ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

COMME EXIGENCE PARTIELLE
À L'OBTENTION DE LA
MAÎTRISE EN GÉNIE
M. Ing.

PAR
Jean-François FRANCHE

OPTIMISATION D'ALGORITHMES DE CODAGE VIDÉO SUR DES PLATEFORMES À PLUSIEURS PROCESSEURS PARALLÈLES

MONTRÉAL, LE 10 JANVIER 2011

PRÉSENTATION DU JURY

CE MÉMOIRE A ÉTÉ ÉVALUÉ

PAR UN JURY COMPOSÉ DE

M. Coulombe, directeur de mémoire
Département de génie logiciel et des TI à l'École de technologie supérieure

M. Tony Wong, président du jury
Département de génie de la production automatisée à l'École de technologie supérieure

Eric Paquette, membre du jury
Département de génie logiciel et des TI à l'École de technologie supérieure

IL A FAIT L'OBJET D'UNE SOUTENANCE DEVANT JURY ET PUBLIC

LE 14 DÉCEMBRE 2010

À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Je tiens à remercier chaleureusement mon directeur de recherche, monsieur Stéphane Coulombe, professeur à l'École de technologie supérieure, pour son grand support dans la réalisation de ce travail. Par son soutien moral et technique ainsi que par sa disponibilité, il a grandement contribué à l'aboutissement de ce projet. Je remercie aussi Steven Pigeon, professionnel de recherche à la chaire de recherche industrielle Vantrix en optimisation vidéo, pour l'aide qu'il a apportée durant ce projet, notamment, lors de sa rédaction.

Merci à Vantrix, entreprise québécoise spécialisée dans l'optimisation de services vidéo mobile et dans la livraison de solutions en matière de vidéo en continu, de navigation web et de messagerie, pour sa contribution financière, technique et matérielle. Merci aussi au conseil de recherches en science naturelles et en génie du Canada (CRSNG) pour son apport financier.

Je remercie également Chantal Gamache, Conseillère au Service d'appui en communication (SAC) à l'École de technologie supérieure, pour son appui à la rédaction lors de la révision du texte.

Enfin, j'aimerais remercier mes parents, mes frères, ma sœur et mes amis pour leur soutien moral tout au long de ce travail.

ALGORITHMS OPTIMIZATION FOR VIDEO CODING ON MULTI-PLATFORM PARALLEL PROCESSORS

Jean-François FRANCHE

ABSTRACT

H.264 is the most recent and efficient standard for video compression. This standard increases the compression ratio of its predecessors, by a factor of at least two, but at the cost of higher complexity. To reduce the encoding time, several H.264 encoders use a parallel approach. In this project, our primary objective is to design an approach that offers better acceleration than the parallel approach implemented by Intel in their H.264 encoder, delivered as sample code in the IPP library.

We present our multi-frame, multi-slice parallel video encoding approach and its motion estimation modes that offer various tradeoffs between acceleration and visual quality loss. The first mode, the fastest and the one that affects quality the most, performs motion estimation within the limits of the current slice. The second mode, slower than the first but offering better video quality, extends motion estimation to neighboring reference slices which have already been processed. The third mode, slower than the second, but providing even better quality, processes a slice only when all its reference slices have been processed.

Our experiments show that the first mode of our approach provides an average acceleration which is higher by about 55% than that obtained by the Intel approach. They also show that we obtain accelerations comparable to those obtained by state-of-the-art approaches, but without the disadvantage of having to use bidirectional frames. Furthermore, our approach can be implemented quickly in an H.264 encoder, which, like Intel's, is based on a multi-slice approach.

Keywords: video encoding, H.264, parallel algorithms, multi-core processor

OPTIMISATION D'ALGORITHMES DE CODAGE VIDÉO SUR DES PLATEFORMES À PLUSIEURS PROCESSEURS PARALLÈLES

Jean-François FRANCHE

RÉSUMÉ

H.264 est le standard de codage vidéo le plus récent et le plus puissant. Ce standard permet, par rapport à ses prédécesseurs, d'augmenter le taux de compression par un facteur d'au moins deux, mais au prix d'une complexité plus élevée. Pour réduire le temps d'encodage, plusieurs encodeurs H.264 utilisent une approche parallèle. Dans le cadre de ce travail de recherche, notre objectif premier est de concevoir une approche offrant une meilleure accélération que l'approche implémentée dans l'encodeur H.264 d'Intel livré en code d'exemple dans sa librairie IPP.

Nous présentons notre approche d'encodage vidéo parallèle multi-trames et multi-tranches (MTMT) et ses modes d'estimation de mouvement qui offrent un compromis entre l'accélération et la perte de qualité visuelle. Le premier mode, le plus rapide, mais dégradant le plus la qualité, restreint la région de recherche de l'estimation de mouvement à l'intérieur des limites de la tranche courante. Le second mode, moins rapide, mais dégradant moins la qualité que le premier, élargit la région de recherche aux tranches voisines, quand les tranches de référence y correspondant ont été traitées. Le troisième mode, moins rapide que le second, mais dégradant moins la qualité, rend une tranche prête à l'encodage seulement quand les tranches de référence couvrant la région de recherche ont été traitées.

Nos expériences montrent que le premier mode de notre approche offre une accélération moyenne environ 55 % plus élevée que celle obtenue par l'approche d'Intel. Nos expériences montrent aussi que nous obtenons une accélération comparable à celle obtenue par l'état de l'art sans l'inconvénient de forcer l'utilisation des trames B. De plus, notre approche s'implémente rapidement dans un encodeur H.264 qui, comme l'encodeur H.264 d'Intel, est basé sur une approche multi-tranches.

Mots clés : encodage vidéo, H.264, algorithme parallèles, processeur multicœur

TABLE DES MATIÈRES

Page

LISTE DES TABLEAUX

Page

LISTE DES FIGURES

Page

LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

BS	boundary-strength
CAVLC	context-adaptive variable-length codes
CABAC	context-adaptive arithmetic coding
CBR	constant bit rate
DCT	discrete cosine transform
LBCS	large block consistency score
HD	haute définition
HT	Hyper-Threading
IDR	instantaneous decoder refresh
IPP	integrated performance primitives
MTMT	multi-trames et multi-tranches
NAL	network abstraction layer
PSNR	peak signal to noise ratio
RBSP	raw byte sequence payload
SVU	système visuel humain
UC	unité central
VBR	variable bit rate
VCL	video coding layer
ZMCS	zero motion consistency score

INTRODUCTION

Problématique

H.264 est le nom du standard de codage vidéo le plus récent et le plus puissant. Ce standard, développé conjointement par l'ITU-T et l'ISO/IEC, deux organismes internationaux de normalisation, a été conçu dans le but d'obtenir des taux de compression au moins deux fois plus élevés que ceux obtenus, pour un même niveau de qualité et pour une même séquence vidéo, par ses prédécesseurs, les standards H.263 et MPEG-4. Pour obtenir de telles performances, les concepteurs du standard H.264 ont introduit plusieurs nouveaux outils de compression qui ont l'inconvénient d'augmenter la complexité d'encodage. L'avènement de la télévision à haute résolution (HD) et la croissance continue de la bande passante des réseaux, qui ont pour effet d'augmenter la quantité, la qualité et la résolution des vidéos à encoder, contribuent eux aussi à l'augmentation de la complexité d'encodage. Cette complexité est telle, que plusieurs encodeurs H.264 ont de la difficulté à répondre aux exigences des utilisateurs. Par exemple, certains encodeurs n'arrivent pas à encoder une séquence vidéo à HD en temps réel. D'autres encodeurs sont incapables de fournir un nombre de canaux d'encodage qui répond aux besoins de l'utilisateur.

Pour réduire la complexité des encodeurs H.264, et ainsi pour mieux répondre aux besoins des utilisateurs, plusieurs solutions ont été proposées. Une première catégorie d'approches consiste à développer des algorithmes moins complexes, souvent au prix d'une perte de qualité. Parmi ce type d'algorithme, nous comptons, notamment, les algorithmes d'estimation de mouvement rapide et les algorithmes de présélection rapide des modes d'encodage. Une seconde catégorie d'approches exploite le jeu d'instruction de l'architecture cible, par exemple, en utilisant les instructions *Single Instruction Multiple* Data (SIMD) pour traiter en parallèle certaines données.

Le travail présenté dans le cadre de ce mémoire porte sur une troisième catégorie d'approches qui consiste à paralléliser un encodeur vidéo au niveau des tâches. Plus

spécifiquement, nous traiterons des approches logicielles basées sur des processeurs généraux à mémoire partagée, tels que les multiprocesseurs, les processeurs multicoeurs et les processeurs dotés d'une technologie multifil simultané (*Simultaneous Multithreading* (SMT) en anglais). Les systèmes à mémoire partagée ont l'avantage d'offrir des faibles coûts de communication entre les unités d'exécution composant le système, donc de permettre des échanges intensifs et rapides de données entre ces unités, puisque ces échanges s'effectuent directement au niveau de la mémoire. Le traitement multifil (*multithreading* en anglais), une technique qui permet d'exécuter simultanément plusieurs fils d'exécution[1] (*threads* en anglais) sur différentes unités d'exécution, est l'approche habituellement utilisée pour paralléliser des applications sur ce type de système.

Nous développerons une approche parallèle pour des processeurs multicœurs d'Intel et nous baserons nos travaux sur l'encodeur H.264 livré en code d'exemple avec la librairie *Integrated Performance Pritmitives* (IPP) d'Intel (Intel, 2010b). Cet encodeur a l'avantage d'implémenter une approche parallèle, de type multi-tranches, en plus de contenir du code optimisé pour des processeurs Intel. Nous avons choisi d'utiliser l'interface de programmation OpenMP. Cette interface, qui a été conçue pour des architectures à mémoire partagée, permet de paralléliser des applications en offrant une couche d'abstraction qui cache, en partie, la gestion des fils d'exécution.

Motivations

La motivation première de ce projet est de réduire le temps de traitement d'un encodage H.264 dans le but d'améliorer le service offert par un encodeur vidéo. Cette amélioration se traduit, par exemple, par une capacité à traiter en temps réel des séquences vidéo, par l'augmentation du nombre de canaux d'encodage vidéo disponibles sur un système ou encore par la possibilité d'utiliser des outils du standard H.264 qui sont inutilisables dans un encodeur insuffisamment optimisé.

---

[1] Dans ce document, *fil* et *fil d'exécution* sont des termes équivalents.

Nous avons opté pour une approche parallèle au niveau des tâches, puisque le paradigme actuel pour améliorer les performances des processeurs consiste à augmenter le nombre de cœurs qui composent un processeur. Notre approche aura donc pour avantage d'offrir des performances qui augmenteront graduellement avec l'évolution des processeurs pour les prochaines années.

Objectifs

L'objectif de la recherche est de développer une approche parallèle offrant une bonne accélération et une perte de qualité non significative (ou du moins acceptable compte tenu des gains en vitesse) pour l'encodage de séquence à haute définition (HD) dans un contexte temps réel (on désire minimiser la latence). Plus spécifiquement, nous avons les objectifs suivants :

- Développer une approche d'encodage parallèle offrant une meilleure accélération, au prix d'une perte de qualité plus élevée, que celle obtenue par l'approche d'Intel.
- Développer une approche d'encodage parallèle la plus transparente possible aux yeux de l'utilisateur, c'est-à-dire une approche qui ne réduit pas significativement les options d'encodage qui sont disponibles à l'utilisateur.
- Développer une approche d'encodage parallèle supportant un nombre variable d'unités d'exécution (cœurs) tout en offrant de bonnes performances. Idéalement, un encodage effectué avec quatre cœurs devrait être deux fois plus rapide qu'un encodage effectué avec deux cœurs, par exemple.

Organisation du mémoire

Ce mémoire est composé d'un chapitre d'introduction, de cinq chapitres de développement et d'un chapitre de conclusion. Les deux premiers chapitres ont pour objectif de transmettre les notions fondamentales du traitement parallèle et du standard H.264 au lecteur. Ainsi, au chapitre 1, nous introduirons le lecteur au traitement parallèle. Nous commencerons ce

chapitre par une introduction générale sur le traitement multifil et son utilisation dans un contexte parallèle. Par la suite, nous décrierons quelques architectures à mémoire partagée. Puis, nous discuterons de mesures d'évaluation et de prédiction des performances d'une application parallèle. Enfin, nous terminerons ce chapitre sur une brève explication du développement d'applications multifils. Au chapitre 2, nous présenterons les concepts importants du standard de codage vidéo H.264. Dans cette présentation, nous mettrons l'accent sur les éléments importants pour paralléliser un encodeur H.264, soit la structure hiérarchique d'une séquence vidéo, les relations de dépendances entre les fonctions et les données et les outils de prédiction du standard.

Au chapitre 3, nous présenterons l'état de l'art de l'encodage vidéo parallèle sur des systèmes à mémoire partagée. De plus, nous aborderons quelques solutions d'encodage parallèle employées sur des systèmes à mémoire distribuée. Ce chapitre débute par une présentation générale portant sur les exigences d'un encodeur vidéo parallèle et sur les différents types de décomposition utilisés pour paralléliser un encodeur vidéo. Par la suite, nous présenterons des solutions plus spécifiques.

Au chapitre 4, nous analyserons l'approche multi-tranches implémentée par Intel dans son encodeur H.264. Nous verrons que cette approche affecte peu, par rapport à une approche non parallèle, la qualité visuelle de la séquence vidéo encodée, mais a cependant l'inconvénient d'obtenir une accélération qui est nettement au dessous de l'accélération théorique maximale. Aussi, nous proposerons dans ce chapitre une modification simple, au niveau du filtre de déblocage de H.264, pour améliorer l'accélération de l'approche d'Intel.

Enfin, au chapitre 5, nous présenterons notre approche parallèle de type multi-trames et multi-tranches (MTMT) et ses trois modes qui offrent des compromis sur le plan de l'accélération et de la perte de qualité. Nous verrons que notre approche offre une accélération significativement supérieure à l'accélération obtenue par l'approche d'Intel au prix d'une perte de qualité supplémentaire, mais tout de même faible. Ainsi la méthode

proposée est très attrayante pour intégration dans des produits de compression vidéo H.264 temps-réel.

CHAPITRE 1

LE MULTIFIL SUR LES SYSTÈMES MULTICŒURS

Dans ce chapitre, nous introduirons brièvement quelques notions fondamentales du traitement parallèle et concurrent. Plus spécifiquement, nous mettrons l'accent sur le traitement parallèle appliqué à des systèmes multicœurs grâce au traitement multifil. Le lecteur désirant approfondir ces sujets pourra se référer aux ouvrages de (Akhter et Roberts, 2006; Butenhof, 1997; Reinders, 2007).

Nous commencerons par une introduction générale sur le traitement multifil et son utilisation dans un contexte parallèle. Puis, nous présenterons trois types de système (les systèmes multiprocesseurs, les systèmes multicœurs et les systèmes dotés de la technologie Hyper-Threading (HT)) supportant l'exploitation parallèle du multifil. Par la suite, nous discuterons de mesures d'évaluation et de prédiction des performances d'une application parallèle. Nous invitons le lecteur à porter une attention particulière sur ces mesures de performances puisqu'elles seront appliquées à plusieurs reprises dans ce mémoire. Enfin, nous expliquerons brièvement le développement d'applications multifils. Plus spécifiquement, nous traiterons du cycle de développement d'une application parallèle, de la conception d'un algorithme parallèle et des interfaces de programmation parallèle.

