Implantation optimisée d'estimateurs de mouvement pour la compression vidéo sur plates-formes hétérogènes multicomposants

Fabrice Urban

N° d'ordre : D 07 - 24

Thèse

présentée devant
L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE RENNES

pour obtenir le titre de
Docteur
spécialité : *Electronique et Traitement du Signal*

Implantation optimisée d'estimateurs de mouvement pour la compression vidéo sur plates-formes hétérogènes multicomposants

par
Fabrice URBAN

Soutenue le 6 Décembre 2007 devant la commission d'Examen

Composition du jury

Rapporteurs

Marco MATTAVELLI	Professeur à l'EPFL, Lausanne
Michel PAINDAVOINE	Professeur à l'Université de Bourgogne, Dijon

Examinateurs

Olivier SENTIEYS	Professeur à l'ENSSAT de Lannion
Thierry COLLETTE	Chef de service au CEA LIST, Saclay
Ronan POULLAOUEC	Ingénieur RD Thomson, Rennes
Olivier DEFORGES	Directeur de la thèse, Professeur à l'INSA de Rennes
Jean-François NEZAN	Maître de conférence à l'INSA de Rennes

Invités

Michel KERDRANVAT	Video Technology Officer, Thomson Grass Valley, Rennes

Institut d'Electronique et de Télécommunications de Rennes
Institut National des Sciences Appliquées de Rennes

Résumé

L'estimation de mouvement est une opération clé pour la compression vidéo, mais implique une complexité de calcul conséquente. Sur un encodeur vidéo, jusqu'à 60% de la charge de calcul est dédiée à cette opération. Le contexte de la haute définition et l'évolution des standards de compression vidéo (ex. MPEG-4 AVC/H.264) contribuent à accroître les contraintes pour une exécution temps-réel. L'implantation optimisée d'estimateurs de mouvement doit alors être étudiée dans ce nouveau contexte. Nous nous proposons, dans cette thèse, d'apporter des éléments de réponse génériques, en étudiant à la fois les algorithmes et les architectures multicomposants, dans un cadre méthodologique.

Dans une première partie, nous commençons par présenter un état de l'art sur les techniques de compression vidéo en général et sur les différentes méthodes d'estimation de mouvement en particulier. Les algorithmes de mise en correspondance de blocs prédictifs HME et EPZS apparaissent comme les mieux adaptés à une implantation optimisée. Les architectures matérielles susceptibles d'exécuter ce type d'algorithmes sont ensuite détaillées. L'utilisation de plates-formes multiprocesseurs programmables dans le cadre du prototypage d'applications hautes performances est alors justifiée.

Dans une deuxième partie, nous nous intéressons à l'implantation d'estimateurs de mouvement sur plates-formes multicomposants. Nous présentons d'abord la méthodologie de prototypage rapide AAA et la méthode de développement utilisée. Ensuite, nous étudions l'implantation et l'optimisation d'estimateurs de mouvement sur processeurs de type DSP. Une description flot de données de l'application est réalisée, de façon générique pour être facilement parallélisable et évolutive. Le nouvel algorithme d'estimation de mouvement HDS est proposé pour améliorer les techniques HME et EPZS dans le contexte de l'étude. L'implantation des algorithmes sur DSP est optimisée afin d'évaluer précisément les performances atteignables en monoprocesseur. La méthodologie utilisée facilite le portage de l'application.

Enfin, l'opération d'estimation de mouvement pour la haute définition étant trop complexe pour atteindre des performances temps-réel sur cible monoprocesseur, nous étudions son implantation sur diverses plates-formes multicomposants. Un coprocesseur est tout d'abord conçu pour réaliser le raffinement subpixélique, opération identifiée comme bon candidat à une implantation sur FPGA. L'application est prototypée sur une architecture hétérogène DSP-FPGA. Ensuite, les traitements sont divisés suivant des bandes d'images et parallélisés. Pour cela, certaines dépendances de données sont modifiées. Les outils de prototypage intègrent naturellement ce genre de parallélisme et permettent de porter rapidement l'application sur une plate-forme multiprocesseur. La généricité des travaux permet d'étendre cette approche à de nombreuses autres applications.

Abstract

Motion estimation is a key operation in most video coding schemes. It is also known to be very computationally intensive. On a H264 video encoder, up to 60% of the computational load is dedicated to this operation. High definition context and the evolution of compression standards increase real-time constraints further. The implementation optimization has to be studied in this new context. In this thesis, algorithms together with multicomponent architectures are evaluated in a methodologic prototyping framework.

In a first part, a state of the art of video compression and existing motion estimation techniques in particular are firstly presented. Predictive bloc-matching algorithms HME and EPZS appear to be well suited to an optimized implementation. Hardware architectures are then detailed. The use of programmable multicomponent platforms is then justified for high-performance application prototyping.

In a second part, we focus on the implementation of motion estimation algorithms on multicomponent platforms. The AAA rapid prototyping methodology is firstly presented. Then the implementation and optimization of motion estimation algorithms on DSP processors. A data-flow model of the application is designed in a generic approach to be parallelizable and upgradable. The new motion estimation algorithm HDS is proposed to improve HME and EPZS techniques in the research context. The implementation of these algorithms is optimized on DSP to evaluate performances on a single processor. The methodology eases the functional verification stages, the porting in the optimization stage and improve design reliability.

Finally, motion estimation operation for high definition being to computationally intensive to be handled in real-time on a single DSP, we focus on its implementation on various multicomponent platforms. Firstly, a coprocessor is designed to handle subpixel refinement. Indeed, this operation had been identified to be a good candidate for an FPGA implementation. The application is then prototyped on an heterogeneous platform comprising a DSP and an FPGA. Secondly, calculations are divided according to image stripes and parallelized. Some data dependencies are thus modified. Prototyping tools naturally manage this kind of parallelism and accelerate the implementation of the application on a multicomponent platform. Thanks to the genericity of the work, this approach can be extended to many other applications.

Remerciements

Je tiens à remercier tout d'abord les membres du jury qui ont accepté d'examiner ce travail : Marco MATTAVELLI et Michel PAINDAVOINE en qualité de rapporteurs, Olivier SENTIEYS, Thierry COLLETTE, Ronan POULLAOUEC et Michel KERDRANVAT en tant que membres du jury.

J'exprime également toute ma reconnaissance à Joseph RONSIN et Olivier DEFORGES pour m'avoir proposé de mener cette thèse et m'avoir accueilli au sein de leur équipe dans le laboratoire IETR.

Je remercie chaleureusement Michel KERDRANVAT, chef du laboratoire compression vidéo à THOMSON Corporate Research Rennes au moment où j'ai commencé ma thèse, sans qui ces travaux et la collaboration avec l'IETR n'auraient pas été possibles.

Je tiens également à exprimer toute ma gratitude à mes encadrants universitaires : Olivier DEFORGES et Jean-François NEZAN et industriels : Michel KERDRANVAT et Ronan POULLAOUEC, ainsi qu'à Mickaël RAULET et Paul KERBIRIOU pour leurs support, leurs conseils, leurs bonnes idées et leurs remarques constructives lors de leur relecture de ce manuscrit.

Merci à tous les membres de l'IETR et de THOMSON pour leur support, leur présence et leur bonne humeur.

Merci également aux étudiants que j'ai eu le plaisir d'encadrer : Serigné DIENG et Alain MACCARI.

Mes derniers remerciement, et non les moindres, s'adressent à ma famille et mes amis pour leur soutient et leur présence.

Table des matières

Introduction générale

La télévision numérique, présente dans la plupart des foyers, occupe aujourd'hui une place privilégiée. L'avènement de la vidéo numérique a été rendue possible principalement grâce aux possibilités de compression. La compression vidéo permet en effet de réduire le débit nécessaire à la diffusion d'une émission, et facilite donc son transport sur tout type de réseau (hertzien, internet, satellite, téléphone mobile, ...). Le coût de diffusion ainsi diminué permet d'envisager une meilleure qualité de service. L'évolution des standards de compression vidéo aboutit à l'amélioration de l'efficacité de codage. Ainsi, MPEG-4 AVC ou H.264 abaisse le débit nécessaire jusqu'à 50 % par rapport à son prédécesseur MPEG-2, ouvrant la voie à de nouveaux services, tels que la vidéo haute définition. L'amélioration des performances de codage est obtenue au détriment de la complexité des traitements. La quantité de calculs se révèle ainsi de plus en plus élevée à chaque génération de codeur. Le passage à des formats haute définition augmente encore très fortement cette complexité.

La normalisation d'une technique spécifie entièrement la phase de décodage, mais laisse toute liberté pour la définition du codeur. Dès lors, des recherches, même sur le plan algorithmique, continuent pour H.264, plusieurs années après son adoption. L'estimation de mouvement est une opération clé pour la compression vidéo. Cette opération contribue pour une grande partie à l'efficacité de la compression en éliminant les redondances temporelles. C'est aussi l'opération nécessitant le plus de puissance de calcul : jusqu'à 60% de la charge de calcul d'un encodeur vidéo H.264. Un effort particulier doit alors être mis sur l'estimation de mouvement, tant sur le plan algorithmique que sur l'implantation.

La quantité de calculs nécessaires à l'estimation de mouvement pour la compression vidéo haute définition dépasse les capacités des processeurs génériques actuels. Ceci va encore se vérifier dans le futur, du fait de l'augmentation de la complexité des standard de compression. Des composants dédiés sont alors développés afin de satisfaire les contraintes temps-réel. La conception de tels systèmes est complexe et implique un temps de cycle de développement conséquent. Une alternative, dans le cadre d'un prototypage à un stade précoce, consiste en l'utilisation de plates-formes multicomposants hétérogènes de type DSP/FPGA. Les DSP sont utilisés pour leur simplicité de programmation et les FPGA pour leurs performances. La programmation de ces plates-formes n'est toutefois pas triviale compte tenu du caractère distribué de l'architecture. Les questions qui se posent sont de plusieurs ordres dans un tel contexte.

– Quel algorithme utiliser pour réduire la complexité sans modifier de façon notable les résultats ?

– Quelle cible matérielle choisir en fonction des algorithmes à exécuter et des contraintes imposées (rapidité, coût, flexibilité, consommation) ?
– Comment obtenir une solution d'implantation qui satisfasse le temps réel (parallélisation, distribution, ordonnancement) ?
– Quelle démarche adopter pour réduire significativement le temps de cycle de développement et favoriser la réutilisation de l'existant ?

Le traitement des images, et plus particulièrement la compression vidéo est une thématique commune aux deux laboratoires *IETR (Institut d'Electronique et de Télécommunications de Rennes) - groupe Images* et *Thomson Corporate Research - Content Delivery and Compression*. L'équipe *Images* du laboratoire *IETR* travaille depuis plusieurs années sur des méthodologies de prototypage permettant un passage quasi-automatique de la description fonctionnelle d'une application de traitement d'images à son implantation sur une architecture multicomposant pouvant comporter des PC, des DSP et des FPGA. L'équipe *Compression* du laboratoire *Content Delivery and Compression* de *THOMSON Corporate Research Rennes* se penche quant à lui sur les nouvelles méthodes de compression vidéo. La nécessité d'implantations efficaces des applications de traitement d'image est réelle, aussi bien dans le but d'évaluer des solutions d'architectures pour les BU[(1)] (BU Compression par exemple), que pour accélérer les simulations de mise au point d'algorithmes. Le développement de démonstrateurs temps réel est également nécessaire pour valider la faisabilité d'une solution ou présenter une technologie à un stade précoce, par exemple lors d'un salon. Les compétences en architectures matérielles doivent alors être renforcées dans un laboratoire plutôt orienté algorithmie.

Ces travaux visent à étudier l'implantation optimisée d'estimateurs de mouvement sur plates-formes multicomposants hétérogènes dans le cadre de la compression vidéo H.264 haute définition. Les aspects algorithmiques et matériels sont abordés dans cette thèse, afin de trouver une adéquation entre les architectures existantes et les algorithmes étudiés. Le contexte applicatif, à savoir des calculs intensifs et des quantités de données conséquentes, se voit comme un élément de validation et d'évolution de la démarche méthodologique.

Dans la première partie de ce mémoire, nous présentons un état de l'art sur les différents points abordés. Le domaine de la compression vidéo est tout d'abord introduit, avec notamment le standard H.264, successeur de MPEG-2 pour la diffusion de vidéo numérique. Les techniques d'estimation de mouvement sont ensuite décrites, avec un accent particulier sur les différents algorithmes de mise en correspondance de blocs, développés en vue d'améliorer les performances. Finalement, une étude sur l'offre en matière d'architectures matérielles existantes est proposée, afin d'appréhender les contraintes d'une implantation optimisée.

Dans la seconde partie, nous présentons les travaux effectués dans le cadre de cette thèse. Tout d'abord, la méthodologie de prototypage AAA est présentée. Nous définissons les fondements et proposons des améliorations des outils afin de prendre en compte les spécificités des applications et architectures considérées. Les travaux réalisés visent ainsi à améliorer la méthode de développement de sorte à accélérer et fiabiliser le prototypage d'une application sur une plate-forme multicomposant. Nous

[(1)] BU : Business Units, entités chargées de la conception et de la production des produits

abordons ensuite l'implantation d'estimateurs de mouvement sur DSP. Les algorithmes sont modélisés sous la forme d'un graphe flot de données, première étape nécessaire à la méthodologie. Les algorithmes HME et EPZS sont étudiés en termes de complexité et de qualité des champs de vecteurs destinés à la compression vidéo. A partir des résultats obtenus, un nouvel algorithme, appelé HDS, est élaboré, combinant les atouts des deux premières techniques. Les optimisations réalisées sur DSP permettent de réduire considérablement les temps d'exécution. Dans le même temps, compte tenu des contraintes imposées notamment par le contexte de la haute définition, les résultats obtenus ne permettent pas d'envisager une implantation temps-réel sur monocomposant.

Finalement, l'application est portée sur différentes plates formes. Le raffinement subpixélique a été identifié comme un point bloquant du fait de la quantité de calculs et de la bande passante mémoire nécessaire. Nous nous proposons donc de réaliser un coprocesseur sur FPGA pour exécuter cette opération. La solution envisagée se doit d'être compatible avec les contraintes imposées, à savoir une bande passante réduite sur le bus externe afin de la connecter à un DSP, des performances temps réel pour la haute définition et des ressources logiques réduites.

La parallélisation de l'application en fonction des opérations permet de mettre en adéquation les spécificités des composants de l'architecture avec celles des traitements à réaliser. Toutefois, chaque opération élémentaire doit satisfaire les contraintes de temps-réel. Le contexte de la haute définition pose ici un problème majeur : l'exécution d'une opération sur une image complète n'est pas toujours possible dans le temps imparti. Nous nous proposons donc de paralléliser l'opération d'estimation de mouvement en découpant l'image en plusieurs bandes, de sorte à relâcher les contraintes matérielles propres à chaque processeur. L'application d'estimation de mouvement a ainsi été prototypée sur une plate-forme de huit DSP. Les outils de prototypage, en particulier la génération automatique de code, permettent l'implantation quasi-automatique et l'augmentation progressive du nombre de processeurs.

Enfin, la conclusion fournit une synthèse des travaux effectués et présente des perspectives de travaux de recherche.

Première partie

Etat de l'art

Chapitre 1

Image numérique et compression vidéo

Le but de ce chapitre est d'introduire les notions générales nécessaires à cette étude. Nous définissons dans un premier temps les images numériques et la compression vidéo en général. Le transport d'un flux vidéo numérique nécessite de compresser celui-ci afin de l'adapter au débit du canal de transmission ou à la capacité du support. Des généralités sur la vidéo numérique et les principales techniques de codage sont présentées. Dans un deuxième temps, les spécificités du dernier standard de compression vidéo de JVT (Joint Video Team ITU/MPEG) : MPEG-4 AVC ou H.264, sont présentées. Les différents modes de codages sont brièvement présentés, et les opérations d'estimation et de compensation de mouvement sont détaillées. En complément, une technique d'évaluation de la qualité des champs de vecteurs dans un cadre de compression vidéo est décrite.

1.1 Introduction aux techniques de compression vidéo

1.1.1 Généralités sur les images numériques

Une image numérique est définie par une matrice de points appelés pixels (figure 1.1). La représentation d'une image numérique est introduite ici afin de bien comprendre les techniques de traitement vidéo.

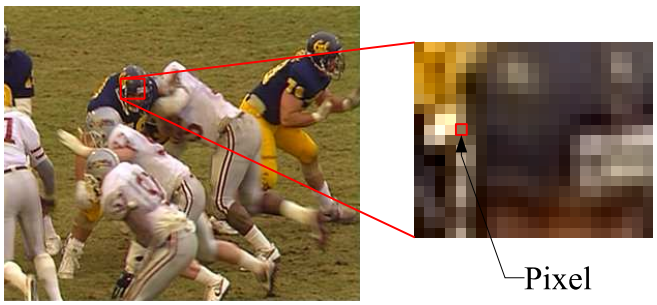
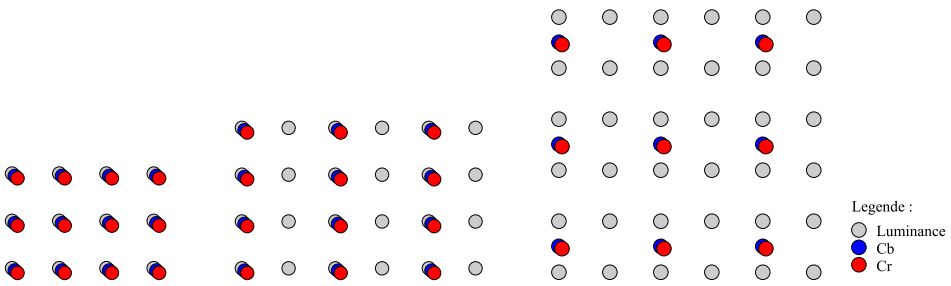


FIG. 1.1 – Représentation numérique d'une image

Une vidéo numérique non compressée se caractérise par son système de représentation des pixels, sa taille et son rafraîchissement. Plusieurs formats existent selon l'application visée.

Représentation des pixels Les pixels peuvent être représentés de plusieurs manières. Dans une image à niveaux de gris, un pixel possède la plupart du temps 256 niveaux de gris (codés sur 8 bits). Le niveau 0 code le noir et le niveau 255 le blanc. Une image couleur est représentée avec trois composantes. Il existe principalement deux systèmes :

- **R :V :B** : un pixel de couleur est la somme des trois composantes Rouge Verte et Bleue. Dans le cadre de la vidéo, ce format est principalement utilisé pour l'affichage. Il peut toutefois être utilisé en post-production de cinéma.
- **Y :Cb :Cr** : Un changement de repère est effectué par rapport au RVB. Un pixel est représenté par sa composante de luminance (Y) et deux composantes de chrominance (Cb et Cr). Ce format est aussi appelé YUV.
  Le système YCrCb est historiquement utilisé dans les systèmes audiovisuels car il permet de diffuser une image couleur tout en gardant une compatibilité avec le système noir et blanc. De plus, l'oeil humain est plus sensible à la luminance qu'à la couleur, ce qui permet de réduire la définition des composantes de chrominance et donc d'obtenir facilement une première réduction des données. On distingue trois formats (figure 1.2) :
  - 4 :4 :4 : chaque composante est codée de la même manière, il n'y a pas de sous-échantillonnage. Ce format est utilisé lorsque toute la qualité de la vidéo doit être gardée, c'est à dire pour le stockage ou la post-production en studio, le cinéma numérique.
  - 4 :2 :2 : on ne garde qu'une ligne sur deux pour les composantes de couleur. Seulement la moitié de l'information de couleur est gardée. C'est un format de qualité studio.
  - 4 :2 :0 : on ne garde qu'une ligne sur deux et une colonne sur deux pour les composantes de couleur. Un quart de l'information de couleur est conservé. C'est le format le plus utilisé pour la vidéo grand public.



(a) Echantillonnage 4 :4 :4 (b) Echantillonnage 4 :2 :2 (c) Echantillonnage 4 :2 :0

FIG. 1.2 – Représentation des pixels

Chaque composante du signal vidéo peut être codée sur 8 à 10 bits. 8 bits sont suffisants pour représenter une composante puisque l'oeil ne distingue pas deux niveaux voisins. Cependant, en post-production 10 bits peuvent être utilisés pour une meilleure précision et une meilleure gestion des différents niveaux de post-production. D'une manière générale, la distribution des niveaux est linéaire, mais il se peut toutefois qu'une échelle logarithmique soit utilisée pour offrir une plus grande dynamique, notamment pour le cinéma numérique qui utilise des caméras pouvant avoir un contraste élevé.
D'une manière générale, les composantes du signal vidéo sont le plus souvent codées sur 8 bits avec un format 4 :2 :0. On a donc en moyenne 1,5 octet par pixel.

Le balayage Le balayage peut être progressif ("progressive scan" ou p) ou entrelacé ("interlaced scan" ou i). Pour une séquence progressive, toute l'image est affichée d'un seul coup, alors que pour une séquence entrelacée les lignes paires et impaires sont acquises et affichées alternativement.

La taille La taille ou définition d'une image est le nombre de lignes et le nombre de colonnes. Pour des définitions standard dans les applications audiovisuelles, seulement le nombre de lignes est spécifié : (ex : 720p : 1280 colonnes $\times$720 lignes en balayage progressif).

Le tableau 1.1 récapitule les paramètres des formats standard. Le débit nécessaire à la transmission des données non compressées est calculé pour un format en 4 :2 :0 avec 8 bits par composante (1.5 bits/pixel).

Définition	Balayage	Fréquence	Débit
QCIF : 176x144	progressif	10 à 30 Hz	380 Ko à 1.1 Mo/s
CIF : 352x288	progressif	10 à 30 Hz	1.5 à 4.6 Mo/s
SD (NTSC) : 640x480	entrelacé	60 Hz	13.8 Mo/s
SD (PAL, SECAM) 768x576	entrelacé	50 Hz	16.6 Mo/s
SD (DVD)(D1) 720x576	entrelacé	50 à 60 Hz	15.6 à 18.6 Mo/s
	progressif	25 à 30 Hz	
HD (720p) : 1280x720	progressif	24 à 60 Hz	33.2 à 83 Mo/s
HD (1080i) : 1920x1080	entrelacé	50 et 60 Hz	78 à 93 Mo/s
HD (1080p) : 1920x1080	progressif	24 à 30 Hz	75 à 93 Mo/s

TAB. 1.1 – Formats standards

Comparées aux débits actuels des communications numériques (de quelques centaines de Kbits/s pour le téléphone portable (3G) à quelques dizaines de Mbits/s pour une chaîne satellite), les images non compressées représentent d'énormes quantités de données. Les débits sont ainsi beaucoup trop élevés par rapport aux bandes passantes disponibles pour leur diffusion. Il est donc nécessaire de compresser les données vidéo pour permettre leur diffusion avec des débits réalistes.

1.1.2 Compression vidéo

La compression (ou codage) d'une séquence vidéo a pour but de réduire son débit et par conséquent les coûts de stockage ou rendre possible sa diffusion sans surcharger le réseau. Cela est rendu possible en cherchant à éliminer les redondances grâce à un codage spécifique. Le codage intra-image exploite les redondances spatiales et le codage inter-image les redondances temporelles. La séquence vidéo est alors décrite de manière plus compacte. Les techniques de compression vidéo actuelles codent l'image par bloc (appelés macro-bloc).

1.1.2.1 Codage intra-image

Le codage intra-image utilise les techniques de compression d'images fixes :

- La redondance inter-pixels. Il est possible d'obtenir une bonne approximation de la valeur d'un pixel (ou d'un bloc de pixels) à l'aide des pixels voisins, et ne transmettre que les différences (résidus) entre les valeurs réelles et les valeurs prédites.
- La transformation. Une transformation sur les résidus (en général fréquentielle, DCT) est appliquée par bloc de pixels afin de concentrer l'énergie des pixels dans un nombre réduit de coefficients.
- La quantification. L'oeil possède des limites de perception. Il est alors possible de réduire le nombre de niveaux des coefficients de la transformée en introduisant des erreurs le moins visible possible.
- Le codage statistique. Les codes à longueur variable permettent d'allouer des codes plus courts aux coefficients les plus probables. Le débit est ainsi statistiquement réduit.

Ces quatre techniques sont utilisées pour compresser des images fixes, la taille des données peut être contrôlée en faisant varier les paramètres de quantification. Ainsi, une quantification élevée va réduire le débit, mais des défauts vont apparaître. Dans le cas de séquences vidéos, les redondances temporelles sont aussi exploitées grâce au codage inter-images.

1.1.2.2 Codage inter-image ou compensation de mouvement

Les différences entre les images d'une séquence vidéo sont principalement dues aux mouvements des objets de la scène et de la caméra. Le but de la compensation de mouvement est de prédire l'image à coder en fonction d'une image de référence et d'un champ de vecteur (un vecteur par bloc) (Fig. 1.3). Le résidu est ensuite encodé avec les techniques présentées en 1.1.2.1 (transformation et quantification). De manière générale, les données à transmettre (champ de vecteurs et résidus) sont réduites par rapport à un codage intra-image.

La compensation de mouvement peut être faite à partir d'une ou plusieurs images de référence afin d'améliorer la prédiction. Chaque bloc peut être reconstruit à partir d'une seule référence ou d'une combinaison de deux références. On parle alors de bi-prédiction.

La compensation de mouvement permet de réduire considérablement le débit d'une séquence vidéo. Une opération d'estimation de mouvement est donc nécessaire afin de produire un champ de vecteur le plus précis possible. Les performances d'un encodeur

(a) Image à encoder (b) Image de référence (c) Champs de vecteurs

(d) Image compensée (e) Résidus

FIG. 1.3 – Compensation de mouvement

vidéo dépendent pour beaucoup de la qualité des champs de vecteurs. Cependant, l'estimation de mouvement est une opération qui requiert de la puissance de calcul. Par exemple, l'estimation de mouvement peut représenter à elle seule jusqu'à 60% de la puissance de calcul d'un encodeur vidéo H.264 [CZH02].

1.1.2.3 Décomposition de la séquence vidéo

Une séquence vidéo compressée est généralement structurée en GOP (Group Of Pictures). Un GOP est une suite d'images codées suivant trois méthodes : codage intra-image (I-frame), prédictif (P-frame) et bidirectionnel (B-frame). L'ordre de ces codages est fixe (Fig. 1.4), avec une image I en début de GOP, puis un motif répétitif d'images B et d'images P jusqu'à la fin du GOP. La fréquence des images I est variable, elle donne la taille du GOP. Les images I sont indispensables pour que le décodeur puisse commencer la chaîne de prédiction.

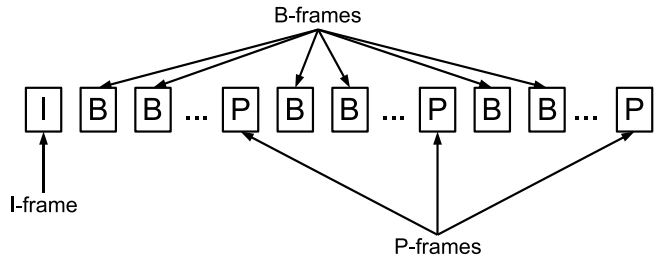


FIG. 1.4 – Structure d'un GOP

1.1.2.4 Standards de compression vidéo

Les standards ont pour but d'assurer la compatibilité entre encodeur et décodeur, tout en laissant le champ libre aux fabricants pour développer des produits innovants et compétitifs. Un standard ne définit pas un encodeur, mais le résultat qu'un encodeur doit produire. Autrement dit, il porte sur la syntaxe du flux et la manière de le décoder.

Il est nécessaire d'établir des standards afin d'homogénéiser les flux et de faciliter la diffusion de vidéos et l'interopérabilité entre constructeurs et fournisseurs de contenus. Le tableau 1.2 récapitule les principales normes de compression vidéo et leur applications.

Standard	Application
MPEG-1	Vidéo-CD, CDI
MPEG-2	Télévision numérique, Satellite, TNT TV Haute Définition, DVD
MPEG-4 Visual	Internet, vidéo mobile, VOD (Video On Demand), Studio
MPEG-4 AVC H.264	Internet, vidéo mobile, Haute Définition VOD (Video On Demand), Studio

TAB. 1.2 – Les normes MPEG et leurs applications

Le groupe MPEG (Moving Picture Expert Group) a été fondé en 1988 pour travailler sur des standards vidéo. La première norme MPEG-1, finalisée en 1992 a pour but la mise en mémoire de documents audiovisuels. MPEG-2 a été développé pour la télévision numérique. Figé en 1994, c'est aujourd'hui un standard largement répandu pour la diffusion.

MPEG-4 a été développé pour répondre à un large spectre de problématiques. Il est divisé en plusieurs parties couvrant la robustesse aux erreurs de transmission, l'interactivité ou encore le codage d'images de synthèse. MPEG-4 Visual (Part 2) a été finalisé en 1999. MPEG-4 AVC (Advanced Video Coding ou Part 10) a été développé en collaboration avec ITU-T (International Telecommunication Union) en tant que recommandations H.264. Alors que MPEG-4 Visual met en avant la flexibilité, H.264 affiche de très bonnes performances. Les taux de compression sont 50% supérieurs à ceux de MPEG-2, à qualité visuelle équivalente. H.264 a été retenu pour la diffusion de la télévision numérique haute définition en France et est en train de devenir le standard préféré en matière de vidéo numérique dans le monde.

1.2 Standard MPEG4/AVC - H.264

Le standard vidéo le plus attractif aujourd'hui est le MPEG-4 AVC ou H.264. Malgré une complexité très supérieure aux précédents standards (MPEG-2, MPEG-4 visual), il est particulièrement apprécié pour ses performances en terme de compression.

Cette section décrit brièvement les caractéristiques de H.264 et présente plus précisément ses spécificités en matière de compensation de mouvement.

1.2.1 Schéma global

La figure 1.5 donne une description générale de haut niveau d'un encodeur H.264. Les opérations classiques des codeurs vidéos sont reprises. Une vue d'ensemble du codec peut être retrouvée dans [WSBL03] et une description plus détaillée dans [Ric03]. Les opérations décrites dans la norme apparaissent sur fond blanc : T est la transformée appliquée sur les résidus, la quantification Q permet de réduire la taille des données, et le codeur entropique élimine les redondances restantes. Pour cette dernière phase, deux techniques sont définies : CABAC (Context Adaptive Binary Arithmetic Coding) ou CAVLC (Context Adaptive Variable Length Coding). Les données en sortie subissent une transformation inverse et sont conservées en mémoire pour la prédiction (intra ou inter après compensation de mouvement). Le "deblocking filter" est optionnel, il permet de réduire les effets de bloc. Les opérations sur fond coloré ne sont pas normalisées. Elles sont laissées à l'initiative du fabriquant. Ce sont des opérations de décision, très importantes pour garder le maximum de qualité à un débit le plus faible possible. Une grande partie des performances du codeur en dépend. On retrouve les opérations de contrôle de débit, de décision de codage et d'estimation de mouvement.

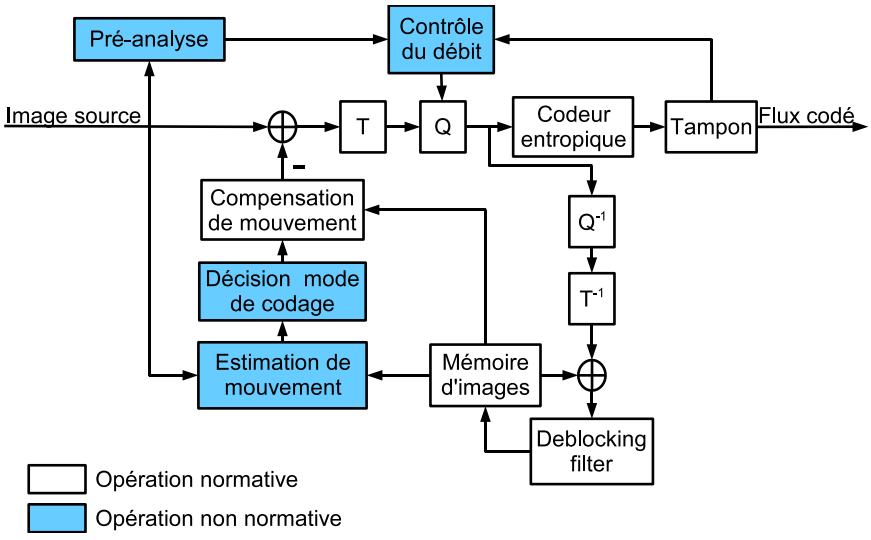


FIG. 1.5 – Diagramme en bloc du codeur H.264

Dans le but d'augmenter ses performances de compression, H.264 introduit de nouveaux outils. La complexité du codeur est supérieure aux précédents standards de plus d'un ordre de grandeur, et celle du décodeur de deux fois supérieure [SDL+04].

Tous les outils de la norme ne sont pas indispensables pour une application donnée. Pour limiter la complexité des décodeurs, des groupes d'outils appelés *profiles* ont été définis. Pour être conforme à la norme, un décodeur peut ne supporter que les outils d'un *profile* donné. Le tableau 1.3 définit les trois *profiles* initialement introduits dans la norme.

Les *profiles baseline* et *main* ont été introduits respectivement pour la vidéo-conférence et la télévision numérique. En 2004, l'amendement FRExt (Fidelity Range Extensions) définit de nouveaux outils avec quatre *profiles* additionnels (High, High 10, High 4 :2 :2 et High 4 :4 :4)[STL04]. Initialement destiné aux environnements

Tools	Baseline	Main	Extended	High
I and P Slices	X	X	X	X
CAVLC	X	X	X	X
CABAC		X		X
B Slices		X	X	X
Entrelacé (PicAFF, MBAFF)		X	X	X
Enh. Error Resil. (FMO, ASO, RS)	X		X	X
Further Enh. Error Resil (DP)			X	X
SP and SI Slices			X	X
Weighted prediction		X	X	X
Intra prediction	X	X	X	X
Variable block size MC	X	X	X	X
1/4 pel MC	X	X	X	X
Multiple ref frames	X	X	X	X
In loop deblocking filter	X	X	X	X
8x8 Transform				X
10 bits				X
4 :2 :2 and 4 :4 :4				X
Lossless				X

TAB. 1.3 – *Profiles* dans H.264

studio, les profiles *high* sont aussi utilisés pour la télévision numérique à la place du profile *main*.

En plus des *profiles* qui restreignent les outils, des *levels* ont été définis afin de borner la puissance de calcul et la mémoire nécessaire pour les décodeurs. Un *level* spécifie des bornes en termes de taille d'image, de fréquence d'image et de débit compressé.

La norme MPEG-4 AVC permet un fort taux de compression au prix d'une complexité élevée. Tout en respectant la norme, la complexité d'un décodeur peut être bornée grâce aux profiles et levels. Quant à l'encodeur, l'implantation des outils est au choix du fabriquant, indépendamment de la norme. En effet, un encodeur respectant la norme à un profile et un *level* donné peut ne pas implémenter tous les outils afin de réduire sa complexité. Cependant, cela se traduit généralement par une perte d'efficacité. Par exemple, avec une puissance de calcul pouvant aller jusqu'à 60% de l'encodage, des compromis sont souvent réalisés sur l'estimation de mouvement.

1.2.2 Compensation de mouvement dans H.264

La compensation de mouvement ou codage inter-image permet d'exploiter les redondances temporelles afin de réduire la quantité d'information à transmettre. L'image courante est reconstruite bloc par bloc à partir d'une image de référence et d'une information de mouvement. H.264 introduit de nouveaux modes de prédiction afin de réduire à la fois les résidus et l'information de mouvement.

1.2.2.1 Image de référence

H.264 peut utiliser une ou plusieurs images parmi un nombre d'images précédemment codées, comme référence pour la compensation en mouvement d'un macrobloc. Un macrobloc d'une image P peut être prédit à partir d'une image de référence, et un macrobloc d'une image B à partir de deux images de référence (la prédiction est alors obtenue en faisant la moyenne entre chaque pixel des deux blocs de référence). La "weighted prediction" est également introduite. Elle permet d'affecter un poids à chacune des prédictions au lieu de prendre la moyenne. Un poids est affecté par image de référence. Ceci est très utile par exemple dans des cas de fondu. Dans H.264, les images B peuvent aussi servir de référence (Fig. 1.6). Deux listes d'images de référence sont gérées au codeur et au décodeur. Plusieurs images de chaque liste peuvent servir comme référence pour prédire l'image courante.

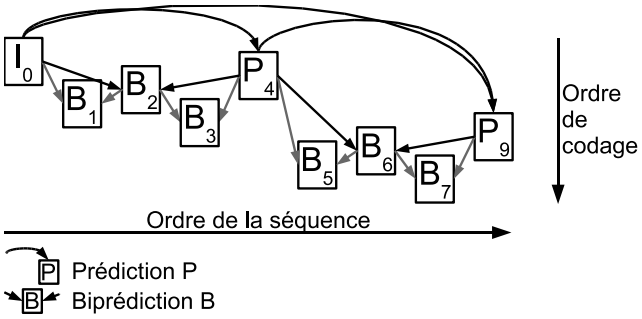


FIG. 1.6 – Images de référence dans un GOP hiérarchique (GOP 4)

1.2.2.2 Taille de bloc variable

La composante de luminance de chaque macro-bloc (rectangle de 16 pixels sur 16 pixels) peut être partagée (Fig. 1.7) en sous partitions 16x8, 8x16, ou 8x8 pixels. Si le mode 8x8 est choisi, chaque bloc peut être à son tour partitionné pour avoir une taille minimale de 4x4 pixels. Ceci conduit à une compensation de mouvement à taille de bloc variable. Un vecteur de mouvement distinct est nécessaire pour chaque sous-partition. Le choix du mode de partition ainsi que chaque vecteur doivent être codés et transmis.

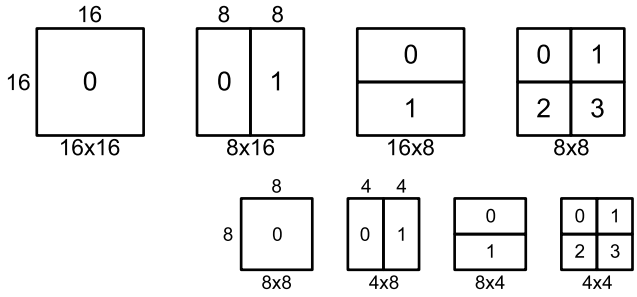


FIG. 1.7 – Décomposition d'un macro-bloc

Lorsqu'une grande partition (16x16, 16x8 ou 8x16) est choisie, un petit nombre de bits est utilisé pour coder l'information de mouvement mais le résidu peut être conséquent. Inversement des petites partitions conduisent à de faibles résidus mais demandent beaucoup d'information de mouvement. Il existe ainsi de nombreuses combinaisons possibles. Le choix de la taille des partitions est donc déterminant pour les performances d'encodage. Des grandes partitions conviennent en général pour les zones homogènes de l'image alors que des partitions réduites favorisent les zones détaillées.

1.2.2.3 Vecteurs de mouvement

Chaque partition est prédite à partir d'une zone de même taille dans une image de référence. Le vecteur de mouvement donne la position relative du bloc de référence. Il peut dépasser les limites de l'image. Dans ce cas les pixels du bord de l'image sont étendus pour reconstruire le bloc de référence (Fig. 1.8).

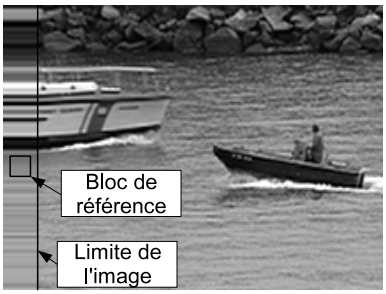


FIG. 1.8 – Reconstruction d'un bloc hors des limites de l'image

Les vecteurs de mouvement sont également compressés. Ils sont prédits à partir d'un médian, calculé avec les vecteurs des blocs voisins. Seule la différence est codée et transmise au décodeur. Ainsi, plus un champ de vecteur est homogène, moins il coûte cher à coder.

La précision des vecteurs de mouvement est d'un quart de pixel pour la luminance et un huitième de pixel pour la chrominance dans H.264 (pour un échantillonnage en 4 :2 :0) (Fig. 1.9). Les mouvements dans les images naturelles n'étant pas limités à un nombre entier de pixels, la qualité de la prédiction est donc améliorée.

Les valeurs des échantillons aux positions fractionnaires ne sont pas disponibles, il est nécessaire de les interpoler. Le filtre d'interpolation est fixé par la norme afin qu'il soit identique au codeur et au décodeur. Les valeurs de luminance demi-pixel sont interpolées avec un filtre séparable à six coefficients (1,-5,20,20,-5,1). Elles sont calculées à partir de six pixels adjacents horizontalement et/ou verticalement. Par exemple b, h, et j dans la figure 1.10 peuvent être calculées de la manière suivante :

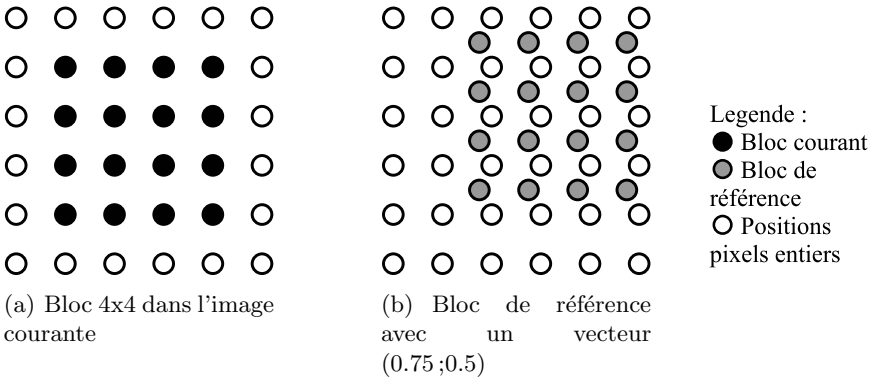
$b = \frac{E-5F+20G+20H-5I+J}{32}$

$h = \frac{A-5C+20G+20M-5Q+S}{32}$

$j = \frac{aa-5bb+20b+20s-5gg+hh}{32}$ ou $j = \frac{cc-5dd+20h+20m-5ee+ff}{32}$

Les valeurs de luminance quart de pixel sont interpolées en faisant une moyenne entre deux échantillons demi-pixels (Fig. 1.10). Par exemple, a et e sont calculés de la manière suivante :

$a = \frac{G+b+1}{2}$ and $e = \frac{h+b+1}{2}$



(a) Bloc 4x4 dans l'image courante

(b) Bloc de référence avec un vecteur (0.75 ;0.5)

FIG. 1.9 – Exemple de prédiction subpixélique

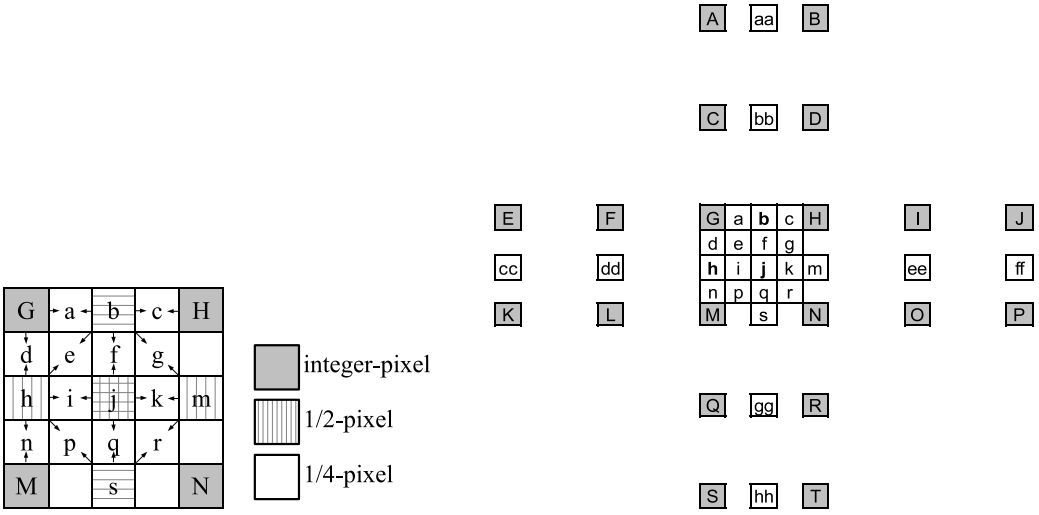


FIG. 1.10 – Filtre subpixélique de luminance H.264

Les échantillons de chrominance sont calculés plus simplement, avec un filtre bilinéaire.

L'introduction du filtre à six coefficients et l'augmentation de la précision au quart de pixel permettent à H.264 d'augmenter ses performances de codage grâce à une prédiction de meilleure qualité par rapport à MPEG-2, par exemple où la précision est au demi-pixel. La compensation de mouvement du standard MPEG-4 AVC est améliorée par rapport à ses prédécesseurs au détriment de la complexité. L'introduction de nombreux modes, tels que la taille de bloc variable et l'amélioration de la précision des vecteurs par exemple, augmentent en effet la complexité. La puissance de calcul nécessaire est alors très élevée, notamment au codeur où tous les modes doivent être évalués. Afin de réduire l'impact du nombre de modes sur la puissance de calcul, des compromis peuvent être effectués : réduction du nombre de modes ou de la précision, estimation de mouvement plus grossière et plus rapide. Cela a bien sûr un impact sur les performances d'encodage.

1.2.3 Qualité de la compensation de mouvement

L'estimation de mouvement a un impact sur les performances de compression. En effet le débit peut sensiblement augmenter si les vecteurs ne sont pas adaptés à la séquence, et dans le cas de la compression avec pertes (comme H.264) la qualité de la vidéo peut baisser. Compte tenu des nombreux modes de codage possibles, la compensation de mouvement dans H.264 a un impact considérable sur la compression. Cette section vise à introduire des outils de mesure et de comparaison de la qualité d'un estimateur de mouvement pour la compression vidéo.

1.2.3.1 Evaluation de la qualité de l'image

La compression vidéo avec pertes exploite les failles du système visuel humain pour réduire la quantité des informations à transmettre. La qualité des images est donc réduite de façon plus ou moins visible. L'évaluation de la qualité d'une image ou d'une vidéo est très complexe. Il existe des métriques de qualité objectives, basées sur un calcul de distorsions comme le PSNR (Peak Signal to Noise Ratio) très utilisé dans le traitement du signal en général :

$$PSNR = 10 \times log\left(\frac{MaxAmplitude^2}{\frac{1}{N}\times\sum_0^{N-1}(originale-reconstruite)^2}\right)$$

Le PSNR mesure une erreur quadratique moyenne non pondérée sur toute l'image. Une altenative est proposée par la SSIM (Structural SIMilarity) [WBSS04] qui pondère les défauts en fonction de leur voisinage. En effet l'oeil humain perçoit moins les défauts dans les zones texturées que dans les zone homogènes. D'une manière générale, ces métriques sont très limitées pour évaluer la façon dont l'oeil humain perçoit les défauts dans une image et la façon dont le cerveau les interprète.

1.2.3.2 Méthode d'évaluation de la qualité du champ de vecteur

Le contexte étant la compression vidéo, les performances des estimateurs de mouvements ne doivent pas être considérées intrinsèquement, mais en prenant en compte à la fois la qualité de la vidéo et le débit du flux. Il faut donc considérer l'ensemble de l'encodeur associé à l'estimateur de mouvement. Malgré ses limitations, la métrique débit/PSNR est très utilisée dans la compression vidéo et sera donc conservée dans cette étude. La compensation de mouvement, dans la compression vidéo vise à réduire la différence entre une prédiction et l'image source, indépendament du contenu, afin de réduire le débit.Comme le but n'est pas d'évaluer la qualité de l'image, mais bien l'efficacité d'estimateurs de mouvement dans un encodeur vidéo, le PSNR, conjointement avec le débit de la vidéo fournira alors une métrique pertinente.

Pour une comparaison des techniques d'estimation de mouvement uniquement, il convient de s'affranchir de certaines décisions de l'encodeur. Par exemple le module de régulation de débit doit être désactivé, et il faut travailler avec une quantification constante. De plus, les paramètres doivent être les mêmes pour tous les estimateurs à tester.

Pour chaque encodeur et pour différents pas de quantification (quatre ici) le PSNR moyen et le débit moyen sont mesurés sur chaque séquence. On peut donc en extraire des courbes débit/PSNR (Fig. 1.11). Plus la courbe est haute, plus la qualité est conservée à un débit donné, et donc plus l'encodeur est performant. Les performances

relatives des encodeurs peuvent ensuite être calculées grâce à l'aire définie par l'axe des abscisses et la courbe de PSNR. Elles peuvent être exprimées en gain de compression à qualité (PSNR) constante ou en gain de qualité à débit constant (Tab. 1.4).

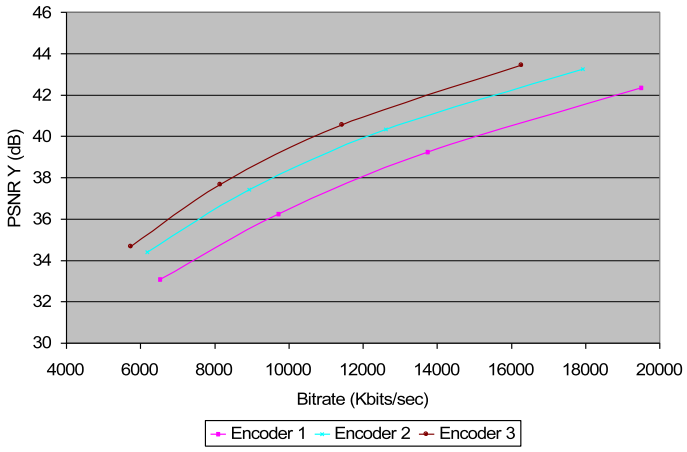


FIG. 1.11 – Courbe débit/distorsion pour la séquence 1

Encodeur \ Séquence vidéo	Seq. 1	Seq. 2	Seq. 3
Encodeur 1	0 (Encoder de référence)		
Encodeur 2	10%	7%	5%
Encodeur 3	15%	12%	20%

TAB. 1.4 – Gains de débit à qualité constante

Comme les paramètres de l'encodeur sont maîtrisés et fixés, seul l'estimateur de mouvement change. On peut donc en déduire que si globalement l'encodeur affiche de meilleures performances, c'est que l'estimateur de mouvement est mieux adapté. Le choix de la séquence vidéo a également un impact sur les résultats, il convient donc de prendre un échantillon significatif de séquences vidéos.

1.3 Conclusion

Dans ce chapitre, nous avons défini les notions de base concernant les images numériques et la compression vidéo. Les standards de compression ont pour but d'assurer une compatibilité de services de codage. Leur évolution permet un codage de plus en plus efficace, ouvrant la voie à la télévision haute définition. Une des avancées les plus notables dans H.264 concerne la phase de compensation en mouvement qui autorise un nombre de références et de modes de prédiction important.

Le gain en termes d'efficacité de codage est toutefois obtenu au détriment de la complexité de la phase d'estimation de mouvement des encodeurs vidéo. Le passage à des formats haute définition, qui introduisent à la fois plus de données à traiter et une cadence plus importante, amplifie encore très fortement cette complexité.

De nombreux efforts de recherche ont été concentrés sur la réduction de la quantité des calculs nécessaires à l'opération d'estimation de mouvement, tout en préservant le maximum de qualité. Le chapitre suivant présente un état de l'art des techniques utilisées sur le plan algorithmique.

Chapitre 2

Techniques d'estimation de mouvement

Ce chapitre présente un état de l'art des techniques d'estimation de mouvement, qui cherchent à évaluer les mouvements entre deux images. L'information de mouvement peut être utilisée pour la conversion de standard, le désentrelacement, le débruitage, la vision informatique, ou encore la compression vidéo. Ces techniques sont généralement classées en trois groupes : les méthodes basées gradient, basées fréquence, et la mise en correspondance de blocs.

Dans le cas des techniques standard de compression vidéo (MPEG-2, MPEG-4), chaque image est encodée bloc par bloc. Les vecteurs de mouvement sont exploités afin de réduire les redondances temporelles et ainsi réduire la taille nécessaire au codage de la séquence. H.264 prévoit de plus une compensation de mouvement avec des blocs de taille variable afin d'adapter l'information de mouvement aux données.

L'opération d'estimation de mouvement s'avère cruciale dans un encodeur vidéo. En effet, d'une part la qualité de l'estimation influe sensiblement sur les performances de compression, et d'autre part c'est l'opération qui nécessite le plus de puissance de calcul. Notons que nous ne considérons ici que des mouvements de translation. Des modèles de mouvements prenant en compte des paramètres supplémentaires pour chaque bloc conduiraient à une complexité de calcul démesurée vis à vis des processeurs actuels. De plus, la compensation de mouvement du standard de compression vidéo H.264 n'accepte que des vecteurs de translation.

Les techniques basées gradient sont tout d'abord exposées, suivies des méthodes fréquentielles. Ensuite, les techniques de mise en correspondance de blocs sont plus largement présentées. Enfin les algorithmes utilisés pour l'estimation de mouvement dans H.264 sont étudiés.

2.1 Méthodes basées gradient

Ces méthodes sont basées sur l'hypothèse de l'invariance de la luminance d'un objet en mouvement.

Soient $x = (x_1, x_2)$ un point de l'image, $I_n(x)$ son intensité lumineuse pour l'image n et d le déplacement du point x entre les images $n-1$et n. La différence inter-image, ou DFD (Displaced Frame Difference) est définie par :

$$DFD(x,d) = I_{n-1}(x+d) - I_n(x) \tag{2.1}$$

En théorie, DFD atteint la valeur 0 lorsque d est égal au déplacement réel. En pratique, d prend des valeurs fractionnaires, nécessitant une interpolation et générant des erreurs. De plus il existe du bruit dans l'image, des variations d'illumination et des occlusions. Le but est donc de trouver le déplacement d qui minimise la DFD. Les techniques d'estimation de mouvement basées gradient s'appuient sur le réarrangement de l'équation (2.1) afin d'avoir accès au déplacement d itérativement, grâce à un développement limité en série de Taylor au premier ordre en d_i (vecteur déplacement à l'itération i) :

$$I_{n-1}(x+d_{i+1}) = I_{n-1}(x+d_i) + (d_{i+1}-d_i)^T \bigtriangledown I_{n-1}(x+d_i) + e_{n-1}(x) \tag{2.2}$$

Où $\bigtriangledown$ est l'opérateur gradient bidimentionnel, T est l'opérateur transposé, et $e_{n-1}(x)$ représente les termes de plus hauts ordres négligés. On obtient donc d'après (2.1) et (2.2) :

$$DFD(x,d_{i+1}) = DFD(x,d_i) + (d_{i+1}-d_i)^T \bigtriangledown I_{n-1}(x+d_i) \tag{2.3}$$

Comme il n'existe en général pas de solution analytique, on a recourt à des méthodes récursives [BLBP87, CR83, KK04b, NR79, WR84] où le déplacement d est estimé itérativement grâce à la méthode de la plus grande pente [NR79]. Le principe est de considérer que l'opposé du gradient d'une fonction pointe dans la direction où la fonction décroît le plus rapidement (Fig 2.1). On peut donc écrire :

$$d_{i+1} = d_i - \epsilon \cdot DFD(x,d_i) \cdot \bigtriangledown_{d_i} DFD(x,d_i) = d_i - \epsilon \cdot DFD(x,d_i) \cdot \bigtriangledown I_{n-1}(x-d_i) \tag{2.4}$$

ou plus simplement

$$d_{i+1} = d_i - \epsilon \cdot sign(DFD(x,d_i)) \cdot sign(\bigtriangledown I_{n-1}(x-d_i)) \tag{2.5}$$

où ϵ est un terme de mise à jour. Le choix de ϵ requiert un compromis : si une grande valeur est choisie, la convergence est rapide mais oscille. A l'inverse, si ϵ est faible, la convergence est plus précise mais aussi plus lente. Ce terme peut être constant [NR79] ou adaptatif [CR83, WR84].

Les algorithmes utilisés dans [BLBP87, KK04b] sont basés sur la technique de Wiener. Le mouvement d est calculé itérativement, en utilisant une approche de pseudo-inverse de matrice. Cette méthode obtient les meilleurs résultats au prix de calculs coûteux.

Les méthodes pel-récursives basées gradient conduisent à un champ de vecteur dense. C'est à dire qu'un vecteur est calculé par pixel. Elles sont donc très sensibles au bruit puisque la DFD n'est calculée que sur un pixel. Afin d'augmenter la robustesse, plusieurs pixels voisins peuvent être pris en compte. Comme les méthodes sont basées sur des développements limités, ces méthodes ne sont pas adaptées à l'estimation de grands déplacements. Une approche hiérarchique peut améliorer les résultats grâce à une représentation multirésolution des images.

Le champ de vecteur résultant est adapté à des traitements pixel par pixel, tel que du filtrage ou de la conversion de standard.

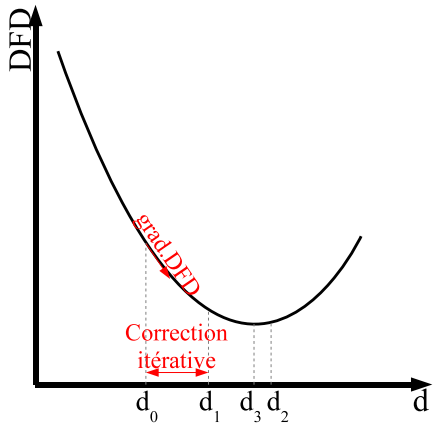


FIG. 2.1 – Descente de gradient

2.2 Méthodes fréquentielles

Ces méthodes d'estimation de mouvement sont basées sur des transformations qui permettent de prendre en compte les propriétés fréquentielles du mouvement à savoir qu'une translation dans le domaine de l'image se caractérise par un déphasage dans le domaine fréquentiel [CM87, Sto86, Tho87]. Ces méthodes basées sur une "mesure" du mouvement sont robustes au bruit et donnent le mouvement réel.

Soit f une image et g sa translatée de (dx, dy).

$$g(x,y) = f(x+dx, y+dy)$$

En appliquant la transformée de fourrier on a :

$$G(u,v) = F(u,v)\, e^{-2\Pi j(u\times dx + v\times dy)}$$

On peut écrire :

$$TF\left[f(x,y) \otimes g(x,y)\right] = F(u,v) \times G^*(u,v) = \left|F(u,v) \times G^*(u,v)\right| e^{j\theta}$$

où $\otimes$ est l'opérations de convolution et * est la conjugéé.
On a donc

$$e^{j\theta} = \frac{F(u,v) \times G^*(u,v)}{\left|F(u,v) \times G^*(u,v)\right|}$$

En appliquant la transformée de Fourier inverse on obtient :

$$TF^{-1}\left[e^{j\theta}\right] = \delta(x-dx, y-dy) = TF^{-1}\left[\frac{F(u,v) \times G^*(u,v)}{\left|F(u,v) \times G^*(u,v)\right|}\right] \tag{2.6}$$

La corrélation de phase est basée sur les transformées fréquentielles de deux images successives. La différence entre les phases donne la corrélation dont la transformée inverse (équation (2.6)) révèle des pics. La position de ces pics correspond à des déplacements entre les deux images. Les propriétés du domaine fréquentiel font que le mouvement est mesuré avec précision, mais les points d'application du mouvement

sont inconnus. Donc en pratique, la corrélation de phase est suivie d'une étape de mise en correspondance [Tho87]. Chaque pic (il existe un pic par mouvement dans le voisinage considéré) révélé dans la surface de corrélation est un vecteur candidat pour la mise en correspondance.

La taille de la transformée contraint le déplacement maximal détectable. Les avantages de cette méthode est qu'elle est peu sensible au bruit, aux variations d'intensité lumineuse (éclair, explosion) et que les vecteurs résultants sont précis (précision subpixel). Cette technique est intéressante pour du recalage d'image par exemple, mais implique beaucoup de calculs.

