTIG : a new develoment environment and langage for Interactive Multidimensional Signal Processing*, International Conference on DSP. Berlin, Octobre 1991

RESUME :

Cette thèse présente une contribution aux méthodologies de programmation pour l'implantation d'algorithmes de traitement d'images en temps-réel sur des architectures de type DSP parallèles. Nous nous intéressons plus particulièrement aux architectures TMS320 C80 et C62 de Texas Instruments. Pour ces DSP, nous présentons un éventail de techniques logicielles qui permettent l'optimisation de l'implantation des opérateurs algorithmiques de traitement ainsi qu'une méthodologie originale pour la gestion des flux de données 2D. Cette méthode vise l'automatisation et l'optimisation de l'utilisation du paradigme matériel couplant les co-processeurs de transfert (DMA) avec les mémoires internes. Dans le contexte multi-processeurs du C80, nous cherchons plus précisément à automatiser la parallélisation SPMD des flux en optimisant le partitionnement des données pour le paramétrage et l'encapsulation synchrones des requêtes de transfert DMA. Avec cette methodologie, nous avons réalisé une librairie de traitement d'images bas niveau pour le C80. Cette librairie innove dans sa conception qui rend transparente la mise en œuvre du DMA. Elle est paramétrable sur le plan de la taille des images traitées et du nombre de processeurs affectés au traitement SPMD, reconfigurable dynamiquement et générique dans l'éventail des algorithmes bas niveau qu'elle permet d'implanter. Deux applications développées avec l'outil sont étudiée s: un algorithme de détection de contours optimal et une technique de détection de mouvement markovienne. Les performances temps-réel (TR) obtenues pour le 1er algorithme sur C80 et C60 montrent l'intérêt de notre démarche pour la gestion des flux et l'apport des techniques d'optimisation utilisées dans la programmation des opérateurs. Pour le 2ème algorithme, les performances TR obtenues sur le C80 nous permettent d'exploiter la détection de mouvement dans un algorithme original de compression de séquences d'images qui fait l'objet du dépôt d'un brevet industriel.

MOTS-CLES : METHODOLOGIES DE PROGRAMMATION, FLOTS DE DONNEES SYNCHRONES, DMA, DSP PARALLELES, TMS320C80, TMS320C6X, DETECTION DE CONTOURS D'IMAGES, DETECTION MARKOVIENNE DE MOUVEMENT, COMPRESSION DE SEQUENCES D'IMAGES

SUMMARY :

This thesis stands as a contribution to the programming methodologies for the real-time implementation of image processing algorithms on parallel DSP architectures. Two key processors are targeted: the TMS320 C80and C62 from Texas Instruments. For these architectures, we present a range of software optimization techniques yielding an efficient programming of the algorithmic processing operators as well as a novel methodology to manage the 2D I/O data streams involved during the processing. This methodology aims at automating and optimizing the use of the DSP-specific hardware paradigm that combines data-transfer co-processors (DMA) and internal memories. More specifically, in the multi-processors context of the C80, our ultimate goal is to automate an SPMD-oriented parallelization scheme that requires to partition data among processors after which we synchronously encapsulate all the generated DMA data transfer requests. Based on this work, we developed a low-level image processing library for the C80. Thanks to its structure, this library innovates as it eases implementing the DMA. Processed image's sizes as well as the number of processors assigned to the SPMD parallelization can be parameterized in a dynamic and reconfigurable way. The library is also generic in the range of low-level image algorithms it can implement. Two sample applications developed with this tool were studied: an edge detection algorithm and a markov-based motion detection algorithm. The real-time (RT) performances we gain for the first application using the C80 and C60 emphasize the benefits of our I/O handling methodology and the gain achieved through optimization techniques when implementing operators. The RT performances we also got for the second algorithm on the C80 allowed us to further embed the motion detection algorithm in a novel video-stream compression algorithm subject to an industrial patent.

KEY-WORDS : PROGRAMMING METHODOLOGIES, SYNCHRONOUS DATA FLOWS, DMA, PARALLEL DSP, TMS320C80, TMS320C6X, IMAGE'S EDGES DETECTION, MARKOV-BASED MOTION DETECTION, VIDEO-SEQUENCES COMPRESSION