1.1 Introduction

En informatique, la concurrence (*concurrency*) « […] is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other. The computations may be executing on multiple cores in the same die [...]. » (Gu *et al.*, 2009). Un système concurrent doit gérer efficacement l'utilisation et le partage de ses ressources entre les différentes applications en cours d'exécution. Dans la plupart des cas, cette gestion a pour objectif premier de réduire le temps de réponse du système et de maximiser l'utilisation du processeur. Le concept de concurrence est étroitement relié aux concepts du multitâche et du multifil. Ces deux techniques parentes se différencient

essentiellement par leur niveau de granularité. Le multitâche permet, à un ou à des utilisateurs, d'exécuter en simultané[2], plusieurs processus sur un même processeur. En informatique, un processus correspond à l'instance d'une application. Quant à lui, le traitement multifil, d'une granularité plus fine, permet d'exécuter plusieurs fils d'une même application à la fois. Le multitâche a été introduit dans les années 1960 à l'intérieur des ordinateurs centraux (*mainframe*) pour supporter l'interaction avec plusieurs utilisateurs en même temps, par l'intermédiaire de terminaux, grâce à des techniques de partage de temps. Pour sa part, le traitement multifil est devenu populaire au milieu des années 1990 (Hennessy, Patterson et Goldberg, 2003). Les concepteurs d'applications l'utilisaient alors, et l'utilisent toujours, pour permettre à une application d'effectuer du traitement en attendant la fin d'un blocage occasionné, par exemple, par des opérations d'entrées/sorties (E/S) dans un autre fil d'exécution.

La gestion de l'exécution simultanée des processus et des fils d'exécution est effectuée, sur les systèmes Unix, notamment, par un ordonnanceur préemptif. Ce type d'ordonnanceur alloue pour chacun des fils, qu'il exécute à tour de rôle, une tranche de temps maximale suffisamment courte pour donner l'illusion à l'utilisateur que les applications s'exécutent en parallèle et suffisamment haute pour minimiser la charge de travail du système. Si le fil en cours d'exécution est bloqué par une interruption ou une variable conditionnelle, l'ordonnanceur effectue alors un changement de fil pour permettre au processeur d'exécuter à nouveau des instructions.

D'un point de vue du système de type Unix, un processus est, entre autres, constitué d'au moins un fil d'exécution, d'un espace mémoire réservé et protégé (non-accessible par les autres applications) et de descripteurs de fichiers. Un fil d'exécution est composé de plusieurs instructions et représente conceptuellement une tâche de l'application. Les fils d'un

---

[2] Dans ce document, nous utilisons le terme « simultané » pour désigner des processus (ou des threads) qui sont exécutés, du point de vue utilisateur, en même temps. Cette exécution simultanée est dite parallèle, lorsque l'exécution simultanée est réelle. Dans le cas contraire, la simultanée de l'exécution est simulée.

même processus partagent une plage de mémoire commune, mais chaque fil conserve ses propres états (pointeur sur l'instruction courante, pointeur sur la pile, registre, etc.) nécessaires à son exécution. Chaque fil est associé par le système à l'un des quatre états d'exécution suivants : prêt, bloqué, en exécution et terminé. Pour l'ordonnanceur, il est moins coûteux d'effectuer un changement de fil qu'un changement de processus, puisqu'un changement de processus implique plusieurs mécanismes coûteux, dont le vidage potentiel de la cache. Enfin, notons qu'il existe trois niveaux de fil d'exécution : les fils niveau-usager, les fils niveau-noyau et les fils niveau-physique. Le premier niveau représente les fils qui sont créés et manipulés à l'intérieur d'une application : ce sont des fils gérés par les développeurs d'applications. Le second niveau représente les fils visibles depuis le système d'exploitation : ce sont des fils gérés par le noyau du système, et plus particulièrement par l'ordonnanceur. Finalement, le dernier niveau représente les fils physiques d'un système. Chaque fil physique est, entre autres, constitué d'un état d'exécution et d'un système d'interruption. Un fil physique est constitué d'une unité d'exécution dédiée ou partage une unité d'exécution avec un autre fil physique. Par exemple, un système composé de deux processeurs ayant chacun quatre cœurs dotés de la technologie HT aura 16 fils physiques (2 processeurs × 4 cœurs/processeur × 2 fils/cœur).

Grâce à l'arrivée récente des processeurs multicœur, qui occupent maintenant la majorité des parts du marché (IDC, 2006), le multifil, en plus d'être un mécanisme efficace pour gérer la concurrence des ressources, est devenu un modèle populaire de développement d'applications parallèles (Akhter et Roberts, 2006). Le modèle de programmation parallèle du traitement multifil appartient à la famille du parallélisme effectué au niveau des tâches. Ce type de parallélisme exploite l'idée que différentes parties d'un programme peuvent être exécutées simultanément et en parallèle sur plusieurs unités d'exécution. Une unité d'exécution correspond à une composante physique qui est capable de traiter un flot d'instructions. Ainsi, un processeur quad-cœur sera composé de quatre unités d'exécution et permettra à l'ordonnanceur d'exécuter en parallèle jusqu'à quatre fils.

1.2 Autres types de parallélismes

Il existe deux autres types de parallélisme sur les processeurs d'aujourd'hui : le parallélisme effectué au niveau des instructions et le parallélisme effectué au niveau des données (Hennessy, Patterson et Goldberg, 2003). Le premier type de parallélisme exploite l'indépendance qui existent entre certaines instructions rapprochées, afin d'exécuter plusieurs instructions en parallèle. Le traitement d'instructions en pipeline et le traitement superscalaire, qui peuvent être combinés dans une même architecture, permettent d'exploiter ce type de parallélisme. Le traitement d'instructions en pipeline divise les instructions en nombre fixe d'étapes simples qu'il exécute à la manière d'une chaîne de montage, tel qu'illustré à la figure 1.1.a. Le traitement superscalaire en pipeline est une technique améliorée du pipeline et peut être vue comme plusieurs chaînes de montage, tel qu'illustré à la figure 1.1.b. Le second type de parallélisme, plus connu sous le nom *Single Instruction Multiple Data* (SIMD) dans la taxonomie de Flynn (Flynn, 1966), permet de traiter plusieurs données en une seule instruction. Il est typiquement utilisé par des applications effectuant du traitement intensif sur les données, telles que les applications de traitement de signal numérique, les applications de traitement d'images et les applications multimédias traitant de l'audio et/ou du vidéo. D'abord utilisé par certains super ordinateurs et processeurs DSP (Hennessy, Patterson et Goldberg, 2003), ce type de parallélisme est maintenant offert par la majorité des ordinateurs récents sous la forme de différents jeux d'instructions étendus : MMX, SSE, SSE2, SSE3 et tout récemment SSE4 pour les processeurs Intel et AMD (Intel, 1999; Peleg et Weiser, 1996; Ramanathan *et al.*, 2006); Altivec (Diefendorff *et al.*, 2000) pour les processeurs PowerPC; et NEON (ARM, 2010) pour les processeurs ARM.

Contrairement au parallélisme au niveau des instructions, le parallélisme au niveau des tâches nécessite généralement une participation active de l'équipe de développement, et cela, dès le départ, car il implique une conception architecturale globale pensée en terme de parallélisme. Comme nous le verrons plus loin, les concepteurs doivent choisir une architecture parallèle correspondant à leurs besoins et doivent décomposer les tâches en fonction de l'architecture choisie et du type d'application développée. Les programmeurs

doivent apprendre le fonctionnement des outils de développement parallèle utilisés et, dans certains cas, les fondations de l'architecture parallèle choisie.

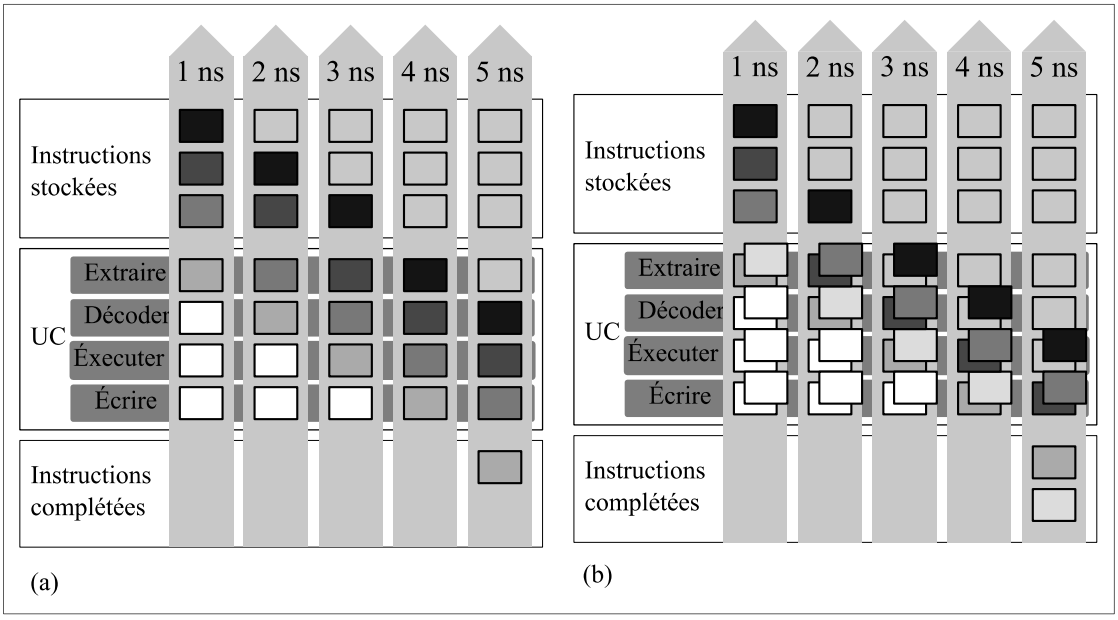


Figure 1.1 a) Schéma d'un pipeline à quatre étages et b) schéma d'une architecture superscalaire combiné avec un pipeline à quatre étages.
Adaptée de (Stokes, 2007)

1.3 Systèmes parallèles adaptées au traitement multifil

Les systèmes multiprocesseurs symétriques, les systèmes multicœur et les systèmes implémentant une forme de multifil simultané, par exemple, les systèmes dotés de la technologie HT d'Intel, sont, de nos jours, les trois types de technologies parallèles les plus adaptés à l'implémentation d'algorithmes parallèles basés sur le modèle multifil, puisqu'ils sont basés sur une mémoire partagée. Ces trois technologies, qui peuvent être combinés ensemble sur un même système, partagent plusieurs points communs aux niveaux système et usager, dont :

- Une architecture basée sur un modèle de mémoire partagée. Ce type de modèle mémoire permet aux données de circuler rapidement d'une unité d'exécution à une autre, cependant, ce modèle nécessite l'emploi de mécanismes de cohérences entre les caches des unités.

- Des unités d'exécutions qui sont perçues comme des processeurs logiques par les systèmes d'exploitation communs (tels que Linux et Windows, par exemple).

Comme nous le verrons plus en détail dans les trois prochaines sous-sections, ces trois types de technologie se distinguent essentiellement par leur microarchitecture. Notons qu'un système multiprocesseur (figure 1.2.b) est composé d'au moins deux processeurs intégrés sur le même support et reliés à la même mémoire par un bus, qu'un processeur HT (figure 1.2.c) est constitué de deux fils physiques qui se partagent un même ensemble d'unités d'exécution et une même cache et qu'un système multicœur (figure 1.2.d) contient au moins deux cœurs gravés sur une même puce. De plus, notons que ces technologies peuvent être intégrées dans un même système. Par exemple, la figure 1.2.e illustre un système multi-cœurs où chaque cœur est doté de la technologie HT.

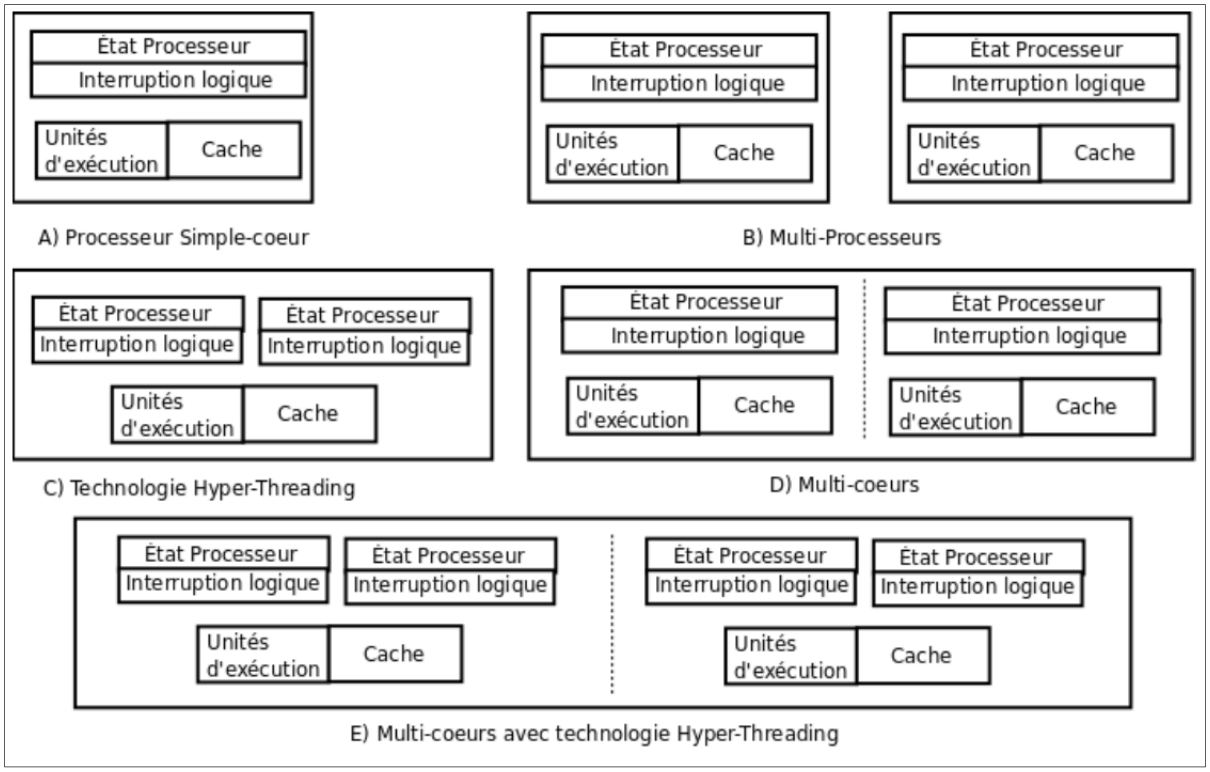


Figure 1.2 Différences architecturales entre les systèmes multiprocesseurs, multicœur et dotés de la technologie Hyper-Threading.
Adapté de (Akhter et Roberts, 2006)

1.3.1 Systèmes multiprocesseurs symétriques

Un ordinateur composé de deux ou de plusieurs processeurs est appelé un système multiprocesseur. Si tous les processeurs du système sont identiques, on parle alors d'un système multiprocesseur symétrique. Dans le cas contraire, le système est dit asymétrique. Une stratégie pour réduire les accès effectués vers la mémoire centrale du système consiste à équiper le système d'une cache de grande capacité. Malgré l'utilisation de ces caches, la mémoire centrale reste un goulot d'étranglement qui limite l'extensibilité de ces systèmes, qui dépassent rarement les 32 processeurs (Hennessy, Patterson et Goldberg, 2003; Hennessy *et al.*, 2003).

1.3.2 La technologie Hyper-Threading

L'HT est le nom donné par Intel à son adaptation du multifil simultané (Tullsen, Eggers et Levy, 1995). Cette adaptation, apparue en 2002, consiste à intégrer deux processeurs logiques sur une même puce pour remplir plus efficacement le flot d'instructions du processeur physique. Selon (Marr *et al.*, 2002), cette technologie permet d'augmenter la quantité d'instructions du processeur de 15 % à 30 % pour une augmentation de seulement 5 % de son nombre de transistors. En pratique, selon les mêmes auteurs, le gain de vitesse varie en fonction de l'application et peut, dans certains cas exceptionnels, être négatif.