2.3 Mise en correspondance de blocs

Les méthodes de mise en correspondance sont les plus naturelles. Elles sont basées sur la corrélation spatiale entre deux images : l'image courante et l'image de référence. Les valeurs des pixels sont directement exploitées afin de mettre en correspondance deux régions dans deux images successives. La position relative des deux régions donne directement le mouvement.

Chaque image est découpée dans un premier temps en blocs de taille $N \times M$, puis pour chaque bloc de l'image courante, le bloc le plus ressemblant (au sens de la minimisation d'un critère d'erreur) est recherché dans l'image de référence (figure 2.2). Contrairement aux techniques précédentes ou le mouvement est évalué à partir de certains paramètres, ici on part d'une solution et sa pertinence est évaluée.

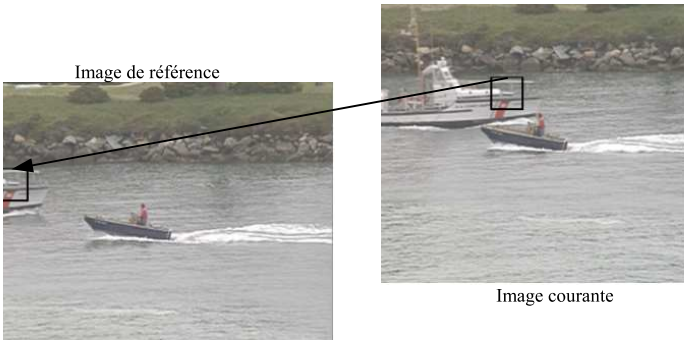


FIG. 2.2 – Mise en correspondance de blocs

2.3.1 Formulation mathématique

Le but est de trouver le bloc le plus ressemblant au bloc courant dans une image de référence. On définit un critère d'erreur f pour mesurer la différence $\varepsilon_{t,\tau}$ entre deux blocs :

$$\varepsilon_{t,\tau} = f\left(g_t\left(x\right) - g_{t-\tau}\left(x + d_{t,\tau}\left(x\right)\right)\right), \forall x \in B \subset R$$

où $g_t$et $g_{t-\tau}$ sont respectivement l'image courante et l'image de référence, $d_{t,\tau}$ est le vecteur déplacement associé à l'image courante, x est un point du bloc B de l'image

R. Le vecteur d optimal minimise la mesure d'erreur ϵ. Comme il n'est en général pas possible de définir $\varepsilon_{t,\tau}$, l'opération d'estimation de mouvement s'appuie généralement sur une méthode par tâtonnement (*trial and error*) : ϵ est évaluée pour toutes les valeurs de d candidates.

Ce calcul demande une grande puissance de calcul car il implique tous les pixels du bloc ($M \times N$) et est calculé pour chaque déplacement. Il existe plusieurs critères permettant d'évaluer l'amplitude de l'erreur de prédiction ϵ. Les plus courants sont présentés ici :

– La Somme des Différences Absolues (SAD)

$$SAD(d) = \sum_{x \in B} |g_t(x) - g_{t-\tau}(x + d_{t,\tau}(x))|$$

– La Somme des Erreurs Quadratiques (SSE)

$$SSE(d) = \sum_{x \in B} (g_t(x) - g_{t-\tau}(x + d_{t,\tau}(x)))^2$$

– La Somme des Différences Transformées (SATD)

$$SATD(d) = \sum |H(g_t(x) - g_{t-\tau}(x + d_{t,\tau}(x)))|$$

où H est la transformée de Hadamard.

La SAD est le critère le plus utilisé car sa complexité est réduite (nécessite seulement des soustractions et additions). La SSE représente l'énergie résiduelle. Dans un problème de compensation de mouvement, minimiser la SSE correspond à maximiser le PSNR (Peak Signal to Noise Ratio). Ce critère est moins utilisé que le précédent à cause de sa complexité (nécessite des multiplications) et de sa sensibilité au bruit (une grande différence due au bruit sur un pixel de l'image a un grand impact sur la SSE). La SATD est largement répandue dans un contexte d'encodage vidéo avec compensation de mouvement : elle représente le coût de codage du résidu grâce à une transformée rapide. Elle nécessite néanmoins l'implantation de la transformée de Hadamard, qui peut s'avérer coûteuse pour une estimation de mouvement temps réel. D'autres critères simples et en général peu robustes existent tels que la somme des bits qui diffèrent entre la référence et le bloc courant.

Le coût des vecteurs est généralement pris en compte également. En effet, dans un contexte d'encodage vidéo, l'estimation de mouvement permet de réduire la taille des données en éliminant les redondances temporelles existant entre deux images successives. Au lieu de décrire un macro-bloc avec la valeur des pixels, il est reconstruit à partir d'une image de référence, d'un déplacement et d'un résidu. Il est donc nécessaire de transmettre les vecteurs déplacements et les résidus pour reconstruire chaque image. Par conséquent, l'erreur d'estimation (SSE, SAD ou SATD) n'est pas le seul paramètre à prendre en compte, pour réduire le débit nécessaire à la transmission du flux compressé : le coût de codage des vecteurs doit aussi être pris en compte. Cela aura pour effet de réduire l'entropie du champ de vecteur. La fonction à minimiser devient [SW98] :

$$J_{MOTION} = D_{DFD} + \lambda_{MOTION} \cdot R_{MOTION} \tag{2.7}$$

Ainsi l'estimation de mouvement prend en compte simultanément la minimisation de l'erreur $D_{DFD} = \varepsilon_{t,\tau}$(SSE, SAD ou SATD) et du coût de codage du vecteur correspondant R_{MOTION} en utilisant un multiplicateur de Lagrange λ_{MOTION} adaptatif. Ce coefficient varie en fonction du pas de quantification q associé à la correction : plus la quantification est faible, plus l'erreur coûte cher, et moins le coût du vecteur à un impact. Donc plus q est grand plus λ_{MOTION} est grand.

Dans ce qui suit, la SAD a été choisie pour sa simplicité d'implantation en vue d'obtenir des performances temps réel. Cependant un raisonnement similaire peut être conduit avec un autre critère à minimiser.

L'estimation de mouvement par mise en correspondance consiste à minimiser l'équation (2.7) en fonction du déplacement d. Différents algorithmes ont été inventés afin d'augmenter les performances de l'estimation du mouvement aussi bien en terme de qualité du résultat qu'en terme de vitesse d'exécution. Un état de l'art des méthodes existantes est présenté ci-dessous.

2.3.2 Recherche exhaustive

L'algorithme de mise en correspondance par recherche exhaustive est très simple et a l'avantage de donner la solution optimale dans un voisinage donné. Le voisinage (ou fenêtre de recherche) est borné. La formulation mathématique de la méthode est donnée par l'algorithme 2.1 avec une fenêtre de recherche de +/- p et un bloc de taille $N \times M$.

Algorithme 2.1 Recherche exhaustive

$for\ \Delta i = -p..p$ (espace de recherche vertical)
 $for\ \Delta j = -p..p$ (espace de recherche horizontal)
  $for\ k = 1..M$ (hauteur de bloc)
   $for\ l = 1..N$ (largeur de bloc)
    $SAD(\Delta j, \Delta i) += |x_1(k,l) - x_2(k + \Delta i, l + \Delta j)|$
   $end\ l$
  $end\ k$
 $end\ \Delta j$
$end\ \Delta i$

x_1 est l'image courante et x_2 l'image de référence

Cette méthode est de loin celle qui demande le plus de puissance de calcul. En effet, une erreur d'estimation est calculée pour tous les déplacements possibles d'amplitude maximale p (Fig. 2.3), soit au total $(2p + 1)^2$ candidats. Le vecteur de mouvement résultant est celui qui minimise ϵ. Le tableau 2.1 donne l'ordre de grandeur du nombre d'opérations nécessaires à l'estimation de mouvement (en tenant compte uniquement du calcul de SAD). Pour une image HD, cela correspond à plus de 50 processeurs pentium bi-coeurs à 2.7 GHz, en considérant les calculs optimisés et le processeur utilisé à 100%.

La recherche exhaustive demande énormément de puissance de calcul, notamment pour la haute définition où la fenêtre de recherche doit être élargie pour prendre en compte tous les mouvements possibles.

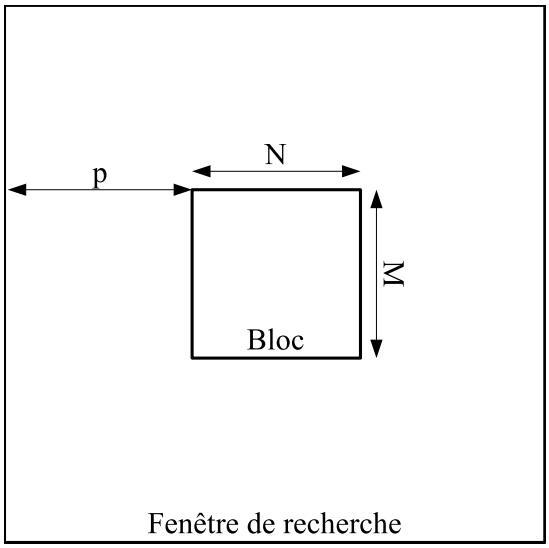


FIG. 2.3 – Fenêtre de recherche

Taille de l'image	Taille de bloc	Amplitude p	Nombre d'opérations par seconde (Gops)
CIF : 352x288	8x8	+/- 8	1.45
25 i/s	16x16		1.46
HD 1280x720	8x8	+/- 64	1522
50 i/s	16x16		1531

TAB. 2.1 – Nombre d'opérations pour l'estimation de mouvement avec la SAD

Pour palier au grand nombre d'opérations nécessaires et à une amplitude p bornée, plusieurs techniques peuvent être utilisées :

- stopper les calculs le plus tôt possible (éviter les opérations inutiles).
  Le calcul peut être arrêté dès que les valeurs déjà accumulées dépassent le minimum courant. Ainsi les boucles Δi et Δj peuvent être arrêtées prématurément si $SAD(\Delta j, \Delta i) > minimum$. Comme la SAD peut augmenter très vite lorsque l'on s'éloigne de l'optimum, de nombreux calculs sont évités (élimination de candidats),
- limiter la recherche à un nombre restreint de candidats.
  La recherche n'est alors plus exhaustive, l'idée générale est de réduire le nombre de candidats en supposant que la SAD varie de façon monotone dans la fenêtre de recherche. Il est alors possible de suivre une direction de descente privilégiée (algorithmes prédictifs et récursifs),
- simplifier les calculs effectués.
  Il est possible de simplifier le critère d'erreur afin de réduire les besoins lors de l'implantation. Par exemple le nombre de bits peut être réduit pour abaisser le coût d'une implantation câblée ou bien le nombre de pixels pris en compte dans le critère d'erreur afin de réduire le nombre de calculs.

Les sections suivantes présentent les techniques existantes dans la littérature pour accélérer l'estimation de mouvement.

2.3.3 Élimination de candidats

Ce type d'algorithme a été développé afin de réduire le nombre de calculs de la recherche exhaustive. Une estimation rapide du critère d'erreur peut permettre d'éliminer certains candidats sans effectuer la totalité des calculs. Le résultat est le même que pour la recherche exhaustive avec statistiquement beaucoup moins de calculs. Différentes techniques existent dans la littérature.

L'algorithme proposé par Li W. et Salari E.[LS95] est initialisé pour chaque bloc avec un vecteur prédit pour lequel la SAD complète est calculée. Ensuite pour les autres candidats dans la zone de recherche $+/-p$ une estimation rapide de la SAD est calculée. Le calcul rapide est choisi tel que l'estimation soit toujours inférieure à la SAD. Si l'estimation est supérieure au minimum courant, le candidat peut être éliminé, sinon le calcul complet doit être effectué pour pouvoir prendre une décision. Si le résultat est toujours inférieur, le candidat est le nouvel optimum sinon il est éliminé. La technique de prédiction permet d'augmenter la probabilité d'éliminer des candidats en choisissant un vecteur d'initialisation ayant une grande probabilité de conduire à une SAD faible. Les performances de cet algorithme dépendent énormément de l'initialisation, dans le cas ou elle est mauvaise, le nombre de calcul peut alors être supérieur à celui de la recherche exhaustive.

L'algorithme de Y-S Chen [CHF01] est basé sur la technique "Winner-update". Une liste de tous les candidats est définie. L'initialisation est effectuée avec une SAD partielle pour tous les candidats et la liste est triée suivant des valeurs croissantes de SAD partielles. Ensuite le candidat dont la valeur est la plus faible voit sa SAD incrémentée puis la liste est à nouveau ordonnée. L'algorithme s'arrête lorsque le candidat de plus faible valeur a une SAD complète, c'est alors l'optimum. Cet algorithme évite beaucoup d'opérations dues au calcul de SAD, cependant il introduit beaucoup d'opérations de manipulation de liste.

Le temps d'exécution de ces algorithmes peut varier énormément suivant les séquences vidéos, ce qui n'est pas favorable à une implantation temps réel. Ces algorithmes permettent de réduire théoriquement le nombre de calculs tout en donnant un optimum global. Cependant leurs temps d'exécution restent élevés pour atteindre le temps réel avec les contraintes de MPEG-4 haute définition (large fenêtre de recherche, précision subpixélique, taille de bloc variable).

2.3.4 Méthodes récursives

Les méthodes récursives permettent d'estimer le mouvement à l'aide de plusieurs passes. Le but est de ne pas parcourir toute la fenêtre de recherche. Leur inconvénient principal est de donner des résultat sub-optimaux. En effet le critère d'erreur n'est en général pas monotone dans la fenêtre de recherche, la méthode peut donner un optimum local. Il existe dans la littérature des méthodes systématiques et prédictives. Ces dernières se servent des résultats précédemment obtenus pour réduire davantage la fenêtre de recherche et augmenter les performances.

2.3.4.1 Méthodes systématiques

Pour chaque bloc, le centre de la recherche est la position initiale, c'est à dire le vecteur nul.

Algorithme 2D logarithmique Cet algorithme proposé par Jain et Jain [JJ81] est basé sur l'hypothèse que la valeur du critère d'erreur croît de façon monotone lorsque l'on s'éloigne du minimum. L'espace des candidats est sous échantillonné pour réduire le nombre de calculs. L'optimum est recherché par pas successifs. A chaque étape l'erreur est calculée pour un candidat dans chacune des quatre directions principales, d'abord avec un pas grossier. La recherche se poursuit autour du candidat ayant la SAD la plus faible. Lorsque qu'un minimum grossier est trouvé, le pas est réduit de moitié et la recherche reprend avec quatre candidats plus raprochés (figure 2.4). Lorsqu'un minimum est trouvé avec un pas de deux pixels, les huit voisins sont testés et l'algorithme s'arrête. Cinq erreurs sont comparées à chaque étape (le centre et quatre directions suivant une quatre-connexité), sauf à la dernière étape ou neuf comparaisons sont effectuées (8 connexité). Le nombre d'étapes est dépendant des données.

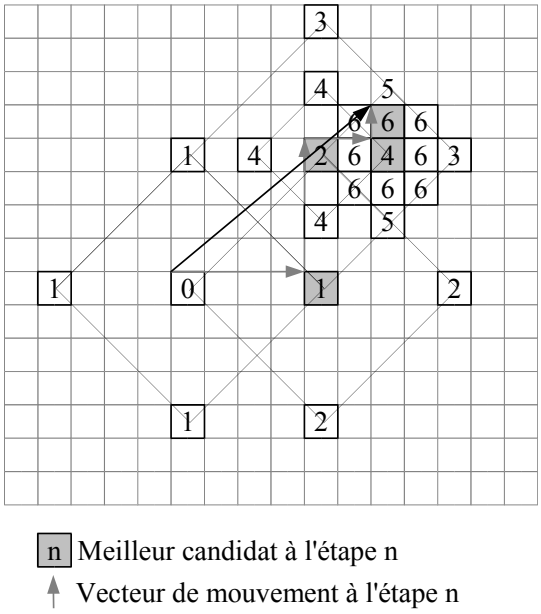


FIG. 2.4 – Algorithme 2D logarithmique

Algorithme à 3 pas Cette méthode proposée par Koga et Linuma [TKA+81] est une recherche du bloc candidat avec un nombre d'étapes fixes (3 pas). A la première étape, la recherche est effectuée sur les huit blocs candidats distants de $d = p/2$ pixels du bloc de référence, le meilleur est retenu comme nouveau centre de recherche. A l'étape suivante la distance d est divisée par deux et le même calcul est effectué. L'algorithme est itéré jusqu'à ce que $d = 1$ (figure 2.5).

Le nombre de calculs effectués est constant (25 candidats), ce qui permet d'évaluer le temps d'exécution avec précision si le temps d'accès aux données est déterministe.

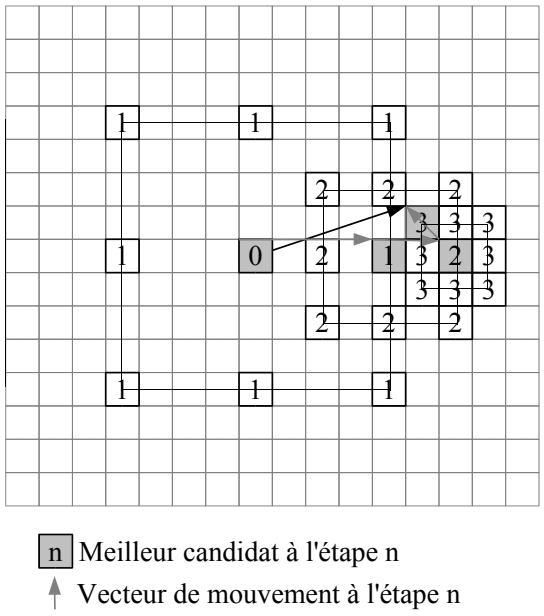


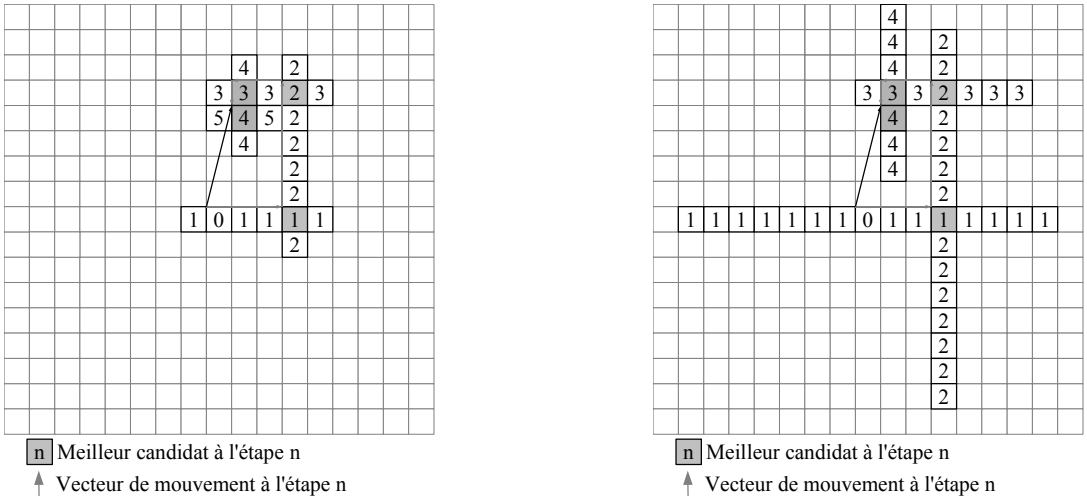
FIG. 2.5 – Algorithme à 3 pas

Des variantes ont été développées, par exemple pour rendre dynamique le nombre de pas grâce à un seuil [KLW87], pour réduire le nombre de candidats [GHA90, LL99] ou bien pour privilégier les faibles mouvements [LZL94, PM96]. Dans ces variantes, la notion d'arrêt prématuré et de seuil sont introduites. La recherche s'arrête dès que la solution trouvée est acceptable, c'est à dire dès que le critère d'erreur est en dessous d'un certain seuil. De nombreux calculs inutiles sont ainsi évités, mais la durée de l'estimation de mouvement devient variable, alors que pour une implantation temps réel, le pire cas doit être pris en compte.

L'inconvénient majeur de ces deux techniques est la faible valeur de l'amplitude du mouvement, en particulier pour les méthodes à pas fixe. En effet l'algorithme fonctionne bien pour des mouvement de faible amplitude mais est très sujet à tomber dans des minima locaux dans le cas de grands déplacements notamment dans le cas de séquences vidéo haute définition.

Recherche directionnelle Développées pour réduire la complexité algorithmique par rapport aux précédentes techniques tout en étant capable d'augmenter la fenêtre de recherche, ces techniques procèdent à une minimisation dans une direction à la fois. La minimisation du problème à deux variables est ramené à un problème à une seule variable. L'algorithme OTS (One-at-a-Time Search) proposé par Srinivasan [SR85] suppose que le critère d'erreur croît de façon monotone à mesure que l'on s'éloigne de l'optimum. La recherche est effectuée dans une direction à la fois. Le meilleur candidat est d'abord recherché sur la ligne du centre de la fenêtre de recherche (horizontalement), en suivant une direction de descente, puis verticalement, jusqu'à tomber dans un minimum (figure 2.6-a). Cet algorithme limite effectivement le nombre de calculs, mais tombe souvent dans un minimum local. De plus il n'est pas assez régulier pour être réalisé dans un composant câblé. Afin de palier à ces problèmes, l'algorithme

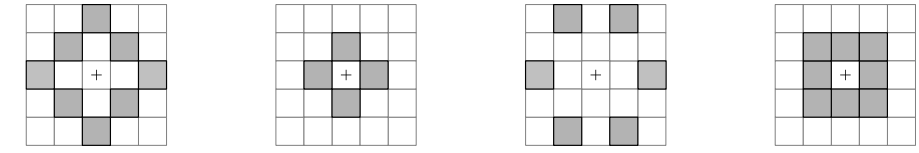
proposé par Chen [CCC94] s'inspire de OTS avec l'idée de la recherche exhaustive (figure 2.6-b). La ressemblance est évaluée pour tous les déplacements sur une ligne horizontale. Puis l'emplacement du meilleur candidat est choisi comme nouveau centre pour une recherche verticale. Cette étape est répétée en réduisant l'amplitude de la recherche de moitié (recherche horizontale plus verticale).



(a) Algorithme OTS (b) Algorithme 1D exhaustif (1DFS)

FIG. 2.6 – Recherche directionnelle +/-7 pixels

Diamond Search Proposé initialement par Lurng-Kuo Liu and Ephraim Feig [LF96] avec le nom "algorithme d'estimation de mouvement par descente de gradient basé bloc" puis formulé "recherche en diamant" par Shan Zhu and Kai-Kuang Ma [ZM97] cette technique est basée sur le suivi d'une direction de descente du critère d'erreur. Le motif en diamant (Fig. 2.7) est basé sur l'étude de la distribution de vecteurs de mouvement à partir de séquences test largement utilisées. Comme indiqué dans [ZM97] de 52.76% à 96.09% des vecteurs de mouvement sont compris dans un cercle de diamètre deux pixels, centré sur la position initiale du bloc. De plus, en considérant le critère d'erreur monotone dans le voisinage de l'optimum, il est possible d'estimer des mouvements réduits avec précision tout en limitant le nombre de calculs.



(a) Diamant large (b) Diamant réduit (c) Hexagon (d) Carré

FIG. 2.7 – Motifs de recherche

L'algorithme débute à la position co-localisée (vecteur nul) dans l'image de référence, et calcule huit SAD autour du centre suivant le motif en diamant large. Si le minimum se trouve sur le bord (il existe une direction de descente), alors l'opération est réitérée avec la position ayant la SAD la plus basse comme nouveau centre. Il est à noter qu'il y aura moins de SAD à calculer que lors de la première itération du fait du recoupement des points du motif avec l'itération précédente. Lorsque le minimum se trouve au centre du motif large, l'opération est une dernière fois exécutée avec le motif réduit pour conduire à l'optimum.

Comme le mouvement est statistiquement souvent très faible, le résultat est obtenu avec peu d'itérations. Quelques optimisations évidentes comme un nombre d'itérations borné, des critères d'arrêt anticipés basés sur un seuil de la SAD sont également proposées dans [ZM97].

Plusieurs dérivés de cet algorithme ont été développés afin d'accélérer la recherche et d'augmenter sa robustesse [TRRK98]. Un motif hexagonal (Fig. 2.7) peut être utilisé pour augmenter la robustesse pour les mouvements de grande amplitude [AH02, ZLC02, ZLCP04].

Ces algorithmes rapides ont été développés pour détecter des mouvements de quelques pixels d'amplitude. Lorsque les mouvement ont une grande amplitude ces techniques perdent de leur efficacité et tombent facilement dans des minima locaux. De plus, le nombre d'itérations et donc la charge de calcul augmentent dans ce cas significativement.

2.3.4.2 Méthodes prédictives

Dans les séquences vidéo naturelles, il existe une corrélation spatiale et temporelle des champs de vecteurs. Il est donc possible d'accélérer la recherche tout en augmentant la robustesse des algorithmes en utilisant cette corrélation. La principale innovation est de sélectionner un centre de recherche pour effectuer une recherche en diamant. Le centre n'est donc plus le vecteur nul, comme dans une recherche en diamant classique, mais un vecteur intelligemment sélectionné dans le voisinage. La méthode est initialement introduite par *De Haan et Al.* dans [dHBHO93].

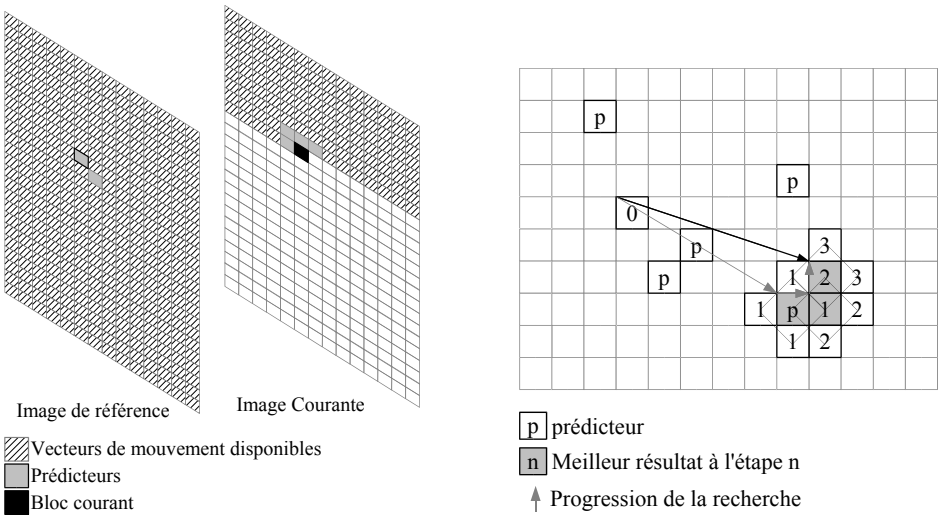
L'algorithme comprend deux étapes (illustrées sur la figure 2.8) :

1. La phase de prédiction

   Une liste de prédicteurs est construite à partir des vecteurs de mouvement déjà calculés (blocs voisins, image précédente). Ces candidats sont évalués en calculant la SAD résultante. Le plus pertinent est sélectionné comme centre d'une recherche en diamant.

2. La phase de raffinement

   Une recherche locale est effectuée autour du nouveau centre, pour raffiner le mouvement prédit. Une technique récursive avec un motif de recherche en diamant est souvent utilisé pour cela.



(a) Prédiction (b) Raffinement (recherche en diamant)

FIG. 2.8 – Algorithme EPZS

PMVFAST [TAL01b] (Predictive Motion Vector Field Adaptive Search Technique) est un algorithme prédictif populaire. L'algorithme utilise les vecteurs estimés des blocs voisins : le bloc supérieur, gauche, supérieur droit, et le co-localisé dans l'image précédente. Un médian est également calculé à partir de ces quatre vecteurs. Le principal avantage est une convergence plus rapide qu'une recherche en diamant conventionnelle grâce à la phase de prédiction.

Beaucoup de variantes ont été proposées pour accélérer la recherche ou augmenter la robustesse [HM99, TAL01a, CZH03, ZLCP04, BdHSvM03, VKM+05].

L'estimateur de mouvement prédictif le plus populaire est sans doute EPZS (Enhanced Predictive Zonal Search) [Tou02]. Il est une amélioration de PMVFAST [TAL01b] et de APDZS (Advanced Predictive Diamond Zonal Search) [TAL01a]. De nouveaux prédicteurs sont considérés, tels qu'un vecteur accélération et des vecteurs voisins du bloc co-localisé dans l'image précédente. La phase de prédiction est considérée très bonne, par conséquent la phase de raffinement est simplifiée en utilisant seulement qu'un motif réduit, soit en diamant (EPZS), soit en carré ou cercle (EPZS^2). Cet algorithme est implanté dans plusieurs logiciels de compression vidéo tels que X.264 ou le JM [TTT02] avec un ensemble de prédicteurs supplémentaires (Fig. 2.9) centrés sur (0 ;0) ou un médian, destinés à améliorer l'estimation de séquences avec beaucoup de mouvements (lorsque la phase de prédiction conventionnelle est inefficace).

2.3.4.3 Méthodes de "vector-tracing"

Dans les algorithmes prédictifs, la pertinence des vecteurs initiaux (ou prédicteurs) a une grande influence sur leur performances. Dans le but d'améliorer cette étape de prédiction, l'interpolation des trajectoires à partir des champs de vecteurs déjà calculés peut fournir des prédicteurs robustes. Dans [MZ00], une technique dite de "vector-

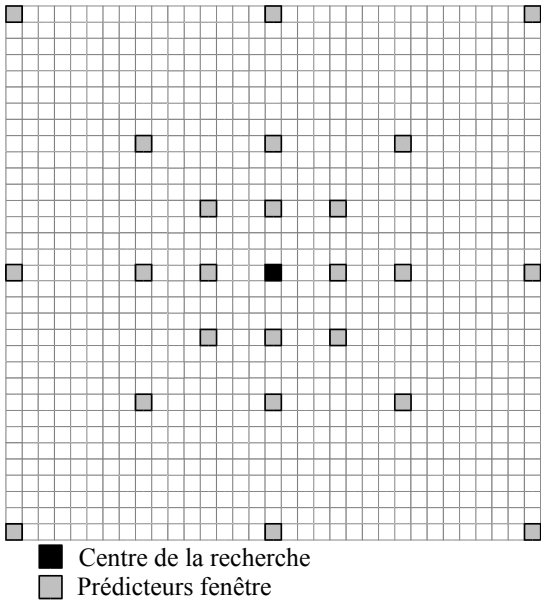


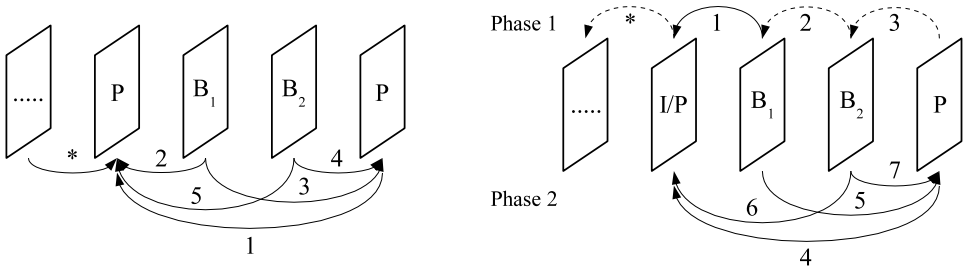
FIG. 2.9 – Prédicteurs supplémentaires pour EPZS

tracing", compatible avec toutes les structures de GOP (cf. paragraphe 1.1.2.3), est associée à un algorithme génétique.

Algorithme de "vector-tracing" : L'algorithme de "vector-tracing" consiste à déterminer des vecteurs initiaux pour une recherche récursive à partir des vecteurs déjà calculés. Pour la compression vidéo, l'image courante et l'image de référence ne sont pas forcément successives, ce qui augmente la disparité entre les images et également l'amplitude des vecteurs de mouvement. Dans [MZ00], les auteurs décrivent deux stratégies d'interpolation des trajectoires : en une phase ou en deux phases (Fig. 2.10 et 2.11). La stratégie en une phase consiste à ne réaliser que les estimations de mouvement nécessaires au processus de codage, alors que celle en deux phases comprend dans la première phase l'estimation de mouvement des images successives dans le but de prédire les trajectoires des estimations de mouvement de la deuxième phase (champs de vecteurs nécessaires pour la compression vidéo).

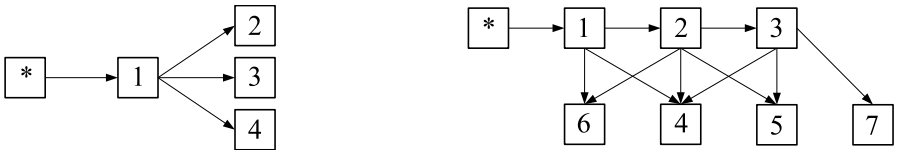
Les mouvements issus des estimations déjà réalisées fournissent des vecteurs initiaux à un processus récursif basé sur un algorithme génétique.

Algorithme génétique : L'algorithme génétique consiste à faire évoluer les vecteurs des estimations récursivement jusqu'à obtenir les vecteurs de mouvement recherchés. L'algorithme est constitué de deux étages (Fig. 2.12) : initialisation et évolution. Dans l'étage d'initialisation, un certain nombre de vecteurs constituant la population initiale sont choisis à partir de l'algorithme de "vector-tracing" et d'élément aléatoires. La phase d'évolution consiste à évaluer un critère d'adaptation (fitness) des individus au problème (dans notre cas c'est la SAD), puis de croiser les individus pour obtenir des mutations, et donc une nouvelle population. Dans cette phase, les meilleurs individus sont croisés afin de converger vers une solution pour le problème donné. Des



(a) 1 phase (b) 2 phases

FIG. 2.10 – Diagrammes schématiques des stratégies de "vector-tracing" 1 phase (a) et 2 phases (b). Les flèches pleines représentent les champs de vecteurs nécessaires pour la compression vidéo et celles en pointillées représentent une étape d'estimation de mouvement utilisée seulement dans le but de prédiction des trajectoires.



(a) 1 phase (b) 2 phases

FIG. 2.11 – Schémas des dépendances pour l'estimation de mouvement avec "vector-tracing" 1 phase (a) et 2 phases (b). Les informations notées "*" viennent de l'estimation de mouvement de la dernière image du GOP précédent.

individus aléatoires sont également générés pour élargir la population autour de ces meilleurs candidats.

En pratique, le nombre de candidats à chaque génération et le nombre d'itérations de l'algorithme sont fixés pour limiter la complexité de l'algorithme. La prédiction temporelle suppose une vitesse constante sur les images considérées. Les étapes de mutation et de génération aléatoires permettent également de prendre en compte des mouvements plus complexes tel que l'accélération.

Grâce aux techniques de "vector-tracing", il est possible d'estimer les mouvements de grande amplitude avec une complexité très inférieure à la recherche exhaustive. L'algorithme nécessite cependant un temps de convergence, au même titre que EPZS, lors des changements de scènes. Dans le cas de l'algorithme à deux phases, la détermination des trajectoires est plus précise compte tenu de l'estimation de mouvement de la première phase, et conduit donc à de meilleurs résultats. Cependant, certains champs de vecteurs sont estimés dans le seul but de prédiction et ne sont pas réutilisés par la suite. La complexité globale de l'estimateur de mouvement s'en trouve donc augmentée.

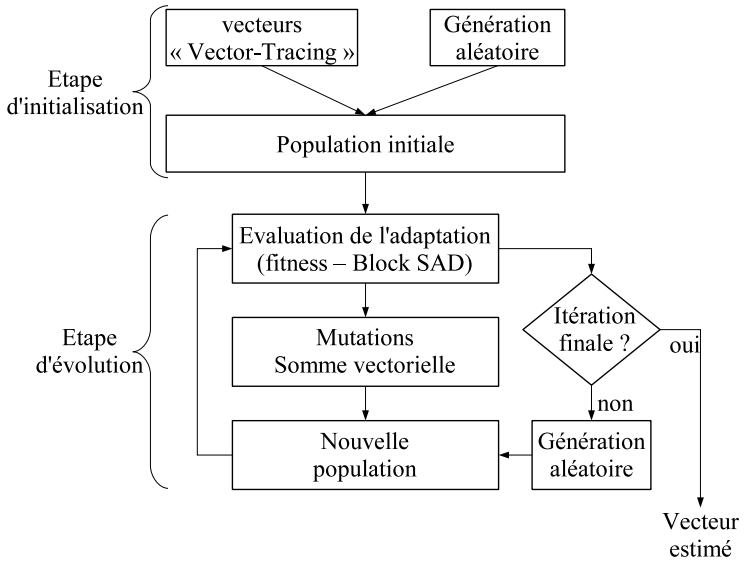


FIG. 2.12 – Diagramme schématique de l'algorithme génétique

2.3.5 Approche multirésolution

Les méthodes exhaustives posent clairement un problème de complexité de calcul, surtout lorsque les déplacements à prendre en compte peuvent aller jusqu'à plusieurs centaines de pixels comme c'est le cas dans certaines séquences en haute définition. Or ces séquences justement posent également problème aux méthodes rapides. Les méthodes prédictives précédentes ne convergent pas et tombent facilement dans des minima locaux, ce qui a pour effet de réduire les performances des encodeurs vidéo : le champ de vecteur ne correspond pas au mouvement réel, ce qui a tendance à augmenter l'entropie et donc le coût de codage des vecteurs. De plus les résidus restent élevés, et des défauts de blocs apparaissent.

Une solution pour combiner une robustesse à des mouvements complexes et de grande amplitude avec un calcul rapide est d'utiliser une technique hiérarchique basée sur une décomposition multirésolution de l'image.

2.3.5.1 Présentation générale

Dans la littérature, on retrouve le terme *estimation de mouvement hiérarchique* associé à différentes techniques : estimation de mouvement à taille de bloc variable, recherche en diamant avec un motif de taille variable, ou bien les approches multirésolution [GRA83, SR99]. C'est plus spécifiquement la troisième technique qui nous intéresse dans cette partie. Par la suite, le terme *estimation de mouvement hiérarchique* sera associé à une technique multirésolution.

Cette technique consiste à réaliser des estimations successives des champs de vecteurs sur des représentations des images à des niveaux de résolution de plus en plus fins. Les correspondances obtenues à un grain élevé permettent de restreindre l'espace de recherche des niveaux plus fins. Les opérations sur le niveau de résolution le plus

élevé consiste en une recherche prédictive avec des prédicteurs fiables issus des niveaux supérieurs.

Les différentes variantes de cette technique résident dans le choix des paramètres :
- la méthode de filtrage et de sous-échantillonnage pour passer d'un niveau à l'autre,
- le nombre de niveaux,
- la taille des blocs à travers la pyramide d'image,
- la prédiction à chaque niveau (propagation des vecteurs entre les niveaux),
- la méthode de recherche à chaque niveau.

Nous allons nous intéresser plus particulièrement à l'estimateur de mouvement développé à Thomson. Il s'agit du HME (Hierarchical Motion Estimator).

2.3.5.2 HME

Une décomposition pyramidale multirésolution des images permet un raffinement successif des vecteurs de mouvement. La corrélation spatiale du champ de vecteur est prise en compte avec une méthode prédictive à chaque niveau.

La première étape de la technique consiste à calculer les pyramides d'images. Pour l'image courante et l'image de référence, les représentations à différents niveaux de résolution sont calculées. Une image d'un certain niveau est obtenue en sous-échantillonnant l'image du niveau inférieur d'un facteur deux (Fig. 2.13). Un filtre passe-bas Gaussien est au préalable appliqué afin d'éviter des problèmes liés au sous-échantillonnage (repliement de spectre notamment).

Ensuite, l'estimation de mouvement proprement dite peut commencer, en partant du niveau le plus élevé (résolution la plus faible). Comme l'image est de taille réduite, une recherche exhaustive peut être effectuée, avec prédiction du centre de la fenêtre de recherche grâce aux vecteurs des blocs voisins déjà calculés. La taille des blocs reste constante à travers la pyramide. Seuls les mouvement dominants sont donc estimés dans les niveaux de faible résolution. Des blocs trop petits ne prendraient pas en compte assez de données et introduiraient du bruit dans le champ de vecteur.

Les vecteurs sont ensuite propagés et raffinés à travers la pyramide (Fig. 2.13). Pour chaque niveau, une technique prédictive est utilisée : des prédicteurs spatiaux (blocs voisins, médian) et hiérarchiques (issus du niveau supérieur et mis à l'échelle) sont évalués. Un recherche très réduite est effectuée autour du meilleur prédicteur (celui donnant le critère d'erreur le plus faible) pour raffiner la prédiction. Pour augmenter la robustesse de l'estimation, une deuxième recherche réduite peut être effectuée autour par exemple du second meilleur prédicteur, mais cela augmente la charge de calcul. Cette technique permet d'obtenir un champ de vecteur à la précision du pixel pour l'image de résolution initiale.

Cet algorithme limite les calculs à effectuer en restreignant la zone de recherche à chaque niveau. La recherche finale autour du meilleur prédicteur peut être précise. La zone de recherche couverte est très importante du fait du mécanisme hiérarchique. La technique est donc peu sensible aux minima locaux. De plus, grâce au mécanisme de prédiction, le champ de vecteur est homogène, et correspond au mouvement réel, ce qui est un atout pour un encodeur vidéo.

En général, ce type d'algorithme est moins populaire qu'un algorithme prédictif classique, car il requiert plus de puissance de calcul pour la construction de la pyramide

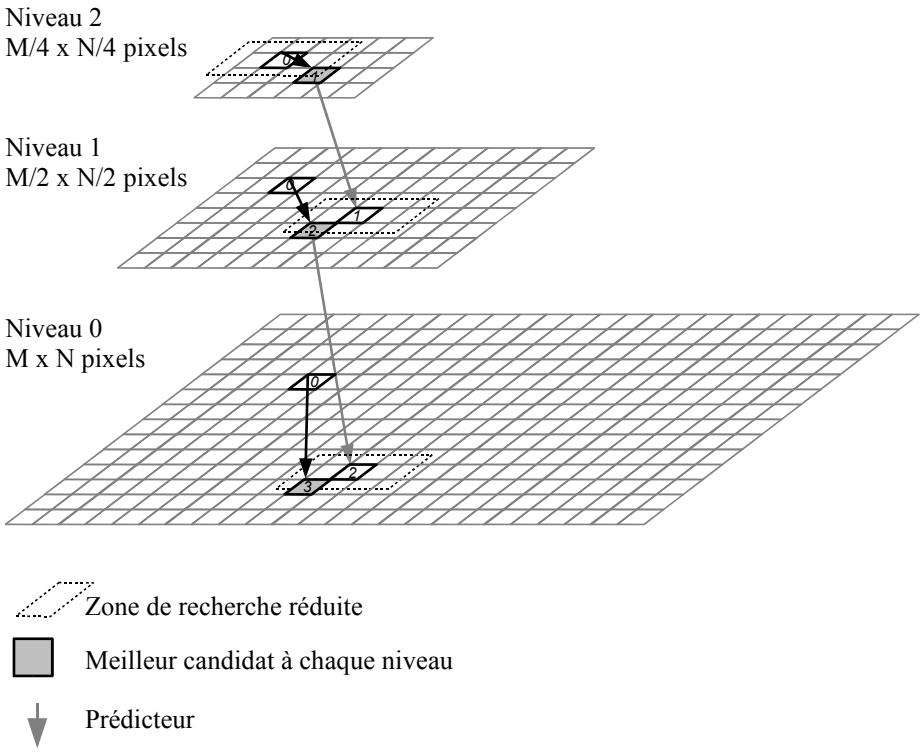


FIG. 2.13 – Estimation de mouvement hiérarchique

et des traitements dans les niveau hiérarchiques. Cependant, il offre une meilleure robustesse, et détecte les mouvement de grande amplitude. Le résultat est très proche du mouvement réel et demeure en règle générale homogène, ce qui lui permet d'être plus efficace qu'une recherche exhaustive dans un contexte de compression vidéo. De plus c'est un algorithme régulier, dans le sens où le nombre de calcul est relativement constant, indépendant de la séquence.

2.3.6 Synthèse des méthodes d'estimation de mouvement basées bloc

Les méthodes d'estimation de mouvement basées bloc correspondent le mieux à la compression vidéo, elle-même généralement basée bloc (MPEG-2, MPEG-4). Les méthodes de mise en correspondance de blocs sont les plus simples à mettre en oeuvre et les moins coûteuses en temps de calcul. Cependant une recherche exhaustive implique une puissance de calcul considérable pour garder l'amplitude nécessaire dans les séquences haute définition. De nombreuses techniques rapides ont été mises au point dans le but de réduire le nombre de calculs tout en donnant une solution la plus proche de l'optimal.

Les techniques prédictives permettent de réduire considérablement la fenêtre de recherche en utilisant le voisinage spatial et temporel pour prédire le centre de la zone de recherche. Cependant dans le cas de mouvements complexes ces méthodes ne convergent pas et se contentent d'un optimum local. Les méthodes multirésolution, un

peu plus complexes, permettent d'introduire des prédicteurs hiérarchiques robustes et éliminent les problèmes de convergence.

Dans la suite de cette étude, l'algorithme HME présenté ci-dessus sera privilégié pour ses performances, sa vitesse et sa régularité. L'algorithme EPZS sera également étudié pour ses calculs réduits.

Les techniques permettant de mettre en correspondance des blocs d'une image courante avec une image de référence ont été présentées. Pour la compression vidéo, et notamment le standard H.264, ces techniques sont utilisées avec des fonctionnalités supplémentaires.

2.4 Impact des fonctionnalités H.264 sur l'estimation de mouvement

Le standard H.264 permet d'obtenir des meilleurs performances de compression par rapport à ses prédécesseurs, notamment grâce de nombreuses spécificités de la compensation de mouvement. Pour utiliser ces modes de prédiction, l'estimation de mouvement doit les prendre en compte. Leurs impacts sur les algorithmes, ainsi que sur la complexité de calul, sont présentés dans ce paragraphe

2.4.1 Taille de bloc variable

La norme H.264 réduit le débit des vidéo en ajoutant des modes de codage par rapport aux anciens standards. La compensation de mouvement peut être faite sur des blocs de taille variable. Les macroblocs 16×16 peuvent être divisés en 16×8, 8×16 ou 8×8 et les sous-partitions 8×8 peuvent être à nouveau divisées en 8×4, 4×8 ou 4×4. Le choix de petits blocs améliore la précision de la compensation de mouvement mais nécessite la transmission d'un plus grand nombre de vecteurs. Un algorithme de décision efficace doit donc être mis en place. Il peut faire partie intégrante de l'estimateur ou bien être séparément exécuté. Dans ce dernier cas, l'estimateur de mouvement fournit les vecteurs pour toutes les tailles de blocs. C'est dans ce contexte que nous allons nous placer pour étudier les différentes techniques d'estimation de mouvement à taille de bloc variable.

Il existe différentes techniques pour calculer un vecteur de mouvement par taille de bloc.

2.4.1.1 Méthode exhaustive

La technique la plus simple consiste à recopier le schéma d'estimation pour chaque taille de bloc. Un champ de vecteurs est alors calculé indépendemment pour chaque mode. Aucune optimisation n'est donc effectuée. De plus la corrélation pouvant exister entre les vecteurs des sous-partitions de taille différentes n'est pas exploitée. Le seul avantage est que cela permet de paralléliser aisément les traitements, vu l'inexistence d'interdépendances de données. Cependant cette solution n'est généralement pas retenue car elle ne permet pas d'exploiter la redondance des calculs.

2.4.1.2 Méthode par post-traitement

Une technique peu coûteuse en temps de calcul consiste à réaliser un post-traitement du champ de vecteurs soit pour le sous-échantillonner, soit pour le sur-échantillonner, suivant la taille du bloc voulue. La qualité des champs de vecteurs est réduite car il n'est pas effectué de mise en correspondance pour chaque taille de bloc.

2.4.1.3 Méthode prédictive

L'approche par post-traitement peut également être une première étape pour une recherche rapide. C'est à dire qu'un champ de vecteurs pour une taille de sous-partition donnée est calculé, puis un post traitement fournit les vecteurs pour les autres tailles, qui sont ensuite raffinés. Dans [CJJ03] par exemple, la recherche est effectuée avec un algorithme prédictif pour la plus petite taille de bloc, les mouvements des blocs de plus grande taille sont rapidement déduits.

Dans [ZSH04], l'estimation commence par la taille 8x8 puis les autres tailles en sont dérivées. La taille de départ est choisie à cause de deux observations. Premièrement, plus la taille du bloc est grande, plus la SAD prend d'informations en compte, donc plus le vecteur est pertinent. Il convient donc de ne pas partir d'une prédiction basée sur une taille trop petite. Deuxièmement plus la différence de taille est réduite entre l'estimation à effectuer et la prédiction, plus celle-ci est pertinente. La valeur médiane semble donc un bon compromis.

2.4.1.4 Réutilisation de SAD

Cette technique consiste à calculer les SAD sur la base des blocs les plus petits, c'est à dire 4x4, et en calculant simultanément tous les sous-blocs inclus dans un bloc de la plus grande taille, c'est à dire 16x16. Pratiquement, seize blocs 4x4 sont estimés en même temps, ce qui permet d'obtenir les SAD des blocs plus grands en combinant les résultats (Fig. 2.14).

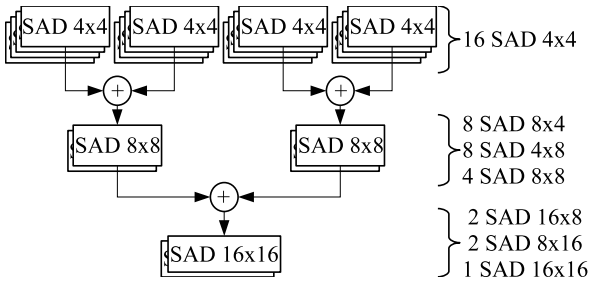


FIG. 2.14 – Principe de réutilisation de SAD

Cette technique est associée avec une recherche exhaustive. Elle a beaucoup moins de sens avec une méthode prédictive, qui supposerait que les vecteurs associés aux blocs de tailles différentes ont le même mouvement, ce qui est contradictoire avec l'intérêt de l'estimation de mouvement à taille de bloc variable. La recherche exhaustive peut néanmoins être réduite si on ne considère que des blocs de tailles similaires, comme par exemple 4x4 à 8x8, ou 16x8, 8x16 et 16x16.

La compensation de mouvement à taille de bloc variable permet de résoudre le compromis coût du résidus versus coût du champ de vecteur d'un encodeur vidéo. Cependant la complexité de calcul augmente. La technique de réutilisation de SAD élimine un certain nombre de calcul, mais nécessite une recherche exhaustive, alors qu'un algorithme par déduction permet de limiter le nombre d'opérations. La technique du post traitement constitue une solution économique, et peut servir de première phase de prédiction à une autre technique rapide.

La complexité de l'estimation de mouvement à taille de bloc variable est d'autant plus importante que la précision des vecteurs est grande. Le raffinement subpixélique augmente en effet considérablement la puissance de calcul nécessaire.

2.4.2 Raffinement subpixélique

Dans H.264, la précision des vecteurs de mouvement atteint le quart de pixel. Un gain de compression significatif est apporté en réduisant considérablement l'énergie des résidus, au prix d'une complexité augmentée pour deux raisons :

- le besoin de calculer la valeur des pixels pour les positions fractionnaires grâce à de l'interpolation,
- l'augmentation de l'espace de recherche, ce qui augmente le nombre de candidats.

On retrouve donc plusieurs solutions possibles, qui ont pour but de réduire la complexité, en s'efforcant de garder la meilleure qualité possible.

2.4.2.1 Stratégie d'interpolation

L'interpolation des valeurs des échantillons pour les positions fractionnaires a des conséquences sur la charge de calcul et la quantité de données à traiter. En effet d'une part le filtre d'interpolation demi-pixel à six coefficients de H.264 induit beaucoup de calculs. D'autre part, le nombre d'échantillons à manipuler est multiplié par quatre pour le demi-pixel et par quatre pour le quart de pixel, soit au total seize fois plus d'échanges de données. Il existe deux approches d'interpolation : effectuer l'opération sur l'image entière ou interpoler les données à la volée. Suivant la technique utilisée l'impact sur les performances n'est pas le même.

Interpolation de l'image entière : Afin d'éviter de nombreux calculs redondants, c'est à dire les échantillons des fenêtres de recherche qui se recouvrent ou bien causés par les estimations successives (taille de bloc variable et multiples images de référence), l'interpolation peut être réalisée au préalable sur l'image entière et stockée en mémoire. Cependant, étant donné la taille considérable des données quart de pixel, les échanges de données entre la mémoire et le processeur créent un goulot d'étranglement. Ceci a une double conséquence, puisque le système se trouve ralenti une première fois lors de l'interpolation (écriture dans la mémoire des données), et ensuite à chaque accès (lecture).

Interpolation à la volée : Pour s'affranchir des goulots d'étranglements dus aux accès à la mémoire, les données peuvent être calculées à chaque fois qu'elles sont nécessaires. Des calculs redondants sont donc à prévoir, mais le jeu de données source

(l'image à la résolution pixel) est réduit et rapidement disponible. Les données interpolées sont stockées temporairement dans des petites mémoires très rapides proches des unités de calcul.

Dans la plupart des cas, cette dernière technique est plus rapide, par conséquent elle est en général préférée.

2.4.2.2 Stratégie de recherche

La technique de base est de considérer une estimation de mouvement pixel entier, puis de raffiner le vecteur à la précision voulue, en plusieurs étapes récursives : d'abord au demi-pixel, puis au quart de pixel, etc. Le raffinement peut aussi être réalisé avec une méthode exhaustive d'amplitude inférieure à un pixel autour de la meilleure position pixel. Bien que susceptible de donner de meilleurs résultats, les méthodes exhaustives engendrent plus de calcul que les méthodes itératives.

De même que pour l'estimation de mouvement au pixel entier, compte tenu de la quantité de calcul engendré, la recherche exhaustive n'est pas utilisée pour la recherche subpixélique, sauf avec une fenêtre de recherche très réduite. Dans [RB05] une solution d'estimation de mouvement subpixélique à taille de bloc variable avec réutilisation de SAD est proposée. Cette solution n'est applicable que si les mouvements sont réduits (jusqu'à quelques pixels), et est donc inadaptée à la compression de vidéo haute définition.

Les méthodes prédictives du type recherche en diamant peuvent prendre en compte les positions fractionnaires directement, comme proposé dans [CJJ03]. Le nombre de distorsions à calculer est réduit. Ceci est vrai lorsque la prédiction est bonne, c'est à dire pour estimer des mouvements homogènes. Dans le cas général plusieurs tailles de motifs en diamant sont successivement utilisés pour raffiner la prédiction, et on se retrouve dans le cas de la technique de base.

2.4.2.3 Sans interpolation

Il est également possible de réaliser le raffinement subpixel sans interpolation, en supposant que le critère d'erreur décrit une surface parabolique près de l'optimum [CCHBC04, HCBC06]. L'estimation de mouvement est réalisée à la précision pixel. Le critère d'erreur (par exemple la SAD) a été calculée au moins pour l'optimum et ses huit voisins. Les paramètres de la surface définie par les valeurs de SAD en fonction de la position sont estimés, puis un optimum en est déduit (Figure 2.15), soit analytiquement, soit en évaluant la distorsion pour chaque position grâce au modèle parabolique. Cette méthode ne donne pas de très bons résultats et une interpolation est nécessaire pour les améliorer [HCBC06].

2.4.3 Multiples images de référence

Le but principal des multiples images de référence est d'augmenter la probabilité de trouver une bonne correspondance, notamment lorsque l'illumination de la scène change, ou lors d'occlusions d'objets en autorisant une recherche dans plusieurs images. Dans le cas d'images de type B, c'est à dire bi-prédiction, l'amélioration de qualité est obtenue en combinant deux références (en moyennant les deux références). L'espace de

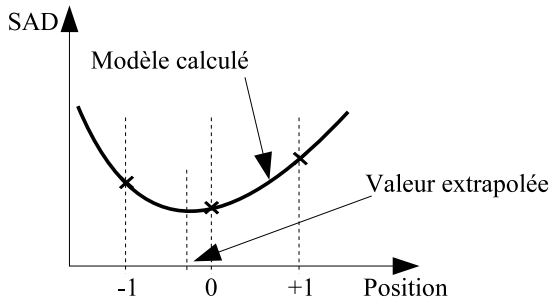


FIG. 2.15 – Raffinement sans interpolation

recherche est donc élargi, ce qui a pour effet d'augmenter la complexité de l'estimateur de mouvement.

Pour les images P, la solution de base est de recopier le schéma d'estimation pour chacune des images de référence. Pour limiter le nombre de calculs, certaines solutions consistent à étendre la recherche en diamant à la dimension temporelle [Tou02]. Une autre solution se concentre sur la réduction de la bande passante mémoire en utilisant la même fenêtre de référence pour plusieurs prédictions [CTHC07]. L'idée est qu'au lieu d'avoir un bloc courant et plusieurs références, on se ramène à un problème avec plusieurs blocs courants et une référence. Etant donné que la référence peut représenter un espace mémoire conséquent par rapport au bloc courant, de nombreux accès mémoire sont ainsi évités. Les contraintes de bande passante sont réduites à celles du problème à une seule image de référence. Cette dernière solution présente beaucoup d'avantages dans un contexte haute définition.

Pour les images B, l'espace de recherche est davantage élargi. Comme une recherche conjointe exhaustive n'est pas pratiquement réalisable, une solution itérative est adoptée où les hypothèses sont successivement raffinées [FWG98] suivant l'algorithme 2.2. Une autre alternative est de considérer directement les résultats d'une estimation de mouvement indépendante pour deux images de références.

Algorithme 2.2 Optimal Hypothesis Selection Algorithm (OHSA) [FWG98]

1. Sous n hypothèses ($n = 2$ pour une prédiction B) la mesure de débit/distorsion $\varepsilon(v_1, v_2) = SAD\left(c - h_1 \dot{r}_1\left(v_1\right) - h_2 \dot{r}_2\left(v_2\right)\right) + \lambda \times \left(R\left(v_1\right) + R\left(v_2\right)\right)$ est à minimiser pour chaque bloc courant c. λ est le multiplicateur de lagrange pour prendre en compte le coût de codage R des deux vecteurs v_1 et v_2. r_1 et r_2 sont les blocs de référence avec leur coefficients de prédiction h_1 et h_2. L'algorithme est initialisé avec deux hypothèses v_1^0 et v_2^0, et $i = 0$.
2. Pour $\mu = 1..2$
   L'hypothèse v_μ^iest concernée, l'autre est fixée. La mesure ϵ est minimisée pour v_μ^{i+1} dans une fenêtre de recherche sélectionnée.
3. L'algorithme continue à l'étape 2 avec $i = i + 1$ tant que ϵ décroît.

2.5 Conclusion sur les techniques d'estimation de mouvement

Nous avons présenté plusieurs techniques d'estimation de mouvement. La mise en correspondance de blocs apparaît comme étant la mieux adaptée à la compression vidéo. Les méthodes d'estimation basées gradient sont très précises, mais ne permettent d'estimer que de faibles déplacements, avec une quantité de calcul très importante. Les techniques fréquentielles sont robustes, mais nécessitent une étape supplémentaire de mise en correspondance.

Dans la suite de l'étude, nous allons nous intéresser seulement à quelques méthodes de mise en correspondance. Ces méthodes sont en effet plus adaptées au codage d'images où le mouvement réel n'est pas primordial, le but étant de minimiser le coût de codage par bloc. Parmi les techniques présentées, l'algorithme prédictif EPZS semble combiner bonnes performances et calculs réduits. HME (Hierarchical Motion Estimator) est robuste et régulier. Ces deux algorithmes constituent ainsi un échantillon représentatif pour l'étude de l'implantation d'estimateurs de mouvement. Leur paramétrage permet en outre d'évoluer d'une estimation grossière très rapide à la recherche exhaustive. Elles sont donc sélectionnées pour la suite de l'étude.