En matière d'architecture, les deux fils physiques se partagent le cœur du processeur, le bus système et une partie de la cache. Chaque fil a ses propres registres, ses états sur le système et son contrôleur d'interruptions. Un algorithme, exécuté automatiquement par le processeur, permet de sélectionner les instructions qui seront utilisées pour remplir le flot d'instructions. Quand les deux fils sont actifs, la sélection s'effectue en alternance. Si l'un des deux fils est inactif, seules les instructions du fil actif seront chargées. La cache est séparée statiquement en attribuant la moitié de la mémoire à chaque fil. Cette approche simple nécessite peu de transistors, mais a pour désavantage d'augmenter le nombre d'absences d'informations dans la cache (*cache miss* en anglais), lorsqu'un fil physique n'utilise pas entièrement la cache qui lui

est allouée et l'autre, au contraire, manque de cache pour conserver l'ensemble de ses données (Tuck et Tullsen, 2003).

L'HT a d'abord été implémenté sur des serveurs Xeon, car ce type de technologie se prêtait bien aux applications multiclients, qui sont pour la plupart des applications multifils. Ensuite, cette technologie a été intégrée à d'autres processeurs Intel pour être remplacée progressivement par la suite par des architectures multicœurs. Aujourd'hui, Intel intègre de nouveau cette technologie dans certains processeurs multicœurs haute-performance, tel que les processeurs Nehalem (Singhal, 2008).

1.3.3 Systèmes multicœurs homogènes

Depuis quelques années déjà, l'industrie des microprocesseurs est passée à l'ère des processeurs multicœurs. Ce type de processeur combine, sur un même circuit, deux ou plusieurs cœurs physiques. Un cœur est une unité d'exécution indépendante, qui permet d'exécuter un flot d'instructions, composé notamment d'états, de registres et d'une cache. Selon la conception, les cœurs d'un même processeur peuvent partager une cache commune, communiquer entre eux par passage de messages (*message passing* en anglais), se partager une même mémoire, etc. Un système multicœur composé de cœurs identiques est dit homogène. Actuellement, Intel et AMD, les deux plus grands joueurs des processeurs utilisés à l'intérieur des ordinateurs personnels, commercialisent ce type de processeur.

Aujourd'hui, les processeurs multicœurs occupent presque 100 % du marché des processeurs utilisés dans les postes de travail, les ordinateurs portatifs et les serveurs. Cette nouvelle tendance, qui est là pour durer encore plusieurs années, est la réponse de l'industrie face aux limites atteintes par les paradigmes qui étaient alors employés pour développer les processeurs monocœur. Parmi ces limites, notons :

- **Le mur de la mémoire :** L'amélioration des performances des processeurs suit une courbe plus rapide que celle de la mémoire. Par conséquent, la vitesse de la mémoire est devenue un goulot d'étranglement qui limite les performances du système. Entre autres,

dans certaines situations, la latence mémoire est trop élevée pour remplir assez rapidement le flot d'instructions.

- **Le mur de la fréquence :** L'augmentation de la fréquence des processeurs dépend, notamment, de la réduction de la taille des transistors, qui dépend elle-même de d'autres avancées technologiques. De plus, l'augmentation des fréquences s'accompagne d'une augmentation déraisonnable de la puissance demandée, menant à d'autres complications comme la dissipation de chaleur. Par conséquent, l'augmentation de la fréquence de l'horloge d'un processeur est peu intéressante pour les ordinateurs personnels et inacceptables pour les systèmes embarqués alimentés par une batterie, et non un réseau électrique.
- **Le mur du parallélisme au niveau des instructions** : Similairement, la majorité des techniques utilisées pour améliorer le parallélisme au niveau des instructions a plafonné. Par exemple, il devient de plus en plus coûteux d'approfondir le pipeline. De plus, le code des applications limite lui-même le parallélisme, notamment, à cause des erreurs de prédictions et des dépendances entre les instructions et les données (Hennessy, Patterson et Goldberg, 2003).

Pour contourner ces limites, l'industrie a modifié son approche. La ligne directrice d'aujourd'hui consiste moins à complexifier davantage le parallélisme au niveau des instructions, mais plus à augmenter progressivement le nombre de cœurs sur les processeurs. Si la tendance se maintient, dans quelques années les processeurs seront composés de 16 cœurs et plus. On parlera alors de systèmes massivement multicoeurs (*many-core* en anglais).

Les processeurs multicœur assureront l'évolution des performances pour encore plusieurs années. Cependant, la forme exacte que prendront les prochaines générations de processeur reste indéterminée. Certains concepteurs croient que les processeurs multicœur prendront une forme hétérogène, similaire à celle des téléphones cellulaires, où chaque cœur est optimisé pour effectuer une tâche particulière. Ces concepteurs expliquent que c'est un excellent modèle pour réduire la consommation d'énergie. Au contraire, d'autres concepteurs croient

que les architectures homogènes gagneront une part importante du marché grâce à un modèle de développement logiciel plus simple.

Merrit rapporte dans (Merrit, 2008) un débat similaire sur l'évolution du nombre de cœurs. Dans ce débat, des concepteurs croient que les processeurs suivront la loi de Moore en doublant le nombre de cœurs tous les 16 mois. Si ce modèle tient la route, on pourrait compter environ 128 cœurs sur un processeur d'ordinateur de bureau, 512 cœurs sur un processeur de serveur et 4096 cœurs sur un processeur embarqué dès 2017. Toujours selon Merrit, d'autres concepteurs, plus conservateurs, prédisent plutôt des serveurs composés de 32 à 128 cœurs pour 2018.

1.4 Autres architectures parallèles

Parmi les autres types d'architectures parallèles, nous comptons, notamment : les grappes de serveurs (*computer cluster*), l'informatique en grille (*grid computing*), les processeurs massivement parallèles (*massively parallel processor (MPP)*) et les ordinateurs parallèles spécialisés. Les grappes de serveurs sont des serveurs, réunis dans un même endroit, qui fonctionnement ensemble comme s'il s'agissait d'une seule unité. Habituellement, ces serveurs sont connectés entre eux par un réseau local Ethernet et communiquent en utilisant le protocole TCP/IP. L'informatique en grille exploite sensiblement le même concept, mais cette fois-ci avec des réseaux d'ordinateurs connectés par Internet. Ce type d'architecture, qui a l'inconvénient d'avoir une forte latence de communication, est souvent utilisé pour résoudre des problèmes complexes qui nécessitent plusieurs calculs sur des ensembles indépendants de données. Les processeurs massivement parallèles correspondent à un système composé de plusieurs processeurs interconnectés par des réseaux de faible latence et de haut débit spécialisés. Dans ce type d'architecture, chaque processeur contient sa propre mémoire et une copie du système d'exploitation et de ses applications. Les ordinateurs parallèles spécialisés sont des ordinateurs développés pour répondre à des besoins d'un domaine particulier ou à quelques classes de problèmes parallèles. Ce type d'ordinateur inclut : les réseaux prédiffusés programmables par l'utilisateur (*Field-Programmable Gate*

Array (FPGA)); les unités de traitement graphique (Graphics processing units (GPU)) et les processeurs vectoriels (*vector processor*).

1.5 Les mesures de performance d'une application parallèle

Les performances d'une application parallèle sont généralement évaluées par deux mesures quantitatives : l'accélération et l'efficience. Ces mesures peuvent être calculées de manière pratique ou théorique. Dans le premier cas, elles sont utilisées pour évaluer les performances réelles d'une application exécutée sur un système spécifique. Dans le second cas, elles sont utilisées pour modéliser le parallélisme d'une application. Cette modélisation servira notamment à expliquer et à valider les résultats pratiques obtenus ainsi qu'à prédire les performances de l'application pour d'autres systèmes (réels ou pas).

1.5.1 L'accélération et l'efficience d'une application parallèle

Soient T_s le temps d'exécution séquentiel d'une application, $T_p(n)$ le temps d'exécution parallèle de cette même application pour n unités d'exécution. L'accélération d'une application se définit alors comme le rapport de T_s sur $T_p(n)$ et s'exprime sous la forme :

$$Accélération(n) = \frac{T_s}{T_p(n)} \tag{1.1}$$

En divisant le résultat de l'accélération par le nombre n d'unités d'exécution utilisées, on obtient la mesure de l'efficience pour un système homogène. Cette mesure s'exprime de la manière suivante :

$$Efficience(n) = \frac{Accélération(n)}{n} \tag{1.2}$$

Pour une application parallèle qui utilise n unités d'exécution et qui s'exécute sur un système homogène, l'accélération et l'efficience idéales (maximales) égalent respectivement n et 1.

Dans la réalité, ces valeurs sont rarement atteintes à cause du traitement supplémentaire nécessaire à la gestion du parallélisme.

1.5.2 La Loi d'Amdahl

Gene Amdahl a proposé en 1967 une loi (Amdahl, 1967), qui porte aujourd'hui son nom, pour estimer théoriquement l'accélération d'une application parallèle. La Loi d'Amdahl stipule que l'accélération d'une application parallèle est limitée par sa proportion *S* de code séquentiel. Cette loi se formule telle que :

$$Accélération(n) = \frac{1}{S + (1 - S) / n} \tag{1.3}$$

Pour un nombre *n* d'unités d'exécution égal à l'infini, l'accélération maximale de l'application se définie telle que :

$$Accélération_{\max imale} = \frac{1}{S} \tag{1.4}$$

La figure 1.3 illustre l'accélération obtenue par trois applications théoriques ayant respectivement une proportion de code séquentiel égale à 10, 20 et 50 %. Le graphique montre que :

- Un code fortement parallèle profite d'une meilleure accélération qu'un code qui l'est moins. Par exemple, le code séquentiel à 10 % a une accélération de 7.8 sur un système de 32 unités d'exécution, alors qu'un code séquentiel à 50 % n'a qu'une accélération de 2.
- Un code fortement parallèle atteint moins rapidement la limite supérieure de son accélération qu'un code qui l'est moins. Par exemple, l'accélération du code séquentiel à 10 % continue à progresser entre 16 et 32 unités d'exécution, alors que l'accélération du code séquentiel à 50 % atteint sa limite vers les 16 unités d'exécution.

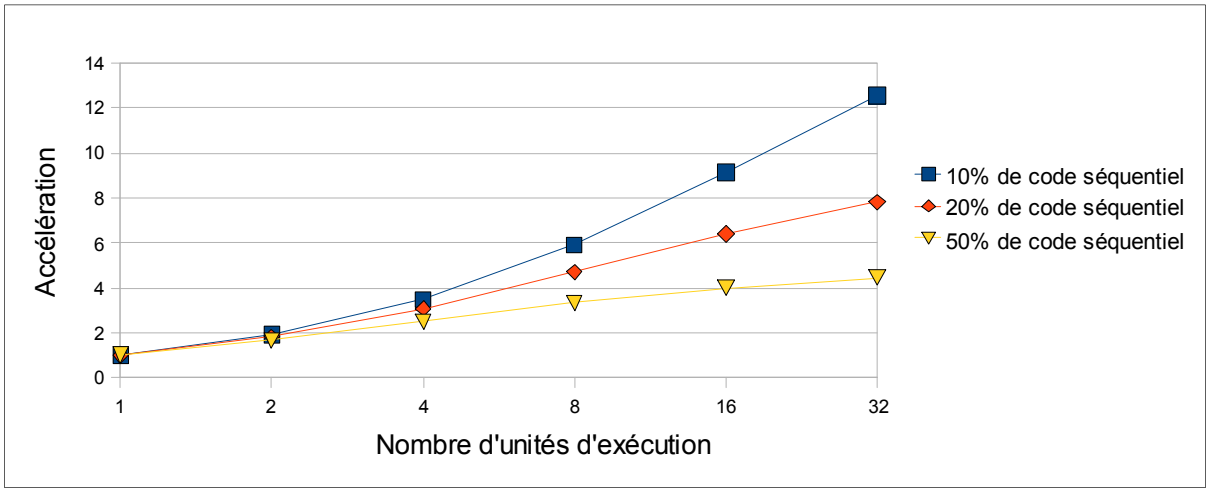


Figure 1.3 Loi d'Amdahl sur l'accélération.

En pratique, l'accélération réelle obtenue est en dessous des prédictions effectuées par l'équation 1.3. Nous pouvons donc rajouter à cette équation une variable *H(n)* qui représente la charge de travail supplémentaire produite par le parallélisme. Pour être le plus fidèle possible au système, cette variable doit tenir compte de facteurs tels que la communication et la synchronisation inter-fils, la gestion système des fils d'exécution et des processus, ainsi que des coûts architecturaux du système (cache, partage de la mémoire principale, limite de la bande passante, etc.). On obtient alors cette équation :

$$Accélération(n) = \frac{1}{S + (1-S)/n + H(n)} \tag{1.5}$$

La conséquence la plus importante de la loi d'Amdahl porte sur l'optimisation. Pour une application séquentielle, la règle d'or consiste à trouver et à optimiser seulement les sections critiques. Avec le traitement parallèle, les choses sont légèrement différentes. Pour maximiser l'accélération, toutes les sections parallélisables doivent être parallélisées. En effet, une section de code non-parallélisé qui représente 5 % du temps d'exécution de l'application a peu d'impact sur l'accélération d'un processeur composé de deux unités

d'exécution. Cependant, cet impact deviendra très significatif sur un processeur composé de 32 unités d'exécution.

1.5.3 La loi de Gustafson

En 1988, Gustafson proposa une loi (Gustafson, 1988) sur l'accélération tenant compte de l'augmentation de la charge de travail au fil des ans. Cette loi a pour équation :

$$Accélération = n - S(n-1) \tag{1.6}$$

où n et S représentent respectivement, comme plus haut, le nombre d'unités d'exécution et la proportion de code séquentiel. Selon ses recherches, la proportion du temps passé dans l'exécution de code séquentiel diminue avec l'augmentation de la charge de travail. Par exemple, une application développée aujourd'hui pour traiter des vidéos CIF (vidéos d'une résolution de 352×288 pixels) sur un processeur quad-cœur, pourra effectuer le même traitement dans quelques années sur des vidéos à HD (d'une résolution de 1280×720 pixels, par exemple) sur un processeur octo-cœur. En somme, l'accélération théorique calculée par l'équation de Gustafson sera plus élevée que celle d'Amdahl, car elle tient compte de la charge de travail.

1.6 Développement d'applications multifil

Cette section est consacrée au développement d'application parallèle multifil. D'abord, elle aborde les deux types de décomposition utilisés pour effectuer du traitement parallèle au niveau des tâches : la décomposition fonctionnelle et la décomposition du domaine. Par la suite, elle décrit un cycle développement adapté à la parallélisation d'applications séquentielles. Puis, elle aborde une méthodologie de conception d'algorithmes parallèles effectuée en quatre étapes. Finalement, elle se consacre à la description de quelques interfaces de programmation parallèle.

1.6.1 Décomposition fonctionnelle et décomposition du domaine

L'objectif du parallélisme par tâche est de diviser l'application en groupes de tâches qui peuvent s'exécuter en parallèle sur plusieurs unités d'exécution. Nous pouvons diviser l'application au niveau des fonctions, on parle alors de décomposition fonctionnelle, ou encore au niveau de la structure de données, on parle alors de décomposition du domaine. Dans (Mattson, Sanders et Massingill, 2004), les auteurs décrivent ces décompositions de la manière suivante :

- La décomposition fonctionnelle : Ce type de décomposition voit le problème comme une série d'instructions qui doivent être divisées en tâches à traiter simultanément. Pour être efficiente, la décomposition doit diviser les instructions en tâches relativement indépendantes.
- La décomposition du domaine : Ce type de décomposition voit le problème comme des données qui doivent être divisées en morceaux pour être traités simultanément. Pour être efficiente, la décomposition doit diviser les données en morceaux relativement indépendants.