La complexité de l'estimation de mouvement pour la compression vidéo augmente à chaque génération de standard de compression vidéo, notamment avec les tailles de bloc variables et la précision subpixélique des vecteurs de mouvement dans H.264. Concernant ces spécificités, nous ne nous arrêtons pas sur une technique précise. Plusieurs solutions seront étudiées, implantées puis analysées.

Le coût de calcul d'un algorithme ne dépend pas seulement du nombre d'opérations, mais aussi du processeur et de la façon dont il est programmé. Pour respecter les contraintes temps réel, il est souvent nécessaire d'augmenter la puissance de calcul disponible. Le chapitre suivant traite des architectures susceptibles d'exécuter une application d'estimation de mouvement.

Chapitre 3

Architectures matérielles

L'opération d'estimation de mouvement pour la compression vidéo H.264 haute définition impose des contraintes matérielles très sévères, notamment en termes de puissance de calcul et de bande passante mémoire. Ainsi, afin de bien appréhender les problèmes que vont poser l'implantation des différents algorithmes présentés au chapitre 2, il est nécessaire d'introduire des notions d'architecture des processeurs. De plus, l'optimisation d'une l'implantation donnée nécessite une très bonne connaissance de l'architecture.

L'évolution des technologies de fabrication permet d'augmenter non seulement la capacité d'intégration, mais aussi la fréquence de fonctionnement des composants. Dans ce domaine, on évoque souvent la "*loi de Moore*", exprimée en 1965 par Gordon Moore, cofondateur d'Intel. Cette loi empirique, selon laquelle la capacité d'intégration (le nombre de transistors dans un processeur) double tous les deux ans, se vérifie depuis plus de quarante ans. En pratique, l'architecture interne des processeurs se révèle de plus en plus complexe, de sorte à améliorer et accélérer l'exécution d'un programme, ou d'un ensemble de programmes.

Ces dernières années, nous assistons à un tournant dans l'architecture des processeurs. La fréquence de fonctionnement n'évolue que très peu, voire même diminue, et les architectures multicœurs sont de plus en plus présentes. Les fabricants ne communiquent plus sur la seule fréquence du processeur pour vanter ses performances, mais sur des notions de puissance de calcul, prenant en compte plus de paramètres.

Nous présentons dans ce chapitre différents éléments de traitement susceptibles de réaliser l'opération d'estimation de mouvement. Dans une première partie, les différentes architectures matérielles sont détaillées, des processeurs génériques aux composants dédiés, en passant par les plates-formes multicomposants. Dans une deuxième partie, nous dressons un état de l'art des unités de traitement développées spécifiquement pour l'estimation de mouvement.

3.1 Etude des architectures existantes

Nous étudions dans un premier temps les solutions d'architectures de manière générale, c'est à dire indépendemment des applications : les processeurs standards, les composants matériels, les architectures émergeantes, et les plates-formes multicomposants.

Ensuite nous discutons sur l'intérêt de ces solutions pour une application d'estimation de mouvement temps réel.

3.1.1 Processeurs

Dans ce paragraphe, différents types de processeurs sont présentés. Ces unités de calcul se caractérisent par leur simplicité d'utilisation. La programmation s'effectue le plus souvent en langage de haut niveau, ce qui simplifie le processus de développement. Leur utilisation est très répandue et leurs outils de développement sont le plus souvent évolués.

3.1.1.1 Processeurs standards

Les GPP (General Purpose Processor) sont des processeurs généralistes qui équipent la plupart des ordinateurs. Leur popularité est liée à leur flexibilité, leur simplicité d'utilisation et leur puissance de calcul. Les innovations dans ce domaine sont conduites par le marché des ordinateurs grand public et professionnels. Cela représente des grands enjeux commerciaux, et pousse ces processeurs à la pointe de la technologie (procédés de fabrication, fréquence d'horloge, architecture interne).

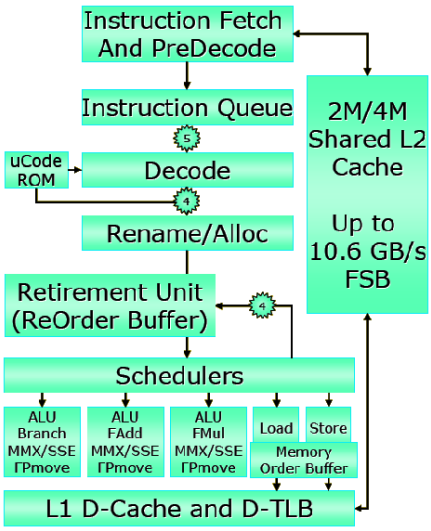


FIG. 3.1 – Architecture du "Intel Core"

La figure 3.1 donne un aperçu de la complexité des GPP actuels les plus puissants. Ils intègrent différentes unités de traitement pour viser de hautes performances.

– **Pipeline d'exécution**
  Le but du pipeline est de découper les opérations en plusieurs étapes afin d'augmenter la fréquence des processeurs, chaque étape étant plus courte. Une instruction pipelinée est donc exécutée en plusieurs cycles. Les architectures du type Pentium 4 ont une fréquence ainsi qu'une longueur de pipeline très élevées. L'inconvénient majeur est l'impossibilité de remplir le pipeline lorsque des instructions consécutives ont une dépendance de données. Comme il est nécessaire

d'attendre le résultat de l'instruction précédente avant de lancer la suivante, des cycles d'horloge sont perdus.
Après avoir poussé à ses limites la taille du pipeline et à la fréquence des processeurs, les concepteurs sont revenus vers des architectures avec des fréquences plus faibles, mais offrant de meilleures performances. C'est le cas d'Intel qui passe de 31 étages de pipeline avec le Pentium 4 à 14 avec le Core 2.

- **Unités de calcul**
  Les données traitées par les GPP peuvent être indifféremment des nombres à virgule fixe ou flottante. Plusieurs unités de calcul dédiées sont embarquées (calcul arithmétique et logique, calcul flottant, accès mémoire, branchement). Cela permet d'exécuter plusieurs instructions par cycle grâce au fonctionnent en parallèle de ces unités (processeurs dits "superscalaires").
- **Unité de réordonnancement des instructions**
  Les dépendances de données entre les instructions séquentielles d'un programme sont analysées, puis les instructions sont parallélisées sur les différentes unités de calcul, et réordonnées si besoin. Cette unité est indispensable pour améliorer le fonctionnement d'un long pipeline et permettre à une architecture superscalaire d'être efficace.
- Unité de réordonnancement des accès mémoire
  De même que pour les instructions, les accès aux données en mémoire sont réordonnés.
- **Prédiction de branchement**
  Un branchement conditionnel dans un programme est très préjudiciable. En effet la condition doit être calculée avant d'effectuer le branchement, induisant une perte de cycles. Pour les limiter, le branchement est prédit. Une analyse statistique en temps réel permet de prédire un branchement et de garder les unités de calcul actives. Lorsque la prédiction est mauvaise, les résultats sont rejetés et le branchement correct est réalisé.
  Ce type d'approche est très efficace pour les boucles dont le nombre d'itérations est inconnu à l'avance. L'exécution spéculative permet de garder le pipeline rempli et les unités de calcul actives.
- **SIMD**
  Les unités de traitement SIMD (Single Instruction Multiple Data) ont été conçues pour les applications multimédia (audio, image, vidéo). Ainsi les extensions Intel MMX (64 bits), SSE (64 bits), SSE2, SSE3 (128 bits) et AMD 3D Now (64 bits) permettent de faire du calcul vectoriel, c'est à dire d'appliquer une même instruction à plusieurs données. Par exemple, il est possible d'ajouter quatre paires d'entiers de 32 bits (ou seize paires d'entiers de 8 bits) en une instruction SSE2. Pour en tirer profit, il est nécessaire de vectoriser l'application à la compilation. Cela est possible grâce au compilateur. Le langage utilisé peut être de moyen niveau comme le C, ou bas niveau comme l'assembleur. Dans ce dernier cas, le temps de développement est alors en général plus important.
- **Large cache**
  Un autre problème présent sur un ordinateur est l'accès mémoire. La mémoire cache des processeurs augmente grâce aux capacités d'intégration toujours plus élevées. Dans le même temps la taille des programmes et des données a ten-

dance à augmenter. Les accès aléatoires de certains programmes et le multitâches rendent alors les mémoires caches trop petites inefficaces.

– **Large bande passante mémoire**
Pour faire face à l'augmentation des mémoires, les bandes passantes augmentent également. Les GPP sont compatibles avec des mémoires rapides, offrant théoriquement jusqu'à 8.5 Go/s de bande passante.

Toutes ces caractéristiques permettent aux GPP actuels d'améliorer leurs performances par rapport aux précédentes générations, à une fréquence de fonctionnement comparable. En effet, ces processeurs sont capables d'optimiser l'exécution d'un programme à la volée pour améliorer les performances des applications qui ne sont pas à la base optimisées pour un type de processeur particulier. Une grande partie de la surface de silicium est utilisée pour cela (prédiction de branchement, unité de réordonnancement, cache). Les points forts de ce type de composants sont sa flexibilité et ses performances, au détriment de la consommation, de l'encombrement et du prix.

3.1.1.2 Processeur de traitement du signal

Les DSP (Digital Signal Processor) sont des processeurs simplifiés dédiés aux applications de traitement du signal. Leur architecture interne est dimensionnée pour le calcul intensif. Suivant les modèles, ils permettent de réaliser des opérations sur des nombres soit à virgule fixe, soit à virgule flottante. En effet les unités arithmétiques et logiques ne sont pas construites de la même manière. Le calcul flottant requiert une architecture plus complexe qui conduit à une fréquence de fonctionnement généralement plus faible. Par conséquent, les flottants sont utilisés seulement lorsqu'une grande précision et une grande dynamique sont requises sur les valeurs traitées. Cependant, une unité de calcul à virgule fixe peut émuler des calculs à virgule flottante, mais avec de très faibles performances.

La logique interne est réduite et dédiée au calcul arithmétique et logique, et la logique de contrôle est réduite (Par exemple on ne retrouve pas d'unité de réordonnancement comme sur le GPP). Le pipeline est très court : les opérations telles que l'addition et la multiplication sont exécutées en un cycle. La figure 3.2 montre l'architecture interne du DSP Texas Instruments TMS320C6416. C'est un des DSP les plus puissants du marché.

Le C64 a deux coeurs de calcul (A et B) qui ont chacun quatre unités (L, S, M et D). Huit instructions peuvent donc être exécutées en un cycle.

Ce DSP est dépourvu d'un ordonnanceur à la volée qui équipe certains GPP. La distribution et l'ordonnancement des instructions sur les différentes unités de calculs sont effectuées au moment de la compilation. L'architecture VLIW (Very Long Instruction Word) lit des instructions de 256 bits pour fournir à chaque cycle jusqu'à huit instructions 32 bits aux huit unités fonctionnelles.

En plus des opérations standard (addition, multiplication), des instructions spécialisées (comptage de bits, rotations, ...) sont implantées. Les unités de calcul 32 bits du C64x permettent d'effectuer une opération 32 bits, deux opérations 16 bits ou quatre opérations 8 bits, grâce aux instructions SIMD. Les unités fonctionnelles peuvent donc être exploitées de manière intensive en utilisant toute la largeur disponible.

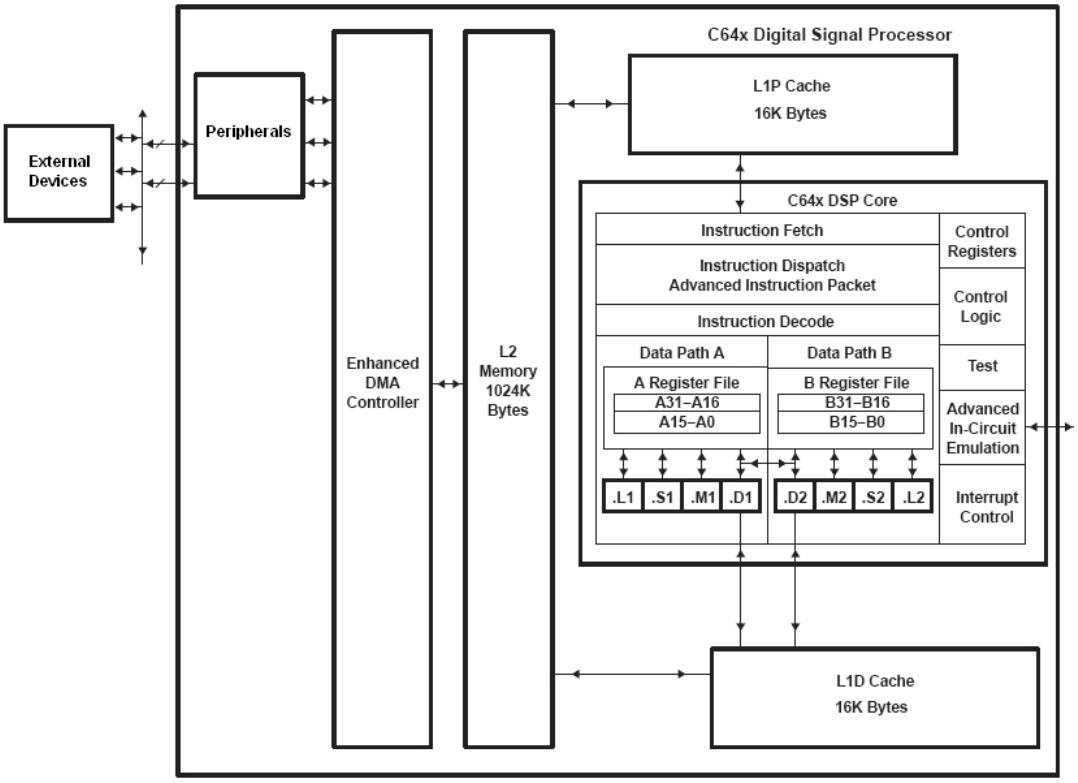


FIG. 3.2 – Architecture du C6416

Les DSP embarquent des périphériques et coprocesseurs dédiés. Des coprocesseurs de décodage Viterbi et Turbo accélèrent les communications numériques. Les transferts entre les périphériques et la mémoire et entre la mémoire interne et la mémoire externe sont réalisés par le DMA (Direct Memory Access), qui permet de décharger le coeur de calcul, et de paralléliser accès mémoire et traitements. Des interfaces série, PCI, mémoire sont également intégrées au DSP.

L'architecture d'un DSP est dédiée au calcul intensif. Une application avec beaucoup de contrôle sera difficilement optimisée, alors qu'un traitement intensif pourra tirer parti du VLIW et du SIMD. De plus les performances sont déterministes, par rapport à un GPP qui réalise des exécutions en désordre et des prédictions de branchement.

Les DSP sont accompagnés d'outils de développement performants. Les débogueurs fournissent des informations de bas niveau pour valider rapidement une application. Les fabricants fournissent en général des compilateurs capables d'optimiser fortement un programme et d'informer le développeur sur le résultat en ajoutant des commentaires dans le programme (boucle optimisée, nombre de cycles, instructions utilisées). Ce retour permet de revenir sur l'écriture du code pour "aider" le compilateur. Des simulateurs permettent d'obtenir des informations supplémentaires sur l'exécution d'une application (charge de calcul, comportement des mémoires caches, temps d'exécution).

3.1.1.3 Comparaison DSP - GPP

Pour faire le choix entre un GPP et un DSP, plusieurs paramètres doivent être pris en compte. En effet un GPP peut effectuer des opérations de traitement du signal, et inversement un DSP est capable de traiter des tâches générales. La figure 3.3 donne un aperçu de l'offre un matière de processeurs.

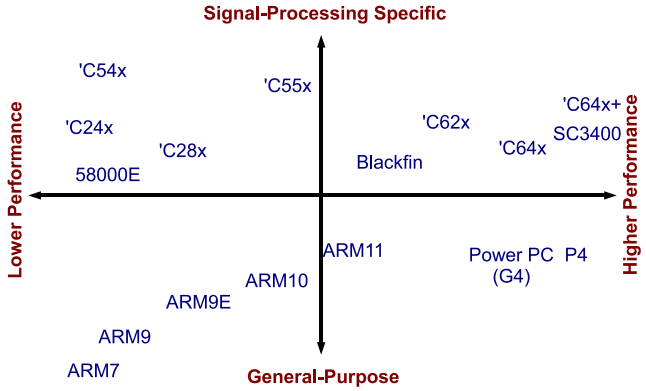


FIG. 3.3 – Exemples de processeurs

En terme de performances, les GPP les plus puissants dépassent les DSP, cependant les DSP conservent une architecture interne dédiée au traitement intensif du signal. Les conséquences prédominantes sont l'encombrement et la consommation (Fig. 3.1). En effet, un GPP consomme jusqu'à cent fois plus qu'un DSP de même performance. Il est plus encombrant et doit être accompagné d'un ventilateur, qu'il faut alimenter et qui est une source de bruit.

Processeur	Pentium core-2 duo E6700	DSP TI TMS320C6455
Fréquence	2.66 GHz	1 GHz
Procédé	65 nm	90 nm
Taille du silicone	143 mm^2	N/A
Consommation	200 Watts	2 Watts
Prix	˜300 euros	˜200 Euros

TAB. 3.1 – Comparaison DSP - GPP

Faire le choix entre un GPP et un DSP nécessite une vue d'ensemble des caractéristiques requises. Un GPP est simple à utiliser, on le trouve "prêt à l'emploi" dans les ordinateurs, avec de nombreux périphériques (écran, clavier, disque dur, réseau) et des systèmes d'exploitations puissants. Les DSP peuvent être fournis sur une carte de développement avec des périphériques, mais une carte dédiée à l'application est quelques fois nécessaire. Enfin, sur DSP, de nombreuses bibliothèques logicielle optimisées tierce partie existent, et les outils de développement sont plus spécialisés.

3.1.2 Composants matériels

Les composants matériels sont utilisés pour leurs performances très élevées. Ils sont développés pour une application spécifique. Ce sont donc des composants spécialisés, répondant à une problématique précise. Ils permettent de résoudre des fortes contraintes temps réel. On peut distinguer deux types : les FPGA et les ASIC.

3.1.2.1 FPGA

Les FPGA (Field Programmable Gate Array) sont des composants électroniques programmables. Grâce à l'évolution des procédés de fabrication, ces composant peuvent actuellement supporter des applications complexes. Ils sont constitués d'un réseau de blocs logiques, de blocs mémoire, de blocs dédiés et d'entrées/sorties.

Les blocs logiques permettant de réaliser des opérations avec quelques variables à travers une LUT (Look Up Table). Le résultat peut être éventuellement stocké dans un registre. Les blocs RAM permettent d'implanter des mémoires adressables et de type FIFO (First In, First Out). Les blocs dédiés permettent de réaliser facilement de nombreuses opérations de traitement du signal (blocs DSP : Digital Signal Processing), de gérer l'horloge, ou des interfaces de communication (RocketIO, Ethernet, PCI Express). Les nombreux ports permettent de connecter des périphériques.

Les FPGA se programment grâce à leurs LUT et leur réseau d'interconnexion. La programmation se fait avec un langage de programmation tel que VHDL (acronyme de VHSIC-HDL : Very High Speed Integrated Circuit Hardware Description Language) ou Verilog. L'outil de développement transforme cette description en un fichier de configuration du FPGA en plusieurs étapes : le HDL doit d'abord être synthétisé (transformé en éléments logiques de base), puis les éléments doivent être placés sur le composant (placement) et interconnectés (routage).

Le cycle de développement est particulièrement long à cause du langage de programmation de bas niveau. Quelques outils existent néanmoins pour convertir automatiquement un programme “C” en VHDL (Impulse C, Handel-C), avec toutefois des performances relatives.

Les FPGA actuels permettent d'intégrer des applications relativement complexes, de la mémoire, des interfaces de communication. La puissance de calcul totale est largement supérieure à celle offerte par un processeur. Ils sont donc particulièrement intéressants dans le traitement d'opérations régulières très intensives.

3.1.2.2 ASIC

Un ASIC (Application Specific Integrated Circuits) est un circuit électronique dédié. Il peut regrouper sur une même puce tous les éléments actifs nécessaires à la réalisation d'une fonction ou d'un ensemble électronique. Ce composant n'est pas modifiable. Comparé à un FPGA, il offre une capacité d'intégration plus élevée, une vitesse plus rapide et un coût plus faible. Cependant, le développement d'un ASIC est très long. La programmation s'effectue avec un langage de bas niveau, et les tests de validation sont intensifs. En effet, comme il n'est pas possible de modifier le composant après sa production, une erreur peut provoquer la nécessité de repasser par la phase de développement et coûte par conséquent très cher. Les ASIC n'ont d'intérêt que pour les grandes séries pour compenser les coûts de développement et de production.

3.1.2.3 IP (Intellectual Property)

L'augmentation de la capacité des composants matériels est telle qu'aujourd'hui il est difficile de concevoir des circuits spécialisés exploitant la totalité des ressources. Les outils de développement sont de plus en plus performants. Cependant, il est indispensable de pouvoir réutiliser des conceptions déjà réalisées. Pour cela, des blocs de base, couramment appelés IP (intellectual property) ou composants virtuels, sont rendus disponibles sous forme de bibliothèques. Leur assemblage permet de décrire rapidement des applications complexes. On peut utiliser des IP sans les avoir conçus, en ayant uniquement une description haut niveau de leur comportement (vue externe fonctionnelle).

Les IP peuvent être paramétrables et décrits en langage synthétisable tel que du VHDL ou Verilog (on parle alors de SoftIP), ou bien dépendant d'une technologie (HardIP). Un SoftIP correspond au code source, alors qu'un HardIP permet de commercialiser un bloc fonctionnel en conservant la confidentialité.

Un IP peut être une fonction de base ou bien un bloc plus complexe. L'intégration de coeurs de processeurs est également possible, tel que le NIOS (Altera) ou le MICROBLAZE (Xilinx). Ces processeurs sont relativement simples, et personnalisables en fonction de l'application.

Les composants matériels présentent certains intérêts. Notamment il est possible de créer des SoC ("System on Chip" : un seul composant intégrant une grande partie, voire la totalité, des fonctions nécessaires à une application) dédiés très performants ou à basse consommation. Le cycle de développement généralement plus long qu'un processeur de type GPP ou DSP doit être justifié par de fortes contraintes temps réel, et les coûts de développement engendrés pour un ASIC doivent être compensés par une production de grande série.

3.1.3 Les nouvelles architectures

Il existe d'autres types d'architectures que des processeurs standard ou des composants totalement programmables. Par exemple, développés notamment grâce au marché du jeux vidéo, les processeurs qui équipent les consoles ou les cartes graphiques peuvent apporter des solutions alternatives.

3.1.3.1 Le Cell

Le processeur Cell est issu d'une collaboration avec Sony, Thoshiba et IBM [KDH+05]. Le but était d'atteindre cent fois les performances d'une *PlayStation2* pour un processeur utilisable très largement (dont le marché dépasse celui des consoles de jeux). Les architectures classiques ne pouvant pas y parvenir, le Cell est composé d'un Power PC (PPE : Power Processor Element) 64 bits, de huit unités de calcul (SPE : Synergistic Processor Element) et d'un bus d'interconnexion (EIB : Element Interconnect Bus) (Fig. 3.4).

Le PPE contrôle globalement le processeur. Il exécute les tâches d'un GPP standard, il gère système d'exploitation et commande les SPE. Les SPE sont des processeurs élémentaires qui réalisent les tâches de calcul. Ils sont composés d'un DMA

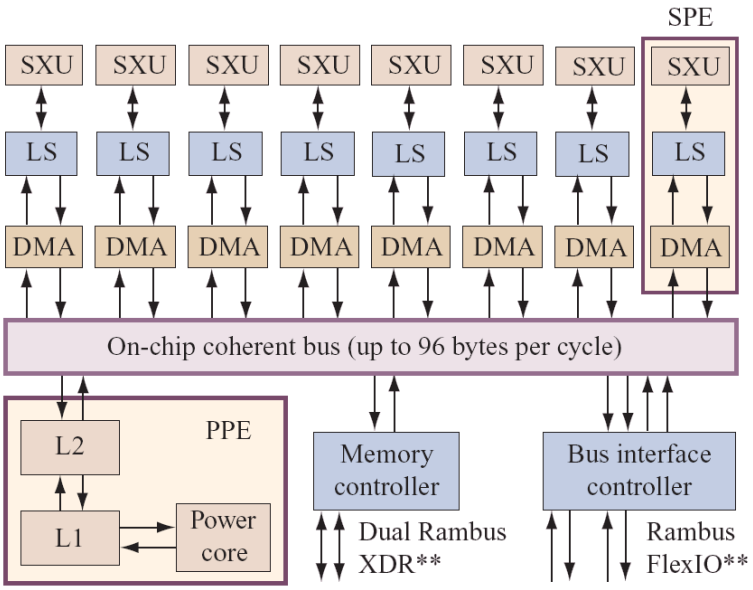


FIG. 3.4 – Architecture du Cell

gérant les accès mémoire à travers le bus, d'une mémoire locale (LS) et d'une unité de calcul SIMD 128 bits et ses registres (SXU) (Fig. 3.5).

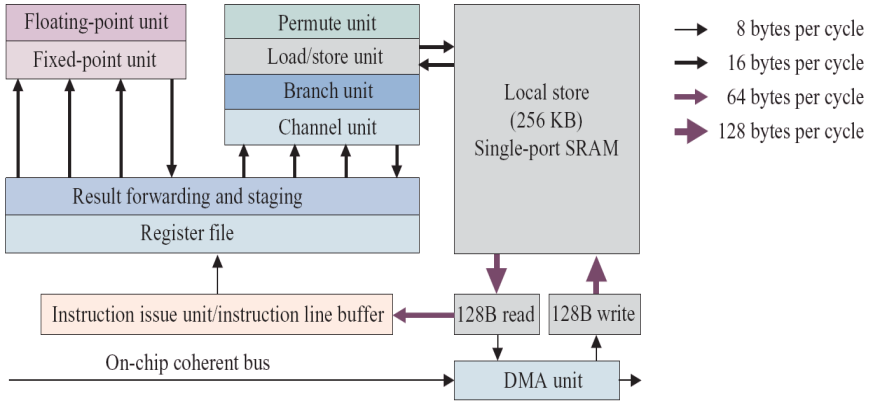


FIG. 3.5 – Architecture détaillée d'un SPE

Pour fournir les données et être cohérent avec sa puissance de calcul, le Cell intègre deux interfaces mémoire Rambus pouvant atteindre au total jusqu'à 25 GO/s, et deux interfaces d'interconnexion FlexIO. La consommation est de 120 W, correspondant à celle d'un Pentium.

Pour tirer parti des performances du Cell, il est nécessaire d'adapter la méthode de programmation. En effet, il est nécessaire de paralléliser les traitements de manière efficace sur les neuf (8+1) processeurs disponibles, tout en contrôlant efficacement la bande passante mémoire. Ainsi plusieurs modèles de programmation sont proposés dans [KDH+05] : GPP avec coprocesseurs, application parallélisée sur les SPE, pipeline des SPE, etc.

3.1.3.2 GPU

Les processeurs graphiques (GPU : Graphic Processing Unit) actuels offrent une très grande puissance de calcul, à un coût relativement réduit. En effet, l'évolution de ces processeurs est telle que leur puissance dépasse largement celle des CPU. De plus ils sont associés à des mémoires très rapides, offrant un grande bande passante. Depuis quelques années, il est possible de programmer les unités de calcul des GPU et de détourner leur finalité première pour faire du calcul numérique. Ce concept est connu sous le nom de GPGPU (General Purpose computation on GPU). Un état de l'art des développements de calcul numérique sur GPU est disponible dans [OLG+05], et un exemple d'application d'estimation de mouvement dans [KK04b, KK04a].

Architecture classique L'architecture des GPU est très spécialisée et la plupart des opérations sont figées. Les données sont traitées massivement en parallèle en appliquant la même opération, dans une unité dédiée, à toutes les données (modèle flot/noyau ou stream processing). C'est cette spécificité qui permet au GPU d'avoir une puissance de calcul très élevée. L'architecture d'un GPU se compose traditionnellement d'un pipeline de calcul (Fig. 3.6).

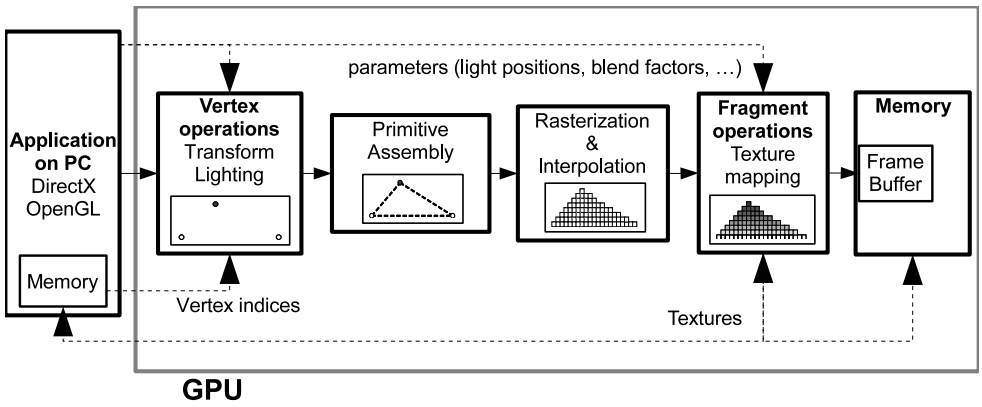


FIG. 3.6 – Architecture classique d'un GPU

Le processeur de sommets (Vertex processor) calcule des transformations sur des points (position 3D, éclairage, ...) qui sont ensuite assemblés sous forme de triangles, puis transformés en objets affichables (Rasterization). Le processeur de fragments applique la texture sur les objets et calcule le rendu final. La puissance de calcul vient de la spécificité des traitements effectués. Les opérations de contrôle sont très réduites et chaque type d'opération est réalisé dans une unité dédiée.

Dans ce modèle d'architecture, le processeur de sommets et le processeur de fragments sont programmables. Initialement introduits pour créer des effets de rendu personnalisés, les possibilités de programmation étendent l'utilisation des processeurs graphiques à d'autres types de calcul. Les programmes, appelés *shaders,* sont donc appliqués sur les sommets et les pixels.

Un stage de fin d'études réalisé par Serigné Dieng et encadré par moi-même a permis d'évaluer la complexité de mise en œuvre d'un algorithme d'estimation de mouvement sur GPU. La programmation est limitée par leur spécialisation : par exemple,

les opérations logiques bit à bit et de décalage n'existent pas et les applications de calcul numérique doivent être encapsulées dans du calcul graphique pour pouvoir être exécutées sur GPU. Les outils de programmation haut niveau pour GPU tels que Cg (C for Graphics) ou GLSL (OpenGL shading Language) nécessitent également une maîtrise de l'interface graphique (DirectX ou OpenGL). Les opérations réellement effectuées sont masquées par l'interface graphique, ce qui ne donne qu'un contrôle limité de l'implantation. De plus, la même application ne donne pas les mêmes résultats sur deux processeurs graphiques différents.

Evolutions La dernière génération de processeurs graphiques de chez NVIDIA (GeForce 8X [NVi06] et Quadro 5600) est basée sur une nouvelle architecture appelée *Unified shader architecture* 3.7. Le but est de réduire les étages de pipeline, d'augmenter la flexibilité et les performances. Il n'y a donc plus qu'un type de processeur : le *unified stream processor* privilégie les boucles plutôt que la séquentialité. Le flot des données passe donc plusieurs fois à travers les processeurs suivant les opérations à effectuer, ce qui a pour effet d'augmenter l'efficacité globale : suivant les calculs à effectuer, la charge de calcul n'est plus déséquilibrée sur les différentes unités.

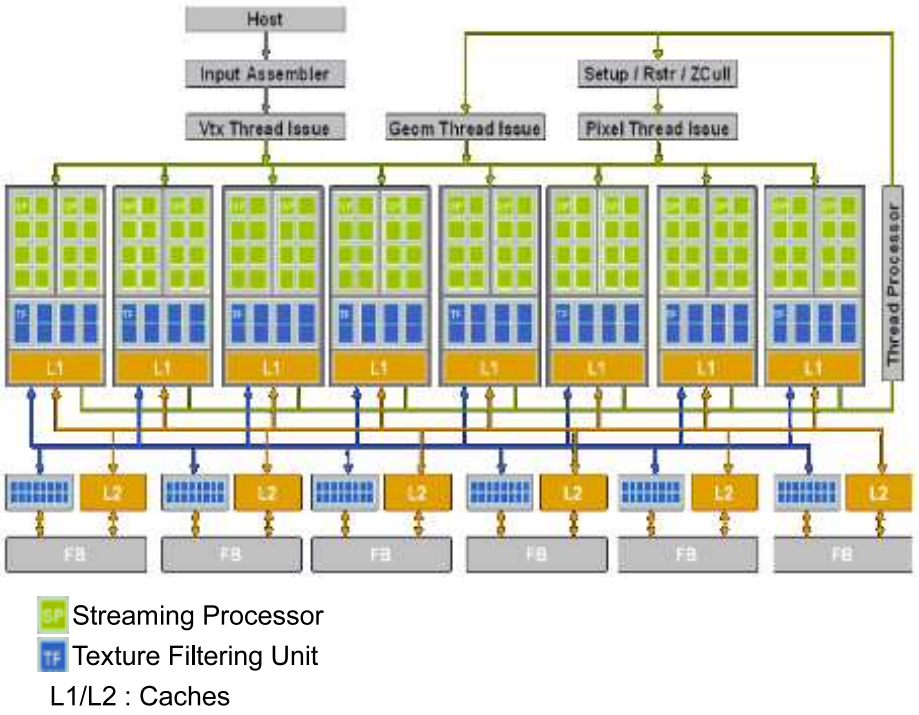


FIG. 3.7 – Diagramme blocs du GeForce 8800 GTX

Chaque processeur est capable d'effectuer des opérations scalaires et vectorielles. Un contrôleur (*Thread Processor*) distribue les opérations sur les différents processeurs pour les garder actifs. Le complexité de l'architecture a pour conséquence une consommation élevée (180 Watts). Nvidia fournit également un kit de développement basé sur le langage C (CUDA : Compute Unified Device Architecture) qui permet le débogage et la simulation sur CPU.

Les processeurs graphiques se présentent comme une alternative intéressante pour le calcul intensif. La puissance de calcul évolue beaucoup plus rapidement que celle des GPP, et offre déjà une puissance de calcul plus de six fois supérieure à un coût relativement réduit.

Cependant, l'approche GPU manque de de généricité et est complexe à mettre en œuvre. Les modèles de programmation sont récents et évoluent. Les outils de programmation haut niveau sont nombreux et requièrent une maîtrise du développement graphique. L'architecture de pipeline graphique fixé est très contraignante pour le calcul numérique. Les futures générations de GPU offrent néanmoins de bonnes perspectives.

Les unités de calcul alternatives confirment une tendance : la multiplication et la spécialisation des unités de calcul. Les calculs intensifs sont ainsi effectués sur des unités SIMD massivement parallèles, et les opérations de contrôle sur un GPP. Pour pouvoir tirer parti d'une telle architecture, les applications doivent être parallélisées à une granularité fine, et être peu conditionnées (avoir peu d'opérations de contrôle).

Les applications complexes de traitement d'image, comme la compression vidéo, combinent des calculs intensifs sur un ensemble de données réduit (bloc, macrobloc), des opérations de décision (tests et conditionnement) et des dépendances de données (prédiction). La parallélisation des traitements sur des architectures massivement parallèles peut s'avérer difficile. Les opérations de contrôle et les dépendances de données peuvent rendre les architectures basées flot/noyau inefficaces. En effet, les architectures Cell et GPU (associé à un PC) ont en commun de nombreuses unités SIMD et un seul GPP, par conséquent la puissance de calcul est déséquilibrée en faveur du calcul intensif, ce qui risque de créer un goulot d'étranglement sur les opérations de conditionnement. Le caractère hétérogène des algorithmes de compression vidéo va plutôt nous orienter vers des plates-formes plus génériques, composées d'éléments standards. La programmation sur des composants connus est de plus simplifiée.

3.1.4 Plates-formes multicomposants

Lorsqu'un processeur seul n'est pas assez puissant pour fournir des performances temps réel, il est naturel de penser à combiner la puissance et la spécificité de plusieurs composants sur des plates-formes multicomposants hétérogènes. La puissance de calcul n'est pas le seul critère de sélection d'une architecture multicomposant. Le coût, le temps et la praticité jouent aussi un rôle important. Cette section détaille les avantages et les inconvénients d'une architecture composée de processeurs et de composants dédiés.

3.1.4.1 Intérêts d'une architecture multicomposant

Différents critères peuvent motiver le choix d'une architecture multicomposant.

- **Puissance de calcul**
  Les architectures multiprocesseurs hétérogènes permettent de combiner la puissance de calcul et la spécificité de plusieurs composants pour atteindre les performances nécessaires.
- **Rapidité de développement**

Un ASIC requiert un temps de développement très long (conception, réalisation, tests) alors qu'une plate-forme multicomposant utilise des processeurs existants. On trouve également ce genre de plates-formes chez certains fabriquants comme Vitec Multimedia ou Sundance. La programmation des composants peut être réalisée dans un langage de haut niveau.
Cette rapidité d'implantation est intéressante pour réaliser des prototypes tôt dans le processus de développement pour accompagner une annonce de produit, ou bien pour lancer un produit tôt sur le marché.

- **Prototypage**
  Le prototypage d'applications est une étape utile dans un processus de développement. Une architecture hétérogène proche du produit final permet de prouver la faisabilité et de mettre en évidence la plupart des goulots d'étranglements sur lesquels les efforts de développement doivent être concentrés.
- **Faible coût**
  Pour les marchés de niche, où le développement d'un ASIC serait trop coûteux face à un faible volume de vente, les solutions multiprocesseurs peuvent s'avérer profitables.
- **La consommation et la puissance dissipée**
  Un équipement embarqué doit consommer peu pour garantir une autonomie suffisante. Les DSP consomment jusqu'à cent fois moins qu'un GPP performant.
- **Evolutivité**
  Les cartes multicomposants sont généralement construites autour d'un ou plusieurs processeurs standards (GPP ou DSP) programmables en C. Les applications sont donc rapidement portées sur cible, de plus l'application comme la plate-forme peuvent être modifiées en limitant les conséquences (redéveloppement).
- **Flexibilité**
  Une carte multiprocesseur est réutilisable pour plusieurs applications beaucoup plus facilement qu'un composant dédié.

Malgré l'augmentation des capacités d'intégration des composants et de l'émergence des composants multicoeurs, les plates-formes multiprocesseurs restent un compromis pour la réalisation de prototypes et les marchés de petites séries.

3.1.4.2 Difficultés de mise en oeuvre

Le portage d'une application sur une architecture multicomposant n'est pas trivial. Les principales difficultés résident dans la parallélisation des calculs et dans les dépendances de données.

Parallélisme et ordonnancement La première difficulté est la parallélisation de l'algorithme. En effet, pour pouvoir tirer parti d'une architecture multiprocesseur il est d'abord nécessaire de distribuer l'application sur les différentes unités de calcul. Les dépendances de données entre les différentes opérations doivent donc être étudiées afin de pouvoir paralléliser des opérations indépendantes.

L'ordonnancement a aussi son importance dans l'efficacité d'une architecture parallèle afin de réduire les attentes de données résultant en une sous-utilisation des processeurs. D'autre part, une utilisation sous-optimale est à craindre lorsque des

opérations critiques sont déportées sur un composant dédié sans avoir adapté l'algorithme ou réordonnancé les traitements. En effet, le processeur commence le calcul, envoie les données sur le co-processeur et attend le résultat, ce qui se traduit par une période d'inactivité et donc à une accélération réduite.

Enfin, un système constitué d'autant d'unités de calcul dédiées que d'opérations critiques est difficile à exploiter de manière efficace dans la mesure où il conduirait à une sous-utilisation de ses composants.

Communications Une application distribuée implique forcément des communications et synchronisations inter-composants. La distribution et l'ordonnancement des opérations doivent prendre en compte les aspects de bande passante sur les liens de communication entre composants. En effet, la saturation des communicateurs conduit à l'inactivité des composants, et donc à une utilisation sous-optimale de l'architecture. Il est donc important de bien dimensionner les bandes passantes inter-processeurs ou de réduire les échanges de données en distribuant les opérations de manière plus efficace.

De plus, les problèmes physiques liés à la propagation des électrons font que les fréquences des bus inter-processeurs sont inférieures à celles des bus internes. Les problèmes liés aux dépendances de données ont donc un impact d'autant plus grand que les processeurs sont sur des composants distincts.

Les systèmes multicomposants sont des solutions couramment employées pour accélérer les temps de calcul. Cependant, les temps de développement restent longs et doivent être diminués pour plus de rentabilité. Pour cela, l'hétérogénéité de l'architecture doit être transparente pour le concepteur. L'utilisation de langages haut-niveau, de bibliothèques spécialisées et d'IP participent à cet effort.

La programmation multitâche permet de tirer profit d'une architecture multicœur mais nécessite un système d'exploitation et une architecture homogène. De plus, cette approche manque de généricité et est utilisée en général après les étapes de conception. Des outils de conception d'applications parallélisables sont donc indispensables pour accélérer les développements sur des architectures parallèles, que ce soit pour des plates-formes multicomposants ou des processeurs avec de plus en plus de cœurs de calculs.

3.1.5 Discussion sur l'adéquation entre les algorithmes et les éléments d'architecture

Plusieurs types de processeurs et d'architectures viennent d'être présentés. Leurs caractéristiques diffèrent et l'intérêt d'un processeur dépend donc de son utilisation finale. Dans cette section, nous allons évoquer les intérêts et les inconvénients des éléments clés du matériel en fonction de l'algorithme visé. L'objectif est de définir une architecture dédiée à l'estimation de mouvement temps réel pour la compression vidéo H.264.

3.1.5.1 Notion d'exécution temps réel pour la compression vidéo

Le temps réel pour la compression vidéo se mesure avec deux notions clés :

– La latence est le temps nécessaire à une image pour traverser tous les éléments de l'encodeur. Dans un contexte de visio-conférence, ou encore celui plus contraignant de télémanipulation (ex : télémédecine), la latence doit être très faible : l'image doit être acquise, compressée, transmise, puis décodée et enfin affichée quasiment instantanément. L'ordre de grandeur est de quelques dizaines de millisecondes. Nous nous fixons des contraintes plus souples étant donné que le contexte est la diffusion vidéo grand public (évènements télévisés en direct). Le délai accepté peut aller de moins d'une seconde à plusieurs secondes. L'estimation de mouvement doit donc avoir une latence bornée pour ne pas ralentir l'encodage. On peut néanmoins se fixer un délai correspondant à plusieurs images. Bien qu'étant bornée, la latence ne sera pas contraignante.
– La cadence. On considère qu'un encodeur vidéo est "temps réel" lorsqu'il est capable de tenir la fréquence d'acquisition des images. C'est à dire jusqu'à 60 images par seconde. La taille de l'image a également un impact important, puisqu'elle représente la taille des données à traiter. Pour la diffusion vidéo grand public, le format affichant les plus hautes contraintes est le 1080p60. C'est à dire une image 1920x1080 pixels à 60 Hz à traiter en moins de 17 ms ($1/60^e$de seconde). Nous nous placerons dans un contexte d'étude et de démonstrateur, à ce titre nous allons nous limiter à atteindre 30 ou 60 Hz en 720p (1280x720) suivant l'implantation réalisée.

La vitesse d'exécution de l'estimation de mouvement dépend à la fois de l'algorithme et de l'architecture. La façon dont l'algorithme est implanté a aussi un impact. Un calcul peut être plus ou moins optimisé sur un processeur donné. Nous n'allons pas le prendre en compte dans ce paragraphe, afin de ne considérer que des performances optimales. Pour évaluer la pertinence d'un algorithme, plusieurs paramètres doivent être pris en compte, tels que la quantité de calcul, la régularité des opérations, les accès mémoires, etc.

3.1.5.2 Algorithmes de mise en correspondance

La recherche exhaustive C'est un algorithme très régulier, les données peuvent être réutilisées d'une position à une autre. Le nombre de calculs est toutefois considérable, seule une architecture câblée massivement parallèle peut être mise à profit.

Les algorithmes d'élimination de candidats Cette technique comporte beaucoup de contrôle et de conditionnement. Ceci laisse peu de place à l'optimisation sur des processeurs classiques car le conditionnement se traduit par des tests et des branchements, ce qui empêche les boucles d'être optimisées (pipelinées). De plus le problème de bande passante mémoire, qui est très limitante, n'est pas pris en compte.

Les algorithmes prédictifs Ils nécessitent un accès aléatoire à la mémoire. Il est donc difficile de réutiliser les données grâce à des registres à décalage. Ces algorithmes sont adaptés aux processeurs avec un modèle de mémoire hiérarchique (cache).

3.1.5.3 Spécificités de H.264

Taille de bloc variable L'estimation de mouvement à taille de blocs variable augmente la charge de calcul. La réutilisation de SAD nécessite une recherche exhaustive, beaucoup trop coûteuse en haute définition. Comme les fenêtres de recherche sont redondantes pour les différentes sous-partitions d'un macrobloc, des calculs séquentiels macrobloc par macrobloc permettent de profiter des mémoires caches sur un processeur. A l'inverse, des traitements indépendants pour chaque sous-partition offrent une parallélisation aisée.

Multiples images de référence Un algorithme prédictif permet de réduire considérablement la charge de calcul induite par les multiples images de référence. Cependant, si pour un bloc courant il faut considérer deux fenêtres de référence, le goulot d'étranglement est sur la bande passante mémoire et la taille des mémoires cache. Un algorithme tel que "multiple current block single reference frame" [CTHC07] permet de réduire ces problèmes d'accès mémoire.

Raffinement sub-pixélique Le jeu de données de l'image de référence étant multiplié par seize pour le quart de pixel, une contrainte lourde est mise sur la bande passante mémoire [UPND07]. Une interpolation à la volée permet de profiter des mémoires rapides et proches de coeur de calcul pour stocker les valeurs interpolées dans le cas d'un processeur. Dans le cas d'une réalisation câblée, les échantillons interpolés ne sont plus stockés, mais calculés à chaque accès. Le nombre de calcul peut donc être augmenté, mais les accès à la mémoire sont largement diminués, et les performances sont globalement améliorées.

Les algorithmes d'estimation de mouvement sont très contraints par l'accès aux données. Les architectures spécialisées permettent de tenir une charge de calcul élevée lorsque l'algorithme est régulier (très peu conditionné). Les algorithmes prédictifs sont adaptés aux processeurs, plus généralistes, avec des mémoires à plusieurs niveaux. De manière générale une architecture adaptée à l'estimation de mouvement doit présenter une bande passante mémoire très élevée. D'une part, les images haute définition représentent de grandes quantités de données devant être contenues dans des mémoires externes. D'autre part l'accès aux données est intensif, une technique efficace de réutilisation de données doit être mise en place (cache).

3.1.6 Synthèse sur les processeurs

Avec l'augmentation des capacités d'intégration, les processeurs deviennent de plus en plus complexes. L'émergence des processeurs multicœurs ces dernières années (Intel, AMD) traduit l'incapacité à améliorer les performances avec une seule unité de calcul. En effet, une architecture puissante et très efficace requiert une spécialisation des unités de calcul et l'optimisation poussée des programmes (ex : calcul SIMD). L'évolution des processeurs a permis d'augmenter les performances des applications grâce aux prédictions de branchement, exécution spéculative et dans le désordre, augmentation des unités dédiées, au détriment de leur taille et leur consommation. Les processeurs dédiés au traitement du signal ont une architecture réduite aux simples unités de calcul dédiés. Leurs performances restent élevées, avec une taille inférieure

et une consommation très réduite par rapport aux GPP, autorisant des applications embarquées. Les composants spécialisés permettent d'obtenir une puissance de calcul élevée au prix d'un temps et d'un coût de développement très longs. Les nouvelles architectures sont des processeurs massivement parallèles possédant plusieurs processeurs SIMD dédiés.

Les plates-formes multicomposants permettent de multiplier la puissance de calcul disponible, avec plusieurs processeurs dont la technique de programmation est maîtrisée, ce qui réduit le temps de développement par rapport à une architecture nouvelle (modèle, langage de programmation, outils). L'hétérogénéité des plates-formes permet de combiner les intérêts des différents composants.

Pour l'estimation de mouvement en haute définition, un des éléments clés est l'accès à la mémoire. Les algorithmes doivent être réguliers pour être optimisés. Depuis l'existence des traitements vidéo numériques, des études ont été menées sur des composants dédiés à l'estimation de mouvement. Les avantages et les inconvénients des travaux les plus récents sont présentés dans le paragraphe suivant. On y retrouve les éléments évoqués dans ce paragraphe, principalement des composants dédiés (SoC).

3.2 Analyses d'architectures dédiées à l'estimation de mouvement

Afin d'obtenir des performances temps réelles, de diminuer la taille et la consommation ou pour réduire le coût, des composants dédiés ont été développés. Ces architectures spécialisées, implantées sur ASIC ou FPGA, permettent de répondre aux différents problèmes spécifiques que pose l'estimation de mouvement pour la compression vidéo.

Cette section présente les caractéristiques générales des processeurs dédiés à l'estimation de mouvement et dresse un état de l'art des solutions apportées au problème de l'estimation de mouvement.

3.2.1 Les techniques de base

Les architectures VLSI (Very Large Scale Integration) adressent les limites de performance grâce à la parallélisation des calculs et à la réutilisation des données pour réduire les accès aux mémoires. En ce qui concerne l'estimation de mouvement (Alg. 2.1-Chapitre 2) le parallélisme peut être extrait à plusieurs niveaux. Les boucles imbriquées de l'algorithme 2.1 peuvent être réorganisées pour être déroulées.

3.2.1.1 Les architectures systoliques

Une architecture systolique est un ensemble de processeur élémentaires (PE) interconnectés organisés sous forme de matrice. Les architectures de ce type gagnent leur performance en éliminant les goulots d'étranglement issus de l'accès aux données dans la mémoire en réutilisant successivement les données. En effet, plutôt que de nécessiter une large bande passante, les données sont propagées d'un PE à l'autre à travers des registres.

Les opérations élémentaires de l'application sont réalisées en parallèle dans les PE, puis les données et les résultats sont propagés dans les PE voisins pour poursuivre

le calcul. Dans une architecture à une dimension (1D), chaque PE est connecté à ses deux voisins horizontalement ou verticalement et dans le cas à deux dimensions (2D) avec ses quatre ou huit voisins.

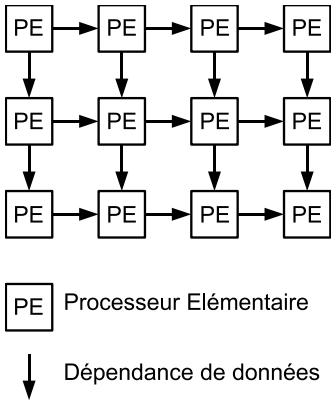


FIG. 3.8 – Exemple d'architecture systolique 2D 4×3.

L'algorihtme d'estimation de mouvement comporte des boucles imbriquées (Alg. 2.1), et l'ordre dans lequel elles sont exécutées peut être interchangé afin de les dérouler. Dans la suite du document, nous allons distinguer deux catégories de parallélisme. Le parallélisme type-1 [DS89] ou inter-candidat [CCH+06], et le parallélisme type-2 [DS89] ou intra-candidat [CCH+06] définis ci-dessous.

3.2.1.2 Parallélisme intra-candidat

Le parallélisme intra-candidat résulte du déroulement des boucles l et/ou k (Alg. 3.1) et de leur parallélisation matérielle. Autrement dit, chaque PE est responsable de la mesure de distorsion d'un pixel particulier dans le bloc courant. Une architecture à une dimension correspond au déroulement d'une seule boucle. Le nombre de PE est en général égal à la taille des blocs et un arbre d'additioneurs est nécessaire pour ajouter les valeurs entre elles.

Algorithme 3.1 Recherche exhaustive avec parallélisme intra

$for\, \Delta i = -p..p$ (espace de recherche vertical)
 $for\, \Delta j = -p..p$ (espace de recherche horizontal)
  $for\, k = 1..M$ (hauteur de bloc)
   $for\, l = 1..N$ (largeur de bloc)
    $SAD(\Delta j, \Delta i) += |x_1(k,l) - x_2(k + \Delta i, l + \Delta j)|$
   $end\, l$
  $end\, k$
 $end\, \Delta j$
$end\, \Delta i$

x_1 est l'image courante et x_2 l'image de référence

La figure 3.9 représente l'architecture développée dans [HL92]. Chaque PE contient la valeur d'un pixel du bloc courant (de taille 4x4 dans cet exemple). Les pixels de la fenêtre de référence sont entrés séquentiellement puis propagés de PE à PE, à

travers des registres. Un arbre d'additioneurs ajoute les valeurs de distorsion issues des différents PEs, et les valeurs finales sont comparées entre elles.

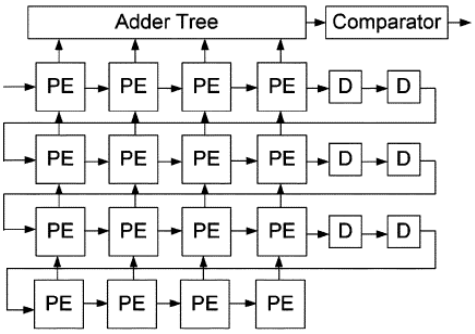


FIG. 3.9 – Exemple d'architecture intra 2D [HL92] pour un bloc 4x4

Dans [DS89], le bloc courant et la fenêtre de référence sont propagés. Une architecture 1D est également proposée afin de réduire la taille du composant.

Une architecture hautement parallèle est proposée dans [CCH+06] où les bus de données sont élargis pour alimenter les PE avec plusieurs pixels à chaque cycle.

3.2.1.3 Parallélisme inter-candidat

Le parallélisme inter-candidat résulte du déroulement des boucles Δi et Δj (Alg. 3.2). Chaque PE calcule la mesure de distorsion complète pour un candidat donné. Le nombre de PE est en général égal à la taille de la fenêtre de recherche.

Algorithme 3.2 Recherche exhaustive avec parallélisme inter

$for\, k = 1..M$ (hauteur de bloc)
 $for\, l = 1..N$ (largeur de bloc)
  $for\, \Delta i = -p..p$ (espace de recherche vertical)
   $for\, \Delta j = -p..p$ (espace de recherche horizontal)
    $SAD(\Delta j, \Delta i) += |x_1(k,l) - x_2(k+\Delta i, l+\Delta j)|$
   $end\, \Delta j$
  $end\, \Delta i$
 $end\, l$
$end\, k$

x_1 est l'image courante et x_2 l'image de référence

La figure 3.10 illustre l'architecture proposée dans [YH95]. Le bloc courant est propagé et la fenêtre de référence est diffusée. Les SAD partielles sont sauvegardées et accumulées dans les PE.

Les deux techniques présentées ci-dessus peuvent être combinées afin d'augmenter le parallélisme. La taille du composant s'en trouve par conséquent augmentée dans le but d'améliorer la puissance de calcul. Dans [CCH+06], une solution avec huit unités de calcul est proposée afin de traiter huit candidats simultanément. Une autre architecture composée de huit unités linéaires avec un parallélisme intra-candidat de

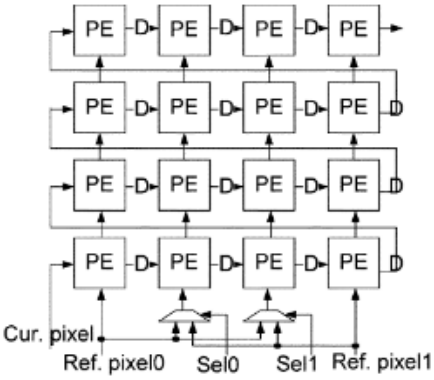


FIG. 3.10 – Exemple d'architecture inter 2D [YH95]

16 pixels de large est décrite dans [SLGI05]. La fenêtre de recherche est alors divisée horizontalement en huit zones dont chacune est traitée par une unité.

Les architectures systoliques permettent de mettre en œuvre efficacement des algorithmes réguliers. C'est à dire que l'accès aux données ne doit pas être aléatoire pour pouvoir profiter du mécanisme de décalage. C'est pourquoi les travaux présentés ci-dessus mettent en oeuvre l'algorithme de recherche exhaustif. Pour les algorithmes rapides, d'autres mécanismes doivent être mis en oeuvre.

3.2.1.4 Algorithmes rapides

Les implantations d'algorithmes rapides tels qu'une recherche en diamant ou hiérarchique nécessitent une gestion améliorée de l'accès aux données.

Une architecture pour des algorithmes de type PMVFAST ou EPZS est proposée dans [LLS05]. Elle est composée de neuf PE pour calculer simultanément neuf candidats (parallélisme inter). Une unité RISC (Reduced Instruction Set) réalise les taches complexes de contrôle (gestion du motif de recherche, sélection des prédicteurs, arrêt anticipé). Une unité de génération d'adresses est responsable de l'accès aux données à travers deux mémoires locales.

Un processeur d'estimation de mouvement programmable est présenté dans [GBG+07]. Il est composé principalement d'une matrice de 8x8 PE interconnectés pour le calcul SIMD des distorsions (parallélisme intra). Des registres dans chaque PE stockent les données utiles. Une unité d'accès aux données gère les transferts entre la mémoire locale et les registres internes aux PE en fonction de l'opération à effectuer.

Ces architectures hétérogènes permettent de traiter les algorithmes rapides en combinant une unité programmable gérant les contrôles complexes à une unité de calcul parallèle. Les différences avec les architectures systoliques classiques concernent principalement les accès aux données. Le caractère aléatoire nécessite une large bande passante et des mémoires locales.

Les réalisations matérielles d'estimateurs de mouvement sont basées sur les techniques présentées ci-dessus. Les opérations sont parallélisées afin de réduire à la fin

le temps de calcul et les accès aux données. Cependant, ce ne sont pas les seules techniques employées dans le cadre de la compression vidéo H.264.

3.2.2 Architectures pour MPEG4-AVC/H.264

Les outils de compensation de mouvement des nouveaux standards doivent être pris en compte dans la conception des architectures matérielles, notamment la taille de bloc variable, le raffinement subpixélique, ou les multiples images de référence.

3.2.2.1 Taille de blocs variables

L'estimation de mouvement à taille de bloc variable doit être capable d'estimer un vecteur pour chaque sous partition, soit au total 41 vecteurs pour un macro-bloc (un 16x16, deux 16x8 et 8x16, quatre 8x8, huit 8x4 et 4x8, et seize 4x4). La solution la plus souvent adoptée est la réutilisation de SAD associée à une recherche exhaustive [CCH+06, YM04, SLGI05].

La solution proposée dans [LL04] est un algorithme rapide (hiérarchique) associé à la réutilisation des SAD pour les différentes tailles de bloc. L'estimation de mouvement au niveau de la pleine résolution est effectué sur une fenêtre +/-2 pixels seulement pour une taille de bloc 16x16. Le calcul des SAD, sur une base 4x4, permet d'associer les résultats pour obtenir les SAD des blocs plus grands. Cette solution conduit à une qualité des champs de vecteur inférieure car elle est basée sur l'hypothèse que les vecteurs des différentes sous partitions sont proches, de 4x4 à 16x16, ce qui est contradictoire avec l'idée des tailles de blocs variables. Une approche différente est utilisée dans [GBG+07] où les différentes tailles sont traitées successivement grâce au caractère programmable de l'architecture en utilisant les données précédemment calculées comme prédiction [LAJ+05]. Le coût de calcul est supérieur car plusieurs recherches successives sont exécutées. Cependant les vecteurs des sous-partitions sont plus pertinents.

3.2.2.2 Raffinement fraction de pixel

La stratégie d'interpolation généralement utilisée pour le raffinement subpixélique est une interpolation à la volée. La mémoire et la bande passante nécessaires sont diminuées. De plus, le calcul des valeurs nécessite seulement une faible latence. Il existe cependant différentes stratégies de raffinement.

Une seule étape : l'architecture proposée dans [RB05] propose une recherche exhaustive d'amplitude $[-3.75; +4]$ directement à la précision quart de pixel avec une réutilisation de SAD pour traiter les tailles de bloc variables. Cette solution est très coûteuse car la fenêtre de recherche est réduite et le nombre de candidats est très élevé (multiplié par 16 au quart de pixel). De plus la réutilisation de SAD pour les tailles de bloc variables conduit à des champs de vecteurs de qualité réduite à cause de la fenêtre de recherche restreinte.

La stratégie de raffinement du vecteur déjà estimé au pixel entier est plus généralement utilisée.

Raffinement exhaustif : Le raffinement exhaustif est utilisé dans [DRS05]. Une fenêtre de recherche de $]-1;+1[$ est généralement suffisante. L'architecture de type inter avec un PE par candidat évite d'avoir à sauvegarder les valeurs subpixel.

Raffinement à deux pas : Le raffinement à deux pas permet de réaliser seulement deux fois neuf calculs et comparaisons de distorsion pour le quart de pixel, au lieu de quarante neuf dans le cas du raffinement exhaustif. Cependant, les échantillons demi-pixel doivent être disponibles pour l'interpolation quart de pixel. Ils peuvent donc être soit sauvegardés, nécessitant des registres, soit recalculés, diminuant ainsi l'efficacité de l'approche. Cette dernière technique est utilisée dans [CHC04]. L'unité de calcul est composé de neuf PE, chacun traitant une ligne de quatre pixels de large. Cette architecture est dupliquée quatre fois dans [YGI06] pour avoir des PE et un filtre d'interpolation de seize pixels de large.

Pour être compatible avec une taille de bloc variable, les vecteurs issus de l'estimateur de mouvement pixel entier sont raffinés successivement taille par taille [CHC04, YGI06]. Les architectures précédentes doivent alors être assez rapides pour prendre cet aspect en compte.

3.2.2.3 Multiples images de référence

Le standard H.264 permet de choisir une image de référence parmi plusieurs dans le but d'améliorer la prédiction inter-image. L'estimation de mouvement doit par conséquent multiplier les fenêtres de recherche par le nombre d'images de référence. Pour prendre cela en compte, les architectures sont développées suivant 2 stratégies.

- Les différentes images de référence sont parcourues séquentiellement. L'architecture de base est prévue pour une seule image de référence. Cependant, les performances sont assez élevées pour traiter plusieurs références séquentiellement [CCH+06]. Une dérivée de cette stratégie est la parallélisation de plusieurs modules indépendants, chacun traitant une image de référence. Les performances requises de chaque module sont moins importantes, mais la solution est plus complexe.
- Avec un changement de variable, il est possible d'inverser le problème : considérer une seule image de référence pour plusieurs images courantes [CTHC07]. Cette technique s'appuie sur le fait que les données d'une image courante (un macro-bloc) sont réduites par rapport à celles d'une fenêtre de référence. Les accès mémoire sont donc réduits en réutilisant successivement la même fenêtre de référence pour plusieurs macro-blocs appartenant à plusieurs images courantes.

Peu de travaux présentent une solution dédiée aux multiples images de référence. Dans [CCH+06] les performances sont suffisantes pour traiter quatre images de référence pour une résolution D1 (720x576), mais ne traite la haute définition qu'avec une seule image de référence. Une solution permettant de réduire les accès mémoire est proposée dans [CTHC07]. Cependant, aucun détail d'implantation n'est donné. On peut supposer qu'une telle solution nécessiterait une unité de calcul très gourmande en taille de silicium pour être efficace avec un algorithme exhaustif.

L'estimation de mouvement est un problème très complexe pour H.264. Les contraintes de charge de calcul et de bande passante mémoire sont très élevées, notam-

ment pour la haute définition. Les différents points de la norme sont adressés par des architectures dédiées. Une solution d'encodeur temps réel est présentée dans [CLC06] avec une seule image de référence en haute définition (720P 30Hz).

Pour atteindre des performances temps-réel, plusieurs compromis sont envisageables : utiliser plusieurs composants (slices), réduire le nombre de modes grâce à un choix à priori afin d'éviter des calculs. La première technique conduit à une augmentation du coût de la solution, la deuxième résulte en une baisse de la qualité ou des performances de compression.

En outre, les images B ne semblent pas être prises en compte dans les architectures d'estimation de mouvement présentées. La solution adoptée est alors de combiner deux vecteurs calculés pour le mode P avec deux images de références. Le résidu n'est donc pas calculé dans l'estimateur de mouvement, mais seulement au niveau de la compensation de mouvement, ou lors du choix du mode d'encodage.

De même que l'algorithme exhaustif pour l'estimation de mouvement est très coûteux, l'estimation exhaustive pour tous les modes de la norme H.264 conduit à des solutions très coûteuses.

3.3 Conclusion

Les processeurs génériques fournissent une solution flexible et évolutive, aussi bien du point de vue logiciel que matériel. La programmation en un langage de haut niveau permet de développer rapidement des algorithmes et d'y apporter des modifications aisément. La prise en compte des spécificités de l'architecture (mémoires caches, registres, unités SIMD, ...) conduit à une meilleure implantation, tout en offrant une indépendance relative vis-à-vis du processeur cible (constructeur, version, ...).

Les composants dédiés permettent de tailler une solution sur mesure. Les performances offertes sont donc presque infinies. Les inconvénients majeurs sont le temps de développement élevé et l'évolutivité très réduite. Le coût de conception est donc très élevé. Les solutions programmables, telles que les FPGA, constituent un compromis entre flexibilité et performance d'une architecture dédiée, avec toutefois un temps de développement toujours plus long que sur un processeur standard.

L'étude des architectures spécialisées pour l'estimation de mouvement fait apparaître les contraintes matérielles et les solutions apportées. L'estimation de mouvement est un problème complexe, impliquant de lourdes contraintes. De plus, cette complexité augmente avec l'évolution des techniques de compression vidéo et la résolution des images. Des composants d'estimation de mouvement dédiés pour la compression vidéo H.264 existent. Cependant, ils ne sont pas évolutifs et ne sont par conséquent pas adaptés au prototypage. Pour cela, une solution à un stade précoce du processus de développement doit être envisagée. Une implantation sous-optimale est donc tolérée, mais elle doit être évolutive pour prendre en compte les futures évolutions vers le produit final, et une possible réutilisation des travaux.

Des solutions multicomposants hétérogènes peuvent combiner la flexibilité des processeurs standard, et la performance de composants dédiés, utilisés pour accélérer des opérations critiques. L'implantation sur processeur est facilitée par un langage de haut niveau et une architecture en général bien maîtrisée. Toutefois, le caractère distribué de l'application implique la parallélisation des opérations, leur ordonnancement

et la mise en œuvre des communications et synchronisations inter-composants. Cela peut s'avérer difficile à mettre au point manuellement, car le développeur logiciel doit maîtriser les communications et synchronisations entre les cœurs de calculs. L'utilisation d'outils de développement spécifiques tend à réduire ces problèmes. Une approche méthodologique adaptée facilite la parallélisation d'une application et son implantation distribuée.

Deuxième partie

Implantation d'estimateurs de mouvement

Chapitre 4

Méthodologie de prototypage

Pour programmer une plate-forme multiprocesseur, mais également des processeurs multicœurs, de plus en plus répandus, il est nécessaire de paralléliser les opérations. Des communications et synchronisations entre les unités de calcul sont alors insérées afin d'assurer le bon fonctionnement de l'application. Cela peut s'avérer complexe, et constituer une source d'erreurs.

Au sein du laboratoire IETR Groupe Image, un thème de recherche sur le prototypage rapide a débuté en 1996. L'objectif est de pouvoir porter des algorithmes de traitement des images, développés en interne, sur des architectures parallèles existantes. Il s'agit ainsi d'ajouter à ces applications la validation de leur exécution temps réel. Dès le début, ces travaux ont été placés dans le cadre de la méthodologie AAA (*Adéquation Algorithme Architecture*). Cette appellation regroupe un ensemble de recherches menées au niveau national, au sein du GdR ISIS (Groupement de Recherche Information, Signal, Images et viSion, thème C), visant à développer des méthodes systématiques de meilleure mise en correspondance entre l'algorithme d'une part, et l'architecture d'autre part. Cette méthodologie est adaptée à notre problématique, à savoir des applications d'images (systèmes orientés données), et une cible matérielle essentiellement composée de plusieurs processeurs (DSP). Nous avons ainsi choisi la méthodologie AAA/SynDEx, pour la réalisation de la phase d'adéquation et de génération de code. Le processus de développement présenté par la suite s'appuie sur cette méthodologie. Le portage d'applications d'estimation de mouvement, demandant une grande puissance de calcul, contribue à l'identification des carences de l'approche méthodologique et permet d'y apporter des améliorations. Dans ce chapitre, il est au préalable nécessaire de montrer les fondements de cette approche.

L'utilisation d'une méthodologie de prototypage vise à faciliter le développement. La distribution et l'ordonnancement des opérations sur les composants de l'architecture, les communications et synchronisations inter-processeurs sont gérées automatiquement, assurant ainsi une implantation fiable et rapide. L'approche est la plus générique possible afin de favoriser la réutilisation des travaux, par exemple lors de l'évolution d'un prototype ou lors du passage d'un prototype à un autre.

Dans une première partie nous présentons la méthodologie AAA/SynDEx, ses fondements et son utilisation. Puis, dans une deuxième partie, nous proposons des solutions à des limitations identifiées.

4.1 Présentation de la méthodologie AAA / SynDEx

Les travaux décrits dans les chapitres 5 et 6 ont été réalisés dans le cadre de cette méthodologie, d'une part pour la valider et résoudre si besoin est certains points bloquants pour une application complexe de traitement d'image avec de hautes performances, et d'autre part dans un soucis de généricité des développements.

La méthodologie s'appuie sur un outil de CAO (Conception Assistée par Ordinateur) universitaire : SynDEx (Synchronised Distributed Executive), développé par l'INRIA (`http://SynDEx.org`). Cet outil permet le prototypage rapide et l'implantation optimisée d'applications temps réel embarquées. Il nécessite en entrée de spécifier l'algorithme de l'application et l'architecture multicomposant. Il réalise ensuite une adéquation correspondant à une implantation optimisée de l'algorithme sur l'architecture, dont le résultat est une prédiction temporelle de l'exécution de l'algorithme sur cette architecture. Il génère enfin automatiquement pour chaque processeur un exécutif temps réel dédié ou un fichier de configuration pour un exécutif temps réel résident standard. Cet outil est basé sur le flot de données, ce qui est bien adapté aux algorithmes d'estimation de mouvement. Il permet de paralléliser une application sur une plate-forme multiprocesseur, en gérant automatiquement les transferts de données et les synchronisations.

4.1.1 Modèle d'algorithme

La spécification fonctionnelle d'une application consiste en général en un algorithme obtenu par composition de plusieurs sous-algorithmes. Dans AAA, un algorithme est défini par un graphe de dépendances de données qui est hiérarchique, conditionné et factorisé [LS97]. Les sommets du graphe sont les opérations et les arcs représentent les dépendances inter-opérations. Le graphe est orienté, les opérations sont partiellement ordonnées par les dépendances de données.

L'algorithme dans AAA/SynDEx est modélisé par un graphe de dépendance de données infini (Fig. 4.1), mais dans lequel un motif infiniment répété est identifié (graphe contracté Fig. 4.2). Le graphe est réduit par factorisation à son motif répété appelé *Graphe Flot de Données* (GFD). Ce modèle est adapté au traitement de l'information, et en particulier au traitement du signal et des images.

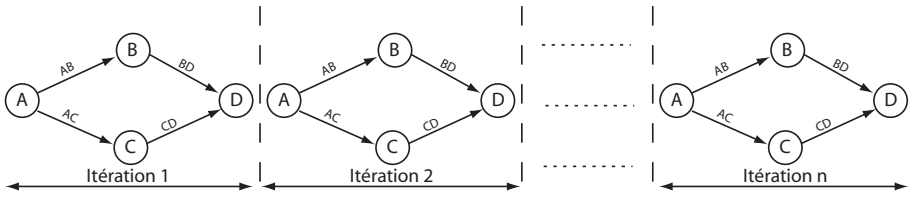


FIG. 4.1 – Graphe de dépendance sur n itérations

Le graphe d'algorithme peut être décrit de manière hiérarchique : chaque opération du graphe peut contenir un sous-graphe permettant une spécification hiérarchique de l'algorithme jusqu'aux "opérations atomiques" (Fig. 4.3). Une opération atomique est une opération élémentaire ne contenant que des ports d'entrée, de sortie, ou d'entrée-sortie. Sur la figure 4.3, les opérations atomiques sont A, B, C,D, E, C_1, C_2, tandis

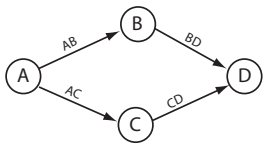


FIG. 4.2 – Graphe flot de données factorisé (contracté)

que F et G sont des opérations hiérarchiques. Les opérations atomiques que l'on peut définir sous SynDEx sont :

l'opération *Constant* qui produit des données identiques lors de chaque itération de l'algorithme. Il suffit donc de l'exécuter lors de la première itération du graphe d'algorithme.

l'opération *Delay* qui permet de spécifier les dépendances inter-itérations du graphe d'algorithme. Il s'agit d'une dépendance de donnée entre chaque répétition du graphe flot de données.

l'opération *Function* qui est une opération atomique, quand elle contient uniquement des ports d'entrée et de sortie. Cependant cette opération peut elle-même contenir des opérations de calcul, de conditionnement et de répétition, elles-mêmes hiérarchiques.

l'opération *Sensor* qui active les entrées du graphe flot de données (capteur). Cette opération peut uniquement produire des données, elle ne contient que des ports de sortie.

l'opération *Actuator* qui active les sorties du graphe flot de données (actionneur). Cette opération peut uniquement consommer des données, elle ne contient que des ports d'entrée.

Le graphe d'algorithme peut contenir des dépendances de conditionnement (sorties de A et B sur la figure 4.3). La valeur portée par une dépendance de conditionnement détermine, à chaque réaction (répétition infinie), le choix parmi un ensemble de sous-graphes alternatifs. Sur la figure 4.3, dans le cas où la sortie de A vaut 1 alors le sous-graphe contenant G est exécuté. Dans le cas où la valeur de A est 2, c'est alors le sous-graphe contenant D qui est exécuté. L'imbrication des conditionnements est traduit par une représentation hiérarchique du graphe d'algorithme. À chaque interaction avec l'environnement, concrétisé par un ensemble d'événements d'entrée, les valeurs des arcs de conditionnement déterminent à partir des valeurs d'entrée l'ensemble des opérations à exécuter pour obtenir les événements de sortie. Chaque sommet non conditionné produit ses événements sur ses sorties dès que tous les événements sur les entrées sont arrivés.

Un sous-graphe du graphe d'algorithme peut être répété un nombre fini de fois et peut contenir, à son tour, un sous-graphe lui aussi répété un nombre fini de fois correspondant à des "nids de boucles". L'imbrication des boucles conduit aussi à de la hiérarchie dans le graphe d'algorithme. Un sous-graphe répété un nombre fini de fois peut aussi être réduit par factorisation à son motif répétitif.

AAA/SynDEx fournit la possibilité de spécifier les opérations répétées sous forme factorisée. La spécification d'algorithme répété sous une forme factorisée repose sur des

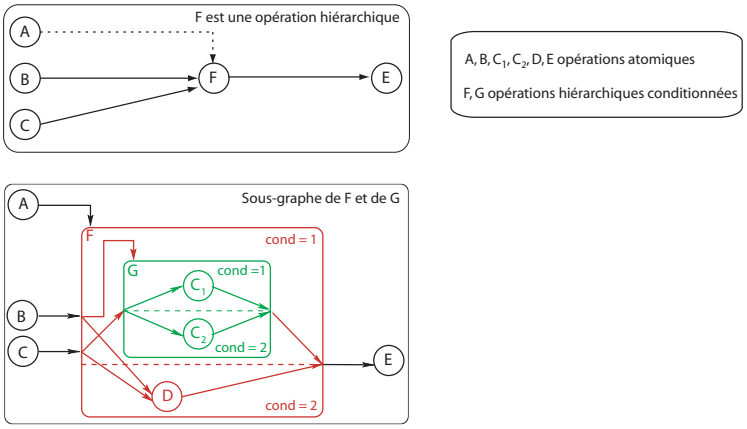


FIG. 4.3 – Graphe hiérarchique et conditionné

opérations particulières, appelées *sommets frontières* car elles délimitent des *frontières de factorisation*. Les parties du graphe d'algorithme, appelées motifs, délimitées par ces frontières sont les parties répétées du graphe. Les différents motifs répétés d'un graphe sont nécessairement disjoints. Les *opérations* ou *sommets frontières* sont créés automatiquement par l'outil SynDEx, et sont définies implicitement lors de la création du graphe d'algorithme.

Les *sommets frontières* sont les suivants :

– **sommet *Diffuse*** : diffuse la donnée en entrée aux motifs factorisés en sortie. Sur la figure 4.4, la donnée entre o_1 et o_2 est diffusée sur chaque opération répétée o_2.

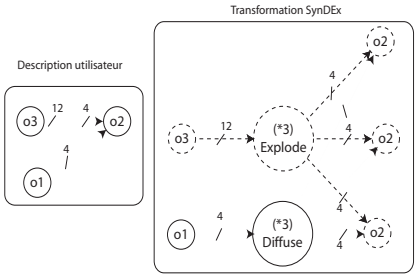


FIG. 4.4 – Sommet *Diffuse*

– **sommet *Fork*** : partitionne la donnée en entrée et distribue les parties aux motifs factorisés en sortie. La figure 4.5 illustre l'expansion du sommet *Fork*, concrétisée par l'opération *Explode*. L'utilisateur décrit sous SynDEx deux opérations $o_3 \rightarrow o_2$ dont le rapport entre la taille de la donnée sortante de o_3 et celle de la donnée entrante de o_2 spécifie implicitement le facteur de répétition de o_2 (ici 3).

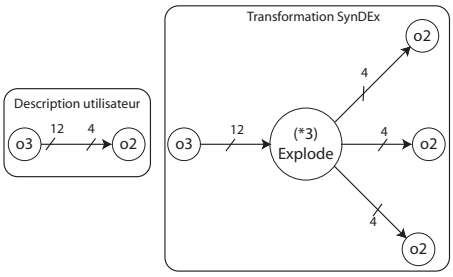


FIG. 4.5 – Sommet *Fork*

– **sommet *Join*** : regroupe les données partitionnées en entrée par les motifs factorisés en un vecteur en sortie. La figure 4.6 illustre l'expansion du sommet *Join* concrétisée par l'opération $Implode$. L'utilisateur décrit sous SynDEx deux opérations $o_2 \rightarrow o_3$ dont le rapport entre la taille de la donnée sortante de o_3 et celle de la donnée entrante de o_2 spécifie implicitement le facteur de répétition de o_2 (ici 3).

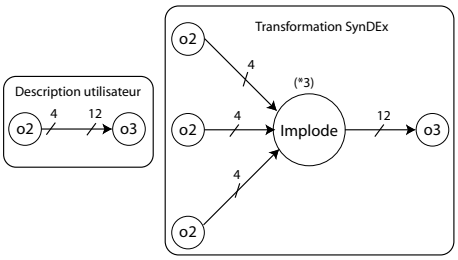


FIG. 4.6 – Sommet *Join*

– **sommet *Iterate*** : prend en entrée la sortie d'un des motifs factorisés ; sa sortie est redirigée vers le motif factorisé suivant. Une valeur d'initialisation est nécessaire. La figure 4.7 illustre l'expansion du sommet *iterate*. L'utilisateur décrit explicitement un facteur de répétition de l'opération o_2. Pour avoir la récursivité de l'opération o_2, il faut spécifier en plus sur un port d'entrée et un port de sortie de o_2 le même nom et les mêmes caractéristiques. L'expansion du sommet frontière *iterate* donne : $o_3 \rightarrow o_2 \rightarrow o_2 \rightarrow o_2$. La récursivité d'une opération peut également être définie implicitement, avec un sommet *Fork.*

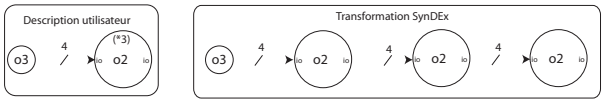


FIG. 4.7 – Sommet *Iterate*

Pour illustrer Ce modèle d'algorithme, nous prenons comme exemple la description d'un produit scalaire. Sur la figure 4.8, les sommets frontières sont représentés et permettent de réaliser le produit scalaire $A.B = s$ où s est un scalaire dont la valeur est :

$$s = A.B = a * c + b * d \text{ avec } A = \begin{bmatrix} a \\ b \end{bmatrix} \text{ et } B = \begin{bmatrix} c \\ d \end{bmatrix}.$$

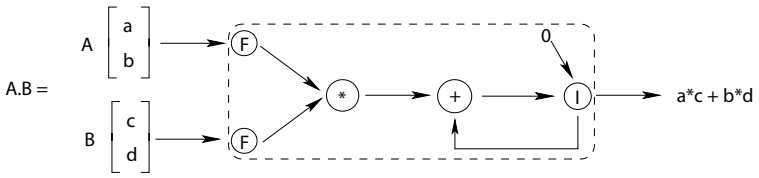


FIG. 4.8 – Graphe flot de données factorisé avec les sommets frontières (F sommet Fork et I sommet Iterate)

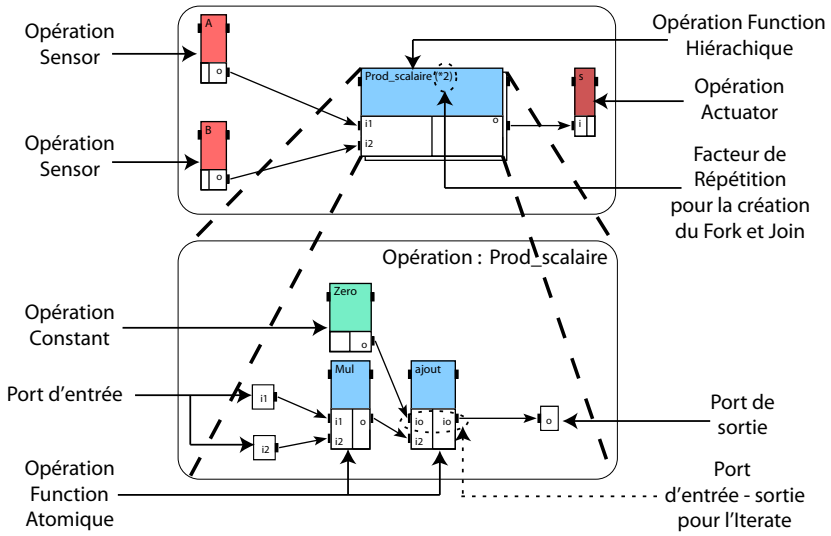


FIG. 4.9 – Graphe flot de données factorisé sous SynDEx

La figure 4.9 illustre la représentation sous SynDEx de la figure 4.8. A et B sont des opérations *sensor*, dont les ports de sortie sont de dimension 2. s est un *actuator* dont le port d'entrée est de dimension 1. $Prod\_scalaire$ est une opération *function* hiérarchique dont les ports d'entrée et de sortie sont de dimension 1. Les opérations $Fork\,(F)$ sont donc créées automatiquement (implicitement) à la frontière entrante de $Prod\_scalaire$, à à travers le facteur de factorisation entre les sorties de A et B et l'entrée de $Prod\_scalaire$. $Zero$ est une constante nulle dont la valeur est définie à l'initialisation du graphe. mul et $ajout$ sont des opérations *function* atomiques. L'opération $ajout$ a une spécificité : elle a le même nom de port io sur une entrée et une sortie, ce qui crée l'opération frontière *Iterate*. Pour la première instance d'$ajout$, la valeur sur l'entrée io de cette opération est la constante de valeur nulle, ensuite l'opération $ajout$ est itérée et la sortie de $ajout_i$ est rebouclée sur l'entrée de $ajout_{i+1}$.

Le conditionnement et la factorisation sont les équivalents, en termes de graphe de dépendances de données, des structures de contrôle que l'on trouve dans les langages impératifs :

– If... Then... Else (*conditionnement*)

– For i=1 to N Do... (*factorisation*)

Le GFD permet d'exprimer les dépendances de données entre les opérations et de faire apparaître les opérations parallélisables. On parle de parallélisme potentiel. Les sommets frontières, principalement $Fork$ et $Join$, ont comme principal avantage d'extraire automatiquement du parallélisme potentiel de données (par opposition au parallélisme potentiel plus général d'opérations).

4.1.2 Modèle d'architecture

Le modèle d'architecture multicomposant [Gra00] choisi dans AAA/SynDEx permet de prendre en compte des architectures hétérogènes. Cela signifie non seulement que les opérateurs peuvent avoir chacun des caractéristiques différentes (durée d'exécution des opérations par exemple), mais aussi que certaines opérations peuvent n'être exécutées que par certains opérateurs. Ceci permet de décrire tout autant des composants programmables (processeurs-FPGA) que des composants non programmables (ASIC) (pas seulement des processeurs, par exemple un port d'entrée vidéo). Une architecture est un graphe, dont chaque sommet est une machine à états finis (machine séquentielle) et chaque arc une connexion physique entre deux machines à états finis. Dans AAA/SynDEx, il existe deux types de sommets :

– l'opérateur pour séquencer des opérations de calcul (séquenceur d'instructions),
– le médium (ou communicateur) pour séquencer les communications.

Dans AAA/SynDEx, il existe deux types de sommets médium :

– la mémoire partagée à accès aléatoire (RAM : Random Access Memory),
– la diffusion à accès séquentiel : SAM (Sequential Access Memory) ayant le fonctionnement d'une FIFO (First In First Out).

La connexion aux communicateurs au sein d'un processeur est considérée se faire par l'intermédiaire d'un séquenceur de communication indépendant du cœur de calcul. Les communications peuvent donc être réalisées en parallèle avec des calculs.

La figure 4.10 donne un exemple de graphe d'architecture de la méthodologie AAA/SynDEx. Les nœuds décrivent :

– $OPRi$ les opérateurs,
– $COMi$ les communicateurs,
– S les communicateurs SAM,
– R les communicateurs RAM.

4.1.3 Mise à plat du graphe d'algorithme

Le graphe d'algorithme entré par l'utilisateur est transformé par SynDEx avant d'effectuer l'adéquation. La transformation du graphe a pour objectif de résoudre les références et de défactoriser les définitions répétées ou conditionnées, mettant ainsi à plat sa hiérarchie : il s'agit donc d'une phase d'expansion du graphe d'algorithme. Les références sur des définitions hiérarchiques sont remplacées récursivement par le graphe d'algorithme de ces définitions. La récursion traite successivement les différents niveaux de hiérarchie. Le graphe obtenu après la mise à plat est un graphe de dépendances portant uniquement sur des opérations atomiques.

La mise à plat d'un "nid de boucles" dans un graphe d'algorithme nécessite la création des différentes opérations implicites situées sur les sommets frontières. Le

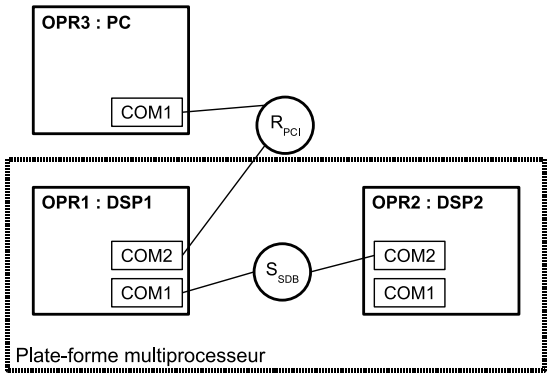


FIG. 4.10 – Graphe d'architecture dans la méthodologie AAA : modélisation d'une plate-forme 2 DSP (Sundance) reliée à un PC via un bus PCI

graphe d'algorithme de la figure 4.9 est transformé par SynDEx, pour ne contenir après la défactorisation que des opérations élémentaires (Fig. 4.11). Le sommet $Fork$ est traduit par l'opération élémentaire $Explode$, pour séparer (exploser) les vecteurs A et B. L'opération $Iterate$ du graphe de la figure 4.9 est itérée sur l'opération $ajout$ du graphe d'algorithme, de manière implicite comparée à l'exemple 4.7, et la dépendance de donnée $inter - ajout$ est créée automatiquement par l'outil SynDEx lors de la mise à plat du graphe d'algorithme.

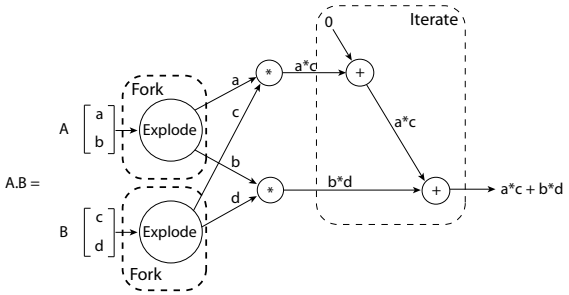


FIG. 4.11 – Mise à plat du graphe du produit scalaire

Les opérations de conditionnement nécessitent également la création de sommets $CondI$ et $Cond0$ (opérations atomiques) pendant la phase de mise à plat :

CondI pour la phase d'initialisation du conditionnement afin d'aiguiller la sortie d'une opération vers le sous-graphe qui doit être exécuté. Il permet surtout de savoir d'où vient la valeur de conditionnement lorsque l'on distribue l'application.

Cond0 pour la phase de finalisation du conditionnement afin de stocker les sorties des sous-graphes conditionnés vers une donnée unique consommée par une opération.

Le graphe d'algorithme de la figure 4.3 est transformé et expansé par SynDEx pour donner la figure 4.12, pour ne contenir que des opérations élémentaires. La sortie de l'opération C est conditionnée, l'opération $CondI_{CF}$ envoie la donnée suivant la valeur de A vers l'opération D ou $CondI_{CFG}$. L'entrée de E ou la sortie de $Cond_{Fo}$ prend

la valeur du sous-graphe conditionné qui vient d'être exécuté. La sortie de $Cond_{Fo}$ prend soit la valeur de D, soit la valeur $Cond0_{Go}$.

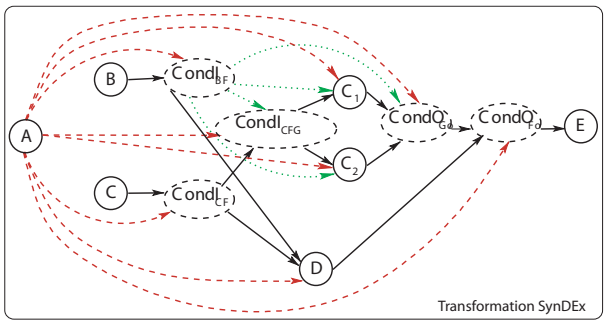


FIG. 4.12 – Sommets de conditionnement : $CondI$ et $CondO$

Effectuer cette transformation avant l'adéquation permet de se ramener pour les traitements de l'adéquation à un formalisme de graphe plus simple, plus proche des graphes flots de données habituels. Le critère considéré pour le choix de la meilleure implantation, dans l'espace des implantations possibles du graphe d'algorithme sur le graphe d'architecture, est de retenir l'implantation donnant une latence globale minimale (minimisant la durée globale de l'application) en tenant compte des coûts de communication inter-processeurs.

4.1.4 Implantation / Adéquation

L'implantation d'un algorithme sur une architecture multicomposant est une distribution et un ordonnancement non seulement des opérations de l'algorithme sur les opérateurs de l'architecture, mais aussi des opérations de communication, sur les communicateurs.

La **distribution** consiste à affecter chaque opération de l'algorithme à un opérateur capable de l'exécuter. Ceci conduit à une partition de l'ensemble des opérations de l'algorithme en autant de sous-graphes que d'opérateurs. Ensuite, il faut affecter chaque dépendance de données inter-opérateurs (c'est-à-dire entre opérations affectées à des opérateurs différents), à une route reliant les deux opérateurs (chemin dans le graphe de l'architecture). Il faut ainsi créer et insérer, entre les deux opérations de l'algorithme, les opérations de communication adéquates.

L'**ordonnancement** consiste à linéariser l'ordre partiel (rendre l'ordre total) associé à chaque sous-graphe de l'algorithme formé d'opérations de calcul et d'opérations de communication.

Une **implantation** est donc le résultat d'une transformation du graphe de l'algorithme (ajout de nouveaux sommets et de nouveaux arcs) en fonction du graphe de l'architecture, lui même transformé (détermination de toutes les routes possibles) [Gra00]. Chacune de ces implantations possibles présente des performances (latence) différentes. La latence est obtenue par le calcul de chemins critiques le graphe de l'implantation étiqueté par les durées d'exécution caractéristiques des opérateurs et des communicateurs de l'architecture.

Le résultat de l'adéquation est l'implantation qui minimise la latence en tenant compte de certaines contraintes de distribution et d'ordonnancement spécifiées par l'utilisateur. Le problème de minimisation ne peut pas être résolu exhaustivement (évaluation de toutes les solutions et sélection de la meilleure) compte tenu de sa complexité. Il est résolu grâce à une heuristique gloutonne détaillée dans [GLS99]. La solution peut être sous-optimale (le résultat optimal n'est pas garanti). Récemment, un algorithme génétique a été implanté pour l'adéquation, dans le cadre de la thèse M. RAULET [RAU06]. Les résultats sont généralement améliorés et offrent des perspectives intéressantes, notamment de pouvoir prendre d'autres contraintes que la latence en compte (cadence, taille mémoire, ...).

4.1.5 La génération de code

La sortie de SynDEx est un macro-code générique (indépendant de l'architecture cible) spécifiant pour chaque processeur :

- la liste des sémaphores de synchronisation entre les séquenceurs d'un opérateur,
- la liste des allocations sur les mémoires partagées et des sémaphores associés à leur synchronisation,
- la liste des allocations internes aux processeurs,
- la liste des opérations sous forme de séquenceur pour chaque communicateur,
- la liste des opérations de calcul.

La génération automatique d'exécutif distribué se fait suivant des règles décrivant la transformation d'un graphe d'implantation optimisé en un graphe d'exécution. L'aspect automatique de la transformation de graphe conduit à des allocations mémoires systématiques pour chaque sortie d'opération. La taille de la mémoire totale nécessaire peut donc être conséquente et influer de manière non négligeable sur les performances. Les travaux de thèse de M. RAULET sur la minimisation mémoire dans la méthodologie AAA [RAU06] favorisent la réutilisation des espaces mémoire grâce à une analyse de l'ordonnancement des opérations.

4.1.5.1 Synchronisations

La génération des synchronisations est détaillée ici car il est nécessaire de bien comprendre leur fonctionnement afin d'appréhender les travaux présentés au paragraphe 4.2.2.

Pour chaque opérateur (resp. communicateur), on construit un programme séquentiel formé de la séquence des opérations de calcul (resp. communication) qu'il doit exécuter. Pour garantir les précédences d'exécution entre les opérations et l'accès en exclusion mutuelle aux données partagées entre deux séquenceurs, on ajoute des opérations de synchronisation avant et après chaque opération qui lit (resp. écrit) une donnée écrite (resp. lue) par une opération appartenant à une autre séquence. Ces opérations de synchronisation utilisent des sémaphores générés automatiquement. Il a été montré à l'aide des réseaux de Petri que ces sémaphores permettent à l'exécutif de respecter l'ordre partiel du graphe d'algorithme initial, n'introduisant ainsi pas d'inter-blocage dans une itération infinie entre la séquence de calcul et celles de communication, ou entre deux itérations infinies [Gra00].

L'exécutif d'un opérateur de calcul est constitué d'une séquence de calcul principale et d'autant de séquences de communication que de média auxquels cet opérateur est connecté. Chaque séquence est exécutée par un séquenceur différent. Le mécanisme des réseau de Petri pour le cheminement des jetons (Fig. 4.13 et 4.14) est repris des travaux de T. GRANDPIERRE [Gra00]. Les séquenceurs sont synchronisés au niveau des dépendances de données les reliant. Des synchronisations sont générées pour chaque dépendance reliant deux opérations exécutées par deux séquenceurs différents (communication1-communication2, calcul-communication, communication-calcul).

Les synchronisations inter-séquences sont de deux types :

- intra-itération : on assure qu'une donnée n'est pas consommée avant d'être produite (synchronisations de type "tampons pleins")
- inter-itération : on assure qu'une donnée produite par une opération à l'itération n du graphe d'implantation n'est pas écrasée par la même opération à l'itération $n+1$ avant que toutes les opérations consommant la donnée à l'itération n soient exécutées (synchronisations de type "tampon vide").

Les synchronisations sont assurées par un mécanisme de sémaphores dont la gestion consiste en deux phases :

- déterminer les sémaphores nécessaires et générer des macros d'allocation pour ces sémaphores ;
- générer les macros d'attente notées *Suc* (équivalentes à *Wait ou P*) et de libération notées *Pre* (équivalentes à *Signal ou V*) sur les sémaphores avant et après les opérations de calcul ou de communication à synchroniser.

Les synchronisations diffèrent avec le type de médium. Dans SynDEx deux types de communicateurs sont autorisés : le modèle SAM et le modèle RAM.

4.1.5.2 SAM : transmission par paquet, FIFO

Les opérations de communication sont des "*Send*" et des "*Receive*" de données transmises entre communicateurs via une SAM (communication par passage de messages). Un "*Send*" sur un processeur est ordonnancé simultanément avec le "*Receive*" associé sur un autre processeur. Les sémaphores générés par SynDEx ne synchronisent que les séquences d'un même opérateur de calcul. Les synchronisations entre deux opérateurs de calcul communiquant des données de l'un vers l'autre sont assurées par les couples d'opérations d'envoi/réception. Dans le cas général le modèle SAM est utilisé lorsque physiquement une FIFO relie deux processeurs. La synchronisation est donc réalisée par le lien de communication matériel lui-même, grâce aux signaux de fonctionnement de la FIFO (full flag et empty flag).

La figure 4.13 représente le réseau de Petri d'une application utilisant un médium de communication de type SAM. Le graphe du processeur produisant la donnée est à gauche (graphes du séquenceur du calcul et du séquenceur de communication), celui du processeur consommateur est à droite. Ces graphes font apparaître les synchronisations logicielles générées par AAA/SynDEx à l'intérieur des processeurs tandis que les synchronisations inter-processeurs sont matérialisées par la FIFO et gérées par le matériel.

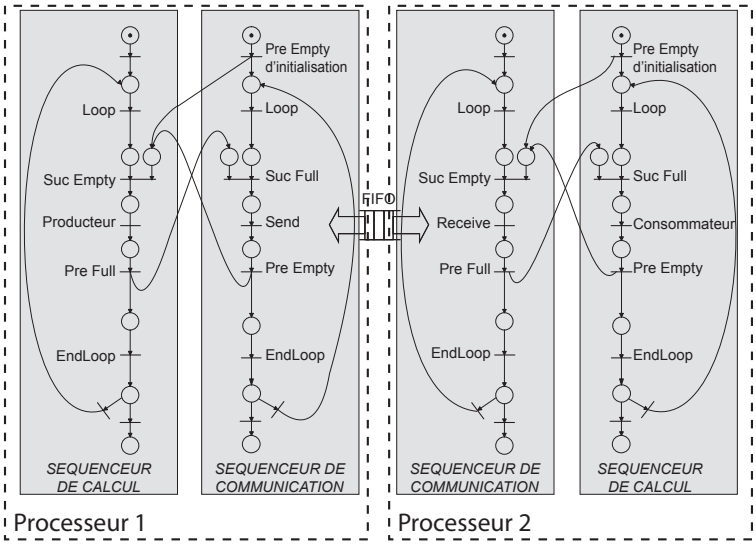


FIG. 4.13 – Réseau de Petri pour le modèle SAM

4.1.5.3 RAM : mémoire partagée

Le médium de communication de type RAM utilise une mémoire indexable à accès aléatoire, on peut donc venir lire les données dans un ordre différent de celui où elles ont été écrites. Les opérations de communication sont maintenant des "*Write*" et des "*Read*". Les sémaphores générées par SynDEx synchronisent les séquenceurs au sein d'un même processeur, comme précédemment, mais aussi entre les processeurs.

La figure 4.14 représente le réseau de Petri de la même application que précédemment, en utilisant un médium de communication de type RAM. Ces graphes font apparaître les synchronisations inter-processeurs prises en compte pour la génération de code.

Il y a deux types de synchronisations pour la modélisation d'un médium de type RAM :

- les synchronisations entre le séquenceur de calcul et le séquenceur de communications (*Pre* et *Suc*) au sein d'un même processeur qui sont les mêmes que précédemment,
- les synchronisations partagées entre les processeurs connectés à la RAM (*PreR* et *SucR*).

4.1.6 Production des exécutifs dédiés

Comme il a été précisé précédemment, les exécutifs générés par SynDEx se présentent sous la forme d'un macro-code indépendant de l'architecture. La dernière étape avant de passer à la compilation ou à la synthèse de l'application, avec des outils dédiés, est de transformer le macro-code en un code compilable (par exemple "C" pour un processeur, "VHDL" pour un FPGA). Cette étape utilise le processeur de macro "M4" afin de traduire les macros dans un langage compilable. Les traductions spécifiques pour les différentes cibles sont contenues dans des dictionnaires appelés *"noyaux d'exécutifs SynDEx".*

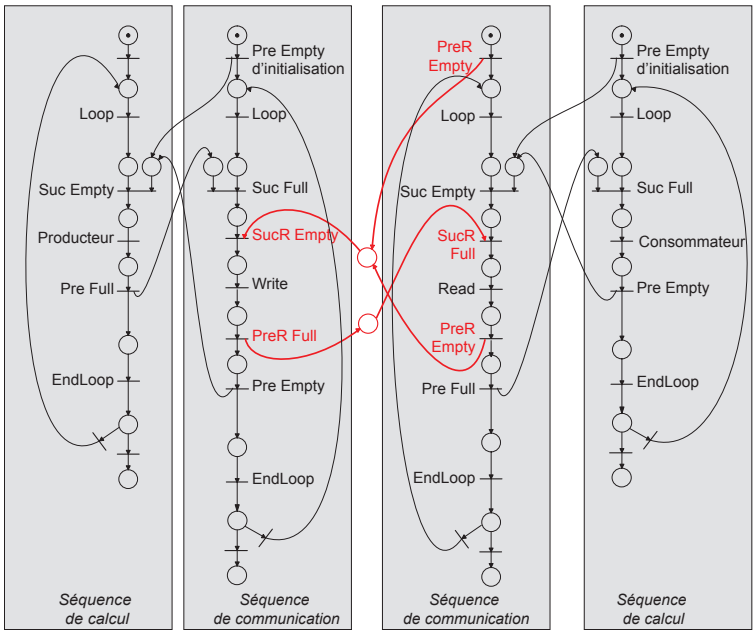


FIG. 4.14 – Réseau de Petri pour le modèle RAM

Historiquement, au sein de l'IETR, le développement des noyaux d'exécutifs SynDEx a débuté en 2001 [LR01, Rau02] par un noyau dédié au DSP de la famille C6x de TEXAS INSTRUMENTS. Mes contributions ont débuté en 2003 lors de mon stage étudiant dans lequel nous avons mis au point un noyau de communication DSP-FPGA et de génération de code pour FPGA [Urb03]. Cela a continué en 2004 lors de mon stage de fin d'étude ou une application multimédia complexe a été portée sur plateforme multiprocesseur [Urb04]. Les noyaux de génération de code deviennent alors de plus en plus opérationnels et les différentes plates-formes de prototypages peuvent être utilisées pleinement, avec un large éventail d'applications [RMU+05, RUN+05].

L'approche utilisée est générique dans le sens où les travaux ont été réalisés sur des plates-formes n'appartenant pas aux mêmes constructeurs. Le changement de plate-forme de prototypage ou l'ajout de nouveaux composants nécessite bien sûr le développement de nouveaux noyaux. Cette tâche est rendue plus rapide grâce aux connaissances acquises au cours du développement de ces divers noyaux [RUN+05]. Nous décrivons brièvement dans la suite les noyaux utiles pour le portage de nos applications. Une description détaillée est faite dans [RAU06].

4.1.6.1 Principe de fonctionnement

Il existe autant d'exécutifs générés que d'opérateurs dans l'architecture, chacun d'eux correspondant à un fichier distinct. Chaque fichier d'exécutif est un code intermédiaire indépendant de l'opérateur, c'est-à-dire du processeur. Il est composé d'une liste d'appels de macros qui seront traduites par un macro-processeur dans le langage source préféré de l'opérateur correspondant (C, ou assembleur par exemple). Chacun de ces programmes sources sera ensuite compilé et chargé sur les opérateur par des outils spécifiques (par exemple *Visual C++* pour PC ou *Code Composer Studio* sur DSP

Texas Instruments). Les définitions de macros qui sont dépendantes de l'opérateur (du processeur) peuvent être classées en deux ensembles. Le premier ensemble est un jeu extensible de macros applicatives réalisant les opérations de l'algorithme. Le second ensemble, que nous appelons *noyau d'exécutif*, est un jeu de de macros système qui supportent :

- le chargement initial des mémoires programmes,
- la gestion mémoire (allocation statique, copies et fenêtres glissantes de macro-registres),
- le séquencement (sauts conditionnels et itérations finies et infinies),
- les transferts de données inter-opérateurs (macro-opérations de communication transférant le contenu de macro-registres),
- les synchronisations inter-séquences (assurant l'alternance entre écritures et lectures de chaque macro-registre partagé entre la séquence de calcul et les séquences de communication),
- le chronométrage (pour permettre la mesure des caractéristiques des opérations de l'algorithme et des performances de l'implantation).

Nous verrons dans la suite qu'un noyau d'exécutif regroupe un type de processeur et les types de média connectés à ce processeur. Nous appellerons bibliothèques les définitions des macros associées à chaque type de processeur et à chaque médium de communication.

4.1.6.2 Organisation des noyaux d'exécutifs en bibliothèques

Comme évoqué précédemment, les bibliothèques SynDEx sont nécessaires pour la génération automatique de code. Elles contiennent la traduction des macros du macro-code généré par SynDEx en des fonctions compilables. Ces bibliothèques sont classées de façon hiérarchique (Fig. 4.15) afin d'être les plus génériques possible et d'être facilement réutilisables pour la définition d'une nouvelle bibliothèque.

Ces bibliothèques sont le cœur de notre chaîne de prototypage, du portage de nos applications dans le monde embarqué (architecture multi-DSP).

Noyau générique Les définitions génériques sont définies dans ce noyau. Cela concerne seulement les macros indépendantes de l'architecture. De ce fait il n'évolue que très rarement.

Bibliothèque de l'application Contient les traductions des appels des fonctions de l'application. Cette bibliothèque est simple et est écrite par l'utilisateur. Les propriétés de l'application sont également contenues dans ce fichier, telles que des options de chronométrages ou d'optimisations spécifiques.

Bibliothèques pour les processeurs Ces bibliothèques prennent en charge les opérations au niveau système. C'est à dire les allocations mémoires, la création et l'exécution des taches, ainsi qu'une couche de base pour la gestion des interruptions lors des communications. Celles qui vont nous intéresser permettent de générer l'exécutif pour :

- un PC sous *Windows,* monoprocesseur ou multiprocesseur (multitâche),

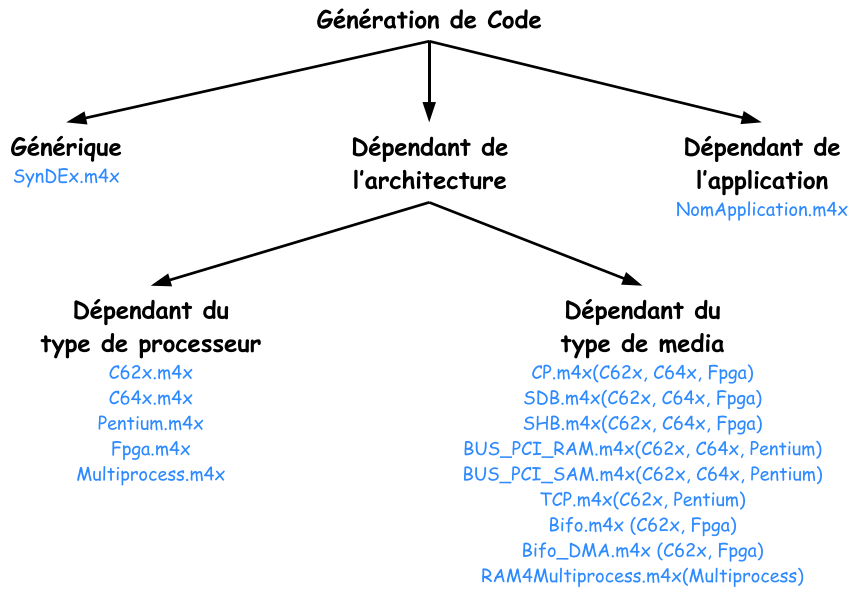


FIG. 4.15 – Arborescence des bibliothèques SynDEx

– un DSP C6x utilisant l'OS temps réel développé par Texas Instruments DSP/BIOS,

Le code est écrit en "C" pour permettre une grande généricité et un maintient (mise à jour) aisé.

Bibliothèques de communications Elles sont très importantes pour distribuer l'application. Pour avoir un intérêt, les communications doivent être rapides et garantir le transfert des données de manière sûre. Les synchronisations étant générées par SynDEx et donc s'appuyant sur une théorie, sont valides par construction (pas d'inter-blocages ou d'erreur de synchronisation). La rapidité est obtenue en utilisant les périphériques disponibles sur les plates-formes de manière optimales : le modèle (SAM ou RAM) le plus adapté est choisi, un DMA (Direct Memory Access) est utilisé si possible. L'utilisation du DMA permet de décharger l'unité de calcul, de synchroniser les média matériellement, et d'utiliser les interruptions pour détecter le fin des transferts, rendant possibles des calculs et des transferts mémoire concurrents. On se rapproche ainsi du modèle théorique utilisé par SynDEx pour calculer la latence. Les synchronisations générées (*Pre* et *Suc*) garantissent un fonctionnement correct. Les bibliothèques disponibles nous intéressant sont :

– le bus SHB (Sundance High speed Bus) et SDB (Sundance Digital Bus) (modèles SAM) , pour réaliser des communications entre DSP sur les plates-formes Sundance,
– le bus PCI (modèle RAM) pour Sundance permettant de faire communiquer un DSP avec le processeur du PC dans lequel est la carte,
– le bus Vitec (modèle RAM) dédié à la carte multi-DSP Vitec permettant d'interconnecter les DSP et le processeur de PC,

- le bus TCP (modèle SAM) permettant d'interconnecter deux PC peut interconnecter des PC mais aussi émuler une plate-forme multiprocesseur sur un seul PC en interconnectant les processus,
- le bus PIPE (modèle SAM) étant une FIFO logicielle pouvant interconnecter des processus ou des tâches,
- le bus RAM (modèle RAM) permettant d'interconnecter des tâches.

4.1.6.3 Conclusion sur la génération de code

L'utilisation de la méthodologie AAA et de SynDEx depuis quelques années a permis le développement d'un noyau temps réel pour C6x, pour FPGA et de noyaux pour les liens de communication inter-DSP et DSP-FPGA. La modélisation d'un FPGA étant différente de celle d'un DSP, celle-ci n'autorise pour le moment qu'une opération à être exécutée sur ce type de composant. Le développement des noyaux PCI et TCP permet aujourd'hui d'intégrer des PC à la plate-forme de développement. Des ressources PC (telles que l'affichage ou la lecture d'un fichier sur disque dur) peuvent donc maintenant être facilement utilisées.

Avec l'expertise acquise au sein du laboratoire, de nouvelles plates-formes sont rapidement modélisées et exploitables, comme notamment la carte multi-DSP de Vitec Multimédia lors du stage de A. MACCARI [Mac06] co-encadré par M. RAULET et moi-même. Les travaux de thèse de G. ROQUIER débutés en 2005 ont débouché sur l'utilisation d'un OS (Operating System) sur les processeurs pour autoriser le multitâche [RRND06] et simplifier les mécanismes de synchronisation. Un code distribué sur plusieurs processeurs peut alors être regroupé sur une même machine en un code multitâche.

Les noyaux de génération automatique de code concernant les plates-formes Sundance et Pentek et Vitec sont maintenant tous disponibles. Ceci permet d'utiliser toutes les ressources matérielles disponibles (plusieurs PC, DSP, FPGA) en ne touchant qu'au code C ou VHDL des fonctions de l'application. Autrement dit, les synchronisations et les transferts de données nécessaires sont gérés automatiquement. De plus les différentes fonctions sont ordonnancées automatiquement par SynDEx de manière optimisée sur les différents composants de l'architecture, afin d'utiliser au mieux le matériel disponible. l'utilisateur peut donc se concentrer sur l'application (débogage, optimisation).

La fonctionnalité croissante de ces différentes bibliothèques et la maturité le la méthodologie AAA/SynDEx permet de définir une méthode pour le développement d'applications multimédia distribuées et leur portage sur plate-forme multiprocesseur de façon progressive.

4.1.7 Méthode de développement

L'automatisation du portage d'une application distribuée sur une plate-forme multiprocesseur a permis de définir une méthode de développement allant de la vérification fonctionnelle jusqu'à l'application distribuée finale. Au fur et à mesure de la maturité des outils et notamment des bibliothèques de génération de code, cette méthode a pu évoluer. Dans les premières phases, des outils comme AVS ou Ptolemy étaient utilisés en amont de AAA/SynDEx pour faire de la vérification fonctionnelle [FDN02].

Il était alors nécessaire de réaliser des traductions pour ensuite utiliser SynDEx. Par la suite la chaîne de prototypage a été simplifiée en intégrant dans SynDEx des outils de vérification fonctionnelle, évitant ainsi des traductions, sources d'erreurs et en réduisant le nombre d'outils (à maîtriser et à acheter) [RMU+05]. Pour ce faire, la technique est relativement simple : ajouter des fonctions permettant de vérifier le bon fonctionnement de l'algorithme (lecture/écriture de fichier, comparaison, affichage,...). Dans le graphe d'algorithme de SynDEx, cela se traduit par l'ajout de nouvelles opérations qui sont connectées aux données à vérifier. Ensuite, peu importe l'architecture utilisée et la distribution des opérations, les données sont automatiquement transférés vers un processeur (de l'architecture cible) capable d'exécuter l'opération de vérification (par exemple un PC pour l'affichage et accès disque). Afin de tenir compte des spécificités du traitement des images d'un point de vue entrée/sortie, des fonctions d'acquisition d'images (par *webcam* ou par lecture de fichier), et d'affichage (une fenêtre associée à chaque appel de fonction de visualisation) sont disponibles, sous environnement standard de développement (*Visual C++*, dev-cpp), sur PC [RAU06]. L'utilisateur est ainsi en mesure d'appeler directement des opérations spécifiques de visualisation dans son graphe d'algorithme. Ces visualisations peuvent être utilisées en association avec les outils de débogage classiques sur PC et sur DSP pour valider fonctionnellement les développements.

Dès le début de ma thèse, la maturité des bibliothèques de communications nous a permis d'utiliser la vérification fonctionnelle dans AAA / SynDEx de manière fiable. Nous avons pu nous placer dans une optique d'utilisateur de la méthodologie et utiliser l'outil de manière intensive. La méthode pour chaque étape de développement a donc pu évoluer avec une sécurité de conception accrue. Dans la suite de ce paragraphe nous présentons les trois étapes de développement (fig. 4.16) : la vérification fonctionnelle, le portage et l'optimisation sur cible monoprocesseur, et le portage sur cible multiprocesseur.

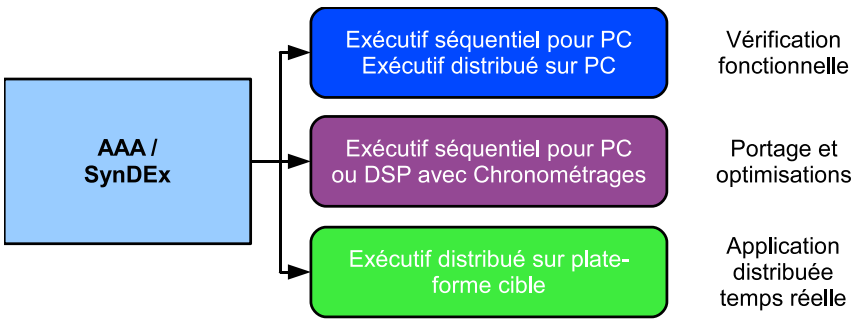


FIG. 4.16 – Etapes de développement

4.1.7.1 Vérification fonctionnelle

Elle permet de valider les algorithmes et la distribution des opérations en quatre sous-étapes.

Modélisation de l'application L'application est décrite sous forme d'un graphe flot de données (GFD), avec une approche descendante. L'algorithme est donc ini-

tialement constitué de macro-opérations. Cela permet d'avoir rapidement un GFD opérationnel pour débuter l'étape suivante du processus. Ensuite les opérations sont raffinées de telle sorte de pouvoir faire apparaître du parallélisme potentiel (i.e. des opérations sans inter-dépendances de données, parallélisables). Il convient de décrire l'application à un grain suffisamment fin pour obtenir un niveau de parallélisme potentiel acceptable.

Vérification fonctionnelle mono-processeur La génération automatique de code fournit un programme en C pour une première implantation sur PC (mono-processeur). Ainsi, l'utilisateur peut créer les fonctions en langage C associées à chacune des opérations du GFD et tester les fonctionnalités de son application avec un environnement de compilation standard. Nous utilisons des primitives de visualisation, permettant d'accélérer la mise au point des algorithmes d'image. Lorsque le fonctionnement de l'application est validé, l'algorithme mono-processeur va servir de référence pour les étapes suivantes.

Exploration architecturale SynDEx permet, avant l'acquisition d'une plate-forme de prototypage, de faire de l'exploration architecturale, c'est-à-dire de dimensionner au mieux le nombre d'opérateurs (processeurs) nécessaires pour le portage de l'application. Il est alors nécessaire d'estimer les performances de l'architecture en terme de temps d'exécution de chaque opération sur chaque type de processeur, et en terme de vitesse de transfert sur les média de communication. Cela permet de vérifier le parallélisme potentiel de l'application et les performances attendues en fonction d'une architecture cible. La figure 4.17 décrit le portage d'une application d'estimation de mouvement sur une plate forme composée de cinq processeurs connectés par une mémoire distribuée. Les fonctions d'entrées-sorties sont exécutées sur le premier processeur, et l'application est distribuée sur les quatre autres processeurs. La description de l'application est hiérarchique, seule la vue du dessus est visible, pour ne pas surcharger la figure. Il est alors possible d'ajouter ou de retirer des processeurs, d'estimer les performances et le nombre de processeurs requis.