1.6.2 Cycle de développement

Dans (Wang, 2007), Wang *et al.* présentent un cycle de développement adapté aux applications séquentielles qui doivent être parallélisées, ce qui correspond au cadre de nos travaux. Leurs cycles se divisent en quatre phases :

- La phase d'analyse : L'objectif de cette phase est d'analyser la version séquentielle du programme afin de déterminer les sections de code qui sont susceptibles d'être parallélisées. Cette analyse s'effectue par une simple lecture du code pour des applications de petite envergure et par des outils de profilage, tel que VTune, pour des applications plus imposantes. Dans les deux cas, il est important d'identifier les sections qui consomment le plus de temps sur le processeur.

- La phase de conception/implémentation : Les objectifs de cette phase sont d'examiner les sections candidates, de déterminer les changements à apporter au code séquentiel et de convertir le code vers sa version parallèle.
- La phase de déverminage : L'objectif de cette phase est de valider l'efficience de l'application parallèle. À cet effet, il est important de déterminer et de corriger les erreurs couramment rencontrées dans les applications multifils, telles que les situations d'interblocages (*deadlocks* en anglais) et les courses d'accès aux données (*data races* en anglais).
- La phase de test/mise au point : L'objectif de cette phase est de tester les performances de l'application. Il est primordial de détecter les sources de ralentissement et de les corriger pour améliorer les performances. Des outils comme Intel Thread Profiler et Intel VTune Performance facilitent la recherche et la découverte de sources de ralentissement (Intel, 2010a).

1.6.3 Conception d'un algorithme parallèle

Dans (Foster, 1995), Foster présente une méthodologie en quatre étapes pour concevoir un algorithme parallèle. Avant de présenter sa méthode, il nous rappelle quelques points clés sur la conception d'un algorithme parallèle. D'abord, il nous indique que la conception d'un algorithme parallèle a pour but de convertir les spécifications d'un problème en un algorithme parallèle en tenant compte de la concurrence et de la mise à l'échelle. Ensuite, il nous précise qu'un passage direct d'un algorithme séquentiel vers un algorithme parallèle n'est pas toujours approprié. Dans certains cas, l'algorithme séquentiel devra être complètement repensé pour être efficace dans un contexte parallèle.

Sa méthodologie est composée des quatre étapes suivantes : partitionnement, communication, agglomération et mappage. Les deux premières étapes sont indépendantes de la machine cible et ont pour but d'explorer les possibilités offertes par le parallélisme. Les deux dernières étapes sont des étapes dépendantes de la machine et ont pour objectif d'adapter la structure

obtenue lors des deux premières étapes aux spécifications de la machine. Ces quatre étapes se présentent ainsi :

- Partitionnement : On décompose l'activité à paralléliser en petites tâches sans tenir compte des spécifications du système. L'activité pourra être divisée en décomposition fonctionnelle ou en décomposition du domaine.
- Communication : On définit les structures de communication nécessaires à la coordination des tâches. Les structures de communication peuvent être locales ou globales, structurées ou pas, statiques ou dynamiques, ainsi que synchrones ou asynchrones.
- Agglomération : On évalue les performances obtenues par le découpage précédent en fonction de la machine cible. Si le coût des communications et de la synchronisation est élevé, on tente de regrouper des tâches pour le réduire.
- Mappage : On assigne les tâches sur le processeur de manière à maximiser les performances et à minimiser les coûts de la communication. Le mappage s'effectue statiquement ou dynamiquement grâce à des algorithmes de balancement de charge.

1.6.4 Interfaces de programmation

Plusieurs interfaces de programmation permettent de développer des applications multifils à l'aide des langages C/C++, cependant seules trois interfaces se démarquent du lot pour le développement d'application parallèle exécuté sur des processeurs généraux, soient : Pthread, OpenMP et Intel Threading Building Blocks *(Barney, 2010; Intel, 2010a; OpenMP Architecture Review Board, 2010)*. La première interface, qui est la plus ancienne et la plus primitive, permet de gérer les fils d'exécution grâce à l'appel de fonction C s'interfaçant au noyau du système d'exploitation. La seconde interface, qui est un standard multiplateforme, permet d'utiliser des commandes du précompilateur C pour paralléliser des boucles et gérer des queues de tâches sans devoir gérer directement les fils d'exécution. Enfin, la troisième interface, plus récente et plus transparente que les deux autres, est une librairie C++ créée par Intel pour offrir une importante couche d'abstraction aux concepteurs et développeurs.

Dans le cadre de nos travaux, nous utiliserons OpenMP, puisqu'il permet de développer rapidement des prototypes d'applications parallèle et parce qu'Intel utilise déjà cette librairie dans le code de son encodeur H.264. Cette interface de programmation permet d'effectuer du calcul parallèle sur des architectures à mémoire partagée. OpenMP est supporté par plusieurs plateformes ainsi que par les langages de programmation C/C++ et Fortran. Cette librairie est constituée d'un ensemble de directives, d'une bibliothèque logicielle et de variables d'environnement. Les directives indiquent au compilateur quelles sont les parties qui doivent être parallélisées et comment elles doivent l'être. La bibliothèque offre des fonctions sur les fils d'exécution et permet d'exécuter le code. Les variables d'environnement permettent de paramétrer l'exécution d'une application, par exemple, en choisissant le type de d'ordonnanceur.

La première version de ce standard fut publiée en 1997. À cette époque, ses créateurs avaient pour objectif de paralléliser des applications sans devoir se soucier des menus détails reliés à la gestion des fils d'exécution, au balancement de charge et à l'ordonnancement. L'approche consistait alors à paralléliser les boucles pour améliorer la performance des applications séquentielles. Dans certains cas, la parallélisation d'une boucle s'effectue simplement en rajoutant une directive au début de la boucle. Dans d'autres cas, le corps de la boucle doit être modifié pour respecter les restrictions d'OpenMP (par exemple, ne pas avoir de saut d'instructions à l'intérieur de la boucle). Pour gérer les boucles, le développeur doit choisir parmi trois types d'ordonnanceur : statique, guidé ou dynamique.

En 2008, OpenMP introduisait la version 3.0 de son standard. Cette version a pour particularité de permettre la parallélisation au niveau des tâches. Ce type de parallélisation présente une granularité plus grossière que celle offerte par la parallélisation par boucle. Notons que la majorité des compilateurs d'aujourd'hui supportent une version d'OpenMP.

OpenMP a pour principal avantage d'offrir une interface qui permet de paralléliser rapidement et facilement une application séquentielle. Cependant, ce standard offre une flexibilité moindre que Pthread, puisqu'une partie importante des détails est cachée dans le

compilateur et dans la librairie d'exécution. Par contre, certaines limites d'OpenMP peuvent être contournées par l'emploi de certaines fonctions Pthread, qui complètent bien OpenMP.

1.7 Conclusion

Dans cette présentation portant sur les notions de base du calcul parallèle, nous avons vu que les processeurs multicœurs occupent pratiquement 100 % du marché des microprocesseurs d'aujourd'hui et que, pour les prochaines années, l'augmentation du nombre de cœurs présents sur un même processeur sera vraisemblablement la principale évolution utilisée pour augmenter la performance des processeurs. Nous avons aussi vu que le modèle multifil se prête très bien à la conception d'applications parallèles exécutées sur ce type de processeur et que l'interface de programmation OpenMP encapsule ce modèle de manière transparente, ce qui en fait un outil de choix. Enfin, nous avons présenté l'accélération et l'efficience, à titre de principales mesures employées, pour évaluer les performances d'une application parallèle. Dans le prochain chapitre, nous présenterons brièvement les grandes lignes du standard de codage vidéo H.264.

CHAPITRE 2

LE STANDARD DE CODAGE VIDEO H.264

Ce chapitre présente les concepts importants du standard de codage vidéo H.264. On y trouve une introduction du standard ainsi que la présentation de la séquence d'encodage, de la structure hiérarchique d'une séquence vidéo et des outils de prédiction et de compression du standard. De plus, nous présentons quelques mesures de qualité vidéo.

L'introduction a pour objectif de présenter les nouveautés présentes dans le standard H.264, alors que le reste du chapitre vise plutôt à aborder le standard selon un point de vue propre au domaine du traitement parallèle. Ainsi, nous en analyserons les différentes relations de dépendance qui touchent aux étapes d'encodage et aux structures de données. Pour le lecteur non initié au codage vidéo, nous suggérons la lecture des ouvrages suivants : *Fundamentals of multimedia* (Li et Drew, 2004; Richardson, 2003; Wang, Ostermann et Zhang, 2002).

2.1 Introduction

Le Video Coding Experts Group (VCEG) de l'ITU-T et le Moving Picture Experts Groups (MPEG) de l'ISO/IEC ont développé conjointement le standard de codage vidéo le plus récent et le plus performant, soit le standard H.264, aussi appelé MPEG-4 Advanced Video Coding (AVC). Ce standard a été publié pour la première fois en 2003 en tant que ITU-T Recommandation H.264 et ISO/IEC MPEG-4 Part 10 (Rec, 2003).

Le standard H.264 a été conçu pour répondre aux besoins et convenir aux moyens de l'industrie de la télécommunication, et plus généralement, des industries concernées par la compression vidéo, actuellement et dans un avenir rapproché. De plus, il a été élaboré pour apporter des contributions significatives par rapport aux standards précédents (H.261, H.263, MPEG-1, MPEG-2, MPEG-4) (ISO/IEC 11172, 1993; ISO/IEC 13818, 1995; ISO/IEC 14496-2, 2001; ITU-T Recommendation H.261, 1993; ITU-T Recommendation H.263, 1998). Plus spécifiquement, les concepteurs du standard H.264 avaient les principaux

objectifs suivants : améliorer le taux de la compression vidéo par un facteur d'au moins de 2 sans affecter significativement la qualité; obtenir une complexité qui tienne compte de l'état de l'art de l'industrie des semi-conducteurs; créer une interface simple et efficace pour supporter l'intégration de différents types de réseaux (Ethernet, LAN, ISDN, réseaux mobiles, etc.) et supports de stockage; avoir un ensemble d'outils et de profiles suffisamment large pour supporter un vaste éventail d'applications (vidéophonie, mobile, télévision, HD, etc.) et de conditions réseaux (perte et corruption de paquets, taille des paquets, etc.).

L'amélioration du taux de compression a été atteinte par l'emploi de deux nouvelles transformées, qui remplacent la *discrete cosine transform* (DCT), soit la transformée entière et la transformée d'Hadamard, ainsi que par l'ajout de nouveaux outils d'encodage qui permettent de mieux exploiter la redondance. Parmi ces outils, on compte, notamment, une compensation de mouvement pouvant utiliser jusqu'à sept tailles de blocs (16 × 16, 16 × 8, 8 × 16, 8 × 8, 8 × 4, 4 × 8 et 4 × 4) (voir la section 2.3.4) et précise au pixel, au demi-pixel, ou au quart-de-pixel; un post-filtrage anti-blocs pour réduire les artefacts (effets de bloc); un codage arithmétique; le support de plusieurs images de référence; une prédiction spatiale améliorée (Luthra et Topiwala, 2003).

L'abstraction du médium utilisé pour transporter ou stocker une séquence encodée s'effectue par l'intermédiaire d'unités de transport, appelées *Network Abstraction Layer* (NAL). Ces unités ont la responsabilité de contenir une charge utile de séquences d'octets bruts, appelées *Raw Byte Sequence Payload* (RBSP). Chaque unité NAL est composée d'un en-tête de la taille d'un octet qui indique le type du RBSP et d'un ensemble de données correspondant soit à des en-têtes d'information, soit à des données vidéo appartenant à la couche *Video Coding Layer* (VCL).

Le standard H.264 définit d'autres outils pour offrir un meilleur support réseau dont le partitionnement des données pour mieux protéger certaines données prioritaires lors du transport, l'utilisation de tranches de types SP et SI pour passer d'un flux vidéo à un autre et

pour effectuer des accès aléatoires efficaces, ainsi que la possibilité d'utiliser des images redondantes.

Enfin, tout comme ses prédécesseurs, H.264 est fondé sur une approche d'encodage par bloc où chaque image (trame) est divisée en plusieurs blocs d'une taille maximale de 16 × 16 pixels (voir section 2.3.4). Cette approche est essentiellement constituée des trois composantes suivantes :

- Un modèle temporel : Ce modèle, basé sur l'estimation de mouvement par bloc (voir section 2.2.1), exploite les similitudes qui existent entre des images (trames) voisines pour réduire la redondance temporelle lors de l'encodage.
- Un modèle spatial : Ce modèle, basé sur des méthodes de prédiction spatiale, exploite les similitudes qui existent entre des blocs voisins pour réduire les redondances spatiales lors de l'encodage. De plus, ce modèle transforme les données spatiales en données fréquentielles sur lesquelles il applique une quantification paramétrable.
- Un module de codage entropique : Ce module applique un codage entropique sur les données précédemment créées (voir section 2.2.3).

2.2 Séquence d'encodage

La séquence d'encodage H.264 est divisée en deux parties. La première se lit de gauche à droite et représente la construction d'une nouvelle trame. La seconde se lit de droite à gauche et représente la reconstruction de la trame encodée, qui sera utilisée ultérieurement comme trame de référence. La figure 2.1 illustre cette séquence d'encodage.

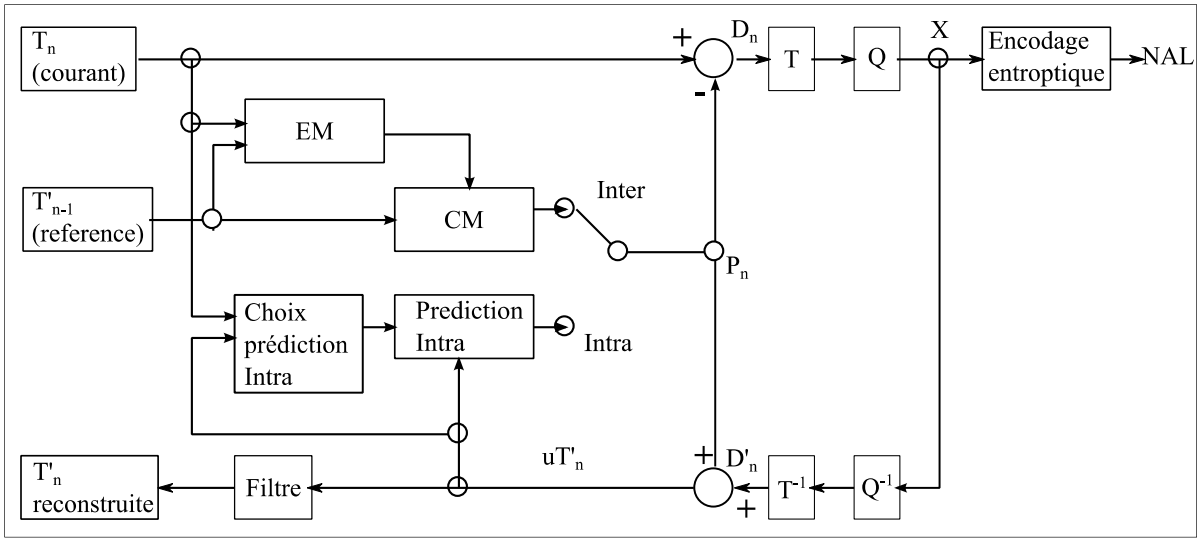


Figure 2.1 Schéma d'encodage H.264.
Adaptée de (Richardson, 2003)

La construction d'une nouvelle trame débute par l'arrivée au temps *n* d'une trame T_n dans l'encodeur. L'encodeur détermine alors le type de cette trame selon la stratégie implémentée. Si cette trame est de type inter (si elle exploite la redondance temporelle), une estimation et une compensation de mouvement, ayant T'_{n-1} pour trame de référence, est alors appliquée sur chaque macrobloc de la trame T_n pour produire des macroblocs P_n. Puis, l'encodeur calcule la différence entre ces macroblocs prédits et ceux de la trame T_n. Cette différence, qui constitue les données résiduelles, est conservée dans des macroblocs D_n. Si la trame est de type intra, les macroblocs D_n correspondent aux données résiduelles produites par la prédiction intra. L'encodeur applique ensuite une transformée et une quantification (voir section 2.2.2) sur les blocs de chaque macrobloc, pour produire les données *X*. Ensuite l'encodeur réordonne et compresse, à l'aide d'un codage entropique, les données qui rempliront le train de bits (*bitstream* en anglais).