Vérification fonctionnelle multiprocesseur Grâce aux bibliothèques SynDEx du processeur PC et des communication TCP, PIPE, et RAM, la distribution de l'application peut être validée sur une plate-forme virtuelle composée de PC multitâche (un seul exécutable, un seul espace d'adressage), de PC multiprocessus (plusieurs exécutables, plusieurs espaces d'adressages) ou de plusieurs PC. Cette étape est nécessaire pour identifier le plus tôt possible des erreurs dues à la parallélisation de l'application (notamment concernant l'accès aux données), et les corriger sur une plate-forme maîtrisée par les développeurs (un PC). Il est alors possible de contrôler le résultat de chaque opération pour vérifier le bon fonctionnement de l'application distribuée. Pour ce faire, le graphe d'algorithme est simplement dupliqué, une instance est exécutée sur mono-processeur séquentiellement (instance référence), et une autre sur la plate-forme virtuelle distribuée, en insérant des contraintes de placement. Des opérations, ayant le rôle de sondes sont connectées entre les deux instances sur certaines données pour vérifier leur cohérence (par exemple : comparaison bit à bit entre les deux jeux de données, les résultats sont fournis sous forme de traces). L'ou-

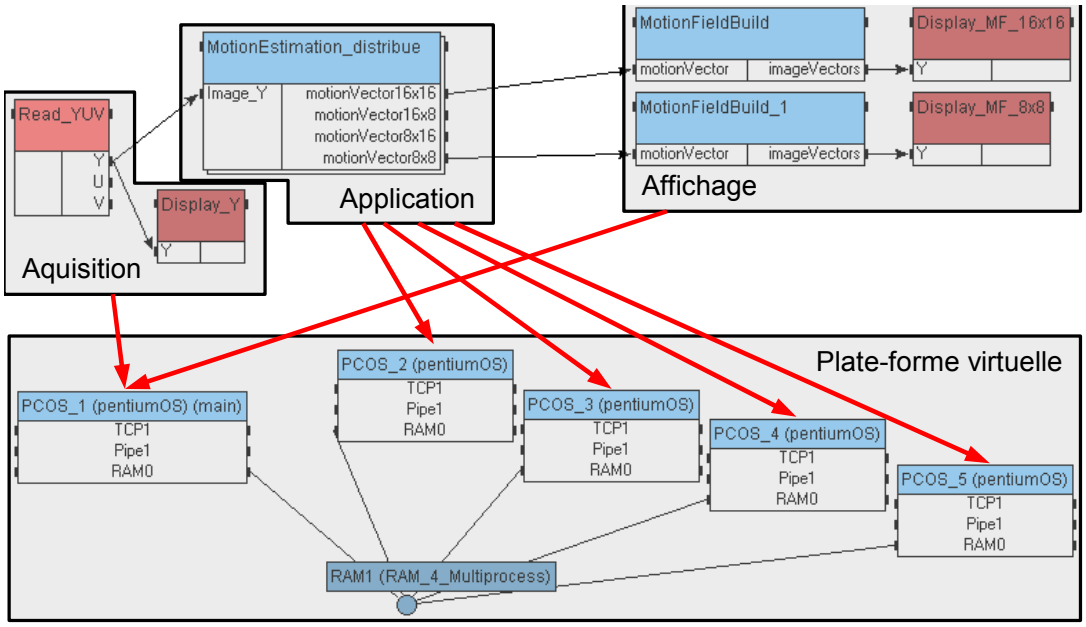


FIG. 4.17 – Exploration architecturale sur plate-forme de type Vitec

til s'assure de l'acheminement des données à travers la plate-forme. La figure 4.18 présente la vérification fonctionnelle de l'application distribuée de la figure 4.17. L'opération d'estimation de mouvement est dupliquée et l'instance de référence est exécutée séquentiellement sur un processeur. Des fonctions de test (comparaison des vecteurs entre les deux instances) permettent de vérifier la cohérence des données en affichant par exemple les caractéristiques des vecteurs qui différent afin d'identifier la source d'erreur. Cette technique de validation est ensuite utilisée tout au long du processus de développement.

4.1.7.2 Portage et optimisation monoprocesseur

Le GFD est ensuite directement utilisé pour porter progressivement les opérations sur processeur cible. L'acheminement des données est toujours géré par l'outil, ce qui permet à l'utilisateur de se concentrer sur le portage et l'optimisation monoprocesseur (optimisation de l'écriture du programme, utilisation du langage assembleur ou de bibliothèques optimisées livrées avec le composant), opération par opération. Pour vérifier que le changement de processeur et les optimisations réalisées n'introduisent pas d'erreur, la technique précédemment utilisée permet de valider les travaux.

Pour mesurer les performances en terme de rapidité, le code généré insère automatiquement les fonctions de chronométrage. L'utilisateur doit juste compiler les sources et lancer l'exécution du programme avec l'outil adéquat à la cible. Les durées de chacune des fonctions (opérations du GFD) sont alors affichées en fin d'exécution. Cette étape doit être réalisée pour chacun des processeurs (PC ou DSP) de la cible finale.

Remarque : Dans certains cas, cette étape peut être conduite avec l'exploration architecturale, lorsqu'il est nécessaire de réaliser des chronométrages afin d'évaluer

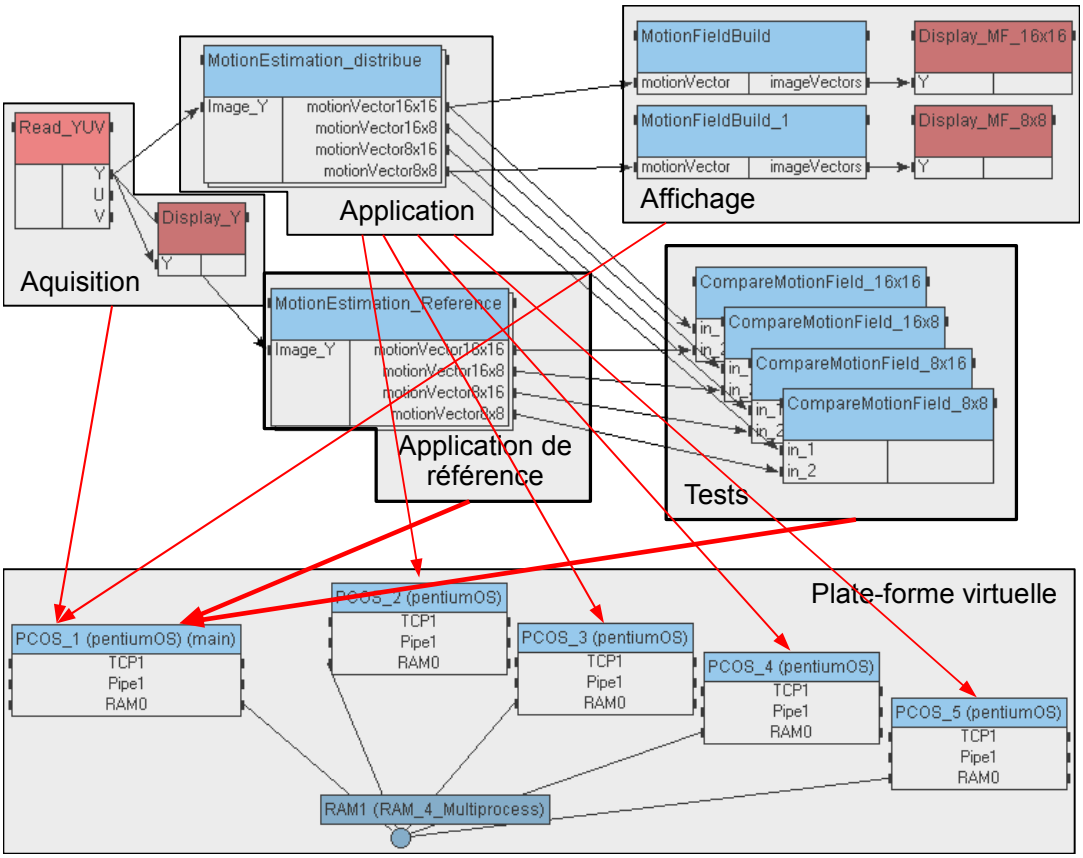


FIG. 4.18 – Vérification fonctionnelle multiprocesseur

précisément les performances. Un premier portage avec les optimisations qui s'imposent sont donc nécessaires.

4.1.7.3 Application distribuée temps réel

SynDEx réalise l'adéquation sur l'architecture parallèle et génère un exécutif distribué temps réel. L'exécutif est optimisé en fonction de la plate-forme multiprocesseur. L'utilisateur peut contraindre l'adéquation en forçant par exemple les entrées/sorties sur un processeur. Plusieurs configurations architecturales peuvent être simulées en faisant varier le nombre et le type des processeurs ou connexions.

L'avantage principal de cette chaîne de prototypage réside dans sa simplicité. La plupart des tâches réalisées par l'utilisateur concernent la spécification de l'application. L'utilisation de SynDEx et des compilateurs est limitée à des opérations simples. Une grande partie des tâches complexes (adéquation, synchronisations, transferts de données et chronométrages) sont effectuées automatiquement.

4.1.8 Synthèse sur la méthodologie

La méthodologie AAA/SynDEx, associée aux différentes étapes de la chaîne de prototypage, permet d'aller de la vérification fonctionnelle à l'implantation temps réelle sur

plate-forme distribuée. Les phases successives de vérification et le portage ainsi que l'optimisation progressifs des opérations offre une sécurité de conception. La distribution et l'ordonnancement de l'algorithme sont réalisés de manière optimisés, visant à minimiser la latence globale. Les transferts de données et synchronisations sont générés de manière automatique et fiable. Les opérations de portage, de vérification et de parallélisation sont accélérées grâce à l'utilisation de bibliothèques de génération de code. La généricité de l'approche permet de prendre en charge de nombreuses plates-formes.

Cet outil universitaire possède néanmoins des limitations qu'il a fallu combler pour aboutir aux résultats escomptés. Le paragraphe suivant propose des solutions apportées à certaines limitations.

4.2 Limites de la méthodologie existante et enrichissement des outils

L'utilisation de cet outil pour des applications de traitement d'images complexes permet de mettre en évidence certaines limitations d'une part, et d'autre part la flexibilité de l'outil permet de contourner les limitations et de faire évoluer la méthodologie. Dans cette partie, des nouveaux besoins en termes de prototypage, identifiés par l'implantation d'estimateurs de mouvement, sont spécifiés et des solutions sont proposées afin de les intégrer dans la méthodologie.

4.2.1 Branchement dans une application - Plugin

Ces travaux d'étude d'estimateurs de mouvement sont réalisés dans le cadre de la compression vidéo. Il est donc intéressant d'intégrer les développements dans un encodeur vidéo, non seulement pour donner une finalité aux travaux, mais surtout afin de comparer les performances des différents algorithmes en terme de qualité des champs de vecteurs pour la compression vidéo, en utilisant comme métrique la performance de compression.

Pour conserver une cohérence dans nos développements, et afin d'unifier notre méthode, nous voulons utiliser AAA/SynDEx pour intégrer l'estimateur de mouvement dans l'encodeur H.264 développé par THOMSON. Cependant, la technique utilisée habituellement ne peut pas être appliquée ici, nous utilisons donc une technique de “plugin”.

Nous utilisons l'encodeur vidéo H.264 développé à THOMSON pour intégrer nos travaux. C'est un kit de développement logiciel (SDK) développé en langage objet C++. La technique usuelle avec AAA/SynDEx serait de décrire le SDK sous la forme d'un graphe flot de données, avec une granularité suffisante pour faire apparaître une ou plusieurs opérations *“estimateur de mouvement”* afin d'appréhender les flots de données entre l'estimateur de mouvement et le reste de l'encodeur. Nos développements s'y substitueraient alors. Cependant, certaines caractéristiques rendent le SDK incompatible avec une description SynDEx :

- nous ne voulons pas perdre de temps à décrire une application trop complexe, notre but n'étant pas de travailler sur l'encodeur dans son ensemble,
- le nombre de personnes impliquées dans le développement du SDK et la maturité du projet ne permettent pas d'envisager une migration vers de nouveaux outils,

– la structure des données complexe (objets contenant des pointeurs vers d'autres objets) permet un accès facile à de nombreuses données, mais cache les dépendances entre les opérations.

Dans notre contexte de travail, nous n'avons pas intérêt à décrire l'encodeur vidéo sous AAA/SynDEx. Cela entraînerait un travail fastidieux pour développer et maintenir l'application. Il n'est donc pas envisageable de décrire le SDK sous SynDEx. Le cas dans lequel nous nous trouvons n'est pas isolé et une solution à notre problème serait réutilisée, par exemple pour le développement d'un décodeur vidéo dans un lecteur multimédia déjà existant.

La technique envisagée, décrite ci-dessous est basée sur la description d'un plugin.

La solution proposée est d'encapsuler la description SynDEx dans une nouvelle fonction d'estimateur de mouvement afin de la rendre transparente. Du point de vue du SDK, rien ne change, l'appel à une fonction du type *"ME_init()"* permet d'initialiser l'application, *"ME_run()"* réalise l'estimation de mouvement, et *"ME_end"* libère la mémoire et termine l'application. Du point de vue SynDEx, le graphe flot de donnée fait apparaître les opérations d'estimation de mouvement et des fonctions d'entrées sorties d'interfaçage. Un noyau SynDEx spécifique pour générer le code du plugin a été développé. Inspiré du noyau "pentium", il permet, lorsque le processeur cible est du type *"SDK"*, de générer le code des fonctions ME_init(), ME_run() et ME_end().

Le graphe d'algorithme contient des opérations d'interfaçage. Ce sont des opérations de lecture d'image, d'écriture des champs de vecteurs et d'accès aux différents paramètres qui ne peuvent être exécutées que sur le processeur du type SDK et qui accèdent aux données de l'encodeur. Il est alors possible d'exécuter l'estimation de mouvement sur le même processeur que le reste de l'encodeur, ou de le porter sur plate-forme multiprocesseur, SynDEx gérant les transferts de données et synchronisations(Fig. 4.19). Cette technique a l'avantage d'être générique et réutilisable pour d'autres encodeurs vidéo, tel que le logiciel de référence MPEG-4 (le JM : Joint Model), ou un encodeur SVC (Scalable Video Coding).

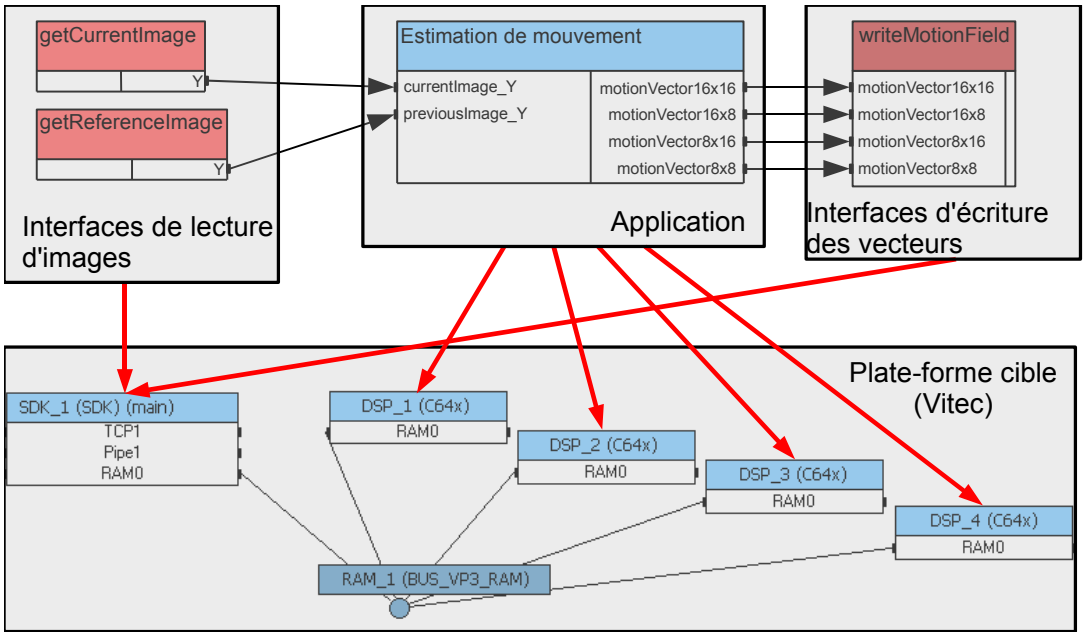


FIG. 4.19 – Utilisation en mode “plugin”

L’encodeur vidéo appelle les fonctions d’estimation de mouvement normalement, sans aucune modification notable. L’application réalise l’estimation de mouvement, les fonctions d’accès aux données sont exécutées sur un processeur particulier (SDK) qui exécute le reste de la compression de manière transparente.

SynDEx permet grâce à ce processus, le portage automatique d’algorithmes d’estimation de mouvement sur des plate-formes multiprocesseurs composées de PC, de DSP et de FPGA. Bien que les applications complexes développées sans l’outil SynDEx sont généralement incompatibles avec celui-ci, il est néanmoins possible d’encapsuler une partie de l’algorithme pour en faire une étude partielle. Par exemple, l’étude d’implantation d’estimateurs de mouvement grâce à SynDEx est possible sans avoir une description de tout l’algorithme de compression vidéo.

4.2.2 Cache automatique sur DSP

Actuellement, la méthodologie utilisée ne prend pas en compte la spécificité de l’architecture mémoire des composants embarqués. Sur un processeur de traitement du signal, il peut exister plusieurs niveaux de mémoire :

- une mémoire très réduite fonctionnant à la vitesse du processeur, contenant des données temporaires (cache de niveau 1 ou “scratchpad”),
- une mémoire interne restreinte légèrement plus lente (cache de niveau 2 ou RAM interne),
- une mémoire externe, dont l’accès est considérablement plus lent mais de grande taille (RAM externe).

Pour des applications de traitement vidéo, les mémoires internes ne sont pas assez grandes, notamment pour la haute définition. Les données sont donc allouées en mémoire externe, dont l’accès est très lent, pouvant réduire les performances d’un fac-

teur cent. Pour accélérer les traitements, un mécanisme d'accès aux données manuel doit souvent être mis en place pour rapatrier les données utiles de manière temporaire en mémoire interne. Cependant cela n'est pas générique, nécessite une bonne maîtrise de l'architecture et de l'algorithme pour mettre en place une technique efficace. De plus le développement d'un tel mécanisme peut être long. Afin de réduire le processus de portage sur DSP, nous voulons utiliser le mécanisme de cache offert par les DSP C64x de Texas Instruments. Dans un contexte d'application distribuée, nous nous heurtons très vite au problème de cohérence de cache. Cette approche est complémentaire avec une minimisation mémoire [RAU06], typiquement dans le cas du traitement d'images haute résolution où la taille d'une image est supérieure à celle des mémoires internes.

Nous présentons dans cette section le mécanisme de cache ainsi que la notion de cohérence de cache. Ensuite la solution proposée est détaillée et des résultats sont présentés pour trois applications de traitement d'image.

4.2.2.1 Le mécanisme de cache

Le mécanisme de cache permet de rapatrier automatiquement les données utiles en mémoire interne.

Architecture mémoire d'un DSP : l'architecture simplifiée d'un DSP est donnée figure 4.20. Le CPU (Central Processing Unit) est le cœur de calcul. Il accède à la mémoire interne à travers un bus rapide. Un contrôleur DMA (Direct Memory Access) gère les transferts mémoire de manière indépendante au CPU. L'interface externe permet de connecter de la mémoire et des périphériques additionnels. L'accès à cette mémoire est très lent à cause de ses spécificités techniques, mais sa taille est grande par rapport aux mémoires internes.

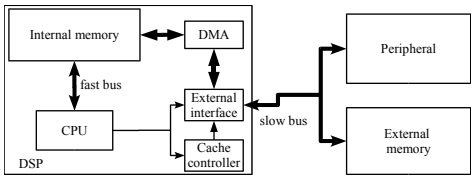


FIG. 4.20 – Architecture mémoire d'un DSP

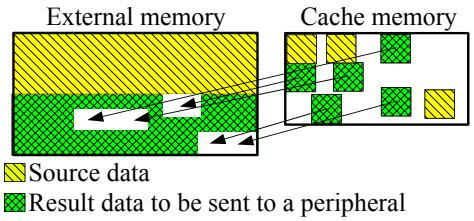
L'accès aux données peut être optimisé en utilisant de manière efficace le DMA (les données sont alors rapatriées en avance de manière efficace). Cependant cela nécessite des développements complexes et longs. L'utilisation du contrôleur de cache peut réduire le temps de développement et est mieux adapté dans un processus de prototypage rapide.

Mécanisme de cache : quelques DSP intègrent un contrôleur de cache pouvant améliorer de manière considérable les performances lors de l'accès à la mémoire externe. Tout ou partie de la mémoire interne est alors dédiée au cache (elle n'est plus adressable). Lorsque le cache est activé, le CPU n'accède plus à la mémoire externe directement. Les requêtes de données sont envoyées au contrôleur de cache qui utilise

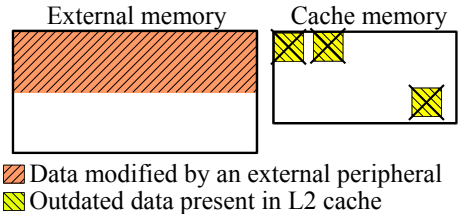
le DMA pour rapatrier les données dans le cache (mémoire interne). Le principal avantage est le caractère automatique, il n'y a donc pas besoin de modifier le programme.

Le contrôleur de cache gère tous les accès du CPU vers la mémoire externe. Il dirige les requêtes de données vers l'adresse cache correspondante, rapatrie les données de la mémoire externe si elles ne sont pas déjà en cache. Lorsque la donnée est retirée du cache, elle est soit simplement invalidée (effacée) ou réécrite dans la mémoire externe en fonction de son statut de modification. Ce mécanisme accélère l'accès à la mémoire externe : en considérant un cas optimal, une donnée n'est rapatriée qu'une seule fois de la mémoire externe et réutilisée à partir du cache, réduisant considérablement le temps de traitement.

La cohérence de cache et les applications multiprocesseurs : avec le mécanisme de cache, il coexiste deux copies d'une donnée (en mémoire externe et en cache). Lorsque l'une ou l'autre est modifiée, les données ne sont plus cohérentes. Un protocole doit être mis en place de façon à s'assurer que des données périmées ne soient pas accédées [TM93]. Le contrôleur de cache ne sait pas gérer la cohérence dans un contexte multiprocesseur. Nous montrons ici comment utiliser SynDEx et la génération automatique de code pour palier à ce manque.



(a) data modified by the CPU



(b) data modified by a peripheral

FIG. 4.21 – Gestion de la cohérence de cache

Nous ne considérons que les mémoires allouées par AAA/SynDEx (les données impliquant une dépendance sue sur le GFD de l'application). Les autres données sont le programme ou contenues dans la pile. Ces premières sont protégées par des sémaphores générés automatiquement dans l'exécutif. Par conséquent une donnée n'est accessible seulement par le CPU ou un périphérique à la fois. Les situations d'incohérence de cache apparaissent dans deux situations identifiées.

1. **Lorsque des données en cache sont modifiées par le CPU et sont ensuite accédées par un périphérique** : la donnée de sortie de la fonction est allouée en mémoire externe cachable (fig. 4.21-a). Le cache est utilisée pour stocker temporairement les résultats et les données source. A la fin du calcul, tous

les résultats n'ont pas encore été écrits dans la mémoire externe. Par conséquent, elle doit être mise à jour avant d'être accédée par un périphérique.

2. **Lorsque des données cachable sont modifiées par un périphérique** : un périphérique a mis à jour la mémoire externe (Fig. 4.21-b). Les adresses correspondantes doivent être invalidées pour empêcher le CPU de les utiliser.

La configuration et la gestion de la mémoire sont généralement faites en utilisant des libraires fournies par le fabriquant de DSP. La technique usuelle de gestion du cache est l'utilisation des fonctions *"WriteBack()"* et *"Invalidate()"* pour les configurations 1 et 2 respectivement. Avant d'être accessible par un périphérique, la cohérence de cache doit être appliquée pour éviter des conflits de données. Dans les applications multiprocesseurs développées avec AAA, les opérations de niveau système sont automatiquement générées. Il est également possible de laisser l'outil gérer la cohérence de cache.

4.2.2.2 Gestion automatique de la cohérence de cache avec AAA

L'outil de prototypage réalise les allocations mémoire, l'ordonnancement, les communications inter-processeur et les synchronisations. La connaissance de ce contexte rend possible une gestion automatique de la cohérence de cache pour maintenir un fonctionnement correct de l'application.

La génération des exécutifs synchronisé Un exemple de génération de code fourni par SynDEx est redonné sous la forme d'un réseau de pétri figure 4.22.

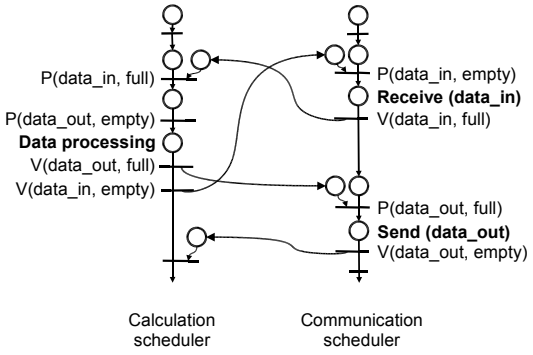


FIG. 4.22 – Synchronisation de l'exécutif

Pour rappel, les opérations P() et V() correspondent à une attente et une libération de sémaphore respectivement. Les emplacements mémoire concernés par un potentiel problème de cohérence de cache sont en mémoire externe. Si une adresse est accédée par le CPU et le DMA, alors la cohérence de cache doit être assurée car le DMA contourne le contrôleur de cache. Dans nos bibliothèques de code, tous les transferts inter-processeurs sur DSP sont optimisés et réalisés par le DMA. La cohérence de cache doit donc avoir lieu lors du déblocage du séquenceur de communication par le séquenceur de calculs. C'est à dire lorsqu'une opération V() est présente dans celui-ci (Fig. 4.22). (un V(mem, empty) est associé à un *"Invalidate"* et un V(mem, full) à un *"WriteBack"*).

Mise en oeuvre Pour garder un fonctionnement simple et rapide, l'appel de macros simples suffit à activer la mémoire cache et à assurer la génération des opérations de gestion de la cohérence des données. L'utilisateur peut utiliser le cache de manière sûre pour accélérer les accès à la mémoire externe des DSP de manière transparente.

DSP Texas Instruments C64x Les DSP C64x intègre un contrôleur de cache [Tex6a]. Nous avons modifié le noyau SynDEx de génération de code correspondant pour prendre en charge automatiquement la configuration et la gestion de la cohérence de cache. Le paragraphe suivant présente des tests concluants réalisés sur différentes applications distribuées de traitement d'images.

4.2.2.3 Validation de la gestion automatique de cohérence de cache

La gestion de cache automatique a été testée et validée sur quelques applications impliquant un PC et un DSP C6416 à 1GHz. Ces applications ont été développées en utilisant la chaîne de prototypage décrite précédemment. Le code généré automatiquement pour le DSP inclue l'activation et la gestion du cache. Les temps d'exécution sont présentés pour la méthode de compression LAR [RBN+03] et un décodeur vidéo MPEG-4 part 2 développés à l'IETR, et une partie de l'estimateur de mouvement qui sera décrit précisément dans la suite de ce mémoire.

Le code C pour PC a été directement compilé pour DSP, donc c'est une application très peu optimisée. Les temps correspondants à la compression-décompression sont donnés dans le tableau 4.1 dans différents contextes. Pour une image CIF (352x288) toutes les données peuvent être allouées en mémoire interne, donnant ici les performances optimales. Pour des images de taille plus grande, l'utilisation de la mémoire externe est nécessaire. L'utilisation du cache accélère les traitements d'un facteur dix par rapport à la mémoire externe seule, et augmente de 16% les temps de traitements par rapport à un cas optimal.

Localisation des données	SD image	CIF image
mémoire externe	800 ms	310 ms
mémoire interne	doesn't fit	26 ms
mémoire externe + gestion de cache automatique	80 ms	31 ms

TAB. 4.1 – Chronométrages du codec LAR

L'application d'estimation de mouvement hiérarchique pour la compression vidéo HD (1280x720) a contribué à valider la technique. L'encodeur est exécuté sur PC et l'estimation de mouvement sur DSP. Les résultats de chronométrage sur DSP sont fournis dans le tableau 4.2 pour une image. Différentes configurations sont testées : le code ne contient que les optimisations du compilateur et les données sont en mémoire externe (1), puis le cache est activée (2), et le code optimisé (optimisations décrites dans la suite) (3), et enfin les données sont rapatriées en mémoire interne manuellement (cache désactivée) (4). L'utilisation du cache donne un temps d'exécution 24 fois plus court. L'accès mémoire optimisé manuellement évite les défauts de cache, et est 22% plus rapide qu'avec le cache, cependant cette technique n'est pas automatique. Elle est développée spécifiquement pour chaque opération et chaque processeur. Elle nécessite donc beaucoup de temps de développement.

Localisation des données	Chronométrages
(1) mémoire externe	5350 ms
(2) mémoire externe + gestion de cache automatique	221 ms
(3) 2 + code optimisé pour DSP	100 ms
(4) 3 + accès DMA manuels	78 ms

TAB. 4.2 – Chronométrage de l'estimation de mouvement

Enfin les résultats pour l'application de décompression vidéo MPEG-4 part2 implantée à l'IETR. Les temps de décodage moyens par image est donné dans le tableau 4.3 pour une séquence CIF (352x288). A cette résolution il est possible d'allouer toutes les données en mémoire interne pour obtenir une temps d'exécution. Le cache réduit considérablement les temps de traitement et induit une perte d'efficacité de 23% par rapport à la mémoire interne seule.

Localisation des données	I frame	P-B frame
mémoire interne	4.3 ms	5.5 ms
mémoire externe sans cache	22 ms	50 ms
mémoire externe + gestion de cache automatique	4.5 ms	7 ms

TAB. 4.3 – Chronométrage du décodeur MPEG-4

La mémoire cache peut être utilisée dans la chaîne de prototypage, et la cohérence des données est gérée automatiquement. Les performances obtenues en terme de temps d'exécution sont améliorées, sans complexité supplémentaire dans le développement. Des modifications dans la plate-forme, la distribution ou l'ordonnancement n'impliquent pas de nouveau développement. Le caractère automatique réduit considérablement les risques d'erreurs, surtout lorsque l'application grandit, avec le nombre de processeurs et de données transférées. Il n'est alors plus possible de faire des modifications sans risque d'erreur. De plus la détection et la correction de ces erreurs sont rendues plus compliquées dans un contexte multiprocesseur. Enfin, l'utilisation du contrôleur de cache rend l'approche générique dans le sens ou cela est compatible avec d'autres processeurs.

Ce travail a été réalisé au niveau des noyaux de génération de code, pour davantage de généricité, il serait intéressant de prendre en compte la spécificité des mémoires dans l'adéquation. C'est à dire pouvoir décrire plus précisément les processeurs (modèle de mémoire interne, scratchpad, mémoire externe cachable ou non) et des contraintes sur des données. Un ordonnancement des opérations en tenant compte de l'optimisation des accès à la mémoire externe avec l'utilisation du cache serait intéressant.

4.3 Conclusion

La méthodologie AAA s'appuie sur des modèles de graphes. L'algorithme est décrit avec un modèle flot de données, ce qui est bien adapté aux applications de traitement d'images contenant peu de contrôle, comme c'est la cas pour l'estimation de mouvement. Le graphe d'architecture permet de définir les processeurs et les liens

de communication, ainsi que d'éventuels périphériques. L'outil SynDEx aide à obtenir une implantation distribuée optimisée quasi-automatiquement. La distribution et l'ordonnancement des opérations sur les différents composants d'une architecture sont réalisés par l'outil. Le fonctionnement de l'application distribuée est correct par construction, grâce à l'analyse des dépendances et à l'insertion de synchronisations.

Le code des processeurs, prenant en compte l'ordonnancement et les communications inter-processeurs, est généré automatiquement. Le portage sur cible multicomposant est donc accéléré. Depuis 2001, les travaux de l'IETR portent sur la génération de code, avec le développement de noyaux couvrant un ensemble de plates-formes de plus en plus large. Ma contribution à ces noyaux, depuis mes stages étudiants, nous a permis d'étendre l'ensemble des composants supportés par l'outil et d'améliorer la stabilité du processus de portage. A titre personnel, cela m'a également permis d'acquérir une expertise dans l'utilisation de plates-formes multicomposants et dans la méthodologie de conception AAA/SynDEx.

Nous avons adapté la méthode de développement associée à AAA pour prendre en charge les étapes du processus de développement. Ainsi, en intégrant des étapes de vérification fonctionnelle tout au long du cycle de développement, nous assurons une sécurité de conception.

Les outils utilisés sont en constante évolution et ne couvrent pas la totalité des besoins à l'heure actuelle. Les limitations doivent être identifiées afin de faire évoluer l'outil. Ainsi, dans le cadre de l'implantation d'estimateurs de mouvement dans une application de compression vidéo, nous avons rendu possible l'intégration d'un sous-ensemble d'opérations dans une application plus conséquente. Cela évite la migration d'outils pour la totalité des développeurs d'une application complexe. De plus, la méthodologie est de fait étendue à la modélisation de modules pouvant s'intégrer à des applications, comme par exemple un module de décodage vidéo dans un lecteur générique.

Nous avons également étendu la méthodologie à l'implantation efficace d'applications de traitement d'images haute définition sur DSP, grâce à l'utilisation du cache. Sa configuration et la gestion de la cohérence dans un contexte multiprocesseur sont gérés automatiquement. Les accès mémoire sont donc automatiquement optimisés, ce qui accélére le portage sur DSP et évite des erreurs de cohérence de cache. Un utilisateur ayant des connaissances limitées sur l'utilisation d'un DSP peut donc obtenir une implantation optimisée.

La chaîne de prototypage présentée dans ce chapitre a été utilisée dans la suite pour le prototypage d'applications d'estimation de mouvement. L'implantation de divers algorithmes sur différentes plates-formes a ainsi pu être étudiée.

Chapitre 5

Etude algorithmique et implantation sur DSP

La méthodologie de prototypage permet d'une part la simplification de la mise au point des algorithmes d'estimation de mouvement, et d'autre part, la réduction du temps de développement de l'implantation sur DSP. L'étape suivante, qui consiste à évaluer les performances et optimiser l'implantation, peut donc être rapidement entamée. Les étapes de vérification fonctionnelles quasi automatiques valident les implantations et les optimisation effectuées.

La modélisation de l'algorithme sous forme d'un Graphe Flot de Données (GFD) est une étape à ne pas négliger. Les performances globales de l'implantation en dépendent à travers la flexibilité et l'expression du parallélisme potentiel. En effet, la démarche entre dans un cadre le plus générique possible afin d'obtenir un produit évolutif ayant un large champ d'applications. En outre, la parallélisation des opérations, nécessaire à l'amélioration des performances, est extraite du GFD. Dans le but de comparer différentes techniques d'estimation de mouvement et de réutiliser les développements déjà effectués, le GFD développé est compatible avec de nombreux algorithmes.

Les estimateurs de mouvement étudiés on été développés en langage C et mis au point sur PC. L'évaluation des différentes techniques existantes fait apparaître leurs avantages et défauts respectifs. Nous verrons ensuite les caractéristiques de l'algorithme développé, tirant parti des avantages des techniques précédemment étudiées.

La génération automatique de code permet ensuite de réaliser rapidement des implantations sur cible. Les performances ont donc été évaluées et les goulots d'étranglements identifiés sur DSP. Nous verrons comment des optimisations spécifiques ont été réalisées dans le but d'exploiter les composants de manière efficace : ordonnancement, utilisation d'instructions spécifiques, accès mémoire, etc. La méthode de développement liée au contexte méthodologique nous a conduit à porter et optimiser les opérations progressivement, en validant les travaux à chaque étape. Tout au long des développements, la comparaison des résultats de l'implantation DSP optimisée avec le modèle PC assure la vérification fonctionnelle.

Ce chapitre se compose de quatre parties : la modélisation GFD des algorithmes, le développement et l'évaluation des algorithmes, l'implantation et l'optimisation sur

DSP, enfin une discussion sur l'implantation et l'optimisation des outils spécifiques à l'estimation de mouvement pour la compression vidéo.

5.1 Modélisation flot de données des algorithmes

La première étape du prototypage est la description du graphe flot de données. Nous allons décrire les fonctions de base de manière le plus générique possible de façon à avoir des briques de base évolutives et paramétrables pour viser plusieurs solutions possibles. Cette description a pour but d'une part d'évaluer la qualité des champs de vecteurs des différents estimateurs, et d'autre part de réaliser des portages optimisés sur cible DSP avec des chronométrages afin d'évaluer la complexité des opérations.

5.1.1 Opération d'estimation d'un champ de vecteurs

L'opération d'estimation d'un champ de vecteur est l'opération de base. Il convient d'apporter un soin particulier à sa modélisation afin de la rendre à la fois flexible et parallélisable, sans faire pour autant dès maintenant un choix d'implantation.

5.1.1.1 Rappels sur les estimateurs de mouvements

Des estimateurs de mouvement pour la compression vidéo ont été présentés au chapitre 2. Nous choisissons d'implanter des techniques prédictives pour leur performances, aussi bien en termes de qualité qu'en termes de charge de calcul. Les algorithmes HME (Hierarchical Motion Estimator) ainsi que EPZS (Enhanced Predictive Zonal Search) [Tou02] sont étudiés. HME a été développé à Thomson, c'est une technique performante, utilisée dans les encodeurs H.264. Le code source est écrit en C++ et ne se prête pas directement à une description flot de donnée. Il a donc été réécrit de manière simple de façon à être optimisé sur DSP facilement. EPZS est un estimateur de mouvement très populaire, implanté dans l'encodeur MPEG-4 AVC de référence JM (Joint Model) [STS06] et dans plusieurs logiciels tels que "XViD" ou "X.264". De même que pour le HME, le code source a été réécrit pour DSP en s'inspirant des développements existants.

5.1.1.2 Opération de base d'estimation d'un champ de vecteurs

L'opération d'estimation de mouvement pour HME a un niveau de résolution donné ressemble beaucoup à EPZS. On retrouve une phase de prédiction et une phase de raffinement. Nous avons cherché à avoir une description flot de données uniforme, le choix de l'algorithme utilisé est réalisé par l'intermédiaire de paramètres, en C, et ne concerne que la phase de raffinement. Cette opération est le coeur de calcul, c'est à dire l'opération qui demande le plus de temps de calcul. Elle doit donc être parallélisable. Pour cela, le nombre de lignes de blocs à traiter est entré en paramètre, ainsi que la ligne de départ. Il est alors possible de partitionner l'image en bandes et de paralléliser les traitements. La dernière ligne de résultats de la bande supérieure peut être fournie à la bande courante pour la prédiction. Cela crée cependant une dépendance de données qui rend les traitements séquentiels.

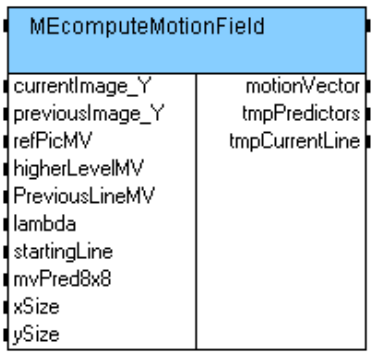


FIG. 5.1 – Opération d'estimation d'un champ de vecteurs

La figure 5.1 présente l'opération d'estimation de mouvement atomique. Ses entrées et sorties sont définies dans le tableau 5.1.

Cette opération d'estimation de mouvement est flexible dans le sens ou elle peut être utilisée pour n'importe quelle taille de bloc, pour EPZS ou HME, et elle peut être parallélisée. Connectée aux opérations d'entrées et sorties, cette opération de base permet d'obtenir les algorithmes souhaités.

5.1.2 Opérations d'entrée sortie

Nous décrivons dans ce paragraphe les opérations d'entrées/sorties développées autour de la fonction d'estimation de mouvement.

5.1.2.1 Elargissement ("padding") de l'image de référence

Dans la norme MPEG-4, les vecteurs de mouvement ne sont pas contraints aux limites de l'image, ils peuvent pointer à l'extérieur d'une image de référence. Les images sont donc élargies en recopiant les pixels du bord de l'image (figure 5.2).

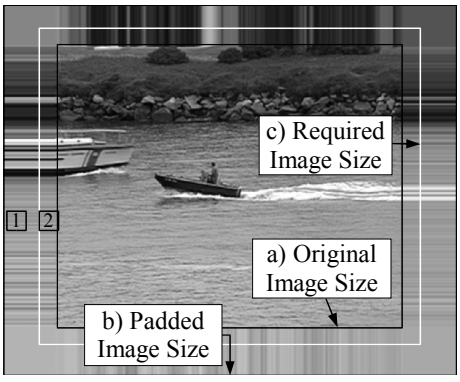


FIG. 5.2 – Elargissement de l'image de référence

Pour ne pas avoir à calculer le bloc de référence à chaque test de vecteur, il est nécessaire d'élargir l'image de référence en mémoire (figure 5.2-b). Comme l'amplitude

Paramètres constants
Largeur de bloc, hauteur de bloc
Largeur de l'image
Nombre de lignes de blocs : Permet de diviser les traitements en plusieurs opérations séquentielles ou parallèles
Entrées
currentImage_Y : Image courante
previousImage_Y : Image de référence élargie (padding) pour ne pas avoir à traiter les effets de bord
refPic_MV : Champ de vecteurs. Prédicteurs temporels (optionnel, "NULL" si non utilisé)
HigherLevelMV : Champ de vecteurs. Prédicteurs hiérarchiques (pour HME, "NULL" pour EPZS)
PreviousLineMV : Vecteurs de la ligne de blocs précédente. Crée une continuité dans le champ de vecteurs (prédiction) (optionnel, "NULL" si non utilisé)
lambda : Multiplicateur de Lagrange
startingLine : Ligne de blocs de départ
mvPred8x8 : Champ de vecteur prédicteur 8x8. Utilisé pour les tailles de blocs autres que 8x8 (optionnel, "NULL" si non utilisé)
xSize : Largeur d'image
ySize : Hauteur d'image
Sorties
motionVector : Champ de vecteurs. résultat
tmpPredictors : Prédicteurs. Utilisé pour allouer une mémoires temporaire
tmpCurrentLine : Ligne de bloc de l'image courante. Utilisé pour allouer une mémoires temporaire

TAB. 5.1 – Description des entrées - sorties

maximale des vecteurs n'est pas restreinte dans la norme H.264, la taille mémoire nécessaire serait trop élevée pour élargir l'image suffisamment. Il est donc nécessaire de mettre en place un mécanisme permettant d'extrapoler les valeurs des pixels à l'extérieur de l'image sans induire un coût de calcul élevé. Pour économiser de l'espace mémoire, l'image est donc élargie seulement de la taille d'un bloc (figure 5.2-c). En effet, comme le montre la figure 5.2, pour un bloc loin à l'extérieur de l'image de référence (1) on peut trouver exactement le même (2) dans l'image 5.2-c. Avec cette technique, bien que l'élargissement soit limité, les vecteurs ne sont pas bornés. C'est à dire que lorsque le vecteur à tester pointe trop loin à l'extérieur de l'image (1), un calcul simple permet d'utiliser un bloc équivalent (2) pour la mise en correspondance tout en gardant la valeur du vecteur. Le nombre d'instructions nécessaires à ce calcul est le même que lorsque les vecteurs sont bornés. En effet, il suffit de borner la position dans l'image de référence. La mémoire nécessaire est donc limitée. Par exemple pour une image 1280x720, une taille de bloc de 16x16, et une amplitude bornée à 64 pixels, la mémoire de l'image des luminances n'est que de 963,5 KO au lieu de 1166 KO et l'amplitude n'est pas bornée.

L'intérêt de permettre aux vecteurs de déborder de l'image est la possibilité d'obtenir un champ de vecteur plus homogène et donc d'en réduire le coût de codage, ce qui est un avantage dans un schéma de compression vidéo.

5.1.2.2 Pyramide d'images multirésolution

L'algorithme HME nécessite de sous échantillonner les images. La pyramide des images aux différentes résolutions est obtenue pour chaque niveau en appliquant un filtre

passe-bas gaussien suivi d'un sous-échantillonage d'un facteur deux sur l'image de niveau inférieur. Une pyramide est créée pour l'image courante et l'image de référence.

5.1.2.3 Construction de l'image du champ de vecteur.

Afin de mettre au point les algorithmes d'estimation de mouvement, d'avoir un visuel permettant d'identifier rapidement un fonctionnement incorrect, une opération dont le but est de créer une image des vecteurs de mouvement a été développée. Cet outil servira dans la suite à visualiser les traitements effectués.

5.1.3 Graphes Flot de Données des estimateurs de mouvement

Nous présentons les graphes complets d'estimation de mouvement pour une taille de bloc donnée en fonction de l'algorithme retenu.

5.1.3.1 HME

La figure 5.3 montre la description hiérarchique de l'algorithme HME. Dans la vue du dessus la fonction d'estimation de mouvement est connectée à une lecture de la séquence vidéo image par image, un registre permet de conserver l'image de référence. En sortie, le champ de vecteurs résultant est transformé en image et affiché. Les traitement des niveaux hiérarchiques sont encapsulés et visibles dans la partie basse de la figure. Pour ne pas la surcharger, seulement deux niveaux hiérarchiques sont affichés, de plus, les constantes et les entrées/sorties non utilisées ne sont pas connectées.

5.1.3.2 EPZS

La description de EPZS est beaucoup plus simple puisque seulement un champ de vecteurs est calculé, directement au niveau pleine résolution. Un registre permet de sauvegarder le champ de vecteurs pour prédire les mouvement dans l'image suivante.

Les estimateurs sont décrits de façon à être évolutifs, et parallélisés aisément. Cette description permet de réaliser la vérification fonctionnelle mono-processeur, et d'implanter notre estimateur de mouvement dans l'encodeur vidéo H.264 de Thomson. Cela permet de comparer les algorithmes en terme de qualité des champs de vecteurs.

5.1.4 Implantation du plugin d'estimation de mouvement

La description flot de donnée est conservée pour l'implantation des estimateurs de mouvement dans un encodeur vidéo. Nous avons expliqué au paragraphe 4.2.1 la manière dont nous intégrons nos développements. Le GFD du plugin d'estimation de mouvements est donné figure 5.5. L'utilisation de la méthodologie AAA/SynDEx permet de générer le code pour la classe C++ réalisant l'estimation de mouvement. Cette opération peut alors être distribuée sur plates-formes multiprocesseurs.

Nous allons dans le paragraphe suivant évaluer les performances des estimateurs de mouvement développés avec un encodeur vidéo H.264.

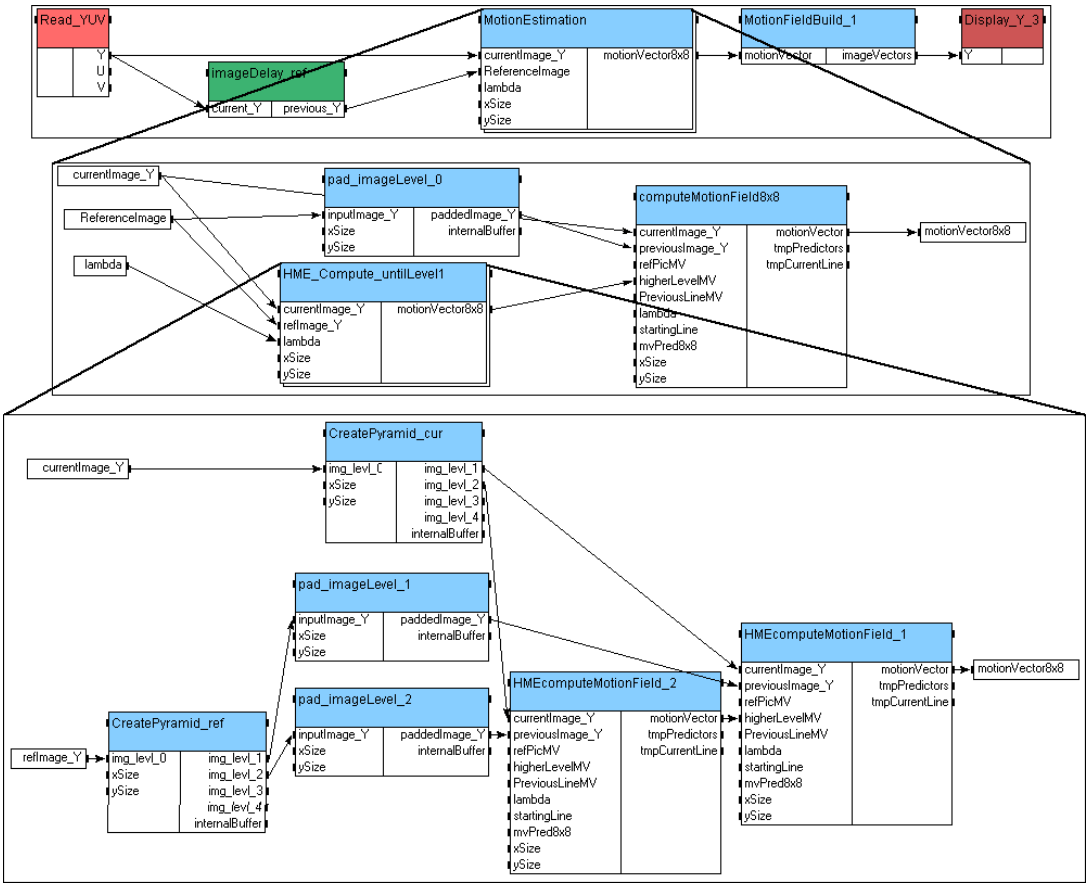


FIG. 5.3 – Graphe flot de données de l'algorithme HME pour des blocs de taille 8x8

5.2 Evaluation des performances des algorithmes développés

La génération automatique de code à partir de la description flot de données permet d'implanter facilement plusieurs estimateurs de mouvements. Pour l'évaluation des performances des algorithmes le code mono-processeur est exécuté séquentiellement avec le reste de l'encodeur vidéo. Des encodages sont réalisés avec des paramètres figés pour confronter les différents algorithmes.

5.2.1 Paramétrage de l'encodeur

Pour faire ressortir la capacité des estimateurs de mouvement à bien mettre en correspondance le bloc courant avec une image de référence, les paramètres de l'encodeur doivent être choisis pour que seul l'impact de l'estimation de mouvement soit visible. La décision des modes de codage est donc forcée, pour ne pas influer sur les résultats. Les paramètres sont les suivants :

- régulation de débit désactivée,
- GOP constant de 24 images (1 I et 23 P),
- Seuls les modes inter8x8 sont autorisés dans les images P

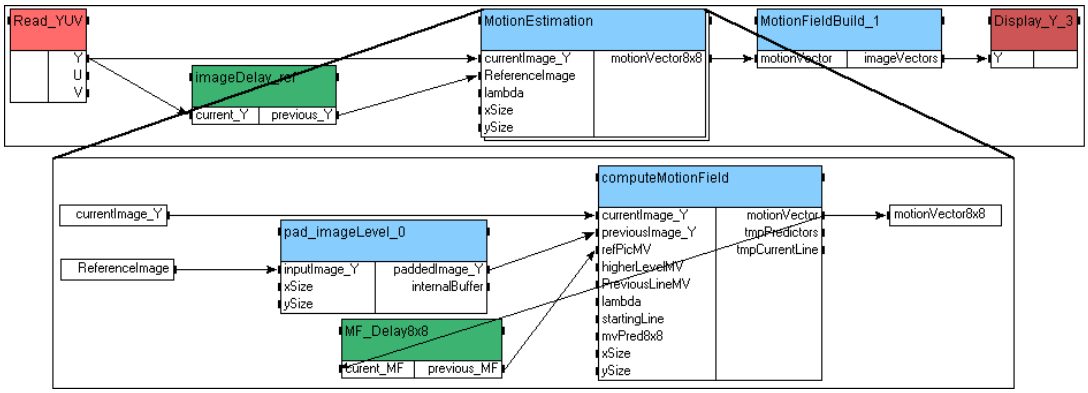


FIG. 5.4 – Graphe flot de données de l'algorithme EPZS pour des blocs de taille 8x8

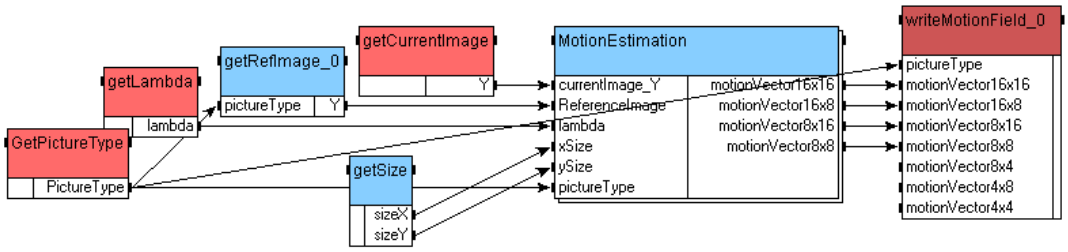


FIG. 5.5 – Graphe flot de données du plugin d'estimation de mouvement

– L'estimation de mouvement est exécutée sur des blocs de taille 8x8, pixel entier. Le but est de comparer les méthodes d'estimation de mouvement développées à partir des algorithmes HME et EPZS. Avec les décision forcées, les résultats seront artificiels, nous pouvons noter que les différences seraient lissées avec des décisions activées.

5.2.2 Algorithmes étudiés

Nous allons évaluer différentes techniques, HME et EPZS, mais aussi un mode intermédiaire permis par notre description flexible. Nous avons développé un nouvel algorithme, appelé HDS (Hierarchical Diamond Search) et présentant des performances proches de HME avec un nombre de calculs réduits. La taille de bloc choisie est 8x8 car cela correspond à un intermédiaire entre toutes les tailles possibles dans la norme H.264. Les autres tailles de bloc ne sont pas traitées tout de suite, nous aborderons ce point dans le paragraphe 5.4.2.

5.2.2.1 HME

L'algorithme est inspiré des développements internes à Thomson. C'est un algorithme prédictif basé sur une description multirésolution des images.

Phase de prédiction Pour chaque niveau de résolution, une estimation de mouvement basée sur des blocs de taille 8x8 est réalisée. La phase de prédiction pour chaque bloc consiste à tester la pertinence de plusieurs vecteurs déjà calculés pour les bloc

voisins et les niveaux hiérarchiques supérieurs. Des vecteurs de l'image précédente peuvent également être considérés comme prédicteurs, mais nous ne les utilisons pas car ils n'apportent pas de gain significatif, la technique hiérarchique étant déjà très performante. Lorsque le meilleur prédicteur est sélectionné, il est ensuite raffiné.

Phase de raffinement Une recherche exhaustive très réduite permet de trouver une meilleure mise en correspondance, si elle existe. Cette recherche s'effectue avec une amplitude de quelques pixels en horizontal et en vertical autour du meilleur prédicteur.

5.2.2.2 EPZS

EPZS est aussi un algorithme prédictif. Les développements sont repris des travaux des logiciels libres x.264 et XViD, puis adapté à nos besoins en terme d'architecture d'algorithme pour conserver une homogénéité et faciliter le portage sur DSP.

Phase de prédiction Comme pour HME, l'estimation de mouvement est basée sur des blocs de taille 8x8. Les prédicteurs proviennent des bloc voisins et de l'image précédente. Lorsque le meilleur prédicteur est sélectionné, il est ensuite raffiné.

Phase de raffinement Une recherche en diamant permet de raffiner le meilleur prédicteur. Ce raffinement est itératif. A chaque étape les huit voisins de la position courante sont testés, puis la nouvelle position courante est celle qui minimise le critère de distorsion. Des modes d'arrêt anticipés peuvent réduire le nombre de calculs à effectuer et accélérer davantage le processus, cependant, le temps d'exécution devient alors très variable et peu prédictible. Pour obtenir la meilleure qualité possible l'arrêt anticipé n'a pas été implanté, mais le nombre d'itérations de la recherche en diamant est borné, afin de garantir un fonctionnement temps réel.

Les deux algorithmes précédents se ressemblent beaucoup et pourtant ils ne conduisent pas à la même quantité de calcul. La recherche exhaustive dans HME, bien que réduite, augmente énormément le nombre de critères de distorsion à calculer. Quelque soit l'algorithme utilisé, la fonction d'estimation de mouvement développée est la même. La liste de prédicteurs est construite en fonction des entrées connectées (c'est à dire en fonction du GFD) et la fonction de raffinement (recherche exhaustive ou en diamant) est choisie selon un paramètre, à la compilation. Cette flexibilité nous a permis de mettre au point de nouveaux algorithmes d'estimation de mouvement.

5.2.2.3 HDS

Les deux algorithmes précédents présentent des intérêts complémentaires. HME est très robuste, de part son approche hiérarchique mais la recherche exhaustive, bien que réduite, demande une grande puissance de calcul. Inversement, EPZS est très rapide grâce à la recherche en diamant récursive, mais la phase de prédiction est moins précise.

Dans le but de combiner les intérêts de ces deux techniques, en éliminant leurs défauts, nous avons cherché à développer un nouvel algorithme basé sur une combinaison entre HME et EPZS. C'est donc une recherche récursive en diamant impliquant une décomposition multirésolution des traitements. HDS (Hierarchical Diamond

Search) est obtenu en utilisant la description flot de données de HME (utilisation des prédicteurs hiérarchiques) et en réglant la compilation avec les paramètres de EPZS (Raffinement local en diamant). Les avantages de cette technique est de conserver des prédicteurs hiérarchiques, robustes, tout en réduisant considérablement la charge de calcul.

Phase de prédiction Comme pour l'algorithme HME, pour chaque niveau de résolution, une estimation de mouvement basée sur des blocs de taille 8x8 est réalisée. Les prédicteurs spatiaux et hiérarchiques permettent d'initialiser la recherche en diamant. Le meilleur prédicteur est sélectionné pour être raffiné.

Phase de raffinement Le meilleur prédicteur est raffiné itérativement selon un motif en diamant, de la même façon que l'algorithme EPZS. A chaque étape, la meilleure position parmi les huit voisins de la position courante est sélectionnée pour devenir la nouvelle position courante. Le nombre d'itérations est borné afin de garantir un fonctionnement temps réel.

5.2.3 Performances obtenues

Plusieurs séquences vidéo SD (576p : 720x576) et HD (720p : 1280x720) ont été encodées en utilisant alternativement les trois estimateurs de mouvement décrits ci-dessus. Les résultats sont présentés dans ce paragraphe en terme de performance d'encodage et de vitesse d'exécution sur PC pour ces premiers portages.

5.2.3.1 Performances d'encodage

Le tableau 5.2 récapitule les résultats obtenus pour trois séquences définition standard et trois séquences haute définition. Les valeurs indiquées dans le tableau correspondent à un gain de débit à qualité (mesuré par PSNR) constante, c'est à dire les différences d'aire entre les courbes débit/PSNR, exprimées en pourcentages par rapport à l'encodeur de référence qui est ici celui qui utilise l'estimation de mouvement HME.

Algorithme d'estimation de mouvement	Variation de débit à qualité constante (%)					
	Séquences SD			Séquences HD		
	Formula1	Raid1 maroc	Raid2 maroc	Seq1	Horses1	SpinCalendar
HME	0	0	0	0	0	0
HDS	3,55	0,29	0,44	-1,53	0,78	2,04
EPZS	17,43	0,72	3,74	13,3	2,12	2,36

TAB. 5.2 – Performances d'encodage pour des séquences SD (576p) et HD (720p)

HME et HDS sont plus performants pour des séquences avec des grands mouvements (Formula1). Les techniques hiérarchiques permettent d'obtenir de bons prédicteurs, alors que EPZS doit trouver une convergence pour avoir des prédicteurs fiables. Lorsque les mouvements sont faibles et homogènes (Raid1maroc, Horses, SpinCalendar), les trois algorithmes ont des performances proches, avec un léger avantage pour HME, puis HDS. Dans certains cas (Seq1) HDS est meilleur que HME. Cela peut

s'expliquer par l'homogénéité du champ de vecteur. En effet la recherche exhaustive réduite peut donner des vecteurs un peu plus dispersés qu'une recherche en diamant tombant plus facilement dans des minima locaux. Or le gain de mise en correspondance ne compense pas toujours le coût de vecteur supplémentaire. Dans la Séquence "Seq1", HME conduit à une meilleure qualité que HDS, mais avec un débit plus important.

Les figures 5.6 et 5.7 donnent le courbes débit/distorsion pour deux séquences choisie pour leur clarté. On peut voir clairement que EPZS est en retrait et que HDS est très proche de HME.

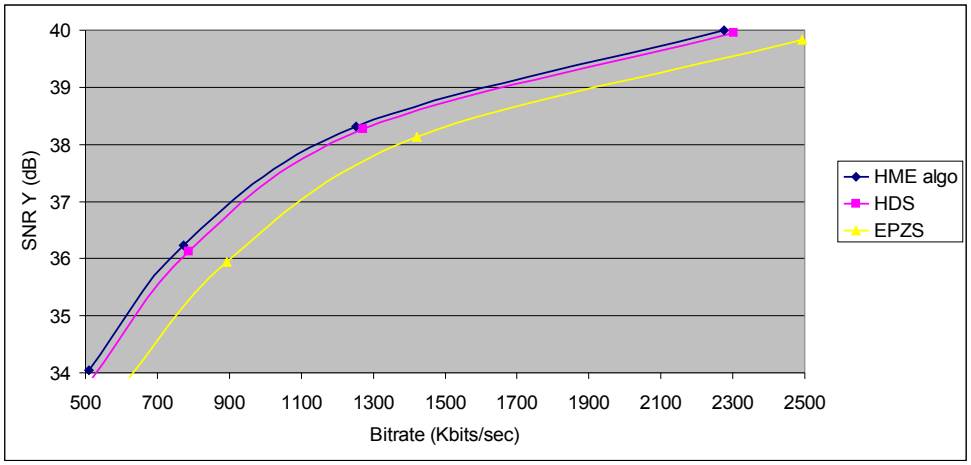


FIG. 5.6 – Courbe débit - distorsion. Séquence Formula1 720x576

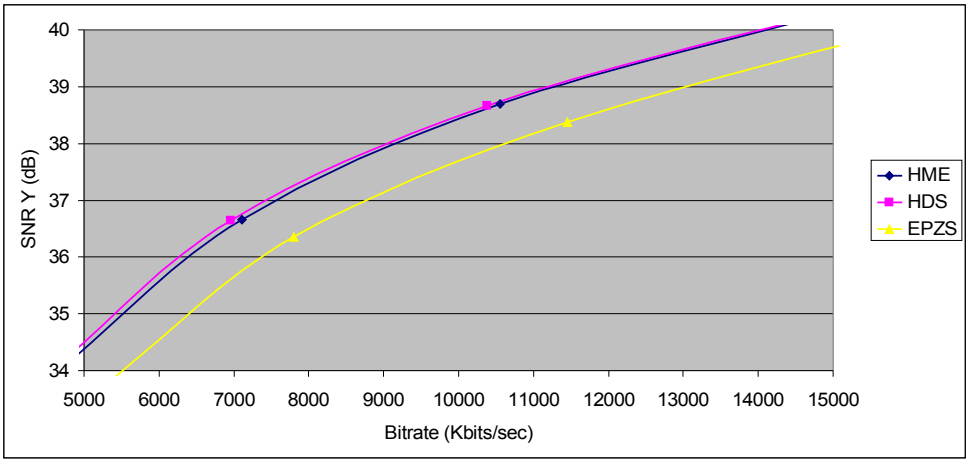


FIG. 5.7 – Courbe débit - distorsion. Séquence Seq1 1280x720

Les trois algorithmes HME, HDS et EPZS ont des performances assez similaires sur beaucoup de séquence dans ces conditions : l'image de référence est l'image juste avant temporellement. L'amplitude des vecteurs est donc limitée, même pour des séquences rapides. Cependant, dès que les mouvement deviennent plus importants, EPZS est moins performant. De plus, dans un cas général, de une à sept images B peuvent

s'intercaler entre l'image courante et l'image de référence, augmentant d'autant l'amplitude des vecteurs et donc les différences de performances entre les estimateurs de mouvement.

Pour compléter l'étude de ces trois algorithmes, leur coût de calcul est maintenant évalué.

5.2.3.2 Chronométrages sur PC

Pour évaluer la charge de calcul de ces trois méthodes, des chronométrages ont été effectués sur un PC, équipé d'un processeur Intel Xeon à 3.6 GHz et de 2 MO RAM. Le code est écrit en C sans optimisations spécifiques, sauf les calculs de SAD qui ont été optimisés avec des extensions SSE (Streaming SIMD Extensions).

Algorithme	HME	HDS	EPZS
Pyramide (x2)	3.4ms	3.4 ms	/
Padding	1 ms (5 nvx)	1 ms (5 nvx)	1 ms
Niveaux 4 à 1 (hiérarchiques)	9 ms (SAD SSE) 20 ms (SAD C)	4 ms (SAD SSE) 8 ms (SAD C)	/
Niveau 0 720p	29 ms (SAD SSE) 60 ms (SAD C)	13 ms (SAD SSE) 23 ms (SAD C)	10 ms (SAD SSE) 19 ms (SAD C)
Temps total	42 ms (SAD SSE) 84 ms (SAD C)	21 ms (SAD SSE) 35 ms (SAD C)	11 ms (SAD SSE) 20 ms (SAD C)

TAB. 5.3 – Chronométrages de HME, HDS et EPZS sur PC. Estimation de mouvement pixel entier pour des blocs de taille 8x8, par image 720p (1280x720)

Les chronométrages, regroupés dans le tableau 5.3 confirment nos suppositions : la recherche exhaustive, bien que réduite conduit à un coût de calcul plus élevé qu'une technique en diamant, et les méthodes hiérarchiques sont plus coûteuse qu'un technique mono-résolution. Au niveau 0 (pleine résolution), bien que HDS et EPZS soient les mêmes algorithmes, le nombre de prédicteurs à évaluer étant plus élevé pour HDS, (prédicteurs hiérarchiques) le temps d'exécution s'en trouve allongé. Bien que EPZS soit moins performant que HME, sont coût de calcul faible justifie son utilisation.

Dans le cas de H.264, l'estimation de mouvement doit être faite pour de multiples tailles de bloc. Le champ de vecteur 8x8 peut servir à initialiser les algorithmes pour d'autres tailles, ainsi que les champs de vecteurs hiérarchiques. HDS est donc un bon compromis entre qualité d'estimation de mouvement et coût de calcul, HME nécessite une puissance de calcul importante, mais peut tirer parti d'une architecture matérielle hautement parallèle, et EPZS est une solution économique.

Les performances globales obtenues pour le moment sur mono-processeur sont au mieux de 11 ms pour un champ de vecteur, avec l'algorithme le plus simple pour une résolution 720p. L'augmentation de la résolution jusqu'à 1080p (1920x1080), et la complexité de la norme H.264 font que ces performances ne sont pas suffisantes pour réaliser l'estimation de mouvement pour H.264 sur mono-processeur. Le choix de porter l'algorithme d'estimation de mouvement sur une plate-forme multiprocesseur est donc justifié pour apporter la puissance de calcul nécessaire.

5.2.4 Conclusion sur les algorithmes étudiés

Dans ce paragraphe nous avons posé les bases nous permettant d'étudier les algorithmes d'estimation de mouvement et les architectures. Des descriptions flots de données adaptées aux algorithmes, flexibles et parallélisables ont été réalisées. La vérification fonctionnelle des algorithmes a été réalisée grâce à la mise au points d'interfaces entre les différents outils : fonction d'acquisition de séquence vidéo (déjà existant), fonction d'affichage des champs de vecteurs, intégration des développement dans un encodeur vidéo. Cela nous a permis d'évaluer les deux algorithmes sélectionnés dans l'état de l'art et de valider nos choix. Nous avons également mis au point un nouvel algorithme d'estimation de mouvement appelé HDS, combinant une qualité élevée et un coût de calcul réduit. Des encodages et des chronométrages permettent d'évaluer les algorithmes de base, en terme de qualité et de vitesse d'exécution. Les performances de l'encodeur avec HDS sont proches, voir supérieures de celles obtenues avec HME, avec un coût de calcul divisé par deux. EPZS est en général en retrait sur des séquences mouvementées.

Les temps d'exécution obtenus lors des premiers portages ne permettent pas d'envisager de réaliser l'estimation de mouvement temps réel pour l'encodage H.264 haute définition sur mono-processeur. L'utilisation d'une plate-forme multiprocesseur est justifiée afin de fournir la puissance de calcul nécessaire.

Avant de pouvoir évaluer le type de plate-forme cible et le nombre de processeurs requis, Nous avons évalué les performances d'un processeur dédié au traitement de signal (DSP). Les algorithmes d'estimation de mouvement ont donc été portés et optimisés, ensuite les spécificités de H.264 ont été étudiées.

5.3 Implantation et optimisations sur DSP

Dans ce paragraphe, nous étudions l'implantation des estimateurs de mouvement décrits précédemment sur DSP. L'application doit être optimisée afin d'exploiter efficacement les ressources de calcul. Des techniques générales d'optimisation sur ce type de processeur sont décrites, puis les optimisations des différentes opération implantées sont détaillées. La méthodologie AAA/SynDEx est utilisée pour porter l'application d'estimation de mouvement rapidement sur DSP. Grâce à des bibliothèques de génération de code existantes, les interfaces d'entrées/sorties sur PC peuvent être directement utilisées pour réaliser des transferts de données de manière efficace (chargement d'images et visualisation).

Les premiers portages se font sur une plate-forme *SMT395* de *Sundance* [Sun04] (Fig. 5.8).

Cette plate-forme est composée d'un DSP Texas Instruments C6416 à 1 GHz, avec 64 MO de mémoire SDRAM externe et d'un FPGA qui gère les interfaces avec l'extérieur (un autre DSP, un FPGA, un convertisseurs, ou un PC). Cette plate-forme est connectée via le bus PCI à un PC.

5.3.1 Généralités sur les optimisations DSP

Les performances d'une application sur un processeur donné dépendent de la complexité de calcul, mais aussi de l'implantation. Afin d'exploiter de manière efficaces

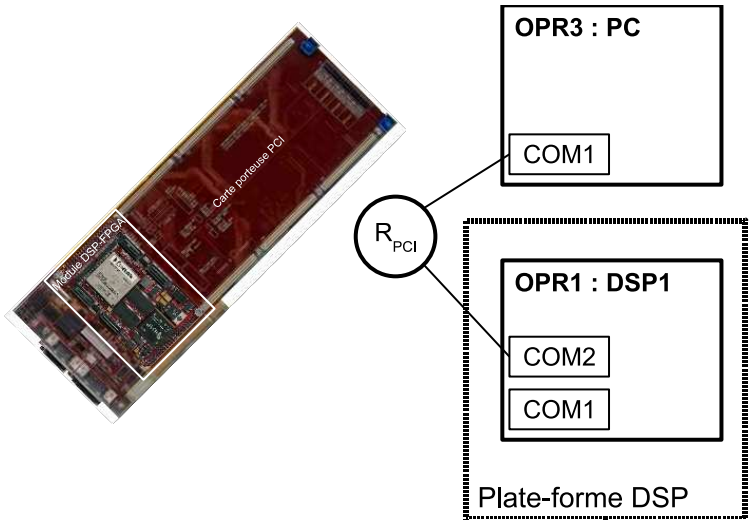


FIG. 5.8 – Graphe d'architecture dans la méthodologie AAA : modélisation d'une plate-forme Sundance (2 DSP) reliée à un PC via un bus PCI

les ressources de calcul, et d'augmenter ainsi les performances, l'implantation d'une application doit être optimisée. Pour cela, le code doit être modifié pour prendre en compte les spécificités du processeur cible. Un code très dédié, écrit en assembleur (langage machine) conduit aux meilleures performances, mais demande un temps de développement long, est difficilement évolutif, et est spécifique à chaque processeur. Inversement un langage de haut niveau (ex : C) est évolutif et générique, mais ne prend pas en compte les spécificités des processeurs. C'est alors au compilateur de réaliser les optimisations. Cependant, les compilateurs sont des programmes automatiques, et bien qu'ils puissent être performants, ils ne conduisent pas à la meilleure implantation. Nous allons décrire brièvement des techniques de base pour optimiser une implantation avec un langage de haut niveau (le C). Nous utilisons l'environnement de compilation dédié "*Code Composer Studio*" (CCS) de Texas Instruments pour programmer sur DSP, cependant les techniques utilisées sont génériques.

5.3.1.1 Optimisation des boucles

Les boucles sont le coeur de l'algorithme de calcul, c'est donc là que le temps de calcul est consommé. L'optimisation des boucles peut donc avoir un grand impact sur le temps d'exécution global. Le compilateur utilisé dispose d'un algorithme d'optimisation de boucles grâce à la vectorisation et au pipeline logiciel.

Vectorisation Selon les opérations à effectuer et la multiplicité des boucles, plusieurs itérations peuvent être déroulées, et les calculs exécutés en parallèle sur les unités de calcul et avec des instructions SIMD (Single Instruction Multiple Data). Les opérations doivent être assez simples pour permettre au compilateur de trouver l'instruction SIMD à utiliser. De plus, la multiplicité doit être connue à la compilation. Elle peut être fournie à l'aide de macros de pre-traitement. L'intérêt est de charger au maximum les unités de calcul et les bus mémoire afin d'approcher les performances

théoriques. Par exemple une unité de calcul 32 bits peut exécuter simultanément la même opération sur quatre paires 8 bits ou deux paires 16 bits.

Pipeline logiciel Le DSP visé intègre plusieurs unités de calcul dédiées (arithmétique, mémoire, multiplication,...). Il est donc possible de paralléliser plusieurs instructions si leur dépendances le permettent. Afin de mieux ordonnancer les instructions, plusieurs itérations de la boucle peuvent être pipelinées. C'est à dire que les calculs d'une itération peuvent être ordonnancés alors que l'itération précédente n'est pas terminée. Cela permet d'imbriquer les itérations et de réduire la latence globale d'une boucle. Celle-ci prend en compte l'initialisation du pipeline (prologue) et sa finalisation (épilogue). La mise en pipeline d'une boucle avec peu d'itérations doit donc veiller à garder un prologue et un épilogue réduits pour être efficace.

Restrictions Pour que le compilateur puisse optimiser la mise en pipeline des boucles, certaines contraintes doivent être respectées :

- Pas de branchement : le nombre de cycle doit être constant et les instructions exécutées doivent être toujours les mêmes pour permettre au compilateur de les ordonnancer. Le conditionnement doit être limité pour ne pas générer de saut conditionnel (utilisation des opérations conditionnelles seulement : l'exécution d'une instruction peut être désactivée en fonction de la valeur d'un registre). De même il ne doit pas y avoir d'appel de fonction.
- Le nombre d'itération doit être connu à l'initialisation de la boucle : la condition de fin de boucle doit rester constante afin de pouvoir pipeliner les itérations.
- Le nombre d'instructions d'une boucle doit être limité (limites de l'outil).

Imbrication de boucles Comme nous venons de le voir, les boucles sont optimisées par le compilateur qui crée un pipeline logiciel. Le temps d'exécution d'une boucle comprend donc un prologue (initialisation du pipeline), le corps d'exécution et un épilogue (vidage du pipeline). Lorsque des boucles sont imbriquées, seul un niveau peut être optimisé. Pour améliorer davantage les performances à la fois en réduisant la latence due au prologue et à l'épilogue, et en optimisant l'ordonnancement sur un ensemble d'instructions plus grand, il convient de pouvoir réaliser les optimisations sur la boucle de plus haut niveau. Lorsque le nombre d'itération est connu à la compilation, les courtes boucles intérieures sont complètement déroulées pour permettre l'optimisation des boucles imbriquées. Ainsi, il est préférable de définir des fonctions spécifiques (par exemple une par taille de bloc) qui peuvent être beaucoup plus performantes.

On peut noter que sur des processeurs du type Pentium, qui intègrent un prédicteur de branchement et une unité d'ordonnancement pour réduire statistiquement le nombre de cycles perdus, la mise en pipeline est effectuée à l'exécution et que ce genre d'optimisation à moins d'impact. A l'inverse, leur unités SIMD étant plus larges, la vectorisation a un impact important.

5.3.1.2 Utilisation du mot clé *"restrict "*

L'architecture mémoire des processeurs cause une latence non négligeable lors de l'accès aux données. Cette latence peut être masquée grâce au pipeline, cependant,

le compilateur prend en compte par défaut le temps nécessaire à la mise à jour de la mémoire entre une instruction d'écriture suivie d'une instruction de lecture, ce qui se traduit dans les boucles courtes à des cycles vides. Pour éviter cela, il est possible de préciser au compilateur que les données écrites ne modifient pas les données lues, c'est à dire que les buffers mémoire en jeu ne se chevauchent pas. Le mot-clé "*restrict*" permet de préciser le type d'un pointeur mémoire en spécifiant qu'il est le seul à accéder à une espace donné. Le compilateur peut donc optimiser les accès mémoire en ne prenant pas en compte les délais d'accès. L'opération type où l'impact est le plus important est la recopie mémoire.

5.3.1.3 Les fonctions "*inline*"

Au lieu de générer un appel de fonction, le compilateur peut choisir d'intégrer directement le corps d'une fonction. Cela peut augmenter la taille du code car la fonction est recopiée autant de fois qu'il y a d'appel, mais les performances sont augmentées en évitant le changement de contexte, les passages de paramètres et le branchement. De plus, la boucle contenant la fonction mise en ligne peut être optimisée par le compilateur, et le cache programme est mis à profit, car les instructions sont proches en mémoire. La fonction doit être définie avec le mot-clé "*inline*", ou bien dans certains cas le compilateur le fait automatiquement.

5.3.1.4 Utilisation des instructions spécialisées

Lors de l'optimisation des boucles, le compilateur peut utiliser des instructions SIMD pour paralléliser les traitements. Cependant lorsque la multiplicité de la boucle n'est pas bonne, en l'absence de boucle, ou si la vectorisation n'est pas triviale, les instructions SIMD ne sont pas utilisées. Il est alors possible de forcer le compilateur à utiliser des instructions SIMD en y faisant directement référence par l'intermédiaire de fonctions intrinsèques.

5.3.1.5 Accès mémoire

Les processeurs sont composés de plusieurs niveaux de mémoire. Les mémoires L1 (niveau 1), proches du CPU (Central Processing Unit), sont rapides, elles fonctionnent à la fréquence du coeur de calcul, mais elles sont de taille réduite à cause des contraintes physiques. La mémoire L2 est une mémoire interne intermédiaire, et la mémoire externe peut être considérée sans limite de taille, mais a une bande passante réduite un temps d'accès long.

Dans le cas général, une image haute définition est contenue en mémoire externe, car les mémoires internes ne peuvent pas la contenir entièrement. Pour éviter les pertes de performance dues à l'accès aux données, les données utiles doivent être rapatriées en mémoire interne (L2). Cela peut être fait manuellement en programmant un mécanisme d'accès aux données, ou automatiquement avec un contrôleur de cache. Dans les deux cas afin de réduire la fenêtre utile d'accès aux données pour éviter les défauts de cache L2 (requête vers une données non mise en cache), et optimiser l'utilisation du cache L1 (de taille réduite), il convient de mettre à profiter la localité des données.

Nous avons décrit les techniques générales utilisées lors de l'implantation des algorithmes d'estimation de mouvement. L'exécution d'un programme non optimisé sur DSP conduit souvent à des performances pouvant être de cinq à dix fois inférieures aux performances attendues. L'étape d'optimisation ne doit donc pas être négligée. Dans le paragraphe suivant nous allons traiter des optimisations spécifiques à l'estimation de mouvement.

5.3.2 Optimisations spécifiques à l'estimation de mouvement

Les calculs les plus coûteux de l'estimation de mouvement ont étés identifiés et les opérations concernées ont été optimisées. Les détails spécifiques sont donnés dans ce paragraphe.

5.3.2.1 calcul de la SAD

La SAD (Sum of Absolute Difference) est choisi comme mesure de distorsion. C'est donc dans cette opération que le processeur passe le plus de temps. Il est donc nécessaire d'y apporter un soin particulier. La SAD peut être générique, afin de réduire la taille du code C, ou bien dédiée à une taille de bloc pour de meilleures performances.

SAD générique Pour n'avoir qu'une fonction de calcul de SAD et simplifier le maintient du code, la SAD peut être écrite de manière dynamique, à l'aide d'une double boucle en fonction de la hauteur et de la largeur d'un bloc (Alg. 2.1). Comme nous l'avons expliqué dans le paragraphe précédent, la taille du bloc étant dynamique, le compilateur ne peut optimiser que la boucle intérieure. La multiplicité de la boucle peut être spécifiée (par exemple 4, pour traiter les tailles de H.264) afin d'autoriser la vectorisation.

SAD dédiée pour une taille de bloc Lorsque la SAD est dédiée pour une taille de bloc, la double boucle peut être complètement déroulée, et le pipeline logiciel est optimisé sur la boucle de niveau supérieur (la fenêtre de recherche). Ainsi les initialisation et vidage de pipeline ne sont pas à prendre en compte pour chaque SAD, l'optimisation prends en compte un nombre plus important d'opérations, et la vectorisation travaille sur une largeur de bloc fixe.

Compte tenu du gain en performance obtenu grâce à la SAD dédiée, nous écrivons une SAD par taille de bloc traité. La vectorisation est de plus obtenue manuellement en utilisant des fonctions intrinsèques dédiées.

5.3.2.2 Fenêtre de recherche (pour HME)

Dans le but de réduire les prologues et épilogues de boucles, et d'agrandir le champ d'optimisation, la double boucle de la fenêtre de recherche peut être transformée en une seule boucle.

Double boucle L'implantation la plus simple est de considérer une double boucle imbriquée pour parcourir la fenêtre de recherche (pour l'amplitude horizontale et verticale). Lorsque le calcul de SAD est entièrement déroulé, la boucle de niveau inférieur seulement est optimisée.

Simple boucle La double boucle est maintenant transformée en une simple boucle. Le nombre d'itérations est le nombre de points à tester. La position de chaque point est lue dans un tableau constant en fonction de l'indice de l'itération. Cela permet d'optimiser toute la partie répétitive. La fenêtre de recherche est de plus rendue davantage flexible, car n'importe quelle forme de fenêtre peut être utilisée, et l'ordre des points à tester est indifférent. Il est alors possible d'envisager des fenêtres circulaires ou un parcours en spirale au lieu de la fenêtre rectangulaire et du parcours en balayage de gauche à droite de base.

Il est possible, avec ce genre de modification de contraindre facilement le nombre de points à tester dans la fenêtre de recherche en fonction du temps de traitement restant. Cela peut être utile pour garantir un fonctionnement en temps réel sur un processeur où les temps d'exécutions peuvent varier en fonction des données (mémoire cache).

5.3.2.3 Recherche itérative (pour EPZS)

Dans l'algorithme EPZS, la fenêtre de recherche est itérativement parcourue, en testant à chaque étape des candidats, suivant un motif en diamant (quatre ou huit connexités). A la première itération, tous les voisins doivent être évalués, alors que ensuite, le motif de recherche recouvre des positions déjà évaluées, pour lesquelles le calcul de SAD est inutile.

Algorithme initial Dans l'algorithme initial (Alg. 5.1), pour chaque itération, tous les candidats sans considérés dans une boucle dont le nombre d'itérations est la taille du motif de recherche. Pour chaque candidat, le calcul est effectué seulement s'il n'a pas déjà été fait à l'itération précédente. Pour cela, chaque position est associée à un masque. Suivant la direction de descente, de un à cinq candidats sur huit sont calculés. Le masque est initialisé pour que toutes les positions soient testées à la première itération. Lorsque le masque a la valeur nulle, un optimum a été trouvé.

Algorithme 5.1 Algorithme EPZS avec un motif à huit connexités

```
int Mask = 0xFF ; // toutes les positions du motif seront testées
while(Mask){ // L'algorithme est itéré tant qu'une direction de descente existe
  int Position ;
  for(Position = 0 ; Position < 8 ; Position ++){ // Boucle 1
    if(Mask & (1 << Position)) // Condition 1 (vérifie que le candidat est à évaluer)
      CHECK_CANDIDATE(Position) ; // Boucle 2 (calcul de SAD)
  }
}
```

Avec une écriture de ce type, les optimisations du compilateur ne peuvent être exécutées que sur le calcul de SAD (boucle 2). Afin de pouvoir optimiser la boucle de niveau supérieur (motif de recherche), il est nécessaire de faire disparaître la condition 1 dans la boucle.

Algorithme optimisé Nous voulons éliminer la condition liée au masque du motif de recherche dans la boucle de test des candidats. Une solution est de construire une liste de candidats avant la boucle de test. Les optimisations du compilateur portent sur la boucle voulue et les performances sont améliorées. Cependant le temps de calcul peut être davantage amélioré en évitant la construction de la liste de candidats. Nous utilisons pour cela des instructions spécifiques. L'algorithme modifié permet l'optimisation de la boucle 1 (Alg. 5.2).

Algorithme 5.2 Algorithme EPZS modifié

```
int Mask = 0xFF ; // toutes les positions du motif seront testées
while(Mask){ // L'algorithme est itéré tant qu'une direction de descente existe
  int Position ;
  int nbPos = COUNT_BITS(Mask) ; // Donne le nombre de points effectif à tester
  for(Position = 0 ; Position < nbPos ; Position ++){ // Boucle 1
    Vector Candidate = GET_NEXT_CANDIDATE(Position, Mask) ;
    // Récupère la position du candidat en fonction du masque)
    CHECK_CANDIDATE(Candidate) ; // Boucle 2 (calcul de SAD)
  }
}
```

L'algorithme ainsi modifié permet de réduire considérablement le conditionnement, et les traitement réguliers sont mieux optimisés par le compilateur. Les performances sont donc améliorées.

5.3.2.4 Buffer interne de l'image de référence

Une image Haute définition (HD) est trop volumineuse pour tenir en mémoire interne, il faut donc la placer en mémoire externe. Les accès à cette mémoire étant très lents, il est utile de rapatrier la zone utile de l'image en mémoire interne. L'activation du cache de niveau 2 améliore considérablement les performances, cependant, un mécanisme manuel peut améliorer les résultats, d'autant plus que le DSP C6416 possède 1 MO de mémoire interne. Nous présentons ici des techniques pour améliorer l'accès à l'image de référence.

Une image HD 720p a une largeur de ligne de 1280 pixels, élargie à 1320 pour prendre en compte la recherche à l'extérieur de l'image (padding de l'image). Pour permettre une estimation de mouvement limitée à 90 pixels en vertical - ce qui est suffisant même pour la cas de la haute définition - il faut environ 200 lignes ($2 \times 90 + 16$(taille d'un macro-bloc) + (bordure pour appliquer le filtre subpixel)) d'image en mémoire interne. Cela représente 264000 octets. Afin d'accélérer les traitements, la zone utile de l'image est copiée en mémoire interne. Lors du déroulement de l'algorithme, à chaque début de ligne, la mémoire interne doit être mise à jour. Seules les nouvelles lignes sont chargées à partir de la mémoire externe, le reste de l'image est géré en interne. Différentes solution de gestion de la mémoire sont présentées ci dessous.

1. Tampon mémoire circulaire

   Un mécanisme de mémoire circulaire est mis en place. Lors du chargement d'une nouvelle ligne, les anciennes données sont écrasées (figure5.9). Une translation d'adresse doit être effectuée pour gérer les accès mémoire lors des calculs.

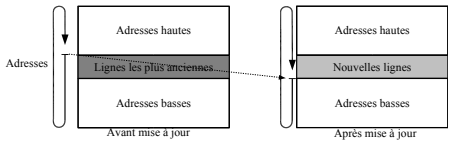


FIG. 5.9 – Gestion du tampon circulaire

2. Recopie interne
   Afin d'éviter la gestion du pointeur de la mémoire circulaire, il est possible de faire glisser les données (figure 5.10). Ce mécanisme évite la gestion dynamique de l'adresse de l'image de référence, par contre les données sont recopiées à chaque nouvelle ligne.

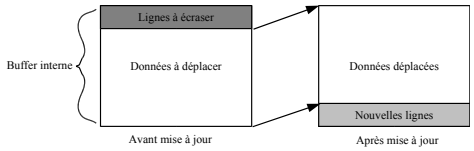


FIG. 5.10 – Gestion du buffer interne

3. Recopie interne améliorée
   Afin d'éviter un grand nombre de recopies dues au glissement des données, il est possible d'augmenter la taille du tampon de mémoire interne (figure 5.11). La recopie des données n'est alors requise que lorsque le tampon est plein. A ce moment les données utiles sont déplacées, écrasant les données périmées, puis les nouvelles lignes sont chargées à partir de la mémoire externe.

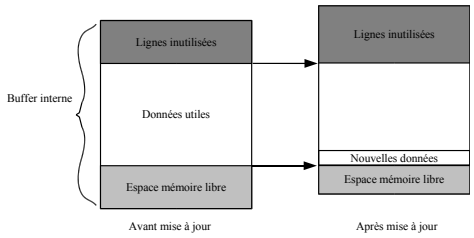


FIG. 5.11 – Gestion du buffer interne amélioré

Ces différentes solutions ne sont pas équivalentes, le choix d'une solution est un compromis entre la complexité introduite par le mécanisme de tampon et la mémoire disponible. La solution 1 est optimale du point de vue des accès mémoire, mais elle introduit des calculs supplémentaires pour gérer les adresses. La solution 2 est certainement trop coûteuse en recopie, elle sera évitée. La solution 3 permet d'éviter des recopies trop nombreuses et une translation d'adresse complexe, elle est un bon compromis si le processeur dispose d'assez de mémoire interne.

Une gestion de mémoire manuelle peut être très efficace, néanmoins cela implique une complexité de l'algorithme supplémentaire et une bonne connaissance de l'archi-

tecture interne du processeur. La taille de la mémoire interne doit également être suffisante. Pour les autres cas, le contrôleur de cache peut offrir un bon compromis, en gérant l'optimisation des accès de manière dynamique en matériel.

5.3.2.5 Synthèses des résultats

Des chronométrages ont été effectués sur DSP C6416 à 1 GHz, pour l'exécution de l'estimation de mouvement HD (720p).

Algorithme et optimisations	Temps d'exécution
Pyramide (4 niveaux) mémoire externe	290 ms
Pyramide (4 niveaux) cache L2	11.7 ms
Pyramide (4 niveaux) accès mémoire manuel en concurrence avec les calculs	2.4 ms
Padding (5 niveaux) mémoire externe	345 ms
Padding (5 niveaux) cache L2	12.4 ms
Padding (5 niveaux) accès mémoire manuel en concurrence avec les calculs	2,7 ms
HME niveau 0 mémoire externe, bloc 8x8	2200 ms
HME niveau 0 cache L2, bloc 8x8	49 ms
HME niveau 0 cache L2, bloc 8x8, SIMD	20 ms
HME niveau 0 cache L2, bloc 8x8, SIMD, fenêtre de recherche en simple boucle	17 ms
HME niveau 0 buffer interne, bloc 8x8, SIMD, fenêtre de recherche en simple boucle	15 ms
EPZS cache L2, bloc 8x8, SIMD	12 ms
EPZS cache L2, bloc 8x8, SIMD, algo optimisé	10 ms
EPZS buffer interne, bloc 8x8, SIMD, algo optimisé	8 ms

TAB. 5.4 – Temps d'exécution des opérations en fonction des optimisations

L'amélioration des temps d'exécution confirme l'intérêt de la phase d'optimisation des algorithmes. L'accès aux données en utilisant la mémoire externe est très important, un gain en performance de plus de quarante est obtenu avec la mémoire cache de niveau 2, et ce gain est d'autant plus important que l'algorithme est optimisé. De manière générale, un gain de trois à cinq est obtenu avec les optimisations.

La mise en place d'une technique d'accès mémoire manuelle optimisée donne un gain de performance non négligeable. Cependant, la taille mémoire nécessaire est conséquente et cette solution n'est applicable que lorsque suffisamment de mémoire est disponible. L'implantation sur DSP Texas Instruments DM642 ne contenant que 256 KO de mémoire interne rend impossible l'utilisation d'une telle technique. A la place, la mémoire cache est utilisée.

L'étape d'optimisation nécessite une bonne connaissance de l'architecture cible et des techniques de programmation. Cette étape peut être longue lorsque les algorithmes sont complexes. Les optimisations sont en général spécifiques à processeur et à un calcul donné. Elles s'effectuent au détriment de la généricité et de l'évolutivité. Néanmoins, les optimisations apportées ici permettent de conserver une certaine évolutivité dans les algorithmes, et apportent un gain en performances indispensable aux opérations de base de l'estimation de mouvement pour réduire le nombre

de processeurs nécessaire. Nous allons maintenant étudier les optimisations apportées aux opérations spécifiques de l'estimation de mouvement pour H.264.

5.4 Implantation des opérations spécifiques à H.264

Nous présentons dans ce paragraphe les implantations optimisées du raffinement subpixélique et le l'estimation de mouvement à taille de bloc variable. Les techniques d'optimisations présentées précédemment sont appliquées et permettent d'améliorer les performances. Des modifications des algorithmes sont également proposées pour réduire la complexité. Les impacts sur la qualité des champs de vecteurs sont analysés.

5.4.1 Optimisation du raffinement subpixélique

H.264 améliore la précision de l'estimation de mouvement par rapport à ses prédécesseurs. La précision des vecteurs dans MPEG-2 est au demi-pixel. H.264 introduit le quart de pixel avec des filtres d'interpolation spécifiques. Le coût de calcul dû au raffinement subpixélique est important. Nous avons présenté différentes stratégies dans le paragraphe 2.4.2. Les techniques sans interpolation [HCBC06] ne donnent pas de bons résultats. Nous utilisons donc une technique à deux pas (demi-pixel puis quart de pixel) avec interpolation des données à la volée, afin d'optimiser les accès mémoire. Nous présentons dans ce paragraphe les implantations de ces opérations.

5.4.1.1 Filtres d'interpolation

L'interpolation des données demi et quart de pixel implique des calculs lourds. Le filtre demi-pixel à six coefficients de la norme est complexe. L'estimation de mouvement peut utiliser n'importe quel filtre puisque seul le vecteur de mouvement est utilisé. L'interpolation est refaite pour calculer le résidus dans l'opération de compensation. Nous pouvons donc évaluer l'intérêt de l'utilisation d'un filtre bilinéaire, beaucoup plus simple à implanter.

Filtres de la norme Les deux filtres demi-pixel et quart de pixel de la norme sont implantés, puis optimisés avec des fonctions intrinsèques. Pour le calcul de SAD demi-pixel, un échantillon sur deux en horizontal et en vertical doit être mis en correspondance avec les pixels du bloc courant. Cela nécessite l'arrangement des échantillons dans les registres afin d'utiliser des instructions SIMD.

L'interpolation quart de pixel est plus simple, l'application du filtre linéaire revient à faire une moyenne entre deux échantillons demi-pixels. Pour pouvoir calculer les huit SAD autour de la meilleure position demi-pixel, les échantillons sont arrangés en huit blocs correspondant aux huit positions.

Filtre bilinéaire Le filtre bilinéaire est plus simple à implanter que le précédent. Sa vectorisation est rapide car chaque valeur de sortie correspond à la moyenne entre deux échantillons d'entrée. Les échantillons demi-pixel et quart de pixel sont arrangés de manière à faciliter la vectorisation du calcul de SAD. Les données correspondant à une SAD sont contiguës.

Le raffinement subpixélique à la volée permet de réduire considérablement les contraintes d'accès mémoire en conservant les valeur interpolées dans les mémoire de niveau inférieures, plus rapides que des mémoires externes.

5.4.1.2 Résultats

Plusieurs séquences vidéos ont été encodées pour évaluer les performances des filtres d'interpolation, et des chronométrages sur DSP ont été réalisés.

Qualité Les encodages vidéo de différentes séquences nous permettent de comparer la qualité des champs de vecteurs issus des différentes implantations d'estimateurs de mouvement HDS (Tab. 5.5) : quart de pixel avec les filtres de la norme, avec des filtres bilinéaire, et pixel entier.

Raffinement quart de pixel	Variation de débit à qualité constante (%)					
	Séquences SD			Séquences HD		
	Formula1	Raid1 maroc	Raid2 maroc	Seq1	Horses1	SpinCalendar
Filtre H.264	0	0	0	0	0	0
Filtre bilinéaire	2,2	5,33	4,8	2,7	4,8	9,4
Pixel entier	9,7	45	30	11	27	60

TAB. 5.5 – Performances d'encodage pour des séquences SD (576p) et HD (720p)

La perte de compression lorsque le raffinement subpixélique n'est pas activé est considérable, pouvant aller jusqu'à 60 %. Un exemple de courbe débit/distorsion est donné figure 5.12. Les performances des estimateurs de mouvement au quart de pixel sont visiblement meilleures qu'au pixel entier. Les résidus sont réduits, donnant une image de meilleure qualité à un débit plus faible. L'utilisation d'un filtre bilinéaire introduit une perte de performance réduite, jusqu'à 9 %.

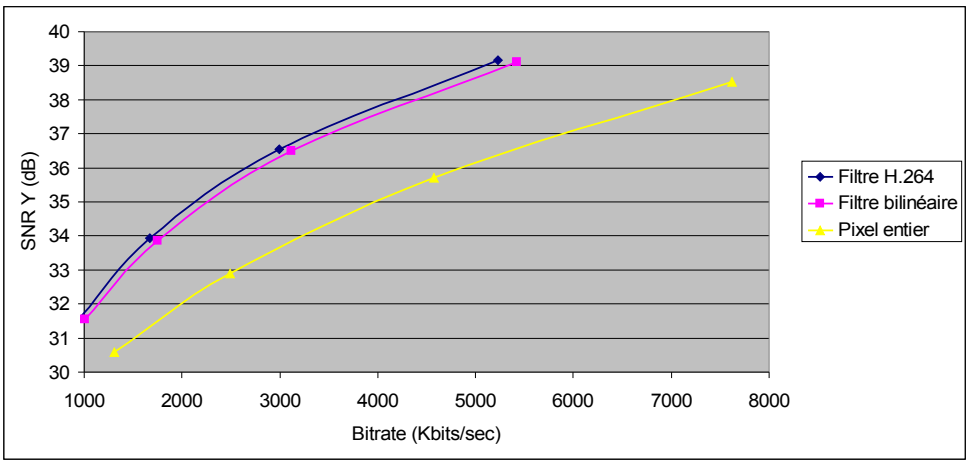


FIG. 5.12 – Courbe débit - distorsion. Séquence Raid1_Maroc 720x576

Chronométrages sur DSP Les temps d'exécution de ces implantations ont été mesurés sur DSP C6416 à 1 GHz. Les résultats sont regroupés dans le tableau 5.6.

Implantation	Filtres H.264 (non optimisé)	Filtres H.264 (optimisé)	Filtres bilinéaire (optimisé)
Filtre demi-pixel	1600 ns	720 ns	150 ns
raffinement demi-pixel SAD sur 8 candidats	820 ns	210 ns	190 ns
Filtre quart de pixel	940 ns	140 ns	150 ns
raffinement quart de pixel SAD sur 8 candidats	560 ns	130 ns	140 ns
Total (bloc 8x8)	3900 ns	1200 ns	630 ns
Total image 720p	56 ms	17.3 ms	9 ms

TAB. 5.6 – Implantations du raffinement subpixélique sur DSP pour un bloc 8x8

Comme prévu, le filtre bilinéaire accélère le raffinement subpixélique d'un facteur proche de deux. Le temps nécessaire à l'estimation d'un champ de vecteur doit prendre en compte la recherche au pixel entier, soit pour une image 720p : 30 ms avec les filtres H.264 et 22 ms avec le filtre bilinéaire. On peut noter que dans le cas du raffinement subpixélique avec le filtre H.264, le raffinement demande plus de temps que l'estimation de mouvement pixel entier, alors que la fenêtre de recherche est très réduite ($amplitude < 1\,pixel$).

L'augmentation de la précision des vecteurs de mouvement apporte un gain en performance de compression, mais aussi une charge de calcul supplémentaire conséquente. L'utilisation d'un filtre bilinéaire réduit de moitié la charge de calcul sur DSP, et limite la perte de performances de l'encodeur. Il peut être un bon candidat à un estimateur de mouvement économique.

5.4.2 Estimation de mouvement à taille de bloc variable

L'estimation de mouvement à taille de bloc variable permet de prendre en compte les mouvements différents de plusieurs objets dans un macro-bloc. Une petite taille de bloc permet d'améliorer la prédiction, mais nécessite plus de débit pour transmettre les vecteurs, alors qu'une grande taille de bloc conduit à une prédiction moins bonne mais réduit le débit nécessaire à la transmission du champ de vecteur. Un algorithme de décision doit donc sélectionner la meilleure combinaison afin de réduire le débit tout en gardant le maximum de qualité.

Lorsque l'algorithme de décision est dissocié de l'estimation de mouvement, celle-ci fourni à la décision les champs de vecteurs correspondant à chaque taille de bloc de manière exhaustive. Pour la haute définition, les tailles de bloc inférieures à 8x8 correspondent à des petits détails et n'apportent pas un gain de compression significatif par rapport à la complexité introduite. Nous ne considérons donc que les sous-partitions 16x16, 16x8, 8x16 et 8x8.

Nous étudions dans ce paragraphe plusieurs techniques d'estimation de mouvement à taille de bloc variable. Nous allons également évaluer la complexité introduite par la précision au quart de pixel de l'estimation de mouvement à taille de bloc variable.

5.4.2.1 Algorithme exhaustif pour les multiples tailles de bloc

La technique exhaustive consiste à faire une estimation de mouvement sur toute l'image pour chaque taille de bloc. La figure 5.13 montre le graphe flot de données correspondant avec l'algorithme HME ou HDS.

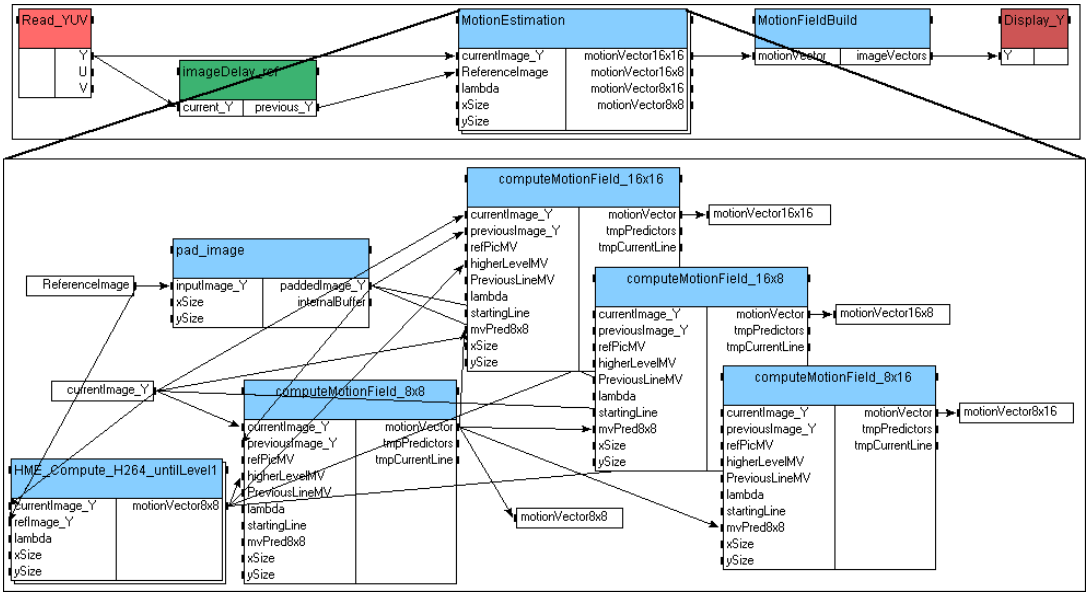


FIG. 5.13 – Graphe flot de données de l'estimation de mouvement à taille de bloc variable exhaustive

L'opération d'estimation de mouvement est définie hiérarchiquement. Les niveaux hiérarchiques ne sont pas détaillés, ce sont les mêmes que dans le cas de la recherche 8x8. L'estimation de mouvement pour chaque taille de bloc est effectuée séparément. Le champ de vecteur du niveau 1 fourni des prédicteurs hiérarchique pour toutes les tailles de bloc. Le champ de vecteurs 8x8 peut également servir de prédiction pour les autres tailles de bloc, comme c'est le cas sur la figure 5.13.

L'algorithme 5.3 décrit le fonctionnement global. N'existant pas de dépendance entre les opérations pour les différentes tailles de bloc (à part la prédiction 8x8 qui est optionnelle) il est possible de paralléliser aisément la boucle 2 sur plusieurs processeurs. L'avantage de cette modélisation est donc une parallélisation presque immédiate.

Algorithme 5.3 Estimation de mouvement exhaustif pour taille de bloc variable

```
pour chaque image
  pour chaque taille de bloc (boucle 1)
    pour chaque bloc (boucle 2)
      Rechercher le mouvement du bloc
    fin pour
  fin pour
fin pour
```

Comme pour la taille de bloc 8x8, l'estimation de mouvement pour chaque bloc consiste à évaluer un certain nombre de prédicteurs, puis à raffiner le meilleur. Le

raffinement peut être effectué avec une recherche exhaustive réduite (HME), ou avec une recherche en diamant (HDS). Lorsque les prédicteurs issus de l'estimation 8x8 sont utilisés, la fenêtre de recherche ou le motif en diamant peuvent être réduits, car la prédiction peut être considérée comme très bonne. La recherche exhaustive comprend un voisinage de un ou deux pixels d'amplitude, et le motif en diamant est réduit à quatre connexités.

Lors de l'implantation de cet algorithme sur DSP, le goulot d'étranglement dû aux accès mémoire se fait remarquer sur chaque parcours de l'image. En effet l'image est parcourue entièrement pour chaque taille de bloc. Cette approche ne permet pas de tirer parti des niveaux de cache de manière efficace. Afin d'utiliser les propriétés de localité des données, il peut être intéressant d'inverser les boucles 1 et 2 de l'algorithme 5.3.

5.4.2.2 Organisation des opérations par macro-bloc

Nous modifions l'algorithme 5.3 en inversant les boules 1 et 2 pour traiter les différentes tailles de bloc par macro-bloc. Nous comptons ainsi utiliser de manière efficace les caches de niveau 1 et 2 présents sur le DSP. Le graphe flot de données apparaît maintenant condensé (Fig. 5.14). La parallélisation des traitements en fonction de la taille du bloc à traiter n'est plus possible au niveau image. Le fonctionnement actuel va permettre d'améliorer les résultats sur chaque DSP en optimisant les accès mémoire.

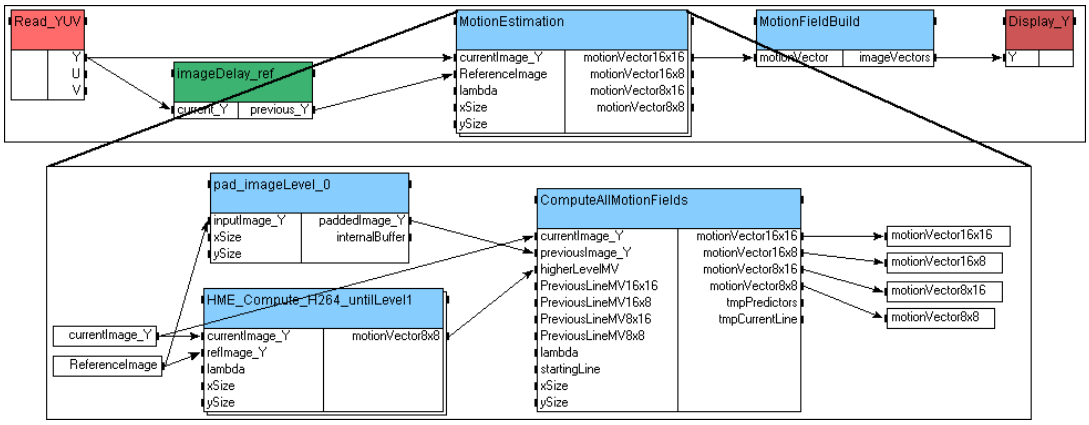


FIG. 5.14 – Graphe flot de données de l'estimation de mouvement à taille de bloc variable exhaustive

5.4.2.3 Réutilisation de SAD

Cette technique a été présentée précédemment (cf paragraphe 2.4.1.4), elle consiste à calculer des SAD partielles et de les combiner pour obtenir des SAD sur différentes tailles de bloc sans calcul supplémentaire. Cette technique doit être associée à une recherche exhaustive pour être cohérente, car une méthode prédictive suit une direction de descente de SAD, qui n'est pas la même pour toutes les tailles de bloc.

Cependant, nous allons évaluer cette technique pour les tailles de bloc 8x16, 16x8, et 16x16. Nous faisons l'hypothèse que lorsque les tailles de bloc sont proches, les

mouvements également. L'estimation de mouvement est réalisée comme précédemment pour les tailles de bloc 8x8 (HME ou HDS), puis les champs de vecteurs du premier niveau hiérarchique et du niveau pleine résolution servent de prédiction à une recherche locale avec réutilisation de SAD. Pour chaque bloc, cinq vecteurs sont calculés.

La SAD est calculé sur la base de quatre blocs 8x8. Ces quatre SAD partielles sont ensuite additionnées pour former deux SAD 16x8, deux SAD 8x16 et une SAD 16x16. Lors de la phase de prédiction, chacun des cinq résultats retient le prédicteur qui minimise sa SAD. La phase de raffinement est effectuée autour du meilleur prédicteur du bloc le plus gros (16x16). Elle consiste en une recherche exhaustive réduite. Les prédicteurs fournissent un point de départ très proche de l'optimum, cependant la fenêtre de recherche est légèrement élargie (amplitude de quelques pixels) pour prendre en compte la multiplicité des tailles de bloc. Une fenêtre de recherche plus grande reviendrait à faire plus de calcul que dans les cas précédents, il serait alors préférable de considérer une recherche indépendante par taille de bloc.

5.4.2.4 Bilan sur l'estimation de mouvement à taille de bloc variable pixel entier

Les variantes de l'estimation de mouvement à taille de bloc variable présentées précédemment sont analysées en terme de qualité des champs de vecteurs pour la compression H.264 et de charge de calcul sur DSP C6416 à 1 GHz.

Nous avons comparé cinq algorithmes :

1. HME : les vecteurs de mouvement pour les tailles de bloc 8x8 sont calculés avec l'algorithme HME. Les autres tailles de bloc (8x16, 16x8 et 16x16) sont analysées exhaustivement avec comme prédicteurs les champs de vecteurs 8x8 et hiérarchique. Le raffinement local est une recherche exhaustive avec une fenêtre de recherche réduite à deux pixels d'amplitude.
2. HDS : les vecteurs de mouvement pour les tailles de bloc 8x8 sont calculés avec l'algorithme HDS. Les autres tailles de bloc (8x16, 16x8 et 16x16) sont analysées exhaustivement avec comme prédicteurs les champs de vecteurs 8x8 et hiérarchique. Le raffinement local utilise un motif en diamant à quatre connexités.
3. Réutilisation de SAD : les vecteurs de mouvement pour les tailles de bloc 8x8 sont calculés avec l'algorithme HDS. Les autres tailles de bloc (8x16, 16x8 et 16x16) sont analysées conjointement avec comme prédicteurs les champs de vecteurs 8x8 et hiérarchique. Le raffinement local est une recherche exhaustive avec une fenêtre de recherche de quatre pixels d'amplitude. La technique de réutilisation de SAD est utilisée pour limiter les calculs.
4. HME 16x16 : l'estimateur de mouvement ne calcule que les vecteurs des blocs de taille 16x16.
5. HME 8x8 : l'estimateur de mouvement ne calcule que les vecteurs des blocs de taille 8x8.

Qualité Les cinq estimateurs de mouvement sont évalués dans l'encodeur H.264. Les estimateur HME 16x16 et HME 8x8 permettent de valider l'intérêt d'avoir plusieurs

tailles de bloc différentes. Le tableau 5.7 synthétise les résultats obtenus en terme de débit/distorsion.

	Variation de débit à qualité constante (%)					
	Séquences SD			Séquences HD		
Implantation	Formula1	Raid1 maroc	Raid2 maroc	Seq1	Horses1	SpinCalendar
HME	0	0	0	0	0	0
HDS	3,3	0,4	0,13	-0,9	-0,5	2,7
Réutilisation de SAD	4	0,4	0,92	0,5	0,7	0,31
HME 16x16	3,7	4,7	7,55	3	2,4	7,9
HME 8x8	13	4,6	8,81	20	20	15

TAB. 5.7 – Performances d'encodage avec les estimateurs de mouvement à taille de bloc variable.

Avec une seule taille de bloc, les performances d'encodage sont moins bonnes. La figure 5.15 fait apparaître les courbes débits/distorsion des différents encodages. Lorsqu'une seule taille de bloc est autorisée, le comportement change à bas débit et à au débit. Une taille de bloc plus grande favorise la compression à bas débit en réduisant la quantité des informations de mouvement à transmettre, et à haut débit la tendance s'inverse car les petits blocs favorisent un faible résidus.

L'estimation de mouvement à taille de bloc variable apporte donc un gain de performance notable. La réutilisation de SAD a des performances légèrement inférieures à HDS excepté pour une séquence HD (SpinCalendar). HDS a des performances proches de HME.

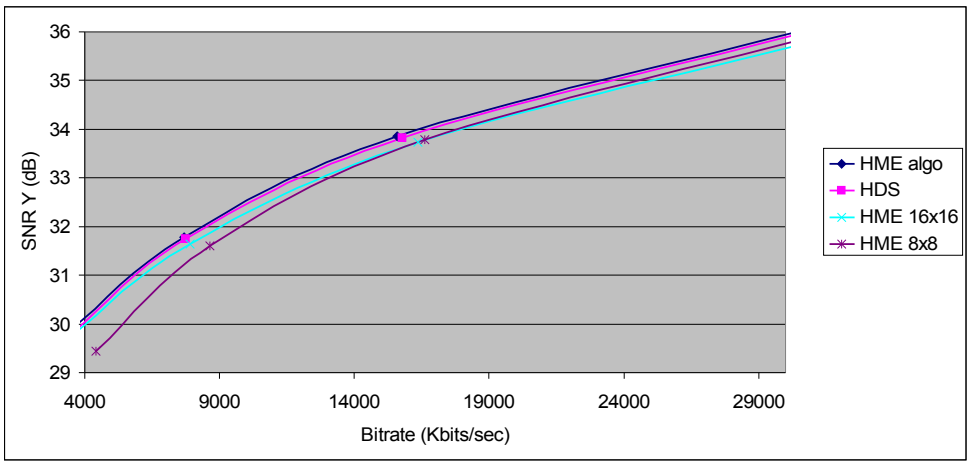


FIG. 5.15 – Courbe débit - distorsion. Séquence SpinCalendar 1280x720

On peut noter que l'estimation de mouvement a taille de bloc variable de 16x16 à 8x8 apporte un gain notable par rapport à une taille de bloc de 16x16 unique, mais inférieur à 8 % dans les exemples choisis. De plus, les petits blocs (8x8) voient leur intérêt dans les hauts débits. Nous pouvons donc valider notre choix de ne pas traiter

les tailles plus petites, qui apporteraient une complexité supplémentaire pour un gain de performance réduit.

Chronométrages Les algorithmes HME, HDS et réutilisation de SAD ont été chronométrés sur DSP. HME et HDS peuvent être implanté de deux manières suivant l'ordre des boucles 1 et 2 de l'algorithme 5.3, c'est-à-dire suivant la factorisation des traitements par image ou par macro-bloc. Pour chaque algorithme, les deux solutions ont été implantées. Les vecteurs calculés par l'une ou l'autre solution sont bit-exacts.

	Temps d'exécution	
Implantation	factorisation image	factorisation macro-bloc
HME 8x8	19 ms	
HME 16x16	12 ms	
HME 16x8	15 ms	
HME 8x16	15 ms	
Total HME	61 ms	51 ms
HDS 8x8	13 ms	
HDS 16x16	10 ms	
HDS 16x8	12 ms	
HDS 8x16	12 ms	
Total HDS	47 ms	35 ms
HDS 8x8 réutilisation de SAD		32 ms

TAB. 5.8 – Implantations des estimateurs de mouvement à taille de bloc variable pour une séquence 720p (1208x720)

Le tableau 5.8 synthétise les mesures pour le calcul des vecteurs du dernier niveau (niveau pleine résolution). La factorisation des calculs pour les différentes tailles de bloc par macro-bloc favorise la localité des données. Les données utiles sont chargées en cache et réutilisées. De ce fait les performances sont meilleures alors que les valeurs calculées sont identiques. L'algorithme HME est accéléré de 10 ms et HDS de 12 ms. La réutilisation de SAD permet de réduire le temps d'exécution par rapport à HME, mais est équivalent à HDS. De plus, HDS est un algorithme plus flexible. Son temps d'exécution peut être davantage réduit. Par exemple il est possible de sélectionner le mode à raffiner avant d'effectuer les calculs (ex : mode 16x8 ou 8x16). Cette réduction de complexité avec une technique de réutilisation de SAD n'a pas d'intérêt.

D'une part l'estimation de mouvement à taille de bloc variable améliore les performance de compression vidéo de façon notable. D'autre part cela introduit une charge de calcul supplémentaire. L'augmentation du temps d'exécution sur DSP est limité en utilisant un algorithme rapide (HDS) et en factorisant les opérations par macro-bloc, ce qui permet de mieux exploiter le mécanisme de mémoire cache. La technique de réutilisation de SAD n'offre pas de performances suffisantes par rapport à HDS. Pour simplifier cette étude, le raffinement subpixélique n'a pas été considéré dans ce paragraphe. Il introduit pourtant un gain en compression et une charge de calcul supplémentaire.

5.4.2.5 Taille de bloc variable et raffinement subpixélique

Le coût de calcul supplémentaire dû au raffinement subpixélique des vecteurs de mouvement est important. D'autant plus que dans un contexte de taille de bloc variable, le raffinement doit être effectué pour chaque taille de bloc. Etant donné que les vecteurs des sous-partitions d'un même macro-bloc sont différents par définition, il est difficile de réutiliser les valeurs déjà interpolées.

Le raffinement subpixélique apportant un gain supérieur par rapport à l'estimation de mouvement à taille de bloc variable (Tab. 5.5 et 5.7), il est préférable d'utiliser un estimateur de mouvement à taille de bloc fixe avec raffinement subpixélique plutôt que d'implanter un estimateur de mouvement à taille de bloc variable sans raffinement subpixélique. Basé sur cette remarque, nous allons donc évaluer les performances d'un estimateur de mouvement allégé.

Description de l'algorithme d'estimation de mouvement H.264 allégé Nous voulons réduire la complexité de l'estimateur de mouvement tout en gardant le maximum de performances de compression. Nous conservons donc à la fois la taille de bloc variable et le raffinement subpixélique dans notre nouvel estimateur de mouvement allégé. Cependant, pour réduire le coût de calcul, seulement un mode est sélectionné pour le raffinement subpixélique. Nous économisons par conséquent les ressources de calcul nécessaires au raffinement subpixélique de trois modes sur quatre. Pour chaque macro-bloc, l'estimation de mouvement à taille de bloc variable pixel entier est exécuté (HDS par exemple). Ensuite, un algorithme de décision simple analyse les résultats des différents modes et choisi le meilleur. Le (ou les) vecteur(s) correspondant(s) est (sont) raffiné(s) au quart de pixel, alors que les autres sont invalidés. Notons que dans cet implantation, de la décision est apportée, chose qui était laissée à l'encodeur vidéo dans les algorithmes précédents. L'estimateur fourni à la fois les modes inter à utiliser et les vecteurs correspondants.

Performances de compression Le nouvel algorithme est évalué en terme de performances de compression. Il est comparé à l'algorithme HDS réalisant le raffinement subpixélique pour chaque taille de bloc, et à une implantation à taille de bloc 16x16 fixe. Les résultats sont regroupés dans le tableau 5.9.

	Variation de débit à qualité constante (%)					
	Séquences SD			Séquences HD		
Implantation	Formula1	Raid1 maroc	Raid2 maroc	Seq1	Horses1	SpinCalendar
HDS $\frac{1}{4}$-pel	0	0	0	0	0	0
HDS $\frac{1}{4}$-pel allégé	0,1	2,5	2,4	0,3	1,9	4,9
HDS 16x16 $\frac{1}{4}$-pel	2	6	6,9	0,9	4,6	7

TAB. 5.9 – Performances d'encodage avec les estimateurs de mouvement à taille de bloc variable au quart de pixel.

Notre algorithme allégé fourni des performances proches de l'implantation complète. Une perte de performance jusqu'à 5% est toutefois mesurée. La perte de compression est toujours inférieure à celle induite par la taille de bloc fixe, ce qui est

une première validation de notre algorithme. L'algorithme de décision a été soumis à peut de tests pour être mis au point, il existe donc une marge d'amélioration possible pour augmenter les performances en analysant plus particulièrement les séquences difficiles.