La reconstruction de la trame T_n compressée, qui servira de trame de référence, débute par l'application d'une quantification inverse et d'une transformée inverse sur chaque bloc *X*. Ces opérations permettent de reconstruire les données résiduelles D'_n. L'encodeur additionne, par la suite, ces données résiduelles aux données prédites *P* pour former la trame reconstruire

uT'_n. Finalement, un filtre est appliqué sur cette dernière pour réduire les effets de blocs et produire la trame T'_n à la sortie.

2.2.1 Estimation et compensation de mouvement par bloc

L'estimation et la compensation de mouvement par bloc est une technique puissante utilisée pour réduire les redondances temporelles qui existent entre des images voisines temporellement en estimant les mouvements qui ont lieu entre ces images. Du côté de l'encodeur, cette technique divise l'image courante en régions de taille fixe, par exemple, en régions 16 × 16, appelées macroblocs. Puis, pour chacun de ces macroblocs, elle recherche et sélectionne dans une image de référence (généralement, l'image précédente) le macrobloc qui lui est le plus similaire (voir figure 2.2). Ensuite, l'encodeur conserve en mémoire la position de ce macrobloc, en calculant son vecteur de mouvement (différence entre sa position et celle de la région courante) et ses données résiduelles (différence entre la valeur de ses pixels et celle du macrobloc courant).

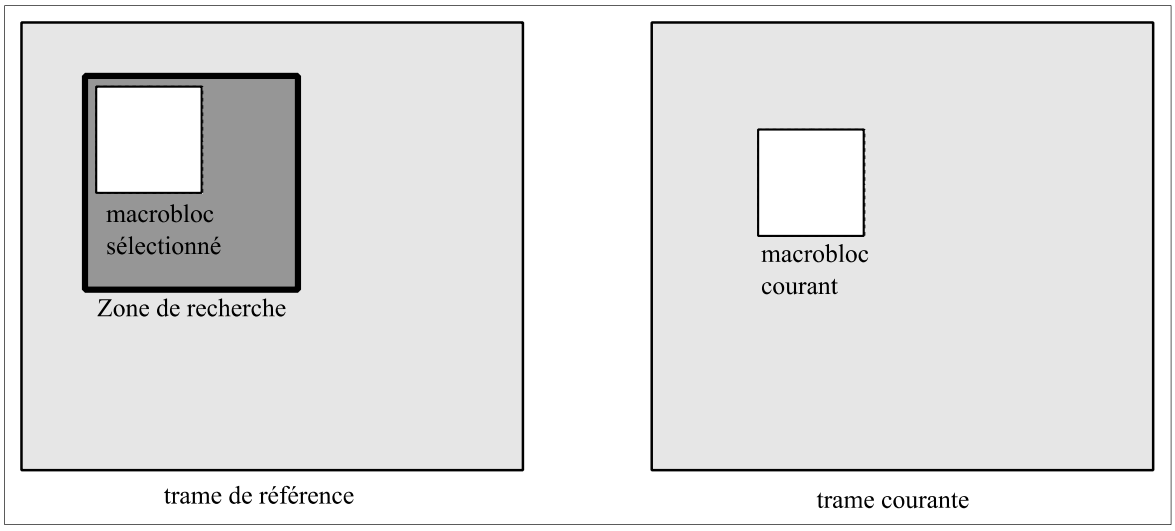


Figure 2.2 Zone de recherche utilisée, lors de l'estimation de mouvement, par le macrobloc courant.

La syntaxe de H.264 permet de raffiner l'estimation de mouvement sur des régions de plus petite taille, appelées blocs, ce qui a pour avantages de suivre plus efficacement le

mouvement d'objets plus petits ainsi que de modéliser plus finement les frontières entre les objets et le décor (ou d'autres objets). De plus, la syntaxe de H.264 offre aussi la possibilité d'effectuer des recherches précises au demi et au quart de pixel, grâce à une interpolation de l'image. Notons qu'une séquence encodée par un encodeur H.264 est composée obligatoirement d'au moins une trame de type intra et généralement de plusieurs trames de type Inter. Seules les trames inter utilisent l'estimation de mouvement. Les trames intra utilisent plutôt la prédiction intra (voir section 2.4.1) qui profitent de la corrélation qui existe entre des blocs voisins pour prédire les valeurs du bloc courant.

2.2.2 Transformées et quantification

Dans le modèle spatial, une texture représente ou bien les pixels d'un bloc non prédit, ou bien les données résiduelles d'un bloc prédit. Pour chaque bloc, ce modèle applique, dans l'ordre, une transformée pour convertir les valeurs de la texture en valeurs fréquentielles, une quantification qui réduit la précision des coefficients (surtout ceux des hautes fréquences) produits par la transformée et, finalement, un ré-ordonnancement en zigzag des coefficients quantifiés.

2.2.3 Codage entropique

H.264 admet l'emploi de deux modes d'encodage entropique alternatifs : un codage de type Huffman (Huffman, 1952), peu complexe, appelé *Context-Adaptive Variable-Length Codes* (CAVLC) et un codage de type arithmétique (Witten, Neal et Cleary, 1987), plus complexe, mais aussi plus efficace, appelé *Context-Adaptive Arihmetic Coding* (CABAC). Contrairement aux standards précédents, qui utilisent des statistiques stationnaires pour produire les codes, ces deux modes de codage utilisent des statistiques adaptées au contexte.

2.3 Décomposition hiérarchique d'une séquence vidéo

H.264 décompose une séquence vidéo en une structure hiérarchique à six niveaux (figure 2.3). Au niveau supérieur de cette structure, se trouve la séquence qui contient un ou plusieurs groupes d'images. Chaque groupe d'images est composé d'une ou de plusieurs trames (ou champs pour un vidéo entrelacé) et commence toujours par une trame de type *Instantaneous Decoder Refresh* (IDR). Enfin, les trames se divisent en une ou plusieurs tranches qui se subdivisent elles-mêmes en macroblocs et en blocs.

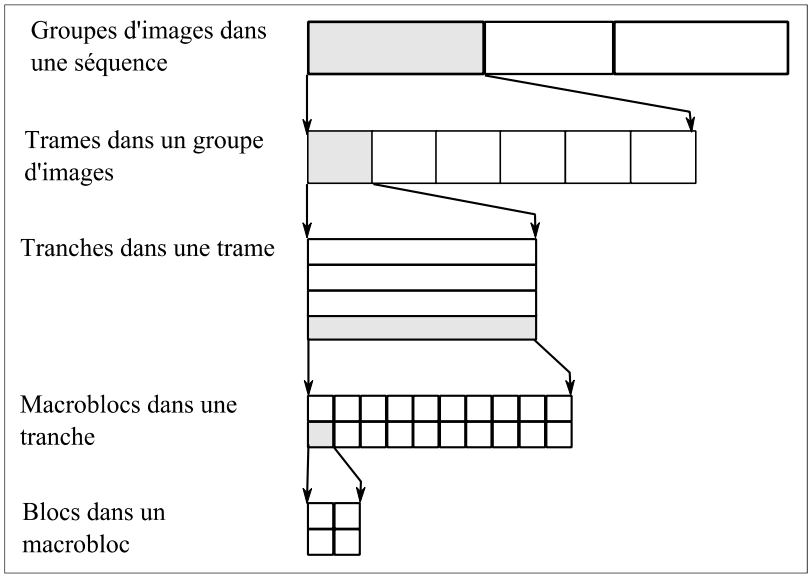


Figure 2.3 Décomposition hiérarchique d'une séquence vidéo H.264.

2.3.1 Groupes d'images

Les groupes d'images sont des séquences d'images qui sont syntaxiquement indépendantes les unes des autres. Chaque groupe d'images débute par une image de type IDR. Ce type d'image est toujours composé exclusivement de tranches I ou de tranches SI et empêche l'encodeur (et le décodeur) d'utiliser les images précédentes à titre de références. Les groupes d'images sont utilisés pour effectuer des lectures aléatoires et pour limiter la propagation d'erreurs dans le temps.

2.3.2 Trames

Une séquence vidéo numérique est composée d'une série d'images, appelées trames, échantillonnées temporellement selon une fréquence équivalente au nombre d'images par seconde. H.264 supporte cinq types de trame : les trames I, P, B, SI et SP. Une trame I est encodée sans aucune référence à d'autres images, ce type de trame est dit intra. Une trame P est une trame encodée où les macroblocs peuvent faire référence à une image passée. Une trame B est une trame encodée où les macroblocs peuvent faire référence à l'une des combinaisons d'images suivantes : une image passée et une image future, deux images passées, deux images futures. Notons que, contrairement aux standards précédents, H.264 permet, mais n'oblige pas, d'utiliser une trame B à titre de référence. Ces deux derniers types de trame sont dits Inter, car ils exploitent l'estimation de mouvement. Enfin, les trames SI et SP sont utilisées, notamment, pour effectuer de la commutation de flux (*switching streams* en anglais) (Karczewicz et Kurceren, 2001).

2.3.3 Tranches

Chaque image d'une séquence est divisée en un nombre de tranches, *slices* en anglais, compris entre un et le nombre de macroblocs présents dans l'image (figure 2.4). Chaque tranche représente physiquement une région donnée de l'image et contient les données encodées des macroblocs correspondants à cette région. Les tranches d'une même trame sont syntaxiquement indépendantes les unes des autres. Cela permet notamment de traiter les tranches en parallèle et d'offrir une meilleure résistance aux erreurs de transport. Tout comme les trames, les tranches peuvent être de types I, P, B, SI et SP.

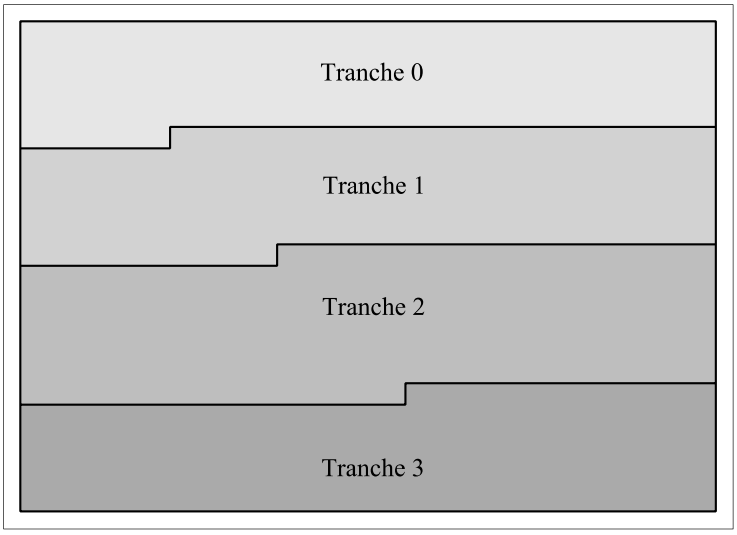


Figure 2.4 Partitionnement d'une image en plusieurs tranches.

Un encodeur H.264 est en mesure d'encoder (et un décodeur de décoder) les tranches d'une trame dans un ordre quelconque grâce au concept d'Arbitraty Slice Order (ASO). De plus, le Flexible Macroblock Ordering (FMO) de H.264 permet de créer des tranches à géométrie irrégulière. À cette fin, le FMO contient une syntaxe permettant d'associer des macroblocs à un groupe tranches. Par exemple, nous pouvons créer, comme à la figure 2.5, le groupe tranche 0 qui servira d'arrière-plan à l'image courante et les groupes tranche 1 et 2 qui serviront d'avant-plan.

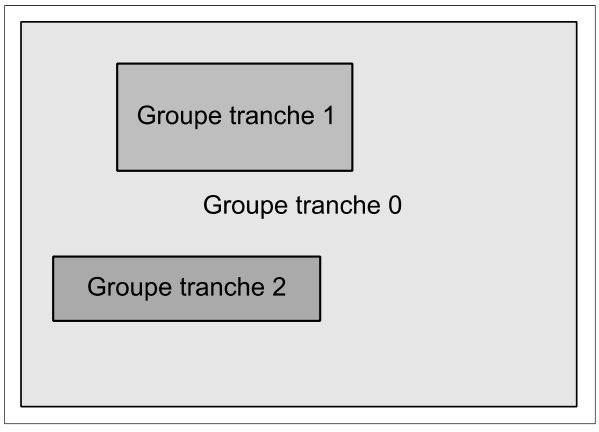


Figure 2.5 Exemple de groupes tranche.
Adaptée de (Richardson, 2003)

2.3.4 Macroblocs

Les macroblocs représentent des régions de 16 × 16 pixels d'une image dans un espace de couleurs YCbCr[3] et suivent généralement un format d'échantillonnage 4:2:0[4] (Li et Drew, 2004). Un macrobloc est composé de 16 × 16 échantillons Y, de 8 × 8 échantillons Cb et de 8 × 8 échantillons Cr. De plus, H.264 permet de partitionner un macrobloc en blocs de taille variable (16 × 16, 16 × 8, 8 × 16, 8 × 8) et de sous-partitionner les blocs 8 × 8 en blocs 8 × 4, 4 × 8 ou 4 × 4 (figure 2.6).

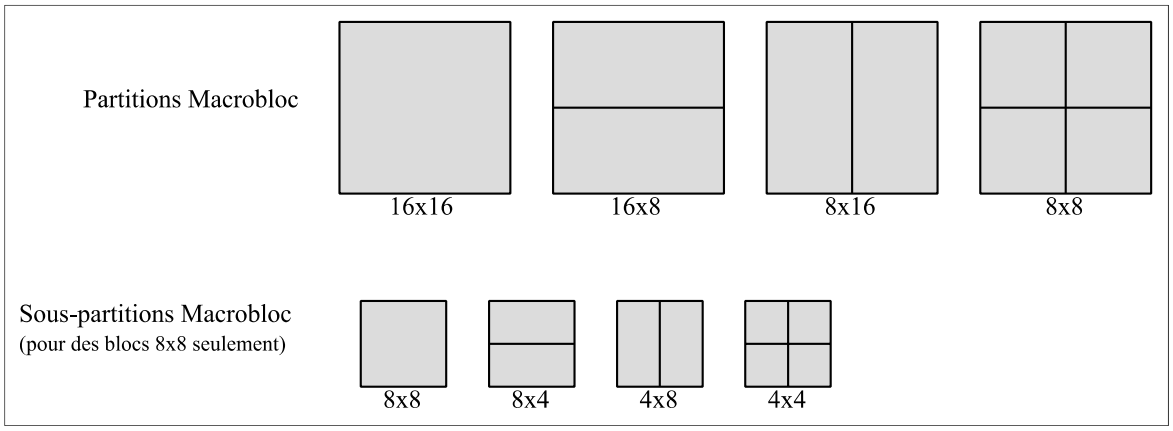


Figure 2.6 Partitions et sous-partitions des macroblocs pour l'estimation et la compensation de mouvement.