Chronométrages Comme nous l'avons remarqué précédemment, le raffinement subpixélique est une opération très coûteuse en ressources de calcul. L'estimation de mouvement à taille de bloc variable amplifie d'autant plus la complexité. Le tableau 5.10 présente les temps d'exécution des différentes implantations sur DSP C6416 à 1 GHz pour une séquence 720p (uniquement l'opération d'estimation de mouvement du niveau de pleine résolution). La factorisation est au niveau macro-bloc afin d'obtenir les meilleures performances possibles.

Implantation	Temps d'exécution
HDS $\frac{1}{4}$-pixel	113 ms
HDS $\frac{1}{4}$-pixel allégé	55 ms
HDS 16x16 $\frac{1}{4}$-pixel	30 ms

TAB. 5.10 – Chronométrages des estimateurs de mouvement à taille de bloc variable et au $\frac{1}{4}$ pixel, par image 720p (1280x720)

L'algorithme HDS allégé permet de réduire le temps d'exécution de l'estimation de mouvement de 113 ms à 55 ms. La moitié des ressources matérielles sont ainsi économisées. L'algorithme de décision permet d'éliminer une grande partie des calculs inutiles. A titre de comparaison, le temps d'exécution de l'estimation de mouvement pour une taille de bloc 16x16 fixe n'est que de 30 ms.

L'estimation de mouvement à taille de bloc variable augmente considérablement la complexité de l'encodeur vidéo, et ce d'autant plus que la précision des vecteurs est au quart de pixel, nécessitant des calculs coûteux pour chaque taille de bloc. La réduction des calculs pour la précision pixel en utilisant une méthode récursive, associé à un algorithme de décision permet de réduire notablement la complexité. La qualité des champs de vecteurs pour la compression vidéo s'en trouve par conséquent diminuée dans une moindre mesure. Le développement de l'algorithme de sélection du mode à raffiner laisse de la place à l'amélioration, notamment pour la bi-prédiction. Le cas des images B n'a pas été traité. Même si, dans ce cas, l'estimation de mouvement est possible indépendemment sur chacune des images de référence, le choix du mode pour un macro-bloc donné doit être le même sur chacune des références pour pouvoir profiter de la bi-prédiction.

5.4.3 Bilan des optimisations algorithmiques spécifiques pour H.264

Nous avons décrit dans les paragraphe précédent les optimisations liées à l'estimation de mouvement en général. Un nouvel algorithme hiérarchique (HDS) a été développé en remplaçant la recherche exhaustive locale de l'algorithme classique (HME) par une technique récursive en diamant. La quantité de calcul est ainsi réduite avec une faible perte de qualité. Les techniques d'optimisation de code utilisées sur DSP ont été décrites pour améliorer les performances mono-processeur.

Dans ce paragraphe, nous avons décrit les implantations des opérations d'estimation de mouvement spécifiques à la compression vidéo H.264. L'augmentation de la précision des vecteurs de mouvement au quart de pixel améliore considérablement les performances de compression mais introduit un coût de calcul supplémentaire important. Grâce à une méthode prédictive avec un motif de recherche réduit, le coût de calcul supplémentaire des techniques d'estimation de mouvement à taille de bloc variable pixel entier est réduit avec un impact négligeable sur les performances de compression. Le raffinement subpixélique associé à la taille de bloc variable multiplie le coût de calcul supplémentaire. Comme ces deux techniques sont nécessaires aux performances de H.264, nous avons développé un algorithme simple de décision permettant de sélectionner le mode inter à partir des champs de vecteurs à la précision pixel entier. Les vecteurs de mouvement sont raffinés pour seule taille de bloc réduisant ainsi la charge de calcul.

Afin de valider les choix d'implantation effectués jusqu'à maintenant, plusieurs séquences sont compressées en augmentant les modes de prédiction. La période des images I est de 30, celle des images P est de 1 avec une image de référence (pas de B pour le moment). Pour les images P, les modes inter 16x16, 16x8, 8x16, 8x8 et skip sont activés ainsi que les modes intra. Cela permet de s'assurer que les modifications de l'algorithme d'estimation de mouvement sont justifiées avec un ensemble d'outils de l'encodeur plus complet. Les performances de compression sont regroupés dans le tableau 5.11 et les temps d'exécution sur mono-DSP dans le tableau 5.12.

	Variation de débit à qualité constante (%)					
	Séquences SD			Séquences HD		
Implantation	Formula1	Raid1 maroc	Raid2 maroc	Seq1	Horses1	SpinCalendar
HME	0	0	0	0	0	0
HDS $\frac{1}{4}$-pel allégé	3,7	2,3	2,1	0,7	1,8	4,9
HDS 16x16 $\frac{1}{4}$-pel	5	5,2	5,3	1,1	4,3	6,9

TAB. 5.11 – Comparaison des performances d'encodage entre l'estimateur de mouvement initial et l'implantation optimisée finale.

Les pertes introduites par les choix d'implantation en terme de performances de compression sont réduites mais peuvent aller jusqu'à 5 %. Il ne faut toutefois pas considérer ce résultat comme définitif, en effet la désactivation des images B ne permet pas d'avoir les performances finales car elles permettent d'améliorer considérablement la compression.

Sur un des DSP les plus performants du marché, l'algorithme initial avec des optimisation d'implantation poussées nécessite plus de 140 ms pour calculer les champs de vecteurs entre deux images 720p. Les optimisations algorithmiques effectuées, telles que l'algorithme hiérarchique HDS pour les champs de vecteurs 8x8, la technique récursive avec un diamant réduit pour les autres tailles de bloc, et la décision du mode à raffiner au quart de pixel, permettent de réduire ce temps d'exécution jusqu'à 65 ms. Cependant, la spécification temps réel la plus proche est de 20 ms pour le format 720p50, soit 3.25 fois moins. Nous pouvons conclure qu'un seul DSP n'est pas suffisant pour réaliser l'estimation de mouvement temps réel pour la compression vidéo H.264 en haute définition.

Implantation	Temps d'exécution		
	HME	HDS	HDS $\frac{1}{4}$-pel allégé
Pyramide 4 niveaux	2,4 ms	2,4 ms	2,4 ms
Padding 5 niveaux	2,7 ms	2,7 ms	2,7 ms
Estimation de mouvement 8x8 niveaux 4 à 1	6,2 ms	4,2 ms	4,2 ms
Estimation de mouvement niveau 0 (taille de bloc variable et $\frac{1}{4}$ pixel)	130 ms	113 ms	55 ms
Total image 1280x720	141,3 ms	122,3 ms	64.6 ms

TAB. 5.12 – Synthèse des chronométrages sur DSP, par image 720p (1280x720)

5.5 Conclusion

Dans ce chapitre, nous avons présenté les étapes de vérification fonctionnelle, d'implantation et d'optimisation monoprocesseur. L'utilisation de la méthodologie de prototypage a facilité le portage sur cible. La chaîne de développement, en particulier la migration progressive des opérations et la vérification fonctionnelle, s'est montrée efficace dans ces travaux. La vérification a ainsi été simplifiée assurant des optimisations fiables. Les communications et synchronisations entre PC et DSP ont été générées automatiquement, offrant la possibilité d'évaluer aisément et rapidement plusieurs solutions d'implantation. Nous avons étudié les performances des algorithmes d'estimation de mouvement HME et EPZS en termes de complexité et de qualité des champs de vecteurs dans un but de compression vidéo. L'analyse des caractéristiques de ces techniques a conduit à la conception d'une nouvelle solution, appelée HDS, combinant robustesse et faible complexité.

Le résultat est donc un estimateur de mouvement performant, optimisé et flexible, sur DSP comme sur PC. Cette technique est utilisée dans plusieurs projets :

- HVS (Human Vision System) du laboratoire *Compression* de Thomson dont le but est de modéliser l'attention visuelle (détecter automatiquement les zones d'intérêt de l'image) [MTCB05],
- l'encodeur vidéo embarqué de l'IETR, groupe image,
- LAR vidéo, qui consiste à étendre une méthode de compression d'images fixes scalable appelée LAR à la vidéo [FABD06].

L'optimisation de l'implantation sur DSP Texas Instruments C64 a été poussée, aussi bien en termes d'exploitation des unités de calculs qu'en termes d'accès mémoire. Toutefois, les optimisations restent évolutives dans le sens ou elles ont été réalisées avec un langage de haut niveau, de façon à pouvoir être adaptées par exemple à un autre modèle de processeur. Des optimisations plus spécifiques peuvent être effectuées, mais conduiraient à un temps de développement supplémentaire conséquent pour une faible amélioration et rendraient de fait l'implantation figée. Cette démarche n'a donc qu'un intérêt limité dans un contexte de recherche et pour la réalisation de prototypes.

La compression vidéo augmente la complexité de l'estimation de mouvement, notamment dans le cadre de H.264. Le traitement exhaustif de tous les modes possibles implique une puissance de calcul conséquente. Le raffinement subpixélique s'avère particulièrement complexe à cause des étapes d'interpolation et de mise en correspondance supplémentaires. Le contexte de haute définition augmente davantage les contraintes

et pose un problème de bande passante mémoire. En effet la quantité des données à traiter nécessite des bandes passantes élevées. L'évolution actuelle des techniques de compression vidéo accroît la complexité de l'estimation de mouvement et des encodeurs en général. Cette tendance se vérifie encore dans les futurs standards tels que MPEG-4 SVC.

L'estimation de mouvement pour la compression vidéo haute définition est trop complexe pour être traitée par un seul processeur, même pour un des DSP les plus puissants du marché. Nous avons donc naturellement cherché à paralléliser les opérations sur une plate-forme multicomposant afin d'augmenter la puissance de calcul disponible.

Chapitre 6

Estimation de mouvement sur plates-formes multicomposants

Dans le chapitre 5, nous avons vu que l'opération d'estimation de mouvement pour la compression vidéo haute définition nécessite une puissance de calcul conséquente. Un seul DSP ne peut pas réaliser cette tâche. Une architecture spécialisée telle celles présentées dans le chapitre 3 peut alors être développée. Cependant, le temps nécessaire au développement ne convient pas à un prototype, qui doit être fonctionnel rapidement. De plus, comme un prototype est souvent réalisé très tôt dans le processus de développement, la solution doit être évolutive, afin de prendre en compte les modifications éventuelles de l'application. Pour répondre au cahier des charges, l'architecture cible idéale doit donc être programmable et disposer d'une grande puissance de calcul. On peut noter également qu'à ce stade, le coût de la plate-forme entre moins en jeu que la complexité de programmation. La puissance de calcul est donc fournie en utilisant plusieurs composants. L'implantation est simplifié et accéléré en utilisant des outils de prototypage rapide (cf. chapitre 4).

Une première solution a été de développer un coprocesseur dédié afin d'accélérer l'opération la plus coûteuse en temps de calcul. Une deuxième solution a été d'étudier différentes techniques de parallélisation à l'aide des outils de prototypage. Nous verrons ici que la parallélisation sur multiprocesseur n'est pas triviale et qu'il est parfois nécessaire d'éliminer des dépendances de données entre opérations afin de faire apparaître du parallélisme potentiel. Cela a un impact sur le résultat, qu'il convient de minimiser. La distribution des opérations sur de multiples composants génère des transferts de données inévitables qui encombrent les liens de communication. De plus, l'accès mémoire étant pénalisant, de meilleures performances sont obtenues en groupant les opérations en fonction des données source, afin d'optimiser l'utilisation des mémoires caches. La solution a été d'effectuer la parallélisation en fonction des données plutôt qu'en fonction des opérations lorsque l'architecture cible est homogène. Toutefois, une opération donnée peut tirer partie des spécificités d'un type de processeur. La parallélisation est alors effectuée en fonction du type de traitement sur une architecture hétérogène.

Dans un premier temps nous allons donner un exemple de parallélisation sur une architecture hétérogène, ou les processeurs sont adaptées à chaque type d'opération.

Dans un deuxième temps nous allons présenter des solutions de parallélisme de données.

6.1 Réalisation d'un coprocesseur

Nous avons vu dans le chapitre 3 qu'il existait plusieurs types de processeurs. Chaque composant présente des caractéristiques matérielles différentes qui peuvent être mises à profit pour un type d'opération donné. Une plate-forme multiprocesseur hétérogène offre non seulement la possibilité de paralléliser les traitements, mais aussi les atouts des différentes architectures. Il est alors important de distribuer les opérations de l'application de manière à exploiter au mieux l'intérêt de chaque processeur.

Dans le chapitre 5, nous avons présenté une implantation optimisée d'un algorithme d'estimation de mouvement sur DSP. L'estimation de mouvement pixel entier est basée sur un algorithme prédictif afin de conserver une grande amplitude des vecteurs de mouvement et une quantité de calculs réduite. Il est donc conditionné et comporte des accès mémoire aléatoires. L'utilisation d'un processeur standard de type DSP est donc adapté.

Inversement, l'opération la plus coûteuse en termes de ressources matérielles est le raffinement subpixélique des vecteurs de mouvements. Malgré une fenêtre de recherche très réduite (amplitude inférieure à un pixel dans chaque direction), beaucoup de calculs sont effectués, notamment à cause de l'interpolation qui impliquent de nombreuses opérations simples (additions et décalages). L'augmentation de la résolution conduit également à une densité de candidats importante, et donc à une quantité de calculs de distorsion et une bande passante mémoire conséquentes. De plus, les calculs systématiques du raffinement subpixélique se prêtent bien à une implantation matérielle. Nous avons donc choisi de concevoir un coprocesseur dédié sur FPGA. Ce type de composant peut assurer une bande passante interne aussi grande que nécessaire : réaliser en parallèle les calculs dès que les données sont disponibles évite d'avoir à stocker les données temporaires.

Dans ce paragraphe nous décrivons le coprocesseur de raffinement subpixélique conçu. La taille de bloc est variable afin de gérer plusieurs modes possibles. La bande passante mémoire est réduite et adaptée au débit de l'interface externe d'un DSP. L'estimation de mouvement pixel entier (IME : Integer-pixel Motion Estimation) est réalisée sur DSP tandis que le raffinement subpixélique (FME : Fractional-pixel Motion Estimation) est accéléré sur FPGA. Dans un premier temps, l'architecture interne est décrite et justifiée, puis dans un deuxième temps, le prototype hétérogène est présenté.

6.1.1 Architecture interne du coprocesseur

Le but du coprocesseur de raffinement subpixélique est de réaliser les opérations d'interpolation et de mise en correspondance de manière efficace. Afin de respecter les contraintes de bande passante avec la mémoire externe, une approche exhaustive est retenue, avec une amplitude de recherche inférieure à un pixel. L'architecture est évolutive dans le sens où le parallélisme peut être augmenté (pour une plate-forme plus performante par exemple) et le filtre d'interpolation peut facilement être modifié (simplification par exemple). La taille du bloc à traiter est modifiable dynamiquement afin de conserver le maximum de flexibilité pour traiter les multiples tailles de bloc.

L'objectif recherche est d'obtenir une architecture atteignant de bonnes performances tout en conservant une bande passante mémoire réduite et avec peu de ressources logiques. Nous décrivons dans la suite l'architecture générale, puis chaque module individuellement.

6.1.1.1 Description générale

L'algorithme sélectionné doit être régulier afin d'exploiter le haut degré de parallélisme. Une approche à deux pas, telle que celle utilisée dans [CHC04] nécessite soit de sauvegarder les demi-pixels pour l'interpolation quart de pixel suivante soit de les recalculer pour éviter l'utilisation d'une mémoire intermédiaire [YGI06]. Dans ce dernier cas, les mêmes données pixel entier sont accédées deux fois de suite, doublant ainsi la bande passante mémoire nécessaire. De plus, comme le filtre est utilisé deux fois pour les mêmes données, son efficacité est donc réduite d'un facteur deux. La bande passante mémoire est un point limitant sur notre plate-forme, c'est pourquoi nous adoptons une recherche exhaustive avec une amplitude de $\frac{3}{4}$ pixel dans chaque direction. 48 candidats $\left((2 \times 3 + 1)^2 - 1\right)$ sont donc évalués autour du résultat pixel entier. Les données nécessaires en entrée du coprocesseur pour chaque vecteur sont la fenêtre de référence, le bloc courant, et optionnellement la valeur du vecteur codé en différentiel pour prendre en compte son coût à travers un coefficient de Lagrange.

Le coprocesseur est un pipeline composé d'un module d'interpolation, d'une matrice de Processeurs Elémentaires (PE), et d'un arbre de décision (Fig. 6.1). Les PE évaluent les distorsions pour chaque candidat. La mesure de distorsion retenue est la SAD car elle nécessite beaucoup moins de ressources qu'une SATD. La taille de bloc variable est supportée grâce à des paramètres de taille du bloc, modifiable dynamiquement.

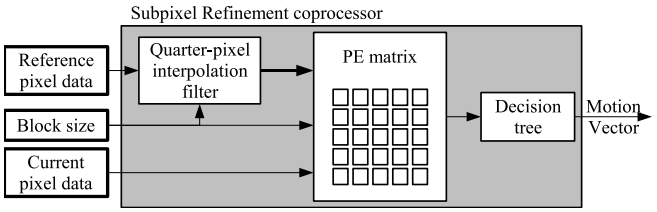


FIG. 6.1 – Description générale de l'architecture

Dans les paragraphes suivants, les unités de calculs sont décrites individuellement.

6.1.1.2 Filtre d'interpolation

Les données subpixéliques sont interpolées à la volée pour éliminer leur mémorisation et l'accès dans une mémoire. Les pixels de la fenêtre de référence sont entrés en série de gauche à droite et de haut en bas dans le filtre. La figure 6.2 décrit la fenêtre de recherche et les dépendances de données pour un bloc 4x4. A gauche, on peut voir les pixels de l'image de référence nécessaires à l'application du filtre d'interpolation, pour obtenir la fenêtre de recherche subpixel. Sans considérer les problèmes d'initialisation du filtre, chaque pixel en entrée permet de calculer seize échantillons quart de pixel, correspondant aux positions encadrées. Sur la droite, on voit la fenêtre de recherche

requise pour un pixel du bloc courant. Compte tenu des dépendances de données pour l'interpolation, cette fenêtre de recherche est divisée en quatre quadrants qui ne sont pas disponibles simultanément.

Dès que le filtre est initialisé (latence liée à la taille du filtre), seize échantillons quart de pixel sont générés pour chaque pixel en entrée (Fig. 6.2-gauche). Pour augmenter le parallélisme, r pixels en entrée sont traités simultanément par cycle d'horloge, de sorte que $16 \times r$ échantillons soient produits par cycle. Une machine d'état contrôle la largeur et la hauteur du filtre, ce qui assure une utilisation des ressources d'interpolation complète et sans recouvrement quelle que soit la taille du bloc à traiter.

Cycle d'horloge	Position des pixels en entrée	filtre 1/2 pel horizontal	filtre 1/2 pel Vertical	filtre 1/4 pel
1	(-3 ;-3) _ (-3 ;-2)	-	-	-
2	(-3 ;-1) _ (-3 ;0)	-	-	-
3	(-3 ;1) _ (-3 ;2)	(-3 ;-1) _ (-3 ;0)	-	-
4	(-3 ;3) _ (-3 ;4)	(-3 ;0.5) _ (-3 ;2)	-	-
5	(-3 ;5) _ (-3 ;6)	(-3 ;2.5) _ (-3 ;4)	-	-
⋮	⋮	⋮		
⋮	⋮	⋮	-	-
21	(1 ;-3) _ (1 ;-2)	-	-	-
22	(1 ;-1) _ (1 ;0)	-	-1 ;-1	-
23	(1 ;1) _ (1 ;2)	(1 ;1) _ (1 ;0)	(-1 ;-0.5) _ (-1 ;0)	-
24	(1 ;3) _ (1 ;4)	(1 ;0.5) _ (1 ;2)	(-1 ;0.5) _ (-1 ;2)	-
25	(1 ;5) _ (1 ;6)	(1 ;2.5) _ (1 ;4)	(-1 ;2.5) _ (-1 ;4)	-
26	(2 ;-3) _ (2 ;-2)	-	-	-
27	(2 ;-1) _ (2 ;0)	-	(-0.5 ;-1) _ (0 ;-1)	-
28	(2 ;1) _ (2 ;2)	(2 ;1) _ (2 ;0)	(-0.5 ;-0.5) _ (0 ;0)	(-0.75 ;-0.75) _ (0 ;0)
29	(2 ;3) _ (2 ;4)	(2 ;0.5) _ (2 ;2)	(-0.5 ;0.5) _ (0 ;2)	(-0.75 ;0.25) _ (0 ;2)
30	(2 ;5) _ (2 ;6)	(2 ;2.5) _ (2 ;4)	(-0.5 ;2.5) _ (0 ;4)	(-0.75 ;2.25) _ (0 ;4.75)
31	(3 ;-3) _ (3. ;-2)	-	-	-
32	(3 ;-1) _ (.3 ;0)	-	(0.5 ;-1) _ (1 ;-1)	-
33	(3 ;1) _ (3 ;2)	(3 ;1) _ (3 ;0)	(0.5 ;-0.5) _ (1 ;0)	(0.25 ;-0.75) _ (1 ;0)
34	(3 ;3) _ (3 ;4)	(3 ;0.5) _ (3 ;2)	(0.5 ;0.5) _ (1 ;2)	(0.25 ;0.25) _ (1 ;2)
35	(3 ;5) _ (3 ;6)	(3 ;2.5) _ (3 ;4)	(0.5 ;2.5) _ (1 ;4)	(0.25 ;2.25) _ (1 ;3.75)
⋮	⋮	⋮	⋮	⋮
46	(6 ;-3) _ (6 ;-2)	-	-	-
47	(6 ;-1) _ (6 ;0)	-	(3.5 ;-1) _ (4 ;-1)	-
48	(6 ;1) _ (6 ;2)	(6 ;1) _ (6 ;0)	(3.5 ;-0.5) _ (4 ;0)	(3.25 ;-0.75) _ (3.75 ;0)
49	(6 ;3) _ (6 ;4)	(6 ;0.5) _ (6 ;2)	(3.5 ;0.5) _ (4 ;2)	(3.25 ;0.25) _ (3.75 ;2)
50	(6 ;5) _ (6 ;6)	(6 ;2.5) _ (6 ;4)	(3.5 ;2.5) _ (4 ;4)	(3.25 ;2.25) _ (3.75 ;3.75)

TAB. 6.1 – Flot de données du filtre d'interpolation H.264 pour un bloc 4x4 ($r = 2$)

Le tableau 6.1 montre le flot de données du filtre d'interpolation H.264. Le point de référence (position 0 ;0) des plages citées est le pixel en haut à gauche du bloc central de la fenêtre de référence. Pour chaque plage, on donne le point supérieur gauche et le point inférieur droit (ex : $(x_1; y_1)$ à $(x_2; y_2)$). A chaque cycle, deux pixels de la fenêtre de référence sont entrés ($r = 2$). Les cycles d'initialisation, correspondant à la latence due à la taille des filtres, sont représentés par des "-". Après que les six premiers pixels soient entrés, soit trois cycles, le filtre demi-pixel horizontal est activé. Le filtre vertical quant à lui nécessite des données sur six lignes. Il est donc activé lorsque la sixième ligne est entrée (cycle 27). Les premiers échantillons sortis correspondent à la ligne supérieure de la fenêtre de recherche et ne nécessitent qu'une interpolation horizontale, ils peuvent donc être produits dès la cinquième ligne (cycle 22). Les échantillons quart de pixel sont calculés après que la sixième ligne soit disponible en entrée du filtre (cycle 28).

Les filtres sont réalisés avec des opérations de décalage et d'addition, pipelinées pour ne pas limiter la fréquence de fonctionnement. Afin de ne pas surcharger le tableau, la latence due à la mise en pipeline des opérations n'est pas répercutée. Il s'agit de deux et trois cycles pour le filtre demi-pixel horizontal et vertical respectivement, et de un cycle pour le filtre quart de pixel. Les données en sortie correspondent à chaque cycle d'horloge à deux rectangles encadrés en rouge sur la figure 6.2-gauche.

Comme la fenêtre de recherche est plus large que la taille d'un bloc, les données sur les bords ne sont pas utiles pour toutes les SAD. Par exemple pour les blocs 4x4, seulement seize des vingt-cinq rectangles sont utiles à chaque SAD. Les PE devrons donc être activés lorsque des données appropriées sont disponibles. Cela est effectué en quatre quadrants visibles sur la figure 6.2-droite.

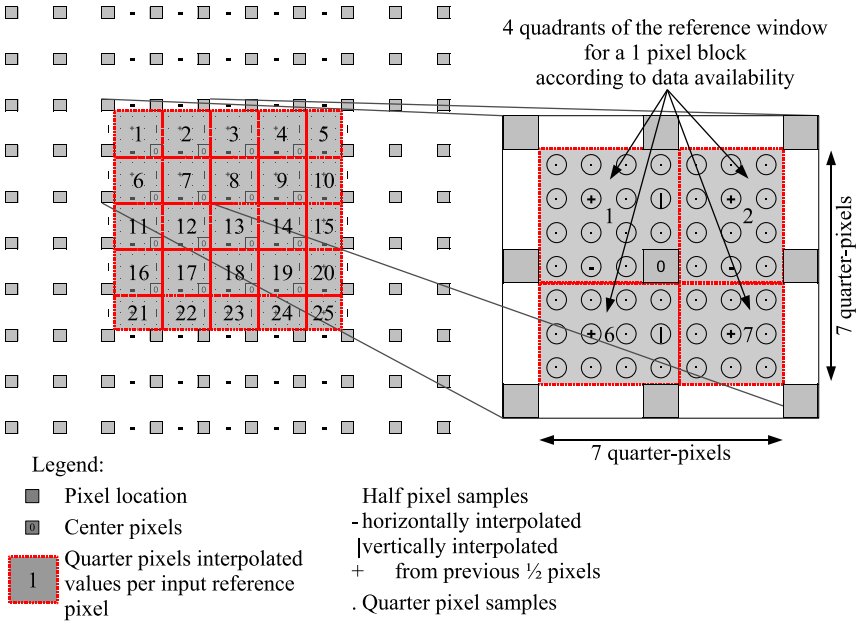


FIG. 6.2 – Disponibilité des données pour les échantillons quart de pixel H.264 avec une taille de bloc 4x4 (gauche) et les dépendances de données pour le calcul des SAD (droite)

La conception du reste de l'architecture tient compte de l'ordonnancement des données issues du filtre d'interpolation afin d'augmenter l'efficacité et de réduire les ressources nécessaires. D'autres filtres d'interpolation que celui de la norme H.264 peuvent être utilisés, par exemple un filtre bilinéaire pour réduire la complexité. Dans la suite, le filtre H.264 est conservé pour garder des performances optimales en termes de qualité. L'utilisation du même filtre qu'au décodeur favorise la minimisation des résidus de compensation de mouvement.

6.1.1.3 Matrice de processeurs élémentaires

De nombreux travaux [DS89, CCH+06, YH95] traitent de l'implantation d'estimateurs de mouvement pixel entier sur des architectures dédiées. Ces travaux ont montré que les architectures utilisant le parallélisme inter-candidat (cf. paragraphe 3.2) sont les

mieux adaptées à une entrée en scan linéaire et minimisent les ressources logiques nécessaires, pour une fenêtre de recherche réduite. En effet, ceci favorise la réutilisation des données entre les calculs de SAD pour des candidats voisins. Le nombre de registres est donc réduit. De plus, le nombre de PE n'étant pas lié à la taille du bloc traité (comme dans le cas intra-candidat), nous pouvons envisager de faire varier le taille du bloc dynamiquement. Nous proposons ici d'adapter la matrice de calcul des distorsions au raffinement subpixélique pour réduire les ressources matérielles, en particulier les registres de propagation des données. Le flot de données du filtre d'interpolation est également pris en compte afin d'optimiser l'architecture de façon globale. Ce choix a pour effet de réduire également les contraintes sur l'arbre de décision.

Modèle de recherche exhaustive pixel entier L'algorithme IME exhaustif avec une amplitude de +/- p pour un bloc de taille $M \times N$ est exprimé avec quatre boucles imbriquées (cf. paragraphe 3.2).

Le parallélisme inter-candidat est obtenu en déroulant les boucles Δi et Δj (Alg. 3.2) en matériel. Les distorsions sont calculées simultanément pour chaque candidat dans $(2 \times p + 1)^2$ PE. Ce modèle résulte de deux hypothèses :

- le pixel courant $x_1\,(k, l)$ est diffusé à tous les PE,
- tous les pixels de référence $x_2\,(k - p, l - p)$ à $x_2\,(k + p, l + p)$ sont disponibles et propagés vers les PE à travers $(2p + 1)$ registres.

De façon à éliminer les registres nécessaires à la propagation des données de référence, l'architecture est inversée. Les données de référence sont diffusées vers tous les PE et le bloc courant est propagé. L'algorithme est transformé avec un changement de variables ($u = k + \Delta i$ et $v = l + \Delta j$) (Alg. 6.1). Cette modification a peu d'impact sur IME : la latence est la même et les cycles d'initialisation des registres de propagation [YH95] sont toujours nécessaires. Cependant, cela devient intéressant lorsque l'algorithme est transposé au quart de pixel.

Algorithme 6.1 Algorithme IME exhaustif inversé

$for\, u = 1 - p..M + p$ (hauteur du bloc)	intra candidat
$for\, v = 1 - p..N + p$ (largeur du bloc)	
$for\, \Delta i = -p..p$ (amplitude verticale)	inter candidat
$for\, \Delta j = -p..p$ (amplitude horizontale)	
$SAD(\Delta j, \Delta i) += \lvert x_1(u - \Delta i, v - \Delta j) - x_2(u, v)\rvert$	
$end\, \Delta j$	
$end\, \Delta i$	
$end\, v$	
$end\, u$	

avec $1 \leq u - \Delta i \leq M$ et $1 \leq v - \Delta j \leq N$

Modèle de recherche quart de pixel Pour la recherche subpixélique, l'image de référence x_2 n'a pas la même échelle que l'image courante x_1. La densité des données est augmentée pour la fenêtre de référence. Par conséquent, lorsque les données de

la fenêtre de référence sont propagées, le nombre de registres nécessaire à leur propagation augmente exponentiellement avec la précision [DRS05] $((2ap+1)^2$ avec a le facteur de précision, c'est à dire $a = 2$ pour le demi-pixel et $a = 4$ pour le quart de pixel) .

Pour réduire le nombre de registres de propagation, l'algorithme 6.1 est considéré et transposé au quart de pixel (Alg. 6.2). Par conséquent le bloc courant est propagé à l'aide de registres et la fenêtre de référence est diffusée vers les PE de manière entrelacée. En effet, chaque donnée quart de pixel est diffusée vers un PE sur quatre en horizontal et en vertical. Les deux boucles Δg et Δh sont ajoutées pour exprimer l'entrelacement. L'amplitude de recherche est inférieure à un pixel dans chaque direction. L'algorithme 6.2 modélise le fonctionnement de la matrice de PE. Δi et Δj sont les coordonnées entières du candidat tandis que Δg et Δh sont les coordonnées fractionnaires. Le nombre de registres de propagation est ainsi indépendant de la précision. Il est lié à la largeur d'un bloc, puisque seul le bloc courant est propagé.

Algorithme 6.2 Algorithme FME inversé

```
for u = 1..M + 1 (hauteur du bloc courant)
 for v = 1..N + 1 (largeur du bloc courant)
  for Δi = 0..1(amplitude verticale entière)
   for Δj = 0..1(amplitude horizontale entière)
    for Δg = −(a − 1)..0 (amplitude verticale fractionnaire)
     for Δh = −(a − 1)..0 (amplitude horizontale fractionnaire)
      SAD(Δj, Δi)+ = |x1(u − Δi, v − Δj)
            −x2(au + Δg, av + Δh)|
     end Δh
    end Δg
   end Δj
  end Δi
 end v
end u
```

avec $-(a-1) \leq a\Delta i + \Delta g \leq (a-1)$ et $-(a-1) \leq a\Delta j + \Delta h \leq (a-1)$
et $1 \leq u - \Delta i \leq M$ et $1 \leq v - \Delta j \leq N$

Les boucles Δi, Δj, Δg et Δh sont déroulées en matériel (par exemple avec une matrice de PE 7×7 pour le quart de pixel). Les échantillons $x_2\,(ak-p, al-p)$ à $x_2\,(ak, al)$ sont diffusés vers les PE appropriés et les pixels de référence $x_1(k,l)$ à $x_1(k-s, l-s)$ sont propagés.

Les inégalités $1 \leq u - \Delta i \leq M$ et $1 \leq v - \Delta j \leq N$ sont assurées matériellement en propageant un signal d'activation des PE en même temps que les données du bloc courant. Les PE sont ainsi activés par quadrant du fait des dépendances de données, en fonction de la position du candidat (Fig. 6.2-droite). Par conséquent, tous les résultats ne sont pas finis de calculer au même instant. Les a^2 premiers coûts sont disponibles après $M\,(N+1)$ cycles, et les suivants après les délais nécessaires. Cela permet de réduire la taille de l'arbre de comparaison dont les ressources peuvent être partagées dans le temps sans introduire de délai supplémentaire.

La figure 6.3 présente la matrice de calcul des SAD. Pour clarifier la figure, seulement une matrice correspondant au demi-pixel est présentée. ($a = 2$ et $p = \frac{1}{2}$) et $r = 1$. Il existe un PE par candidat. La matrice reflète donc la fenêtre de référence.

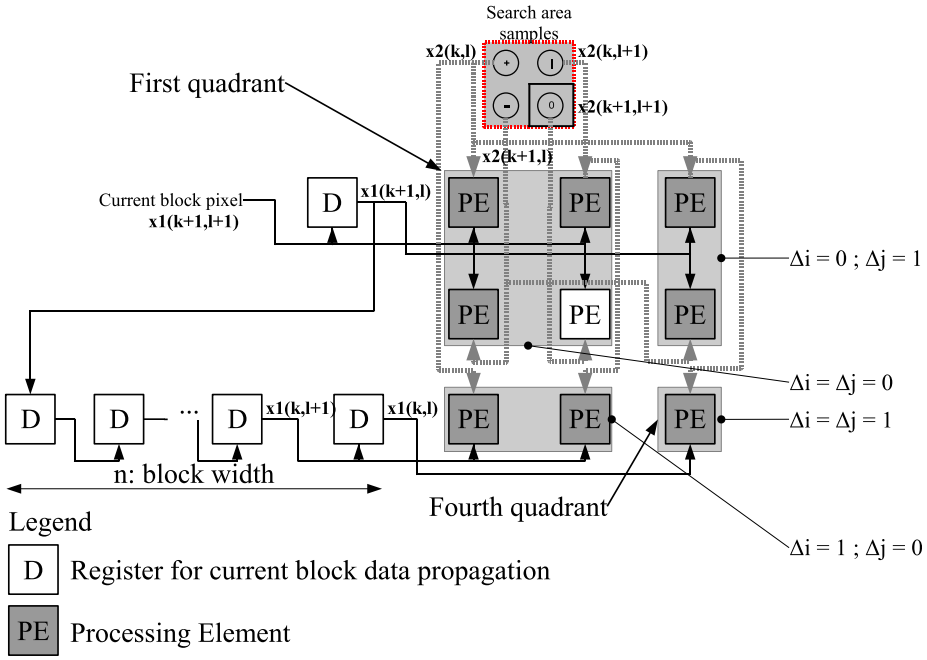


FIG. 6.3 – Matrice de PE ($a = 2$, $p = \frac{1}{2}$ et $r = 1$)

La fenêtre de référence est partitionnée en quatre quadrants en fonction de la disponibilité des données. Chaque quadrant représente le déroulement des boucles Δg et Δh de l'algorithme 6.2. Les quatre quadrants sont le résultat du déroulement des boucles Δi et Δj. Le quadrant du haut à gauche est activé en premier, celui à droite est retardé d'un cycle, et ceux du bas du délai correspondant au calcul d'une ligne (dépendant de la taille du bloc). Les cycles vides correspondant au remplissage des registres de propagation sont éliminés de l'étape d'initialisation. A la place, les résultats sont partitionnés en fonction de la disponibilité des données, offrant la possibilité de réduire la taille de l'arbre de décision.

Augmentation du parallélisme Cette architecture utilise pleinement le parallélisme inter-candidat (un PE par candidat), et supporte de plus le parallélisme intra-candidat (ex $r = 2$ pixels par cycle). Les performances peuvent être accrues en déroulant la boucle Δv de l'algorithme 6.2 d'un facteur r. A la fois le filtre d'interpolation et les PE sont affectés. Le détail d'un PE présenté sur la figure 6.4 correspond à $r = 2$. A chaque cycle, deux différences absolues sont calculées et accumulées. Les performances sont donc doublées avec cependant un coût supplémentaire en ressources logiques (les unités d'interpolation et la matrice de PE s'agrandissent).

Afin de considérer l'optimisation débit/distorsion, l'accumulateur de SAD peut être initialisé avec une estimation du coût du vecteur pondéré par un coefficient de Lagrange. Cette valeur est calculée avec peu de ressources matérielles lors de l'initialisation du pipeline du filtre d'interpolation.

Un haut degré de parallélisme est obtenu avec une bande passante mémoire réduite ($r = 2$ pixels par cycle pour l'image de référence et l'image courante). L'architec-

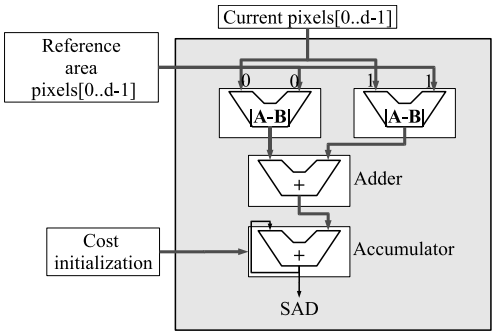


FIG. 6.4 – Détail d'un PE avec $r = 2$

ture est dimensionnable dans le sens où les performances peuvent être améliorées en augmentant le parallélisme intra-candidat si les ressources disponibles le permettent. Les ressources logiques sont relativement indépendantes de la taille du bloc à traiter, puisque seulement les registres de propagation sont impliqués. La taille de bloc variable est donc supportée sans modification matérielle.

6.1.1.4 Arbre de décision

Le vecteur finalement raffiné est choisi en comparant les distorsions calculées par la matrice de PE dans un arbre de décision (Fig. 6.5). Les valeurs disponibles simultanément sont comparées dans une structure d'arbre binaire. Le plus faible des coûts issus de chaque quadrant est séquentiellement comparé à l'optimum courant. Le vecteur ainsi que son coût sont finalement disponibles après que les coûts du quatrième quadrant soient traités.

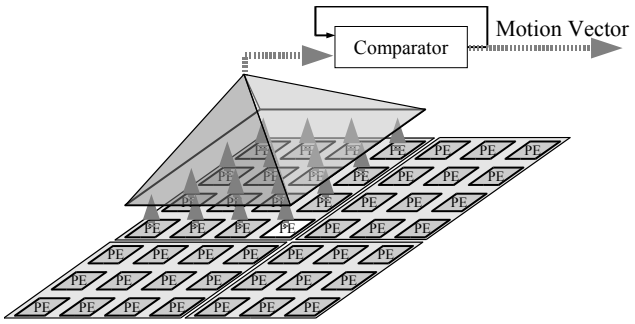
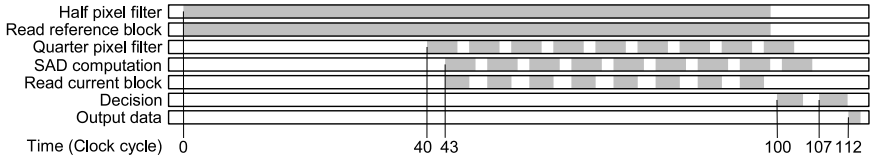


FIG. 6.5 – Arbre de décision

La taille de la base de l'arbre de décision est limitée à a^2 grâce à l'élimination des cycles d'initialisation dans la matrice de PE. Ses ressources peuvent donc être partagées sans introduire de délai supplémentaire. Au contraire, comme le nombre de données est moins grand, le pipeline est moins long.

6.1.1.5 Résultats d'implantation

L'ordonnancement des unités de l'architecture est donné sur la figure 6.6 pour un bloc de taille 8x8 avec $r = 2$.



(a) H.264 filter

FIG. 6.6 – Ordonnancement des unités pour un bloc 8x8 avec $r = 2$

Les cycles d'initialisation du filtre d'interpolation empêchent d'utiliser l'unité de calcul de SAD très efficacement. Il est possible de les réduire mais en faisant un compromis sur qualité de l'interpolation par exemple avec un filtre bilinéaire, ou bien en modifiant le filtre sur les bords de la fenêtre afin d'éliminer une partie des cycles d'initialisation.

Le nombre de cycles d'horloge nécessaire au raffinement quart de pixel d'un vecteur de bloc 8x8 est donné dans le tableau 6.2 en fonction du parallélisme intra r. La latence correspond au temps nécessaire à calculer tous les échantillons requis en fonction du débit d'entrée, auquel il faut ajouter le nombre d'étages du pipeline global. C'est donc une fonction du type $\frac{(M+6)\times(N+6)}{r} + l$, où l est le nombre d'étages de pipeline. La configuration $r = 2$ résulte en une latence de 112 cycles pour raffiner une vecteur 8x8 au quart de pixel avec le filtre H.264.

Largeur d'entrée	$r = 1$	$r = 2$	$r = 4$
Latence	209	112	70

TAB. 6.2 – Latence de l'implantation FPGA pour un bloc 8x8 (cycles)

Pour obtenir des performances temps réel pour une séquence 720p60, un bloc 8x8 doit être traité en moins de $\left(\frac{720\times1280\times60}{8\times8}\right)^{-1} = 1157\,ns$, soit 154 cycles avec une fréquence de 133 MHz. Le paramètre $r = 2$ est retenu pour atteindre cette contrainte, avec un fréquence de fonctionnement et des ressources logiques raisonnables.

Le tableau 6.3 donne des résultats de synthèse et une comparaison avec deux implantations hautes performances conçues par Chen [CHC04] et Yang [YGI06]. Le coprocesseur de raffinement subpixélique a été synthétisé pour un FPGA Xilinx Virtex II pro (XC2VP20). Environ 6000 LUT (Look Up Tables) à quatre entrées sont nécessaires, soit un peu moins de la moitié du composant, et la fréquence maximale est annoncée à 150 MHz. Une valeur supérieure peut être attendue sur un composant plus récent, ou de manière encore plus significative sur un ASIC. La conception a été réalisée en VHDL et brièvement optimisée. Les résultats des travaux de Chen et de Yang sont tirés de leurs pubications [CHC04, YGI06].

La bande passante mémoire des données source de deux pixels par cycle et par image est un atout majeur pour permettre la connexion à travers un bus à bande passante réduite. Par exemple, un DSP avec un bus de 32 bits à 133 MHz peut

	Travaux de Chen	Travaux de Yang	Ces travaux
Ressources logiques (Kgates)			
H.264 filter	23.9	119	14.5
Matrice de PE	34.8	41.1	54
Décision	2.2	8.5	10.4
Total	79.3	188	94
Bande passante mémoire (pixels/cycle)			
bloc courant	4	16	2
bloc ref	10	22	2
Cycles par bloc			
4x4	10x2	5x2	64
8x8	28x2	14x2	112
16x16	88x2	22x2	498

TAB. 6.3 – Comparaison avec des travaux précédents

fournir les données pour assurer un fonctionnement continu. Le coprocesseur cadencé à 133 MHz raffine un vecteur 8x8 au quart de pixel en 840 ns (112 cycles). Le débit maximal est donc de 82 images par secondes pour une séquence 720p (sans prendre en compte les transferts de données).

La bande passante avec l'extérieur est réduite, comparée aux travaux précédents (4 pixels par cycle comparé à 14 [CHC04] et 38 [YGI06]). De plus, les données ne sont accédées qu'une seule fois au lieu de deux grâce au raffinement exhaustif. Les performances de l'architecture proposée, pour une taille de bloc donnée sont inférieures à celles des solutions de Chen et de Yang. Cependant, le fonctionnement avec un DSP permet d'implanter un algorithme de décision et de rendre la recherche exhaustive de toutes les tailles de bloc inutile. Les paramètres de taille de bloc peuvent être modifiés dynamiquement. L'architecture de Chen se base sur une taille de bloc 4x4, résultant en des calculs redondants (recouvrements). L'architecture de Yang élimine cette redondance avec des unité de calculs dédiés pour des blocs de 16 pixels de large, résultant en une utilisation sous-optimale pour les tailles inférieures. Notre architecture prend en compte la taille de bloc variable et les unités de calcul sont dynamiquement configurables pour obtenir le maximum de performances.

Bien que la recherche exhaustive soit efficace pour réduire les transferts de données et le temps de calcul, plus de distorsions sont calculées par rapport à une approche à deux pas, résultant en une augmentation des ressources logiques dans la matrice de PE. Afin de limiter ceci, il est possible de réduire le nombre de candidats considérés en réduisant l'amplitude de recherche, ou encore en adoptant un motif de recherche particulier (par exemple circulaire).

Cette architecture pour le raffinement subpixélique des vecteurs de mouvement réduit les contraintes matérielles tout en donnant des performances optimales. La matrice de PE et l'arbre de comparaison prennent en compte les dépendances de données avec le filtre d'interpolation. Dès que les données sont disponibles, les SAD entre les échantillons quart de pixel et le bloc courant sont calculées. De hautes performances sont atteintes avec des ressources logiques réduites. Elle est dimensionnable dans le

sens où le parallélisme peut être augmenté si le composant cible le permet. Ceci est particulièrement adapté dans notre phase de prototypage.

Le paragraphe suivant décrit le prototype global d'estimateur de mouvement pour H.264 réalisé.

6.1.2 Estimateur de mouvement hétérogène

L'estimation de mouvement est une opération demandant beaucoup de bande passante mémoire et de puissance de calcul. Sur DSP, les algorithmes rapides mettent à profit les capacités à faire des branchements conditionnels et des accès aléatoires à la mémoire, ce qui est difficile à mettre en oeuvre sur une architecture câblée. L'estimation de mouvement pixel entier (IME) atteint de bonnes performances sur un DSP. Par contre, Le raffinement subpixélique (FME), travaillant sur une fenêtre de recherche très réduite, est plus coûteux (cf. paragraphe 5.4). La quantité de données source réduite et les opérations régulières (interpolation et calcul de distorsion) permettent de tirer parti d'une architecture câblée. Une implantation FPGA peu coûteuse permet d'obtenir de meilleures performances qu'un DSP. L'opération d'interpolation décuple les données en interne, et permet d'atteindre un parallélisme élevé avec une faible bande passante en entrée.

Nous présentons dans ce paragraphe un prototype d'estimateur de mouvement hétérogène où IME est exécuté sur un DSP et FME est accéléré sur un FPGA, fonctionnant comme un coprocesseur. Les calculs sont mis en pipeline par bloc afin de les paralléliser.

6.1.2.1 Plate-forme de prototypage

Le matériel utilisé est une plate-forme de prototypage *Sundance* (Fig 5.8) équipée d'un module *SMT395* (DSP Texas Instrument C6416 à 1 GHz avec un FPGA Xilinx Virtex II Pro XC2VP20). Le FPGA gère les transferts entre le DSP et le monde extérieur. Il n'est pas utilisé à 100% et permet donc l'implantation de fonctions dédiées. Le FPGA est branché sur le bus mémoire externe du DSP avec une largeur de 32 bits et une fréquence de 133 MHz. Afin de personnaliser le FPGA, le fabriquant fournit les sources du programme du FPGA (firmware) [Sun]. Cela concerne les liens de communications déjà implantées. La figure 6.7 présente le schéma bloc du FPGA.

On retrouve plusieurs types de blocs :

- le *Processor block* gère l'interface entre le DSP et le FPGA, c'est lui qui implémente le protocole du bus externe du DSP, décode les adresses et génère les interruptions,
- un *Connector block* interface le FPGA avec un élément extérieur (Bus PCI, autre FPGA, LED, ...), il gère le protocole du lien de communication,
- un *Interface block* permet de faire le lien entre les deux précédents blocs.

Nous avons choisi d'intégrer nos développements au niveau d'un *interface block* afin de réutiliser au maximum les éléments déjà existants. Ce bloc est donc modifié pour prendre en compte les spécificités de l'application et devient un *User block*, auquel nous connectons le coprocesseur proprement dit. Ceci permet d'utiliser simplement les développements existants. En effet, sur les plates-formes de type *Sundance*, les communications inter-processeurs sont réalisées par l'intermédiaire de FIFO sur le

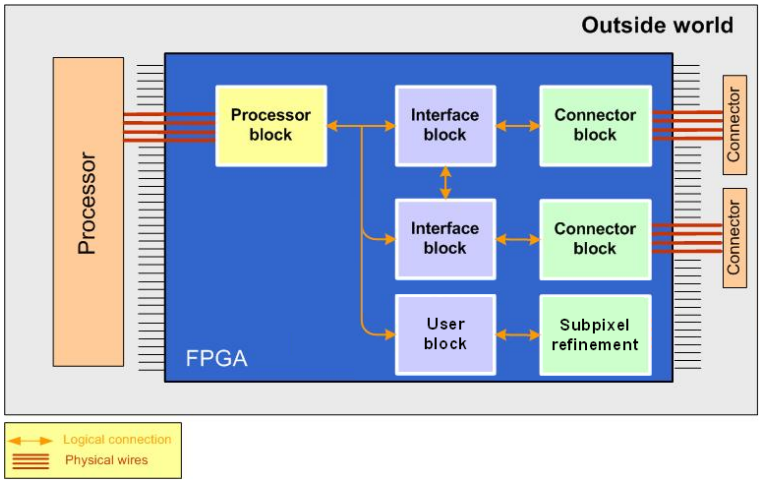


FIG. 6.7 – Schéma général du Firmware Sundance

FPGA. Dans le cadre des bibliothèques de communication (noyaux SynDEx, cf. paragraphe 4.1.6), des fonctions de communication permettant d'envoyer et de recevoir des données sur le DSP, ont donc déjà été développées. Du point de vue du DSP, il faut envoyer les données dans une FIFO, comme pour une communication avec l'extérieur, et se mettre en attente du résultat. Du point de vue du coprocesseur, dès qu'une données est reçue dans le *User block*, celui-ci est activé et et les données sont traitées. Le résultat est ensuite renvoyé vers le DSP en fin de calcul.

6.1.2.2 Parallélisation des opérations

L'estimation de mouvement est basée sur un algorithme prédictif. Les vecteurs de mouvements déjà calculés sont donc nécessaires pour prédire le mouvement courant ainsi que pour calculer le coût du vecteur. Cela crée des dépendances de données entre IME et FME, ce qui résulte inévitablement en une exécution séquentielle. Pour exploiter le parallélisme de la plate-forme et utiliser les composants de manière efficace il est nécessaire d'extraire du parallélisme de l'application, sans introduire de perte de performances de compression. Pour cela il est possible de modifier les dépendances de données pour créer un pipeline au niveau bloc composé de deux étages : IME et FME. Le vecteur du bloc de gauche est donc entré dans l'étage IME à la précision pixel au lieu du quart de pixel. Il est donc possible de paralléliser IME sur le bloc suivant avec FME sur le bloc courant (Fig. 6.8).

Le pipeline au niveau bloc permet donc de réaliser le raffinement des vecteurs au quart de pixel de manière transparente sur FPGA. L'impact de la modification des dépendances sur les résultats et de la latence due à la taille du pipeline sont négligeables au niveau image.

6.1.2.3 Méthode de développement

Le coprocesseur est utilisé de manière transparente avec l'outil de prototypage. En effet certaines limitations nous empêchent à l'heure actuelle de bien exploiter le co-

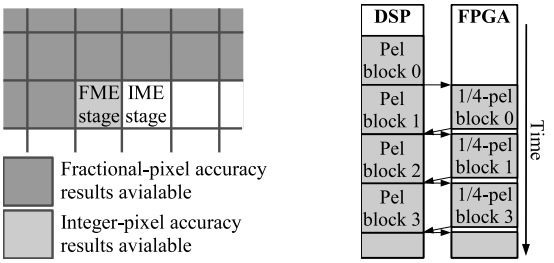


FIG. 6.8 – Implantation du pipeline au niveau bloc

processeur. Il manque une notion de multi-rythme pour dissocier le fonctionnement au niveau image des traitements au niveau bloc et une notion de pipeline permettant de paralléliser IME et FME.

Les résultats de l'utilisation de SynDEx ont montré que la notion de boucle n'est pas bien prise en compte dans l'outil actuel. Par conséquent une description fine au niveau bloc conduit à un nombre d'opérations élémentaires trop important car les boucles sont déroulées exhaustivement. Il n'est alors pas possible de réaliser une description au niveau bloc pour faire apparaître les opérations IME et FME. L'algorithme d'estimation de mouvement est donc décrit au niveau image. De plus, la distribution et l'ordonnancement optimaux sont ici simples à identifier, ce qui limite l'intérêt de l'outil.

Les interfaces entre le DSP et le FPGA sont existantes, il est donc rapide de mettre au point les opérations de transfert et de synchronisation optimisées. Afin de simplifier le développement, le coprocesseur peut être utilisé comme un périphérique même du DSP. Une macro-instruction est donc développée selon l'algorithme 6.3.

Algorithme 6.3 Macro-instruction de raffinement subpixélique

```
si (indice de bloc ≠ 0) (pipeline initialisé ?)
  lire résultat du bloc précédent
fin de si
transférer données du bloc courant (lancer le traitement sur de nouvelles données)
si (indice de bloc = dernier bloc) (fin du pipeline ?)
  lire résultat du bloc courant
fin de si
```

L'outil de prototypage est toutefois utilisé au niveau image pour la description globale de l'algorithme. Il permet d'exécuter l'estimation de mouvement sur la plateforme hétérogène, et de réaliser les opérations de lecture de flux et d'affichage sur PC. Les transferts et synchronisations sont gérés automatiquement. Il permet également de faire la vérification fonctionnelle et des chronométrages (cf. paragraphe 4.1.7).

6.1.2.4 Performances

Une fois que la vérification fonctionnelle a été validée avec le modèle PC, le chronométrage de plusieurs configurations permettent d'évaluer les performances de la solution hétérogène. Les temps d'exécution apparaissent dans le tableau 6.4 avec pour référence les résultats de IME sur DSP, FME sur DSP et FPGA, et l'estimateur com-

plet (IME+FME hétérogène). Les temps d'exécution sont donnés pour seulement une taille de bloc, avec le niveau de pleine résolution de l'algorithme HDS pour IME.

IME est exécuté en 900 ns pour un bloc 8x8 et 2800 ns pour un bloc 16x16. Le raffinement quart de pixel est exécuté sur DSP en 1200 ns pour un bloc 8x8 et 4400 ns pour un bloc 16x16 alors que cela ne requiert seulement que 842 et 1925 ns sur FPGA. De plus les traitements sont parallélisés et le fonctionnement du FPGA est donc à priori transparent. Les mesures de l'application parallélisée sur l'architecture hétérogène donnent des temps d'exécution de 1250 et 3900 ns pour des blocs 8x8 et 16x16 respectivement. Le fonctionnement du FPGA n'est donc pas totalement transparent. Effectivement, le temps global devrait être celui celui du plus long traitement, soit 1200 et 4400 ns et il est supérieur. Le temps supplémentaire est dû au transfert des données entre le DSP et le FPGA. En effet Le DSP place les données (bloc courant, fenêtre de recherche, vecteur et multiplicateur de Lagrange) en mémoire interne pour les envoyer vers le FPGA de manière contiguë. De plus, les transferts de données occupent le bus mémoire et ralentissent donc légèrement les traitements.

Taille de bloc	8x8	16x16
IME sur DSP	900 ns (720p frame : 13 ms)	2800 ns (10 ms)
FME sur DSP	1200 ns (17.3 ms)	4400 ns (15.8 ms)
FME sur FPGA	842 ns (12 ms)	1925 ns (7 ms)
IME sur DSP + FME sur FPGA	1250 ns **(720p frame : 18 ms)**	3900 ns **(14 ms)**

TAB. 6.4 – Chronométrages par bloc (et par image 720p)

L'utilisation d'un FPGA pour réaliser l'opération de raffinement subpixélique des vecteurs de mouvement permet de réduire considérablement l'impact de cette opération sur le temps de traitement. Le coprocesseur cadencé à 133 MHz a des performances du même ordre de grandeur que IME sur DSP, ce qui permet une utilisation des ressources efficace avec une parallélisation des traitements sur les deux composants.

Il est donc possible d'atteindre 55 (resp. 70) images par secondes pour l'estimation de mouvement au quart de pixel d'une image 1280x720 pour des blocs 8x8 (resp. 16x16). L'introduction du coprocesseur permet d'atteindre le temps-réel pour des blocs 16x16 et d'en être très proche pour des blocs de taille 8x8.

6.1.2.5 Estimateur de mouvement complet

Les performances présentées ci-dessus prennent en compte seulement le niveau pleine résolution de l'algorithme HDS, en ne considérant qu'une taille de bloc. Cependant, un estimateur de mouvement complet pour la compression vidéo H.264 suppose de prendre en compte également plusieurs tailles de bloc et les niveaux hiérarchiques. Un module DSP+FPGA n'est alors plus à même de traiter toutes les tailles de manière exhaustive en temps réel.

L'algorithme allégé décrit au paragraphe 5.4.2.5 permet de ne raffiner qu'un seul mode sur les quatre tailles considérées. Cette réduction de complexité permet d'envisager un estimateur de mouvement à taille de bloc variable à 25 images par seconde en 720p. Une réduction de complexité de l'algorithme pixel entier à taille de bloc variable

est nécessaire pour atteindre de meilleures performances. Par exemple un algorithme basé sur EPZS 8x8, avec un regroupement des blocs pour obtenir les vecteurs des autres tailles (sans mise en correspondance), permettrait d'atteindre les 50 images par secondes, avec une perte de qualité des vecteurs de mouvement.

Une autre solution, beaucoup plus coûteuse, consiste à réaliser l'estimation de mouvement des niveaux hiérarchiques sur un processeur dans un premier étage de pipeline, puis de dupliquer le schéma IME+FME pour chaque taille de bloc dans un deuxième étage de pipeline. Cette solution nécessite donc au total cinq DSP et quatre petits FPGA pour réaliser l'estimation de mouvement H.264 pour une image de référence en 720p 50 Hz, ce qui est cher et donc non envisageable dans un contexte industriel.

6.1.3 Bilan sur la conception d'un coprocesseur

Le prototype réalisé permet de mettre en adéquation deux opérations d'estimation de mouvement ayant des propriétés différentes avec deux architectures différentes. Le coprocesseur proposé a pour but de raffiner un vecteur de mouvement au quart de pixel avec peu de ressources logiques. L'architecture du FPGA offre un parallélisme élevé, permettant d'atteindre un débit de données interne soutenu tout en conservant une bande passante mémoire externe compatible avec un bus externe de DSP. Cela permet donc d'obtenir de bonnes performances et d'accélérer les traitements par rapport à une exécution sur DSP. L'estimateur de mouvement pixel entier sur DSP, basé sur un algorithme rapide, tire parti des capacités de branchement conditionnel et d'accès mémoire aléatoire. Les travaux effectués pour une taille de bloc peuvent être réutilisés pour plusieurs tailles de bloc. Une extrapolation des résultats prédit les ressources matérielles nécessaires à l'exécution d'un estimateur de mouvement à taille de bloc variable exhaustif basé sur une solution hétérogène. Grâce à la programmabilité du DSP, il est aisé de modifier l'algorithme. La solution câblée conserve alors de la flexibilité pour permettre une taille de bloc variable et ainsi autoriser des compromis sur la qualité des champs de vecteur au profit des performances et de la réduction du coût de la plate-forme.

La parallélisation en fonction du type de traitement à effectuer sur une architecture hétérogène permet de mettre à profit les spécificités des composants en exécutant chaque opération sur l'unité la plus adaptée. Cependant le parallélisme est limité au nombre d'opérations sans inter-dépendances. Il est donc nécessaire d'utiliser une autre technique pour créer davantage de parallélisme lorsque les contraintes ne sont pas atteintes, ou lorsque la taille de l'image augmente. Le paragraphe suivant présente des solutions de parallélisation en fonction des données.