Le type d'un macrobloc varie en fonction du type de la tranche auquel il est associé. Une tranche de type I contient seulement des macroblocs de type I. Quant aux trames de type P et B, elles contiennent respectivement des macroblocs P et B. De plus, elles peuvent contenir, toutes les deux, des macroblocs I et sautés (*skipped* en anglais).

---

[3] Dans un espace de couleur YCbCr, Y est la composante qui représente le signal luma (ou luminance) tandis que Cb et Cr sont les composantes qui représentent les chromas rouge et bleu.

[4] Un format d'échantillonnage 4:2:0 conserve l'ensemble des composantes Y et sous-échantillonne, horizontalement et verticalement, les composantes Cb et Cr de manière à produire et à conserver une valeur Cb et une valeur Cr pour quatre valeurs Y.

2.4 Mécanismes de prédiction

Le codage prédictif est « une technique qui prévoit la valeur probable d'un élément […] en se basant sur les valeurs antérieures et qui code la différence entre les deux valeurs, passées et présentes pour permettre la compression effective » (Office de la langue française, 1999). Cette technique permet donc de réduire la redondance d'information en encodant seulement la différence entre deux valeurs. On parle de redondance temporelle quand il s'agit d'information qui se répète d'une trame à l'autre et de redondance spatiale pour de l'information liée à l'intérieur d'une même trame. H.264 réduit ces deux types de redondance par l'emploi de différentes méthodes de prédiction intra et Inter.

2.4.1 Prédiction intra

La prédiction intra de H.264 se sert de la corrélation qui existe entre des blocs voisins d'une même trame pour prédire les valeurs des pixels du bloc courant. Essentiellement, cette technique utilise les voisins supérieurs et le voisin de gauche, qui doivent avoir été encodés et reconstruits, du bloc courant pour construire différents blocs prédits. Ensuite, l'encodeur conserve, pour la suite de l'encodage, le bloc prédit qui minimise l'erreur (les données résiduelles) entre lui et le bloc courant.

H.264 définit neuf modes de prédiction pour les blocs Y 4 × 4 (figure 2.7), quatre modes de prédiction pour les blocs Y 16 × 16 et aussi quatre modes de prédiction pour les blocs Cb et Cr. Chaque mode correspond à la construction d'un bloc prédit spécifique. Par exemple, le mode 0 (vertical) de la prédiction intra d'un bloc Y 4 × 4 copie verticalement la valeur des composantes Y situées immédiatement au dessus de bloc courant.

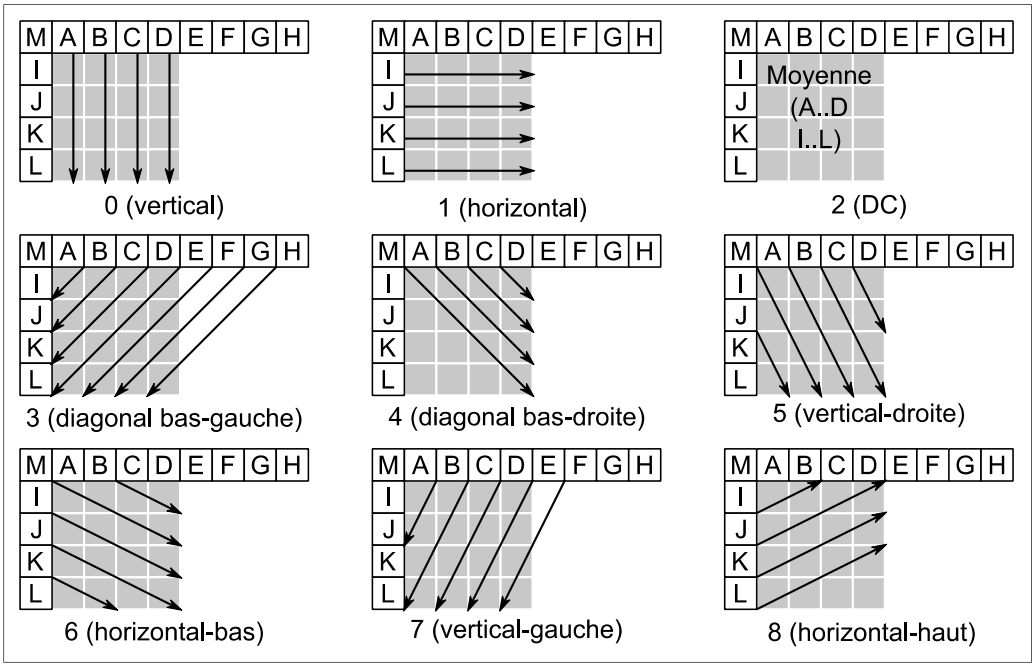


Figure 2.7 Modes de prédiction intra pour des blocs Y 4 × 4.
Adaptée de (Richardson, 2003)

La syntaxe de H.264 contient un drapeau I_PCM qui, une fois activé, force l'encodeur à encoder directement la valeur d'un bloc, plutôt qu'une valeur transformée et prédite. L'encodage direct de cette valeur est parfois nécessaire quand le bloc courant n'est pas fortement corrélé avec ses blocs voisins ou encore quand l'application de la transformée réduit l'efficacité de la compression. De plus, le drapeau I_PCM peut être utilisé, au prix d'un encodage peu efficace, pour éliminer les dépendances entre des blocs voisins et, ainsi, permettre à l'encodeur d'encoder ces blocs en parallèle.

2.4.2 Prédiction Inter

La prédiction inter profite de la similitude entre des trames voisines pour prédire la valeur des pixels d'un bloc en utilisation l'estimation et la compensation de mouvement par bloc (section 2.2.1). H.264 contient plusieurs outils permettant d'effectuer des prédictions de blocs plus ou moins précis. D'abord, H.264 permet de subdiviser un macrobloc en plusieurs blocs de taille variable (section 2.3.4) et d'allouer un vecteur de mouvement à chacun de ces blocs,

pour mieux suivre les mouvements d'objets plus petits dans la scène et modéliser les frontières. Ensuite, la syntaxe d'H.264 rend possible l'utilisation d'algorithmes d'estimation de mouvement non-exhaustifs - et par conséquent sous-optimaux - ainsi que l'utilisation d'une zone de recherche restreinte pour réduire la complexité de la recherche. De plus, H.264 offre l'opportunité d'effectuer une recherche sur plusieurs trames de référence. Ce type de recherche peut être bénéfique, par exemple, dans des scènes qui contiennent des flashs.

2.4.2.1 Prédiction de vecteurs de mouvement

L'encodage complet d'un vecteur de mouvement peut coûter cher en bits. Pour réduire le coût de l'encodage d'un vecteur de mouvement, H.264 définit une méthode de prédiction de vecteurs de mouvement basée sur la corrélation spatiale du mouvement. Cette méthode permet de réduire le coût de l'encodage d'un vecteur lorsqu'il faut moins de bits pour encoder la différence entre le vecteur prédit et le vecteur réel que pour encoder le vecteur réel. La figure 2.8 montre les partitions qui sont utilisées pour prédire le vecteur de mouvement de la partition courante lorsque les partitions voisines sont : a) de même taille et b) de taille différente.

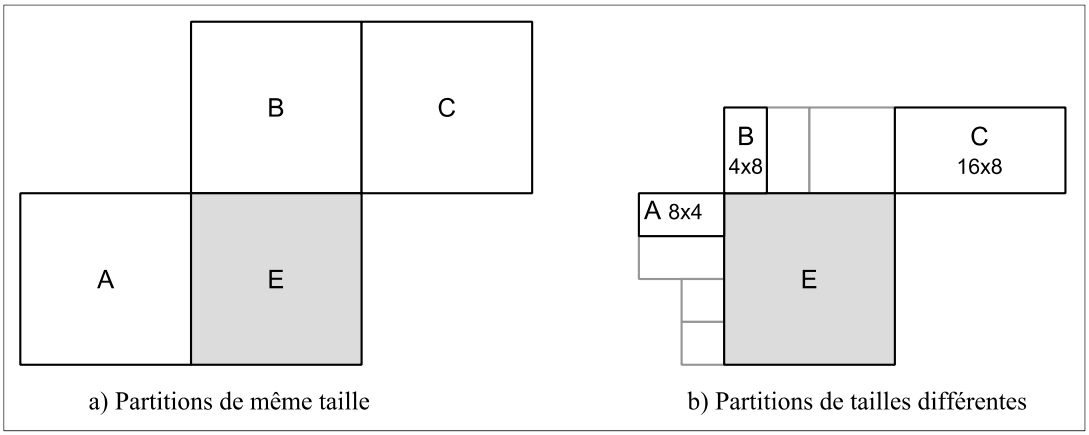


Figure 2.8 Partitions et voisins utilisés lors de la prédiction du vecteur de mouvement courant.
Adaptée de (Richardson, 2003)

La prédiction s'effectue de la manière suivante. Soit E le bloc actuel, B le bloc immédiatement au dessus de E, C le bloc immédiatement à droite de B et A le bloc immédiatement à gauche de E. S'il y a plus d'une partition à gauche de E, alors la partition la plus haute est choisie. S'il y a plus d'une partition au dessus de E, alors la partition le plus à gauche est choisie, et même chose pour la ou les partitions de C. Ensuite, H.264 calcule la valeur de la prédiction selon ces règles :

- La médiane des vecteurs de mouvement des partitions A, B et C est utilisée pour les partitions qui ne sont pas de format 16 × 8 et 8 × 16.
- Le vecteur de mouvement B est utilisé pour la prédiction d'une partition 16 × 8 située au niveau supérieur d'un macrobloc et le vecteur de mouvement A est utilisé pour la prédiction de la partition 16 × 8 inférieure.
- Le vecteur de mouvement A est utilisé pour la prédiction d'une partition 8 × 16 située à gauche d'un macrobloc et le vecteur de mouvement C pour la prédiction d'une partition 8 × 16 située à droite.

2.5 Résumé sur les dépendances

Le réseau de dépendances entre les données compressées d'H.264 est complexe. D'abord, nous notons que les groupes d'images, et les tranches appartenant à une même trame, sont indépendants les uns des autres aussi. Ensuite, nous remarquons que :

- Les macroblocs intra d'une même trame dépendent de leurs voisins supérieurs et de leur voisin de gauche, si de tels voisins existent à l'intérieur de la tranche.
- Les macroblocs inter dépendent de leurs voisins : de gauche immédiat, supérieur immédiat et supérieur droit immédiat. Ils dépendent de plus des macroblocs situés dans la zone de recherche et d'interpolation des trames de références.

2.6 Mesures de qualité vidéo

Pour évaluer la qualité d'une séquence vidéo compressée, nous avons besoin d'une mesure de qualité objective qui calcule la distorsion qui existe entre une séquence vidéo originale et

une séquence vidéo modifiée. Idéalement, la distorsion calculée par cette mesure doit être corrélée avec la distorsion perçue par le système visuel humain (SVH). De plus, cette mesure doit préférablement avoir une faible complexité en calculs. Lors de l'estimation de mouvement, les encodeurs vidéos utilisent souvent la mesure de l'erreur quadratique moyenne (EQM), ou *Mean Squared Error* (MSE) en anglais, qui se définit ainsi :

$$EQM = \frac{1}{NM} \sum_{i=0}^{n} \sum_{j=0}^{m} (C_{ij} - R_{ij})^2 \tag{2.1}$$

où $N \times M$ représente les dimensions des blocs à comparer, C et R représentent respectivement les valeurs des échantillons de l'image (bloc) courante et de l'image (bloc) de référence.

Le rapport signal à bruit, que nous appellerons *Peak Signal-to-Noise-Ratio* (PSNR) dans le reste de ce document, est la mesure de qualité la plus couramment utilisée dans la littérature portant sur le traitement de séquences vidéo. Par conséquent, nous utiliserons cette mesure lors de la présentation de nos résultats. Le PSNR, qui retourne une valeur en décibel (dB) se définit telle que :

$$PSNR = 10 \cdot \log_{10} \left(\frac{d^2}{EQM} \right) \tag{2.2}$$

où *d* représente la valeur maximale que peut prendre un échantillon. Cette valeur est égale à 255 pour de la compression vidéo dont les échantillons sont représentés avec 8 bits de précision. Selon (Wang, Ostermann et Zhang, 2002), un PSNR supérieur à 40 dB indique typiquement une image d'excellente qualité, un PSNR entre 30 et 40 dB signifie généralement que l'image est de bonne qualité, un PSNR entre 20 et 30 dB, une image de mauvaise qualité et un PSNR en dessous de 20 dB, une image inacceptable.

Il existe plusieurs mesures de qualité vidéo plus récentes et mieux corrélées au SVH que le PSNR. Parmi ces mesures, nous comptons : SSIM, NQI, VQM (Xiao, 2000; Zhou et Bovik, 2002; Zhou *et al.*, 2004).

2.7 Conclusion

Dans ce chapitre, nous avons présenté le standard de codage vidéo H.264. Nous avons énuméré ses principales nouveautés qui lui permettent, notamment, d'obtenir un taux de compression environ deux fois supérieur à celui de ses prédécesseurs, pour une même qualité vidéo. Nous avons abordé la séquence d'encodage H.264 et discuté brièvement de ses principales étapes (méthode de prédiction, estimation et compensation de mouvement, transformés, quantification, codage entropique). Puis, nous avons décrit la structure hiérarchique d'une séquence vidéo H.264 (groupe d'images, trame, tranche, macrobloc, bloc) en portant une attention particulière sur les relations de dépendance entre chaque structure. Par la suite, nous avons décrit les principaux mécanismes de prédiction, intra et inter, définis par la syntaxe de H.264. Finalement, nous avons présenté deux mesures de qualité vidéo : l'EQM et le PSNR.

CHAPITRE 3

ÉTAT DE L'ART – PARALLÉLISATION D'UN ENCODEUR VIDÉO

Dans ce chapitre, nous présenterons l'état de l'art de l'encodage vidéo parallèle sur des systèmes à mémoire partagée. De plus, nous aborderons quelques solutions d'encodage parallèle employées sur des systèmes à mémoire partagée.

Nous débuterons par une introduction qui propose une vue d'ensemble sur la parallélisation d'encodeurs et de décodeurs vidéo. Puis, nous aborderons quelques exigences fonctionnelles et architecturales relatives à cette parallélisation. De ces exigences, nous passerons à la présentation des principaux types de décomposition utilisés par les encodeurs vidéo parallèles. Par la suite, nous décrirons les principales solutions proposées dans la littérature pour paralléliser les encodeurs vidéo.

3.1 Introduction

La parallélisation d'un encodeur vidéo dépend principalement des exigences fonctionnelles et architecturales du système. Cette parallélisation se matérialise sous la forme d'une solution adaptée à un contexte spécifique. En effet, comme nous le verrons plus loin, ces exigences correspondent à des contraintes qui limitent la décomposition du problème et, plus généralement, les choix de conception parallèle.