6.2 Parallélisation en fonction des données

Il est possible d'augmenter le parallélisme potentiel d'une application en divisant des opérations conséquentes en des opérations plus petites. Cependant, il est pour cela nécessaire d'éliminer des dépendances de données. Nous avons vu dans le chapitre précédent qu'un ordonnancement en fonction des opérations sur une même image limitait les performances à cause des accès mémoires. Lorsque les opérations nécessitant

des données proches sont groupées, les mémoires caches sont mises à profit. Afin de réduire les pertes de performances dues aux transferts de données inhérents à la parallélisation des traitements, nous allons dans ce paragraphe préférer le parallélisme de données.

Lorsque plate-forme est composée de processeurs avec une architecture mémoire hiérarchique (cache), il est préférable de limiter les accès aux données sur les bus externes en regroupant sur un même processeur les opérations en fonction des données. Cela permet d'améliorer les performances des mémoires caches, et dans le même temps de limiter les échanges de données entre les processeurs.

Les algorithmes de traitement d'images tels que l'estimation de mouvement se prêtent bien à une division des opérations en fonction des données. Par exemple, un découpage de l'image en bandes permet d'obtenir autant d'opérations que nécessaire à distribuer et à ordonnancer. Les dépendances de données entre les bandes doivent toutefois être modifiées lorsque les calculs dépendent des résultats.

Nous décrivons dans ce paragraphe la façon dont le parallélisme est extrait du Graphe Flot de Données (GFD) avec la méthodologie AAA et les résultats obtenus.

6.2.1 Modélisation de l'algorithme parallèle

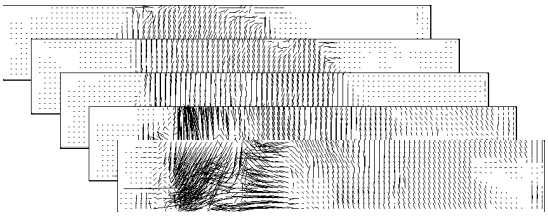
La description de l'algorithme sous forme d'un GFD a été décrite au paragraphe 5.4.2 (Fig. 5.14). L'opération d'estimation de mouvement à taille de bloc variable et au quart de pixel est exécutée macro-bloc par macro-bloc. Le nombre de lignes à traiter ainsi que la position de la ligne de départ sont entrés en paramètres. Il est alors possible de séparer les traitements sur plusieurs bandes. Plus il y a de bandes, moins il y a de lignes à traiter par bande. Ceci permet à la méthodologie AAA d'extraire automatiquement le parallélisme potentiel de données.

6.2.1.1 Parallélisation en bandes indépendantes

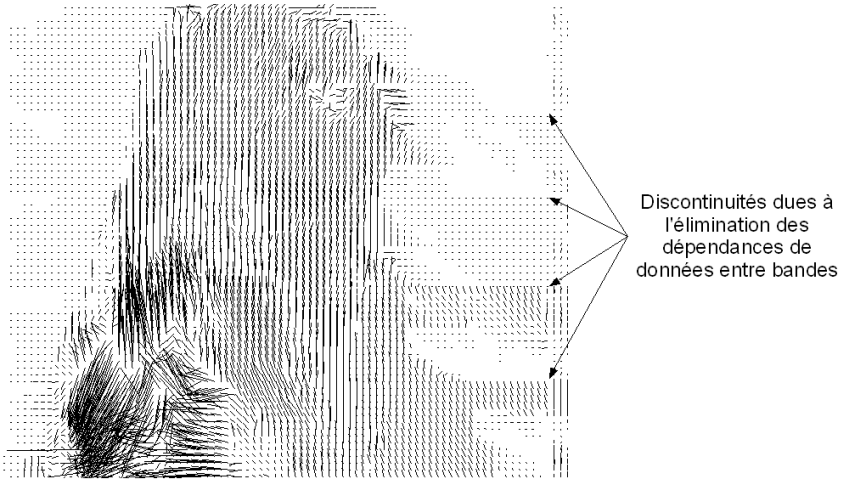
La capacité de l'outil à itérer automatiquement une opération en fonction du rapport de quantité de données entre chaque extrémité d'un arc de dépendance permet de découper naturellement l'image en bandes de taille égale (cf. paragraphe 4.1). Chaque instance de l'opération fonctionne indépendamment des autres, sur une bande donnée. Pour éviter de borner la fenêtre de recherche, toute l'image de référence est diffusée à chaque instance (sommet "*Diffuse*"). L'image courante est découpée en bande, puis chaque bande est associée à l'instance d'estimation de mouvement appropriée (sommet "*Fork*"). Chaque instance fournit les vecteurs de mouvement pour sa bande de calcul (Fig. 6.9-b). Les différents résultats sont ensuite assemblés avec un sommet "*Join*" pour former le champ de vecteurs global (Fig. 6.9-c).

Nous avons donc de cette manière accru le parallélisme potentiel de l'algorithme. Dans l'exemple donné, l'image est découpée en cinq bandes permettant d'exploiter cinq DSP. Le graphe temporel de la figure 6.10 montre l'ordonnancement et la distribution obtenus. Les cinq instances d'estimation de mouvement correspondant aux cinq bandes de l'image sont bien exécutées en parallèle sur des processeurs distincts. La bande passante de la mémoire partagée permet d'assurer le transfert des images et des champs de vecteurs sans ralentir les calculs. Cependant, les processeurs ne sont pas utilisés

(a) Image source



(b) Découpage des traitements en bandes



(c) Assemblage des bandes

FIG. 6.9 – Parallélisation de l'algorithme en bandes

de manière efficace puisque quatre sur cinq sont inactifs pendant environ la moitié de temps.

La parallélisation en bandes ne concerne ici que le niveau de pleine résolution de l'algorithme hiérarchique. Les niveaux intermédiaires et le calcul des pyramides d'images sous échantillonnées sont exécutés séquentiellement à cause de leur dépendances de données. Le champ de vecteurs du niveau 1 doit également être disponible avant l'exécution des cinq instances du niveau 0. Cette dépendance de données contraint à un fonctionnement séquentiel, il serait donc inutile d'ajouter un processeur dans ces conditions.

Il n'est ainsi pas possible de diminuer la latence à cause des dépendances de données. Par contre la cadence peut être améliorée. En effet, en insérant des registres (opérations "*Dela*y") sur les arcs des dépendances de données entre les opérations du niveau 0 et les opérations des niveau hiérarchiques, ces dépendances de données deviennent alors inter-itération : les données qui passent à travers les registres sont retardées d'une itération (Fig. 6.11). De cette façon, les données utiles sont présentes dès le début de l'itération (en fait depuis la fin de l'itération précédente), ce qui permet de combler les inactivités des processeurs. Il est donc possible de définir un pipeline avec la description flot de données. La latence optimisée par AAA/SynDEx représente désormais la durée de l'étage de pipeline le plus long, c'est à dire la cadence du pipeline.

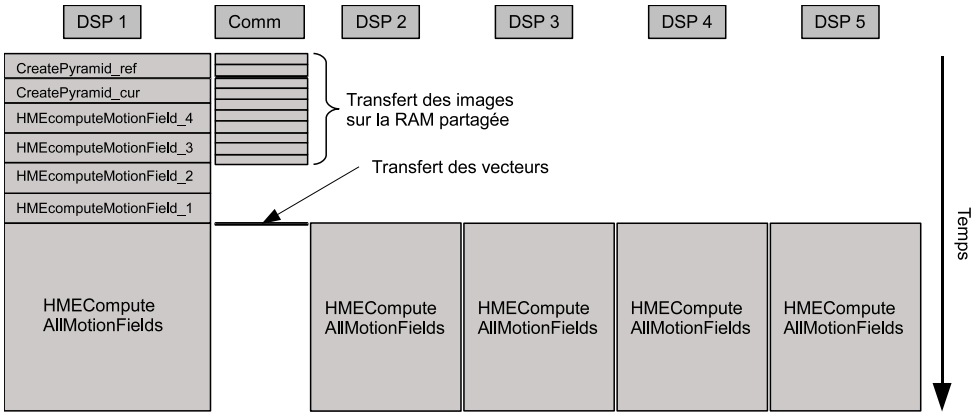


FIG. 6.10 – Ordonnancement et distribution des opérations sur une architecture parallèle

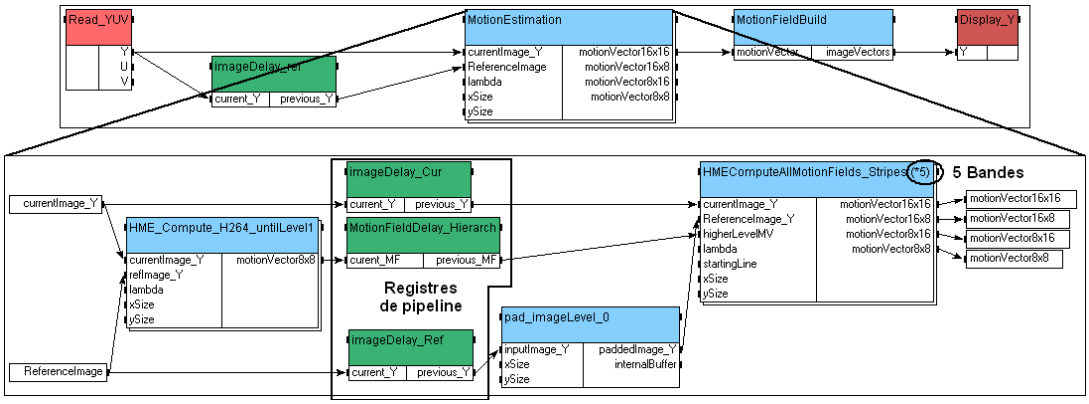


FIG. 6.11 – Graphe flot de données à deux étages de pipeline

Grâce à cette technique de mise en pipeline, les opérations des niveaux hiérarchiques sont exécutées sur un processeur additionnel en parallèle avec les traitement du niveau pleine résolution. La figure 6.12 présente le graphe temporel associé. Les temps d'inactivité des processeur sont éliminés.

Le découpage en bandes permet de paralléliser des opérations normalement séquentielles de manière naturelle. L'élimination des dépendances introduit toutefois des discontinuités dans le champ de vecteurs à la limite des bandes (Fig. 6.9-c). Ces discontinuités peuvent engendrer des artefacts visibles dans l'image compressée, et réduire les performances de compression. Pour augmenter la qualité de l'estimation de mouvement, il est nécessaire de rajouter les dépendances de données qui rendent l'exécution de nouveau séquentielle. Le parallélisme peut cependant être restauré comme nous allons le voir dans le paragraphe suivant.

6.2.1.2 Parallélisation avec pipeline

Nous avons vu au paragraphe 4.1 qu'il était possible d'itérer une opération en redirigeant vers l'itération suivante les résultats de l'itération courante (sommet "*Ite-*

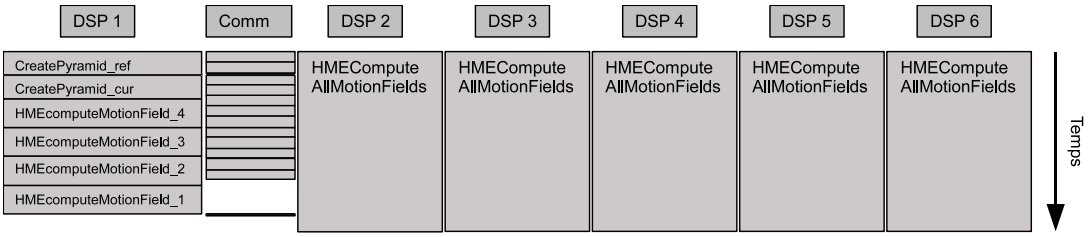


FIG. 6.12 – Ordonnancement et distribution des opérations avec du parallélisme temporel

rate"). Afin de conserver un champ de vecteurs continu et toutes les prédictions spatiales (blocs du dessus) à la frontière des bandes, les champs de vecteurs résultants d'une bande sont redirigés vers le traitement de la bande suivante. Cela implique une dépendance de données et empêche la parallélisation des bandes. Toutefois il est possible d'ajouter un registre sur le chemin de la dépendance (Fig. 6.13).

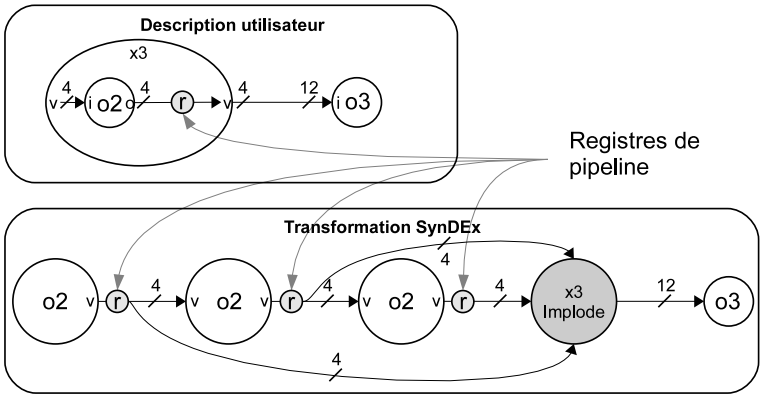


FIG. 6.13 – Obtention d'un pipeline avec AAA

Trois éléments essentiel permettent d'obtenir le pipeline : un sommet "*Join*" pour le découpage en bandes, un sommet "*Iterate*" pour conserver les dépendances de données, et des opérations "*Delay*" pour "casser" les dépendances intra-itération. Appliqué à l'algorithme d'estimation de mouvement, un graphe temporel similaire à celui de la figure 6.12 est obtenu. Les dépendances de données entre les bandes doivent de plus être transmises entre les processeurs.

Le graphe flot de données de l'algorithme est décrit tel que le nombre de bandes soit paramétrable. Il convient alors de choisir un nombre de bandes adapté au nombre de processeurs disponibles. Le mécanisme de mise en pipeline automatique garantit de pouvoir obtenir la cadence voulue avec un nombre de processeur suffisant, sous réserve que les échanges de données ne deviennent pas prépondérants. L'intérêt majeur est que si une augmentation de la latence est tolérée, une cadence élevée peut être rapidement obtenue avec une plate-forme surdimensionnée. Ensuite, à mesure que les calculs sont optimisés, le nombre de processeurs (et par conséquent la latence globale) peut être diminué. Si la latence est également une contrainte, les bandes sont traitées de manière indépendante.

A chaque étape du prototypage, le portage sur plate-forme est obtenu automatiquement. C'est à dire que la parallélisation, l'ordonnancement, les communications et synchronisations sont générés par l'outil. L'utilisateur doit donc seulement modifier son graphe d'algorithme et d'architecture, lancer les opérations de génération de code, le compiler et enfin l'exécuter.

Le paragraphe suivant illustre ces techniques de parallélisation sur une carte multi-DSP.

6.2.2 Application d'estimation de mouvement sur multi-DSP

Dans ce paragraphe, nous donnons des exemples de prototypes réalisés sur une plate-forme *VP3* de *Vitec Multimedia* embarquant huit DSP (Fig. 6.14).

6.2.2.1 Plate-forme

Le graphe d'architecture dans AAA est donné sur la figure 6.14 avec un PC hôte et quatre DSP. La carte de développement embarque huit DSP Texas Instruments DM642 à 600 MHz. Ces processeurs ont le même cœur de calcul qu'un C64 mais ne disposent que de 256 KO de mémoire interne à partager entre la cache de niveau 2 et une mémoire interne adressable pour allouer les données critiques. Chaque DSP est connecté à une mémoire externe de 64 MO cadencée à 100MHz [Mul05]. Les mémoires externes des DSP sont toutes reliées à un bus commun, permettant d'écrire une même donnée vers plusieurs mémoires simultanément. On a donc un modèle de mémoire partagée à accès multiple. SynDEx ne gère pas l'accès multiple, donc nous nous contentons dans un premier temps d'un modèle de RAM partagée.

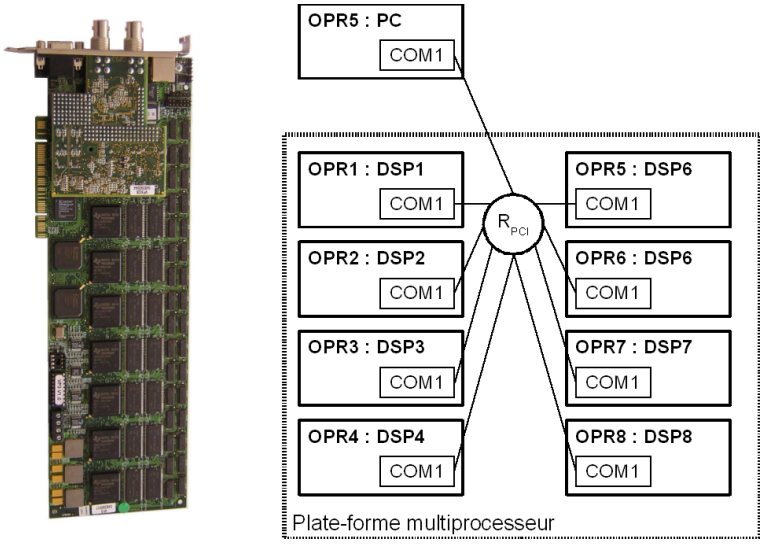


FIG. 6.14 – Graphe d'architecture dans la méthodologie AAA : modélisation d'une plate-forme Vitec (8 DSP) reliée à un PC via un bus PCI

Nous allons implanter plusieurs configurations de l'estimateur de mouvement en utilisant la méthode de parallélisation en bandes indépendantes. Dans un premier

temps, tout est exécuté sur un processeur, sans découpage pour évaluer les performances. Puis progressivement nous augmentons le nombre de processeurs à quatre, six et huit pour évaluer le gain de performances de la parallélisation. Pour chaque implantation, les exécutifs de chaque processeur sont générés automatiquement, ce qui accélère le processus de prototypage.

6.2.2.2 implantation mono-processeur

Les performances obtenues sur des images de taille 1280×720 sont regroupées dans le tableau 6.5. Les résultats sur le DSP Texas instrument C6416 à 1 GHz sont également rappelés dans un but de comparaison.

DSP	DM642 à 600 MHz			C6416 à 1 GHz		
Algorithme	HDS pixel	HDS $\frac{1}{4}$ pixel	HDS $\frac{1}{4}$ pixel allégé	HDS pixel	HDS $\frac{1}{4}$ pixel	HDS $\frac{1}{4}$ pixel allégé
Pyramide	3.5 ms			2.4 ms		
niveaux 4 à 1	8 ms			4.8 ms		
Padding niv.0	2.8 ms			2.1 ms		
Niveau 0	62 ms	200 ms	95 ms	35 ms	113 ms	55 ms
Total	76 ms	214 ms	109 ms	44 ms	122 ms	64 ms

TAB. 6.5 – Chronométrages de l'estimation de mouvement sur un DSP DM642 à 600 MHz avec une séquence 720p

Le rapport entre les temps d'exécution sur les deux processeurs ne correspondent pas exactement aux rapport entre les fréquences de fonctionnement. L'élargissement de l'image (padding, cf. par. 5.1.2.1) et le calcul de la pyramide nécessitent moins de cycles sur le DM642 à 600 MHz que sur C6416 à 1 GHz. Cela s'explique par la fréquence de fonctionnement de la mémoire externe. Sur le DM642, elle est cadencée à 100 MHz, soit 0.167 fois la fréquence du DSP (0.133 pour le C64 avec 133 MHz). Moins de cycles sont donc nécessaires pour les opérations régulières à fort accès mémoire (pyramide et padding). A l'inverse, la mémoire interne étant plus petite sur le DM642, la mémoire cache est réduite. Les opérations d'estimation de mouvement, nécessitant une large fenêtre de référence, sont alors ralenties.

Sur un DSP, les trois configurations de l'estimateur de mouvement conduisent à un temps par image de 76 à 214 ms. Pour accélérer les traitements, nous allons utiliser plusieurs DSP en parallèle.

6.2.2.3 Implantations multiprocesseurs

L'algorithme est maintenant distribué sur plusieurs DSP (sans FPGA) en découpant l'image en bandes indépendantes. Cela correspond à l'utilisation de la notion de "*Slice*" pour la compression vidéo H.264 haute définition. D'après les résultats du tableau 6.5, 11.5 ms sont nécessaires pour les calculs sur les niveaux de 4 à 1. Ces performances constituent donc notre borne inférieure, puisque les opérations ne sont pas parallélisées.

Les traitements sur PC sont également parallélisés avec de la mise en pipeline pour influer le moins possible sur les résultats. Trois processeurs différents sont utilisés grâce au multi-tâche [RRND06] : lecture et affichage de la séquence, affichage des champs

de vecteurs et communications avec la plate-forme. Le processeur utilisé est un bi-processeur Xeon à 3.6 GHz. Les opérations sur PC ne ralentissent pas les traitements sur la plate-forme.

En prenant en considération les communications, pour les niveaux hiérarchiques seuls sur un DSP, 21 ms sont mesurées par images, soit 47 images/s. Il existe donc un surcoût de 10 ms dû aux transferts des données, bien que les transferts et les calculs soient réalisés en parallèle. Les données transférées concernent principalement la séquence d'image et les vecteurs, que l'on va retrouver sur chaque processeur. Par conséquent, ce surcoût est présent sur chaque DSP. Le modèle de communications utilisé dans l'outil de prototypage (RAM partagée) ne permet pas de prendre en compte les spécificités de la plate-forme. Cela se traduit par des recopies redondantes et un ordonnancement non optimal. La plate-forme n'est donc pas exploitée efficacement. L'outil est cependant utilisé car il permet de prototyper rapidement l'application sur une architecture comportant un nombre important de processeurs. On peut noter qu'il est toujours possible de réaliser des optimisations à la main en dernier lieu.

Les performances obtenues (Tab. 6.6) tiennent compte des communications entre DSP et PC, et entre DSP et DSP. Le temps nécessaire à l'estimation de mouvement d'une séquence de 1000 images est mesuré.

Algorithme		HDS pixel	HDS $\frac{1}{4}$ pixel	HDS $\frac{1}{4}$ pixel allégé
1 DSP		89 s 11 images/s	225 s 4.4 images/s	125 s 8 images/s
4 DSP :	niveaux 1 à 4 + 3 bandes	32 s 31 images/s	75 s 13 images/s	43 s 23 images/s
6 DSP :	niveaux 1 à 4 + 5 bandes	23 s 43 images/s	48 s 21 images/s	29 s 34 images/s
8 DSP :	niveaux 1 à 4 + 7 bandes	22 s 45 images/s	38 s 26 images/s	24 s 42 images/s

TAB. 6.6 – Chronométrages de l'estimation de mouvement sur une plate-forme multi-DSP DM642 à 600 MHz avec une séquence 720p

Les résultats sont conformes avec les prévisions. Les communications réduisent fortement les performances sur cette plate-forme. Cependant, les temps d'exécution évoluent de manière linéaire avec le nombre de processeurs, excepté lorsque la borne inférieure de 21 s est approchée. Par conséquent nous pouvons déduire que la parallélisation des opérations ne conduit pas à une surcharge des liens de communication. La génération de code automatique fourni ici une façon de vérifier à postériori les résultats de la parallélisation, de manière simple et rapide.

Les résultats montrent qu'il est possible d'approcher les 25 images par seconde pour l'algorithme HDS avec une recherche subpixélique simplifiée sur quatre DSP. L'estimation de mouvement à taille de bloc variable exhaustif au quart de pixel (HDS $\frac{1}{4}$ pixel) nécessite huit DSP pour dépasser 25 images par seconde.

Le mécanisme de communications inter-processeurs de la plate-forme supporte bien un nombre élevé de DSP (jusqu'à 8) sans être saturé. Cependant les communications sont assez coûteuses, et ne permettent pas de dépasser 45 images/secondes sans appor-

ter d'optimisations manuelles sur les communications. Le modèle de RAM partagée utilisé [MNUR07] doit être amélioré pour exploiter plus efficacement la plate-forme.

6.2.3 Synthèse sur le parallélisme de données

Le modèle d'algorithme flot de données de la méthodologie AAA a été utilisé pour décrire l'algorithme d'estimation de mouvement de manière flexible. Le parallélisme potentiel de données est extrait automatiquement grâce à un découpage en bandes. Ce découpage est défini implicitement par l'utilisateur et doit être en adéquation avec le nombre de processeurs disponibles de la plate-forme.

L'insertion de registres sur les dépendances de données entre deux opérations permet de paralléliser deux opérations normalement séquentielles, en retardant d'une itération les résultats. Un mécanisme de pipeline est donc obtenu grâce à cette technique.

La méthodologie AAA a permis de faciliter l'implantation multiprocesseur en prenant en charge la distribution, l'ordonnancement les communications et synchronisations inter-processeurs. Dans l'exemple donné, la distribution et l'ordonnancement étaient évidents compte tenu du peu d'opérations. Cependant, une description algorithmique flexible permet de porter l'application sur une plate-forme multiprocesseur aisément. De plus la parallélisation et l'accélération progressives sont obtenues rapidement, de manière automatique. Lorsque la distribution et l'ordonnancement sont figés, des optimisations manuelles peuvent être nécessaires afin d'améliorer l'implantation automatique.

Une application d'estimation de mouvement a été prototypée sur une plate-forme de 11 processeurs : un PC multi-tâche émulant 3 processeurs, et une plate-forme de 8 DSP. Sur ce type d'architecture, la quantité des processeurs introduit une complexité d'implantation conséquente, et conduit généralement à un temps de développement long. Ce type d'architecture complexe est également une source d'erreurs potentielles, concernant l'ordonnancement, les transferts de données et les synchronisations. La méthodologie utilisée a permis d'automatiser l'implantation. Le portage sur cible est donc accéléré et réalisé de manière fiable. C'est à dire que les communications ne génèrent pas d'erreur et l'ordonnancement est garanti sans inter-blocage. L'application peut être rapidement prototypée avec des configurations différentes (plate-forme, distribution, ordonnancement) afin de rechercher la meilleure solution.

La méthodologie AAA/SynDEx accélère le développement d'une application multiprocesseur, des étapes de vérification fonctionnelle à l'implantation distribuée sur plate-forme cible. Cependant, l'outil admet des limitations.

- L'utilisation d'un coprocesseur (cf. paragraphe 6.1) peut nécessiter un ordonnancement multi-rythme : les opérations sont parallélisées et mises en pipeline au niveau bloc, alors que le graphe flot de données est géré au niveau image. Ceci n'est pas possible avec le modèle d'algorithme actuel, et le coprocesseur doit être utilisé manuellement. C'est à dire que l'ensemble $DSP + coprocesseur$ est modélisé comme un seul composant du point de vue de la méthodologie.
- Le mécanisme de pipeline est créé grâce à une astuce permettant de retarder les dépendances d'une itération. Ce mécanisme n'est pas géré directement par la méthodologie, ce qui pose des problèmes à l'initialisation et au vidage du

pipeline. De plus il n'est pas possible d'optimiser conjointement la latence et la cadence puisque seule la latence d'une itération est prise en compte. Il est donc nécessaire de bien maîtriser à la fois la méthodologie l'algorithme et l'architecture afin de créer un graphe d'algorithme efficace.
- Le modèle de RAM partagée n'est pas adapté à la carte multi-DSP *Vitec VP3*. Les solutions de l'adéquation peuvent être sous-optimales. De plus, les chronométrages révèlent un surcoût des communications.
- L'adéquation est basée sur l'optimisation de la latence d'une itération. Le temps d'exécution des opérations sur les processeurs de l'architecture et les durées nécessaires aux communications sont prises en compte indépendamment. Or, selon le processeur utilisé, l'ordonnancement peut avoir un impact (utilisation des mémoires caches) sur les durées, et des transferts mémoires peuvent ralentir l'accès aux données. Ces paramètres ne sont pas considérés lors de l'adéquation.

Ces limitations doivent être prises en compte pour rendre plus efficace la méthodologie de prototypage. Les cas simples sont facilement traités, mais lorsque l'algorithme se complique (répétitions, pipeline), il peut être difficile, voire impossible, d'obtenir un résultat. L'outil de conception SynDEx permet d'accélérer les étapes de conception d'applications distribuées. Néanmoins une évolution est nécessaire pour le développement d'applications complexes, et notamment pour le traitement d'images.

6.3 Conclusion

Dans ce chapitre, nous avons présenté deux techniques de parallélisation des calculs permettant d'approcher, et même d'atteindre le temps réel avec des applications d'estimation de mouvement pour la compression vidéo haute définition.

Une plate-forme hétérogène peut combiner les intérêts de plusieurs architectures. L'estimation de mouvement pixel entier est basée sur un algorithme prédictif et nécessite un processeur capable d'effectuer des branchements conditionnels et des accès mémoire aléatoires. L'utilisation d'un DSP est donc adaptée, et permet d'ajouter de la flexibilité. Le raffinement subpixélique requiert une bande passante mémoire conséquente, et des opérations d'interpolation coûteuses. La régularité des calculs en fait un bon candidat pour être accéléré sur FPGA.

La combinaison DSP-FPGA permet de réaliser le raffinement subpixélique de manière transparente grâce à un pipeline au niveau bloc et un algorithme de décision sur le DSP qui évite le raffinement exhaustif pour toutes les tailles de bloc. Le FPGA doit être utilisé en tant que coprocesseur, indépendamment de AAA/SynDEx à cause des limitations de l'outil.

Le découpage de l'image en bandes permet de créer naturellement du parallélisme. Le nombre de bandes s'adapte aisément au nombre de processeurs disponibles. Toutefois il est nécessaire d'éliminer les dépendances entre les bandes grâce à deux solutions : créer un pipeline avec un étage par bande, ou rendre les bandes indépendantes. Ce type de parallélisation est bien adapté aux applications de traitement d'images et permet d'accélérer automatiquement une application sur une plate-forme multiprocesseur, ou un processeur de PC multicœur en utilisant une méthodologie de prototypage rapide.

La méthodologie AAA est bien adaptée aux applications traitement d'images. La modélisation flot de données permet de décrire un algorithme avec peu de contrôle.

Cependant, l'outil SynDEx admet des limitations, d'une part dans les modèles d'algorithme (pas de pipeline ni de multi-rythme) et d'architecture (modélisation de l'architecture interne d'un processeur, modèles d'interconnexions), et d'autre part dans l'adéquation (prise en compte de la cadence).

Le groupe image du laboratoire IETR travaille actuellement sur le développement un nouvel outil de prototypage rapide basé sur la méthodologie AAA. Le but est d'éliminer les contraintes imposées par SynDEx, afin d'obtenir un outil plus ergonomique et évolutif. Ce nouvel outil, "PREESM" (Parallel Real-Time Embedded Executive Scheling Method) a pour objectif d'intégrer de nouveaux modèles d'algorithme et d'architecture.

Conclusions et perspectives

Les travaux présentés dans ce mémoire abordent les problèmes posés par l'implantation efficace d'algorithmes de traitement d'images. L'estimation de mouvement illustre bien les contraintes rencontrées. Les techniques d'optimisation du code sont abordées afin de pousser les performances sur cible mono-processeur, ainsi que les aspects méthodologiques liés au développement d'applications sur plates-formes parallèles. Un premier objectif a été d'étudier l'implantation d'estimateurs de mouvement pour la compression vidéo. Des recherches ont été effectuées à la fois sur les algorithmes et les architectures matérielles. Un deuxième objectif a été de valider et développer les outils de prototypage dans un cadre d'applications complexes.

Le contexte applicatif de l'étude a été décrit dans la première partie. Le chapitre 1 a introduit les techniques de compression vidéo. Le standard vidéo H.264 améliore les performances de compression par rapport à ses prédécesseurs au détriment de la complexité de calcul qui se voit augmentée d'un ordre de grandeur. L'estimation de mouvement, avec l'introduction de nombreux modes, contribue pour beaucoup aux performances comme à la complexité. La puissance de calcul nécessaire peut être réduite de manière conséquente en utilisant une technique adaptée. Un état de l'art des techniques existantes a été proposé dans le chapitre 2. Les algorithmes de mise en correspondance prédictifs comme EPZS ou multirésolution comme HME sont parmi les plus intéressants en terme de réduction de la complexité et en terme de robustesse. La taille de bloc variable et la précision fraction de pixel contribuent à améliorer la définition du mouvement, et dans le même temps augmentent la complexité. Nous avons ensuite présenté les architectures matérielles capables d'exécuter l'application d'estimation de mouvement. L'utilisation de composants programmables est bien adaptée au développement de prototypes, sujets à être modifiés. L'étude de composants dédiés à l'estimation de mouvement fait ressortir la complexité de cette opération et met en avant les points bloquants, à savoir la quantité de données à traiter (bande passante mémoire) et le nombre de calculs importants. Des solutions multicomposants sont alors à envisager, afin d'augmenter la puissance de calcul, et hétérogènes, dans le but d'exploiter l'intérêt de plusieurs types d'architectures.

Dans une deuxième partie de ce document, nous avons présenté des implantations d'estimateurs de mouvement pour la compression H.264 dans un cadre méthodologique. Nous nous sommes tout d'abord attachés à expliquer les fondements de la méthodologie AAA et à présenter les outils de prototypage associés. Le but de cette méthodologie est d'automatiser le portage d'une application sur une plate-forme multiprocesseur. L'algorithme d'une part et la plate-forme d'autre part sont d'abord décrits de manière indépendants. Ensuite, les opérations de l'algorithme sont

distribuées et ordonnancées de manière optimisée. Enfin le code, prenant en charge entre autres les allocations mémoires, les communication et synchronisations inter-processeurs, est généré de manière automatique pour chaque composant. L'automatisation du processus de portage réduit le cycle de développement et apporte une sécurité dans la conception des systèmes. Cette méthodologie est intégrée dans un processus de développement permettant de porter et d'optimiser progressivement les opérations sur cible. Des outils simples assurent la validité des travaux.

Le cadre de l'estimation de mouvement pour la compression vidéo haute définition nous a permis d'identifier et de corriger deux limitations. Premièrement l'interfaçage des travaux avec une application annexe est très complexe. La solution apportée a été une abstraction de l'application annexe afin de pouvoir interfacer l'application d'estimation de mouvement avec un encodeur vidéo. Nous avons pu, ensuite, comparer différentes méthodes de mise en correspondance en terme de qualité dans le cadre de la compression vidéo. Deuxièmement, la mémoire interne d'un DSP étant réduite, l'utilisation de la mémoire externe était inévitable. Il était alors nécessaire d'utiliser un mécanisme de rapatriement des données en mémoire interne afin de ne pas faire chuter les performances. Nous avons utilisé le contrôleur de cache du processeur pour réaliser cette tâche. Nous avons modifié la génération automatique de code pour gérer automatiquement la mémoire cache, notamment les problèmes de cohérence.

Les outils étant été adaptés et mis au point, l'implantation d'estimateurs de mouvement a ensuite pu être étudiée. L'étude algorithmique du chapitre 5 a permis de mesurer la complexité de l'opération d'estimation de mouvement, après une modélisation flot de données adaptée. Une nouvelle technique a été développée, combinant qualité des champs de vecteurs et faible complexité, afin d'améliorer les performances par rapport à l'existant. L'optimisation de l'implantation sur DSP a nettement amélioré les performances. Toutefois un seul DSP n'a pas les ressources nécessaires à l'exécution temps réel pour la haute définition. Des prototypes d'implantations multicomposants ont été réalisés et présentés dans le chapitre 6. L'opération de raffinement subpixélique a été identifiée comme bon candidat à une implantation sur FPGA. Un coprocesseur dédié a été conçu afin d'accélérer la phase de raffinement subpixélique. Les performances obtenues sont élevées avec peu de ressources matérielles. Une implantation hétérogène a donc pu être prototypée sur une plate-forme embarquant un DSP et un FPGA de taille moyenne. La flexibilité de l'implantation permet de prendre en charge la taille de bloc variable et d'utiliser un algorithme de décision permettant de réduire la complexité d'un facteur deux.

La parallélisation des traitements en fonction des opérations implique de nombreux transferts de données et ne permet pas d'utiliser les mémoires caches des processeurs de manière efficace. Afin d'augmenter l'accélération liée à l'utilisation de plusieurs processeurs, nous avons ensuite étudié une parallélisation en fonction des données. Le découpage de l'image en bandes permet de paralléliser naturellement les traitements. La méthodologie prend en charge automatiquement ce genre de parallélisation dans la modélisation de l'algorithme. La mise en pipeline permet de conserver les dépendances mais n'est pas prise en compte en temps que telle dans l'adéquation. Nous avons donc évalué cette technique en prototypant l'application sur une plate-forme de huit DSP. Les performances augmentent bien avec le nombre de processeurs utilisés, toutefois, Le modèle de communications inter-processeurs RAM partagée ne permet pas d'opti-

miser les transferts et constituent donc une limitation. Le modèle de communications nécessite alors d'être adapté plus précisément au matériel.

Le travail de prototypage a été fait de manière générique. Le passage à une autre architecture est donc facilité. En particulier, le nombre et le type de processeurs utilisés n'ont jamais été fixés : ils dépendent de l'application finale. A ce titre, l'estimateur de mouvement réalisé dans le cadre de ces travaux est utilisé dans plusieurs applications. L'optimisation a réduit le temps d'exécution sur différentes cibles, et la modélisation flot de données le rend simple et modulable. Ainsi, dans un projet au sein de Thomson, une application de détection de zone d'intérêt grâce à un critère psychovisuel a pu intégrer des informations de mouvement. Des évolutions de l'estimateur de mouvement spécifiques à cette application sont à prévoir, notamment la prise en compte du mouvement global. Au sein de l'IETR l'estimateur de mouvement est utilisé dans le développement d'un encodeur vidéo MPEG-4 embarqué et dans un codeur LAR vidéo qui consiste à étendre la technique de compression d'images fixe LAR [FABD06] à la vidéo. Une perspective d'application supplémentaire est l'encodeur vidéo associé au nouveau standard de compression vidéo SVC (Scalable Video Coding). Il peut en effet bénéficier d'un estimateur de mouvement optimisé et profiter des différents niveaux de résolution de la technique hiérarchique utilisée.

Le développement d'un estimateur de mouvement optimisé sur une plate-forme multicomposant a permis de valider certains aspects de méthodologie, de proposer des évolutions et d'identifier des limitations. La méthodologie AAA/SynDEx améliore la sécurité de conception, en supportant les phases de développement de la vérification fonctionnelle à l'implantation optimisée. Les évolutions apportées étendent la méthodologie au traitement vidéo haute résolution sur cibles embarquées avec notamment des contraintes mémoires fortes. Des limitations ont été mises en évidence et certaines ont été levées en réalisant la distribution et l'ordonnancement manuellement, par exemple avec l'utilisation d'un coprocesseur matériel. Certaines limitations n'ont pas pu être résolues avec le logiciel. L'IETR travaille actuellement au développement d'un nouvel outil méthodologique dans le but d'introduire de nouveaux modèles d'architecture et d'algorithmes, ainsi que d'améliorer l'ergonomie.

Liste des algorithmes

Table des figures

Liste des tableaux

Publications personnelles

[UPND072] Fabrice Urban, Ronan Poullaouec, Jean-François Nezan and Olivier Deforges. A flexible heterogeneous hardware/software solution for real-time high-definition H.264 motion estimation, IEEE Transactions on Circuits and Systems for Video Technology, submitted, 2nd review, 2007.

[MNUR07] Alain Maccari, Jean francois Nezan, Fabrice Urban and Mickaël Raulet. Interconnected distributed RAM in SynDEx. dans Design and Architectures for Signal and Image Processing Workshop, Novembre 2007

[UPND07] Fabrice Urban, Ronan Poullaouec, Jean-François Nezan and Olivier Deforges. H.264 Fractional Motion Estimation Refinement : a Real-Time and Low Complexity Hardware Solution for HD Sequences, dans 15th European Signal Processing Conference, Septembre 2007.

[UPND06] Fabrice Urban, Ronan Poullaouec, Jean François Nezan and Olivier Déforges. Real-time Multi-DSP Motion Estimator for MPEG-4 AVC/H.264 High Definition Video. dans International Conference on Signals and Electronic Systems. Septembre 2006.

[URND06] Fabrice Urban, Michaël Raulet, Jean François Nezan and Olivier Déforges. Automatic DSP cache memory management and fast prototyping for multiprocessor image applications. dans 14th European Signal Processing Conference, Septembre 2006.

[UPDN06] Fabrice Urban, Ronan Poullaouec, Olivier Déforges and Jean François Nezan. Estimateur de mouvement temps réel multi-DSP pour l'encodage vidéo MPEG-4 AVC/H.264 haute définition, dans CORESA, novembre 2006.

[RUN+05] M. Raulet, F. Urban, J.-F. Nezan, O. Déforges and C. Moy. SynDEx Executive Kernels For Fast Developments Of Applications Over Heterogeneous Architectures. dans XIII European Signal Processing Conference (EUSIPCO). Septembre 2005.

[Urb04] Fabrice URBAN. Transmission de flux vidéo sur systèmes de communication 3G et 4G. Projet de Fin d'Etudes. IETR INSA Rennes - Mitsubishi Electric ITE. 2004.

[RMU+05] M. Raulet, C. Moy, F. Urban, J.-F. Nezan, O. Deforges and Y. Sorel. Rapid prototyping for heterogeneous multicomponent systems, dans EURASIP Journal on Applied Signal Processing. Novembre 2005.

[Urb03] Fabrice URBAN. Développement de noyaux d'exécutifs SynDEx pour plates-formes hétérogènes DSP-FPGA. Rapport de stage. IETR INSA Rennes - Mitsubishi Electric ITE. Juillet-aout 2003.

Bibliographie

[AH02] Yakup Paker Anastasios Hamosfakidis. A Novel Hexagonal Search Algorithm for Fast Block. *EURASIP Journal on Applied Signal Processing*, 6 :595–600, 2002.

[BdHSvM03] A. Beric, G. de Haan, R. Sethuraman, and J. van Meerbergen. A technique for reducing complexity of recursive motion estimation algorithms. In *IEEE Workshop on Signal Processing Systems*, pages 195 – 200, 2003.

[BLBP87] J. Biemond, L. Looijenga, D. E. Boekee, and R. H. J. M. Plompen. A pel-recursive wiener-based displacement estimation algorithm. *Signal Process.*, 13(4) :399–412, 1987.

[CCC94] M.J. Chen, L.G. Chen, and T.D. Chiueh. One-dimensional full search motion estimation algorithm for video coding. *IEEE Transactions on Circuits and Systems for Video Technology*, 4(4) :504–509, 1994.

[CCH+06] Ching-Yeh Chen, Shao-Yi Chien, Yu-Wen Huang, Tung-Chien Chen, Tu-Chih Wang, and Liang-Gee Chen. Analysis and architecture design of variable block-size motion estimation for H.264/AVC. *IEEE Transactions on Circuits and Systems I*, 53(2) :578 – 593, February 2006.

[CCHBC04] Tuan-Kiang Chiew, James T. H. Chung-How, David R. Bull, and C. Nishan Canagarajah. Interpolation-free subpixel refinement for block-based motion estimation. In *SPIE*, volume 5308, pages 1261–1269, Jan 2004.

[CHC04] Tung-Chien Chen, Yu-Wen Huang, and Liang-Gee Chen. Fully utilized and reusable architecture for fractional motion estimation of H.264/AVC. *Internatinal Conference on Acoustics, Speech an Signal Processing*, 5 :9–12, 2004.

[CHF01] Y-S Chen, Y-P Hung, and C-S Fuh. Fast Block Matching Algorithm Based on the Winner-Update Strategy. In *IEEE Transactions on Image Processing*, volume 10, August 2001.

[CJJ03] W. Choi, B. Jeon, and J. Jeong. Fast motion estimation with modified diamond search for variable motion block sizes. In *IEEE Intern. Conf. On Image Proc. (ICIP)* , sep 2003.

[CLC06] Tung-Chien Chen, Chung-Jr Lian, and Liang-Gee Chen. Hardware architecture design of an h.264/avc video codec. In *ASP-DAC '06 : Proceedings of the 2006 conference on Asia South Pacific design automation*, pages 750–757, 2006.

[CM87] E. D. Castro and C. Morandi. Registration of translated and rotated images using finite fourier transforms. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume PAMI9 (5), pages 700–703, Sept 1987.

[CR83] C. Cafforio and F. Rocca. The differential method for image motion estimation. *Image sequence processing and dynamic scene analysis*, pages 104–124, 1983.

[CTHC07] Tung-Chien Chen, Chuan-Yung Tsai, Yu-Wen Huang, and Liang-Gee Chen. Single reference frame multiple current macroblocks scheme for multiple reference frame motion estimation in h.264/avc. *IEEE Transactions on Circuits and Systems for Video Technology*, 17(2), February 2007.

[CZH02] Zhibo Chen, Peng Zhou, and Yun He. Fast integer pel and fractional pel motion estimation for jvt. *JVT-F017, 6th Meeting of Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG*, 2002.

[CZH03] Z. Chen, P. Zhou, and Y. He. Fast motion estimation for JVT. JVT-G016.doc, March 2003.

[dHBHO93] G. de Haan, P.W.A.C. Biezen, H. Huijgen, and O.A. Ojo. True-motion estimation with 3-D recursive search block matching. In *IEEE Transactions on Circuits and Systems for Video Technology*, volume 3, pages 368 – 379, 1993.

[DRS05] Tiago Dias, Nuno Roma, and Leonel Sousa. Efficient motion vector refinement architecture for sub-pixel motion estimation systems. *IEEE Workshop on Signal Processing Systems Design and Implementation*, pages 313 – 318, 2005.

[DS89] Luc De Vos and Michael Stegherr. Parameterizable VLSI architectures for the full-search block-matching algorithm. *IEEE Transactions on Circuits and Systems*, 36(10) :1309–1316, October 1989.

[FABD06] Erwan Flécher, Samir Amir, Marie Babel, and Olivier Déforges. Lar video : Lossless video coding with semantic scalability. In *International Conference on Signals and Electronic Systems*, pages 227–230, Sept 2006.

[FDN02] V. Fresse, O. Déforges, and J.-F. Nezan. AVSynDEx : A Rapid Prototyping Process Dedicated to the Implementation of Digital Image Processing Applications on multi-DSPs and FPGA Architectures. *EURASIP journal on Applied Signal Processing, special issue on Implementationof DSP and Communication Systems*, 2002(9) :990–1002, September 2002.

[FWG98] Markus Flierl, Thomas Wiegand, and Bernd Girod. A locally optimal design algorithm for block-based multi-hypothesis motion-compensated prediction. In *Data Compression Conference*, pages 239–248, 1998.

[GBG+07] Klaus Gaedke, Malte Borsum, Marco Georgi, Andreas Kluger, Jean-Pierre Le Glanic, and Pascal Bernard. Architecture and VLSI implementation of a programmable HD real-time motion estimator. *IEEE International Symposium on Circuits and Systems*, May 2007.

[GHA90] M. GHANBARI. The cross-search algorithm for motion estimation. *IEEE Transactions on Communications*, 38(1) :950–953, 1990.

[GLS99] T. Grandpierre, C. Lavarenne, and Y. Sorel. Optimized Rapid Prototyping For Real-Time Embedded Heterogeneous Multiprocessors. In *CODES'99 7th International Workshop on Hardware/Software Co-Design*, Rome, mai 1999.

[GRA83] F. Glazer, G. Reynolds, and P. Anandan. Scene matching by hierarchical correlation. *Conference Comput. Vision Pattern Recognition*, pages 432,441, 1983.

[Gra00] T. Grandpierre. *Modélisation d'architectures parallèles hétérogènes pour la génération automatique d'exécutifs distribués temps réel optimisés.* Spécialité électronique, Université de Paris Sud, 2000.

[HCBC06] P.R. Hill, Tuan-Kiang Chiew, David R. Bull, and C. Nishan Canagarajah. Interpolation-free subpixel accuracy motion estimation. *IEEE transactions on Circuits and Systems for video technology*, 16(12) :1519–1526, December 2006.

[HL92] Chaur-Heh Hsieh and Ting-Pang Lin. Vlsi architecture for block-matching motion estimation algorithm. *IEEE Transactions on Circuits and Systems for Video Technology*, 2(2) :169–175, June 1992.

[HM99] P.I. Hosur and K.K. Ma. Motion vector field adaptive fast motion estimation. *Second International Conference on Information, Communications and Signal Processing*, 1999.

[JJ81] J. R. Jain and A. K. Jain. Displacement measurement and its application in interframe coding. *IEEE Transactions on Communications*, COM-29(12) :1799–1808, 1981.

[KDH+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. Technical Report 4/5, IBM J. RES. and DEV., JULY/SEPTEMBER 2005.

[KK04a] F. Kelly and A. Kokaram. Fast image interpolation for motion estimation using graphics hardware. In *IST/SPIE Electronic Imaging - Real-Time Imaging VIII*, San Jose, California, USA, January 2004.

[KK04b] F. Kelly and A. Kokaram. Graphics hardware for gradient based motion estimation. *IS and T/SPIE Electronic Imaging - Embedded Processors for Multimedia and Communications*, 2004.

[KLW87] Subhash C. Kwatra, Chow-Ming Lin, and Wayne A. Whyte. An adaptive algorithm for motion compensated color image coding. *IEEE Transactions on Communications*, 35(7) :747–754, 1987.

[LAJ+05] Yongfang Liang, I Ahmad, Luo Jiancong, Sun Yu, and V Swaminathan. On using hierarchical motion history for motion estimation in H.264/AVC. 15 :1594–1603, December 2005.

[LF96] Lurng-Kuo Liu and Ephraim Feig. A block-based gradient descent search algorithm for block motion estimation in video coding. In *IEEE Transactions on Circuits and Systems for Video Technology*, volume 6, pages 292–296, Aug 1996.

[LL99] Jianhua Lu and Ming L. Liou. A simple and efficient search algorithm for block-matching motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology*, 7(2) :429–433, 1999.

[LL04] Jae Hun Lee and Nam Suk Lee ;. Variable block size motion estimation algorithm and its hardware architecture for H.264/AVC. *International Symposium on Circuits and Systems*, 3 :741–4, May 2004.

[LLS05] Tiejun Li, Sikun Li, and Chengdong Shen. A novel configurable motion estimation architecture for high-efficiency mpeg-4/h.264 encoding. In *Design Automation Conference*, volume 2, pages 1264–1267, Jan 2005.

[LR01] Yann Le Méner and Mickaël Raulet. *Développement d'un noyau temps réel pour DSP C6x intégré dans le générateur de code distribué SynDEx pour Architectures Multiprocesseurs*. IETR INSA Rennes - Mitsubishi Electric ITE, July-September 2001.

[LS95] W. Li and E. Salari. Successive elimination algorithm for motion estimation. *IEEE Transactions on Image Processing*, 4 :107–110, 1995.

[LS97] C. Lavarenne and Y. Sorel. Modèle unifié pour la conception conjointe logiciel-matériel. *Traitement du Signal*, 6 :14, 1997.

[LZL94] R.X. Li, B. Zeng, and M.L. Lion. A new 3-step search algorithm for block motion estimation. *IEEE transactions on Circuits and Systems for Video Technology*, 4 :438–442, 1994.

[Mac06] Alain Maccari. *Description sous SynDex de la plate-forme VP3-PMC de VITEC MULTIMEDIA*. IETR INSA Rennes - Vitec Multimédia, July-September 2006.

[MTCB05] Olivier Le Meur, Dominique Thoreau, Patrick Le Callet, and Dominique Barba. A spatio-temporal model of the selective human visual attention. In *IEEE International Conference on Image Processing*, pages III – 1188–91, Sept 2005.

[Mul05] Vitec Multimedia. *VP3 Board user manual*, 2005.

[MZ00] Marco Mattavelli and Giorgio Zoia. Vector-Tracing Algorithms for Motion Estimation in Large Search Windows. *IEEE Transactions on circuit and systems for video technology*, 10(8) :1426–1437, December 2000.

[NR79] A.N. Netravali and J.D. Robbins. Motion-compensated television coding : Part i. *Bell Syst. Technical Journal*, 58 :631–670, mar 1979.

[NVi06] NVidia. Nvidia geforce 8800 architecture technical brief. Technical report, NVIDIA Corporation, November 2006.

[OLG+05] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, aug 2005.

[PM96] Lai-Man Po and Wing-Chung Ma. A novel four-step search algorithm for fast block motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology*, 6(3) :313–317, 1996.

[Rau02] Mickaël Raulet. Hardware Resource and Software Application Description Methodology for Several Processing Boards on TI C6x DSP Processors. Master's thesis, IETR INSA Rennes - Mitsubishi Electric ITE, June 2002.

[RAU06] Mickaël RAULET. *Optimisations Mémoire dans la Méthodologie AAA pour Code Embarqué sur Architectures Parallèles.* Electronique et traitement du signal, INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE RENNES, 2006.

[RB05] Choudhury A. Rahman and Wael Badawy. A quarter pel full search block motion estimation architecture for H.264/AVC. *IEEE International Conference on Multimedia and Expo*, July 2005.

[RBN+03] M. Raulet, M. Babel, J.-F. Nezan, O. Déforges, and Y. Sorel. Automatic Coarse Grain Partitioning and Automatic Code Generation for Heterogeneous Architectures. In *IEEE Workshop on Signal Processing Systems (SIPS'03)*, Seoul, Korea, August 27-29 2003.

[Ric03] Iain E.G. Richardson. *H.264 and MPEG-4 Video Compression : Video Coding for Next-generation Multimedia.* John Wiley and Sons, 2003.

[RRND06] Ghislain ROQUIER, Mickaël RAULET, Jean-François NEZAN, and Olivier DEFORGES. Using RTOS in the AAA Methodology Automatic Executive Generation. In *European Signal Processing Conference (EUSIPCO)*, 2006.

[SDL+04] Sergio Saponara, Kristof Denolf, Gauthier Lafruit, Carolina Blanch, and Jan Bormans. Performance and complexity co-evaluation of the advanced video coding standard for cost-effective multimedia communications. *EURASIP Journal on Applied Signal Processing*, 2 :220–235, 2004.

[SLGI05] Yang Song, Zhenyu Liu, Satoshi Goto, and Takeshi Ikenaga. Motion estimation algorithm modification and implementation in h.264/avc. *18th Workshop on Circuits and Systems,IEICE*, pages 193–198, April 2005.

[SR85] R. Srinivasan and K.R. Rao. Predictive coding based on efficient motion estimation. *IEEE Transactions on Communication*, pages 888–896, 1985.

[SR99] Byung Cheol Song and Jong Beom Ra. A fast motion estimation algorithm based on multi-resolution frame structure. 6 :3361–3364, March 1999.

[STL04] Gary J. Sullivan, Pankaj Topiwala, and Ajay Luthra. The h.264/avc advanced video coding standard : Overview and introduction to the fidelity range extensions. *SPIE Conference on Applications of Digital Image Processing*, pages 220–235, 2004.

[Sto86] R. Storey. HDTV Motion Adaptive Bandwidth Reduction using DATV. Technical Report BBC RD 1986/5, BBC Research Department, 1986.

[STS06] Gary Sullivan, Alexis Michael Tourapis, and Karsten Sühring. *H.264/MPEG-4 AVC Reference Software Manual.* Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, october 2006.

[Sun] Sundance. *SMT6500 package help file.*

[Sun04] Sundance. *SMT395 user manual*, 2004.

[SW98] G. Sullivan and T. Wiegand. Rate-Distortion Optimization for Video Compression. *IEEE Signal Processing Magazine*, pages 74–90, Nov 1998.

[TAL01a] A. M. Tourapis, O. C. Au, and M. L. Liou. New results on zonal based motion estimation algorithms - advanced predictive diamond zonal search. In *IEEE International Symposium on Circuits and Systems*, volume 5, pages 183–186, Sydney, Australia, May 2001.

[TAL01b] Alexis Michael Tourapis, Oscar C. Au, and Ming Lei Liou. Predictive Motion Vector Field Adaptive Search Technique (PMVFAST) Enhancing Block Based Motion Estimation. In *Visual Communications and Image Processing*, 2001.

[Tex6a] Texas Instruments. TMS320C6000 DSP Cache User's guide, spru656a.

[Tho87] G.A. Thomas. Television motion measurement for DATV and other applications. Technical Report BBC RD 1987/11, BBC Research Department, 11 1987.

[TKA+81] T.Koga, K.Linuma, A.Hirano, Y.Iijima, and T.Ishiguro. Motion compensated interframe coding for video conferencing. *Proceedings of National Telecommunication Conference*, NTC81 :G5.3.1–G5.3.5, 1981.

[TM93] M. Tomasevic and V. Milutinovic. A survey of hardware solutions for maintenance of cache coherence in shared memory multiprocessors. In *Proceeding of the Twenty-Sixth Hawaii International Conference on System Sciences*, volume 1, pages 863 – 872, August 1993.

[Tou02] Alexis Michael Tourapis. Enhanced Predictive Zonal Search for Single and Multiple Frame Motion Estimation. *proceedings of Visual Communications and Image Processing*, pages 1069–79, 2002.

[TRRK98] J. Y. Tham, S. Ranganath, M. Ranganath, and A. A. Kassim. A Novel Unrestricted Center-Biased Diamond Search Algorithm for Block Motion Estimation. *IEEE Transactions on circuits and systemes for video technology*, 8, NO. 4, AUGUST 1998.

[TTT02] Hye-Yeon Cheong Tourapis, Alexis Michael Tourapis, and Pankaj Topiwala. Fast motion estimation within the JVT codec. JVT-E023.doc, March 2002.

[VKM+05] K. Virk, N. Khan, S. Masud, F. Nasim, and S. Idris. Low Complexity Recursive Search Based Motion Estimation Algorithm for Video Coding Applications. In *Proceedings of 13th European Signal Processing Conference*, Antalya, Turkey, 2005.

[WBSS04] Zhou Wang, Alan Conrad Bovik, Hamid Rahim Sheikh, and Eero P. Simoncelli. Image quality assessment : From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4) :600–612, April 2004.

[WR84] D. R. WALKER and K. R. RAO. Improved pel-recursive motion compensation. *IEEE Transactions on Communications*, 10 :1128–1134, October 1984.

[WSBL03] Thomas Wiegand, Gary J. Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the H.264/AVC video coding standard. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(7) :560–576, July 2003.

[YGI06] Changqi Yang, Satoshi Goto, and Takeshi Ikenaga. High performance vlsi architecture of fractional motion estimation in h.264 for HDTV. *IEEE International Symposium on Circuits and Systems*, pages 2605–2608, 2006.

[YH95] Hangu Yeo and Yu Hen Hu. A novel modular systolic array architecture for full-search blockmatching motion estimation. *International Conference on Acoustics, Speech, and Signal Processing*, 5 :3303–3306, May 1995.

[YM04] Swee Yeow Yap and John V. McCanny. A VLSI architecture for variable block size video motion estimation. *IEEE Transactions on Circuits and Systems : Expess Briefs*, 51(7) :384–389, July 2004.

[ZLC02] Ce Zhu, Xiao Lin, and Lap-Pui Chau. Hexagon-based Search pattern for Fast Block Motion Estimation. *IEEE Transactions on circuit and systems for video technology*, 12(5) :349–355, May 2002.

[ZLCP04] Ce Zhu, Xiao Lin, Lap-Pui Chau, and Lai-Man Po. Enhanced Hexagonal Search for Fast Block Motion Estimation. *IEEE Transactions on circuit and systems for video technology*, 14(10) :1210–1214, OCTOBER 2004.

[ZM97] Shan Zhu and Kai-Kuang Ma. A new diamond search algorithm for fast block matching motion estimation. In *International Conference on Information, Communications and Signal Processing*, volume 9, pages 292–296, sept 1997.

[ZSH04] Zhi Zhou, Ming-Ting Sun, and Yuh-Feng Hsu. Fast variable block-size motion estimation algorithms based on merge and split procedures for h.264/mpeg-4 avc. *International Symposium on Circuits And Systems*, 3 :725–8, May 2004.