La littérature dénombre plusieurs approches pour paralléliser un encodeur ou un décodeur. Ces approches peuvent être regroupées en trois catégories : (1) les approches logicielles pour les systèmes à mémoire partagée; (2) les approches logicielles pour les systèmes à mémoire distribuée; (3) et les approches matérielles pour les processeurs parallèles spécialisés. La première catégorie est basée sur l'utilisation d'ordinateurs généraux et inclut des solutions logicielles proposées pour des multiprocesseurs symétriques, des processeurs multicœurs et des processeurs dotés de multifil simultané (HT) (Jung et Jeon, 2008; Steven, Xinmin et Yen-Kuang, 2003; Yen-Kuang *et al.*, 2004). Ces solutions ont souvent une mise à l'échelle limitée

et sont parfois basées sur une communication inter-fils intensive, ce qui est approprié sur ce type de système où les échanges de données s'effectuent directement au niveau de la mémoire. La seconde catégorie, elle aussi basée sur l'utilisation d'ordinateurs à usage générale, propose des solutions pour des grappes de serveurs (Luo, Sun et Tao, 2008; Rodriguez, Gonzalez et Malumbres, 2006). Ces solutions offrent souvent une meilleure mise à l'échelle que celles de la première catégorie, mais utilisent rarement une communication intensive à cause du délai occasionné par les échanges de données inter-ordinateur qui s'effectuent par l'entremise d'un réseau. La troisième catégorie est constituée de solutions matérielles (et logicielles) développées sur des processeurs spécialisés. Les solutions de cette dernière catégorie sont les plus variées puisque ce type de système implique moins de contraintes architecturales, donc plus de liberté pour la conception.

3.2 Exigences d'un encodeur vidéo

Les exigences d'un encodeur vidéo se divisent en deux catégories : les exigences fonctionnelles et les exigences architecturales. La première catégorie spécifie ce que l'application doit faire et la seconde spécifie l'architecture sur laquelle elle sera exécutée. Cette architecture, dans son sens le plus large, correspond à une famille d'ordinateurs parallèles, par exemple, aux processeurs multicœurs symétriques et, dans sens le plus étroit, à un ordinateur parallèle spécifique, par exemple, à un quad-cœur d'Intel.

Parmi les exigences fonctionnelles qui influencent la conception d'un encodeur vidéo parallèle, on compte : le ou les profiles (les outils de compression) du standard H.264; le temps d'encodage moyen et maximal d'une trame; le délai moyen et maximal entre la réception d'une trame et son expédition après traitement; la variation des délais de transmission, appelées la gigue (le *jitter*); les tailles maximales du tampon d'entrée et du tampon de sortie de l'encodeur; les outils de compression utilisés et leur configuration, par exemple, l'encodeur pourrait utiliser un nombre fixe de tranches et plusieurs trames de références; la qualité du vidéo produit. À cette liste, on ajoute les exigences fonctionnelles spécifiques au traitement parallèle, telles que l'accélération, la mise à l'échelle et la transparence de l'application.

Les spécifications exactes de chacune de ces exigences dépendent du contexte dans lequel l'encodeur sera déployé. Par exemple, un encodeur H.264 temps réel spécialisé dans le traitement simultané de plusieurs séquences CIF n'aura pas les mêmes spécifications qu'un encodeur H.264 spécialisé dans l'encodage de disques Blu-ray. Dans le premier cas, la parallélisation de l'encodeur aura intérêt à offrir un faible délai et une gigue réduite, au détriment de la qualité vidéo et de l'accélération produite par le parallélisme, pour satisfaire la contrainte temps réel. Dans le second cas, le délai et la gigue ne seront pas considérés et l'accélération devra être maximisée, sans causer une perte de qualité notable (par exemple obtenir une perte de qualité supplémentaire, par rapport à un encodage séquentiel et mesuré par le PSNR, en dessous de 0.1 dB), pour terminer l'encodage le plus tôt possible.

Lors de la conception, on doit aussi tenir compte des caractéristiques de l'architecture cible. Chaque type d'architecture parallèle a ses avantages et ses inconvénients. Par exemple, les architectures basées sur un modèle mémoire partagée sont très bien adaptées à la production d'algorithmes parallèles qui nécessitent beaucoup de communication entre les unités d'exécution. Cependant, ces systèmes ont généralement des capacités de calcul globales limitées par rapport aux systèmes distribuées. Pour les systèmes basés sur un modèle mémoire distribuée, c'est généralement l'inverse. Ces systèmes sont souvent développés pour obtenir des capacités calculatoires plus grandes et, dans certains cas, pour être étendus ultérieurement par l'ajout de nouveaux processeurs. Par contre, le coût de communication de ces système est généralement élevé et parfois non-uniforme, ce qui les rend peu adaptés à l'exécution d'algorithmes effectuant beaucoup de transferts de données entre les unités d'exécution.

3.3 Décomposition d'un encodeur vidéo H.264

La syntaxe et la structure des encodeurs vidéo supportent les deux types de décomposition de base, soient la décomposition fonctionnelle et la décomposition du domaine : la première, parce que les encodeurs vidéo sont naturellement divisés en étapes successives, un peu comme une chaine de montage; la seconde, parce que les séquences vidéo contiennent

suffisamment de données pour être divisés en blocs de données à traiter simultanément. De plus, ces deux types de décomposition peuvent être combinés pour former une décomposition hybride.

3.3.1 Décomposition fonctionnelle

Un encodeur vidéo se divise en plusieurs étapes distinctes et traitées successivement (voir figure 2.1) : estimation de mouvement, prédiction par compensation de mouvement, application d'une transformée, quantification, codage entropique, etc. La décomposition fonctionnelle permet d'identifier ces étapes, de les regrouper dans un nombre de groupes de tâches équivalant au nombre d'unités d'exécution disponibles sur le système et d'exécuter ces groupes de tâches simultanément sous la forme d'un pipeline. Par exemple, un pipeline développé pour un processeur doté de deux unités d'exécution, pourrait être composé de deux niveaux : l'un qui traite les étapes relatives à l'estimation de mouvement et aux techniques de prédiction, l'autre qui traite les étapes restantes, telles que l'application de la transformée, la quantification et le codage entropique.

Pour être efficace, le pipeline conçu doit avoir une distribution de charge équilibrée entre ses unités d'exécution. Dans le cas contraire, les unités qui ont des tâches plus longues à exécuter entraineront des délais pour les unités suivantes. Dans les faits, ce besoin d'équilibrer la charge entre les différentes unités d'exécution demande des efforts de conception importants, et rend les systèmes de ce type peu flexibles aux changements majeurs, puisque de tels changements peuvent déséquilibrer la distribution de la charge de travail entre les unités d'exécution et, par conséquent, impliquer une nouvelle conception. Par exemple, l'implémentation d'un algorithme d'estimation de mouvement deux fois plus rapide que l'algorithme précédemment implémenté rendra, dans notre exemple de pipeline, le premier niveau trop rapide par rapport au deuxième niveau, si la conception a initialement permis d'obtenir des niveaux biens équilibrés. Malgré ces faiblesses, ce type de décomposition a l'avantage de permettre un développement modulaire qui affecte peu les structures de données du programme et qui permet d'utiliser, dans certains cas, des puces spécialisées, par

exemple, en estimation de mouvement, pour accélérer le traitement d'une étape gourmande en calculs.

L'ensemble de ces caractéristiques font de la décomposition fonctionnelle une décomposition appropriée pour les processeurs spécialisés. Ce type de processeur est généralement conçu pour exécuter des applications précises qui nécessitent peu de changements majeurs sur le plan des exigences logicielles, ce qui permet de développer des algorithmes rigides qui n'auront pas à évoluer sur d'autres architectures ni à supporter de nouvelles fonctionnalités qui affecteront la distribution de charge. En contrepartie, ce type d'architecture est très peu approprié pour les processeurs à usage général. Ce type de processeur évolue rapidement, ce qui rend difficile la maintenance et la mise à l'échelle de l'application. De plus, les processeurs à usage général sont utilisés dans trop de contextes différents pour qu'un encodeur basé sur une décomposition fonctionnelle puisse bien s'y intégrer.

3.3.2 Décomposition du domaine

Comme nous l'avons vu à la section 2.3, un encodeur vidéo représente une séquence vidéo sous la forme d'une structure hiérarchique à six niveaux : la séquence, les groupes d'images, les trames, les tranches, les macroblocs et les blocs. La décomposition du domaine se réalise à l'aide de cette structure hiérarchique. Typiquement, le concepteur doit choisir entre les quatre niveaux du milieu. En effet, il n'existe qu'une seule séquence par encodage et les blocs impliquent une synchronisation trop fréquente en plus de rendre une distribution de charge difficile, notamment, parce que les blocs ne sont pas toujours de la même taille. Le ou les niveaux à sélectionner dépendent de plusieurs facteurs, dont l'architecture et le type d'application ciblé. Les prochaines lignes résument les avantages et les inconvénients de chacun de ces niveaux

3.3.2.1 Décomposition au niveau des groupes d'images

La décomposition au niveau des groupes d'images est une approche populaire pour paralléliser l'encodage d'une séquence vidéo sur des systèmes à mémoire distribuée (Luo, Sun et Tao, 2008; Rodriguez, Gonzalez et Malumbres, 2006). Cette approche a pour avantage de nécessiter peu de communication et de synchronisation, puisque les groupes d'images sont des unités d'encodage indépendantes. Cette indépendance permet aussi de paralléliser assez rapidement un encodeur vidéo, car elle affecte peu les structures de données.

Cependant, la grosseur de cette unité d'encodage occasionne trois inconvénients majeurs. D'abord, elle ne permet pas de réduire la latence de l'encodage, car l'encodage du premier groupe d'images n'est pas accéléré. Par conséquent, cette approche ne permet pas de transformer une application séquentielle qui n'est pas temps réel en une application parallèle temps réel. Ensuite, elle rend difficile le balancement de charge, puisque le temps nécessaire pour encoder un groupe d'images est inégal. Finalement, le traitement parallèle de plusieurs groupes d'images occasionne, sur une grappe d'ordinateurs, un trafic élevé et réduit les performances de la cache principale pour les architectures à mémoire partagée, puisque chacun des fils accède alors à des données (images) différentes et, par conséquent, n'est pas en mesure d'utiliser des données qui auraient été chargées dans la cache par un autre fil.

Ces caractéristiques rendent la décomposition au niveau des groupes d'images appropriée pour traiter plus rapidement l'encodage d'une ou de séquences vidéo au prix d'une latence élevée. Cette approche est donc appropriée pour traiter des fichiers, mais inappropriée pour des communications en temps réel.

3.3.2.2 Décomposition au niveau des trames

La décomposition au niveau des trames est une décomposition habituellement utilisée en combinaison avec un autre type de décomposition (Qiang et Yulin, 2003; Steven, Xinmin et Yen-Kuang, 2003). Cette décomposition peut consister à exploiter les trames de référence

multiples qui peuvent être utilisées pour encoder une trame dans H.264. Cette décomposition peut aussi consister à exploiter les relations de dépendance qui existent entre les trames de type I, P et B. (Rappelons qu'une trame I ne dépend d'aucune autre trame, qu'une trame P dépend de la trame I ou P qui la précède et qu'une trame B dépend de la trame I ou P qui la précède et de la trame I ou P qui la suit. Rappelons aussi que les trames I et P servent de références pour d'autres trames et qu'aucune trame ne dépend d'une trame B, sauf dans le standard H.264, qui lui permet d'utiliser une trame B comme trame de référence.) Ces relations de dépendances permettent, par exemple, de paralléliser une série de trames IBBPBBPBBPBBP en suivant le schéma de dépendance de la figure 3.1 lorsque nous n'utilisons pas les trames B à titre de référence. Sur cette figure, les numéros d'index représentent les numéros temporels (l'ordre d'affichage) des trames. Dans ce cas-ci, ces numéros ne correspondent pas à l'ordre d'encodage. Ainsi, l'encodeur traite d'abord la première trame I (0) de la séquence. Puis, il encode la première trame P (3) avant de débuter le traitement parallèle de la seconde trame P (6) et des deux premières trames B (1 et 2) de la séquence.

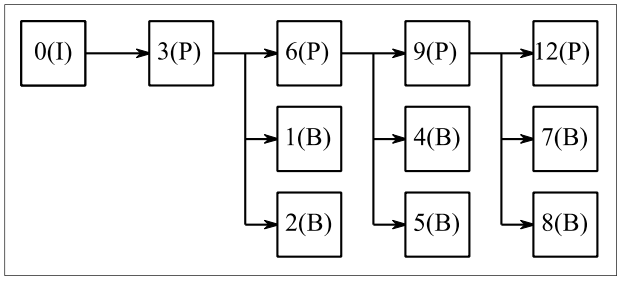


Figure 3.1 Relations de dépendance entre les différents types de trame.
Tirée de (Qiang et Yulin, 2003)

Dans ce contexte, la complexité d'encodage des trames P, par rapport à la complexité d'encodage des trames B, est le point critique du parallélisme de l'application. Si l'encodage d'une trame P est plus lent que l'encodage d'une trame B, alors l'encodeur sera en mesure d'encoder seulement 3 trames en parallèle (une trame P et deux trames B). Au contraire, si l'encodage d'une trame P est plus rapide que l'encodage d'une trame B, alors l'encodeur sera en mesure d'encoder, parfois, plus de trois trames en parallèle. Par exemple, si l'encodeur encode en moyenne une trame P deux fois plus rapidement qu'une trame B, alors l'encodeur devra être en mesure d'encoder cinq trames en parallèles (une trame P et quatre trames B).

Cette approche a pour principaux inconvénients d'avoir une accélération maximale limitée par le temps nécessaire pour traiter les trames de type I et P séquentiellement et une mise à l'échelle limitée par le nombre de trames B placées entre les autres types de trames et le rapport entre la complexité des trames P et des trames B. C'est la raison pour laquelle ce type de décomposition est généralement utilisé en combinaison avec un autre type de décomposition. De plus, cette approche nécessite une utilisation plus fréquente des mécanismes de synchronisation et de communication qu'une décomposition par groupe d'images.

3.3.2.3 Décomposition au niveau des tranches

La décomposition au niveau des tranches consiste à découper une trame en un nombre de tranches habituellement égal ou supérieur au nombre d'unités de traitement disponible et à profiter de l'indépendance qui existe entre ces tranches pour les traiter en parallèle (Jung et Jeon, 2008; Steven, Xinmin et Yen-Kuang, 2003; Yen-Kuang *et al.*, 2004). Cette approche a le double avantage de réduire à la fois le temps total d'encodage et de diminuer la latence, ce qui est parfait pour traiter des séquences vidéo en temps réel.

Cependant, l'utilisation de cette technique affecte le rapport qualité/débit. Car on réduit l'exploitation des corrélations qui existent aux frontières de deux tranches voisines en divisant une trame en plusieurs tranches indépendantes. Et en plus, on augmente la taille des en-têtes. Par conséquent, il existe une relation entre le nombre de tranches par trame et l'augmentation du débit pour une même qualité. Par exemple, (Steven, Xinmin et Yen-Kuang, 2003) indiquent que le débit, pour une même qualité, est environ de 15 à 20 % plus élevé quand une trame CIF est divisée en neuf tranches. De plus, la mise à l'échelle de cette approche est limitée par le nombre maximal de tranches supporté par un encodeur donné.

3.3.2.4 Décomposition au niveau des macroblocs

La décomposition au niveau des macroblocs est, sans contredit, la décomposition la plus difficile à exploiter. Elle demande aux concepteurs des efforts pratiques (programmation) plus importants que les autres types de décomposition, essentiellement pour trois raisons :

- Sa granularité est fine : c'est-à-dire que les tableaux de données distribués aux différentes unités d'exécution sont petits et nécessitent donc une implémentation plus pointue des mécanismes de synchronisation et une étude plus approfondie des structures de données du code.
- Ses dépendances sont nombreuses : par exemple, un macrobloc inter est généralement dépendant de la trame précédente, de ses voisins supérieur et supérieur-droit ainsi que de son voisin de gauche (voir figure 2.8).
- Son balancement de charge est difficile : Il est difficile de bien distribuer le travail, car la durée d'encodage d'un macrobloc est variable.

Malgré ces points négatifs, ce type de décomposition est souvent utilisé, car il offre une excellente mise à l'échelle, une bonne accélération et une faible latence. Ce type de décomposition est souvent utilisé conjointement avec l'exploitation d'un parallélisme inter-trame, c'est à dire un parallélisme qui vise à traiter le début d'une nouvelle trame avant la fin du traitement la trame courante.

3.4 Approches d'encodage parallèle sur systèmes à mémoire partagée

Les approches développées sur des systèmes à mémoire partagée offrent une bonne mise à l'échelle, une bonne accélération et une latence faible. La majorité de ces approches sont fondées, notamment, sur un parallélisme effectué au niveau des tranches (Jung et Jeon, 2008; Steven, Xinmin et Yen-Kuang, 2003; Yen-Kuang *et al.*, 2004). Cependant, la littérature compte aussi des approches d'une granularité plus fine, telle que l'approche de l' « onde de choc » présentée dans (Chen *et al.*, 2006).

3.4.1 Approche parallèle au niveau des trames et des tranches

(Steven, Xinmin et Yen-Kuang, 2003) proposent une méthode de parallélisation basée sur une décomposition effectuée au niveau des trames et des tranches. Dans ce contexte, ces deux types de parallélisme sont complémentaires. Le premier type, basé sur un schéma d'encodage IBBPBB, encode les trames I et P l'une à la suite de l'autre, en simultané, avec des trames B qui ont accès à leurs trames de références. Ce type de parallélisme n'affecte pas la qualité de la compression, mais limite cependant la mise à l'échelle (voir section 3.3.2.2). Le second type améliore la mise à l'échelle en permettant l'utilisation d'un plus grand nombre d'unités d'exécution, mais au prix d'une dégradation de la qualité visuelle de la séquence vidéo. Cette dégradation est, d'une part, occasionnée par les macroblocs situés dans la frontière supérieure (ou inférieure) d'une tranche. Ces macroblocs peuvent seulement utiliser un nombre restreint de modes de prédiction, puisqu'ils n'ont pas accès aux macroblocs de la tranche supérieure (ou inférieure). D'autre part, cette dégradation est causée par l'ajout d'un entête pour chaque tranche dans le train de bits (voir section 4.3.1 pour plus de détails) Réunis ensemble, les deux types de parallélisme ont l'avantage de garder les unités d'exécution presque toujours occupées, ce qui a pour effet de produire une excellente accélération. Cette méthode, illustrée à la figure 3.2, est divisée en deux sections exécutées simultanément.

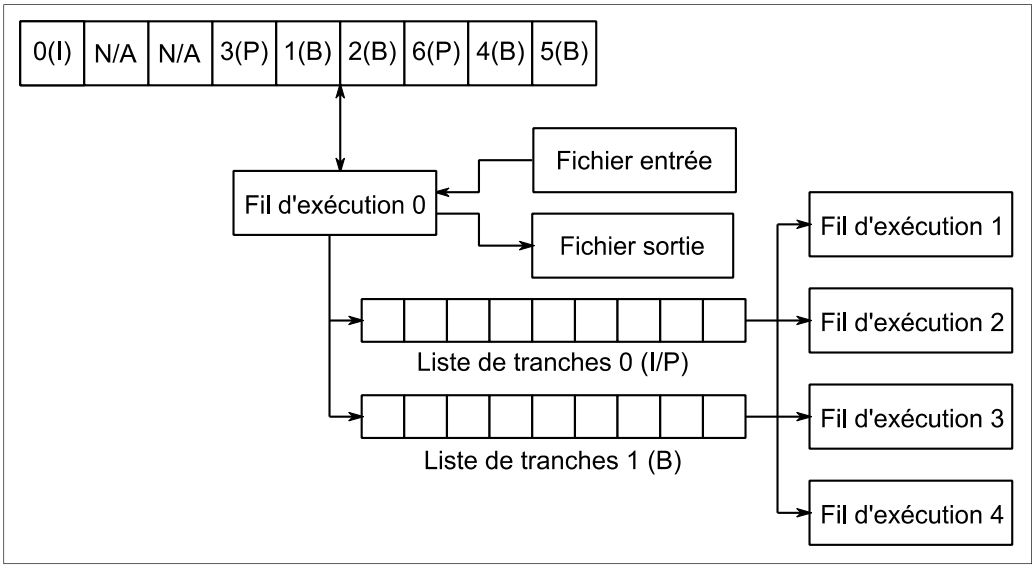


Figure 3.2 Approche parallèle de (Steven, Xinmin et Yen-Kuang, 2003).
Adaptée de (Steven, Xinmin et Yen-Kuang, 2003)

La première section, qui correspond au fil 0 sur la figure 3.2, a pour responsabilité de gérer la lecture des images entrantes et l'écriture des images sortantes. Dans un premier temps, cette section vérifie si la prochaine trame doit être encodée. Si oui, elle lit alors la trame depuis le fichier d'entrée. Puis, elle identifie le type de la trame en suivant un schéma d'encodage IBBPBB. Ainsi la trame 0 sera de type I, les trames 1 et 2 de type B, la trame 3 de type P, etc. Par la suite, le fil 0 place la trame dans un tampon d'images en fonction de son ordre d'encodage. Après, le fil 0 divise la trame en plusieurs tranches qu'il va placer dans la queue I/P ou dans la queue B, ce qui met fin au traitement relatif à la lecture d'une trame. Dans un second temps, la première section vérifie si le tampon d'images contient des trames encodées qui ne sont pas déjà écrites dans le fichier de sortie. Si oui, la section procède à l'écriture de ces données.

La deuxième section, qui correspond aux fils 1 à 4 sur la figure 3.2, a pour responsabilité d'encoder les tranches qui ont été placées, par la première section, dans l'une ou l'autre des listes de tranches. À cet effet, chaque fil exécute en boucle les trois opérations suivantes : (1) si une tranche est disponible dans la liste I/P, alors la récupérer et l'encoder; (2) sinon, si une tranche est disponible dans la liste B, alors la récupérer et l'encoder; (3) sinon, attendre l'arrivée de la prochaine tranche. Les tranches I et P sont traitées en premier, puisque des

trames ultérieures peuvent en être dépendantes, contrairement aux trames B. En d'autres mots, la complexité des trames I et P produit le chemin critique de l'application.

Les auteurs ont testé cette méthode sur un système multiprocesseur Xeon d'Intel composé de 4 processeurs et supportant la technologie HT. Ils annoncent une accélération maximale de 4,69 avec le mode HT activé, et de 3,95 sans ce mode. De plus, ils notent une dégradation graduelle de la qualité au fur et à mesure qu'on augmente le nombre de tranches utilisées. Dans le contexte de leurs expérimentations, les auteurs indiquent que le meilleur compromis accélération/perte de qualité se situe au alentour de 3 à 4 tranches par trame.

3.4.2 Approche parallèle au niveau des macroblocs et des trames

Dans (Chen *et al.*, 2006), les auteurs présentent une approche parallèle effectuée au niveau des macroblocs pour paralléliser un encodeur H.264. Cette approche a la particularité de préserver la qualité vidéo, d'obtenir de bonnes accélérations et de permettre une mise à l'échelle relativement large.

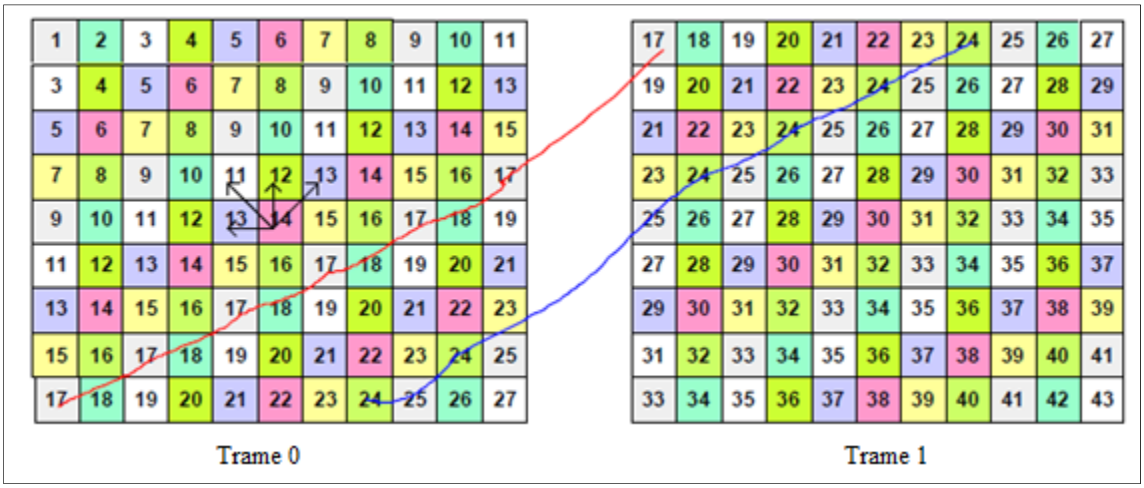


Figure 3.3 Approche parallèle de type onde de choc (Chen *et al.*, 2006).
Tirée de (Chen *et al.*, 2006)

La figure 3.3 montre que le parallélisme effectué au niveau des macroblocs est basé sur un parcours d'encodage suivant une structure que l'on pourrait comparer à une onde de choc. Ce parcours a été créé pour exploiter au mieux les dépendances qui existent entre les macroblocs

(voir section 2.4) et pour permettre un parallélisme effectué en diagonales de macroblocs. Plus spécifiquement, et toujours selon la figure 3.3, cet algorithme fonctionne de la manière suivante : le système encode d'abord les macroblocs numérotés 1 et 2 l'un à la suite de l'autre. Puis, il encode, successivement et en parallèle, les deux macroblocs numérotés 3, les deux macroblocs numérotés 4, les trois macroblocs numérotés 5, et ainsi de suite. Cette méthode utilise donc peu d'unités d'exécution au début de la trame, ce qui produit une faible accélération et atteint son maximum d'unités d'exécution potentiels aux diagonales 11, 13, 15 et 17. Ce maximum correspond au résultat de l'équation $\frac{1}{2}(w + 1)$, où w représente la longueur d'une trame en macroblocs. Cela signifie qu'on peut traiter jusqu'à 40 macroblocs en parallèle pour une séquence vidéo aillant une résolution de 1280×720. Par la suite, les pseudo-diagonales, et par extension l'accélération, s'amenuisent à nouveau. Afin de maintenir une accélération élevée et, contourner le problème des petites pseudos-diagonales, les auteurs combinent cette méthode avec un parallélisme inter-trame. Ce mécanisme supplémentaire permet de prolonger virtuellement les pseudo-diagonales (17 à 27) de la seconde moitié de la trame n, vers les pseudo-diagonales de la première moitié de la trame $n + 1$. Ainsi, l'encodeur a la possibilité d'encoder plus souvent en parallèle des diagonales de $\frac{1}{2}(w + 1)$, macroblocs.

Les auteurs ont testé cette approche sur un système multiprocesseur Xeon d'Intel composé de 4 processeurs et supportant la technologie HT. Ils annoncent des accélérations moyennes de 4,6 et de 3,8 pour des tests effectués avec la technologie HT activée et désactivée, respectivement. Par rapport à l'approche précédente, cette approche a l'avantage de préserver la qualité vidéo, mais obtient cependant des accélérations marginalement moins intéressantes. De plus, les auteurs n'indiquent pas les comportements que prend cette approche dans un contexte où on utilise plusieurs tranches dans une même trame.

3.4.3 Approche adaptative au niveau des tranches

Dans (Jung et Jeon, 2008), les auteurs proposent une approche adaptative effectuée au niveau des tranches pour paralléliser un encodeur H.264. L'objectif de cette approche est d'adapter les dimensions des tranches, à l'aide d'un modèle d'estimation de complexité, afin d'équilibrer

le balancement de charge entre les unités d'exécution. Les auteurs utilisent un modèle d'estimation de complexité basé sur une méthode de présélection rapide des modes de prédiction. Deux mesures de pointage sont utilisées pour effectuer la présélection : le *zero motion consistency score* (ZMCS) et le *largeblock consistency score* (LBCS). La première mesure attribue, à chacun des blocs d'une trame, un pointage basé sur l'historique des vecteurs de mouvement égaux à zéro. Cette mesure utilise d'abord la formule suivante pour calculer l'historique d'un bloc situé à *(n,m)* dans un macrobloc *(i,j)* pour une trame *t* :

$$ZMC_t(n,m;i,j) = \begin{cases} ZMC_{(t-1)}(n,m;i;j)+1, & \text{si } \left|m_x^t(n,m)\right| + \left|m_y^t(n,m)\right| = 0 \\ 0, & \text{si } \left|m_x^t(n,m)\right| + \left|m_y^t(n,m)\right| \neq 0 \end{cases} \tag{3.1}$$

où m_x^t et m_y^t représentent respectivement le mouvement horizontal et vertical du bloc situé à *(n ,m)*. Ensuite, la mesure de pointage est proprement appliquée :

$$ZMCS_t(i,j) = \begin{cases} \textit{Élevé}, & \text{si } ZMC_t(n,m;i,j) \geq T_{motion} \\ \textit{Faible}, & \text{si } ZMC_t(n,m;i,j) < T_{motion} \end{cases} \tag{3.2}$$

où T_{motion} est une constante valant 4. La seconde mesure attribue, à chacun des blocs d'une trame, un pointage basé sur l'historique des macroblocs de dimension 16 × 16. Cet historique est calculé ainsi :

$$LBC_t(i,j) = \begin{cases} LBC_{(t-1)}(i,j)+1, & meilleurMode(i,j) \in \{\textit{sauté}, P16x16\} \\ 0, & meilleurMode(i,j) \notin \{\textit{sauté}, P16x16\} \end{cases} \tag{3.3}$$

Par la suite, le pointage LBC est déterminé par l'équation :

$$LBCS_t(i,j) = \begin{cases} Faible, & LBC_t(i,j) < T_{mode1} \\ Moyen, & T_{mode1} < LBC_t(i,j) < T_{mode2} \\ Élevé, & T_{mode2} < LBCt(i,j) \end{cases} \quad (3.4)$$

Où T_{mode1} et T_{mode2} sont des constantes valant respectivement 1 et 4. Par la suite, les auteurs proposent des règles à appliquer pour présélectionner les modes qui seront testés. Ces règles sont présentées dans le tableau 3.1, où un CHK_{intra} égal à 1 indique qu'on doit vérifier le mode intra du macrobloc, et où la valeur de *g* sera utilisée pour calculer la complexité des modes à tester.

Tableau 3.1 Pré-sélection des modes avec leur complexité *g*

CHK_{intra}	**ZMCS**	**LBCS**	**Modes à tester**	**Valeur de *g***
	Élevé	-	{Sauté, P16×16}	1
	Faible	Élevé	{Sauté, P16×16, P16×8, P8×16}	2
	Faible	Moyen	{Sauté, P16×16, P16×8, P8×16, P8×8}, où P8×P8 est élément de {8×8}	3
	Faible	Faible	{Sauté, P16×16, P16×8, P8×16, P8×8}, où P8×P8 est élément de {8×8, 8×4, 4×8, 4×4}	4
	Élevé	-	{Sauté, P16×16, Intra}	1
	Faible	Élevé	{Sauté, P16×16, P16×8, P8×16, Intra}	2
	Faible	Moyen	{Sauté, P16×16, P16×8, P8×16, P8×8, Intra}, où P8×P8 est élément de {8×8}	3
	Faible	Faible	{Sauté, P16×16, P16×8m, P8×16, P8×8, Intra}, où P8×P8 est élément de {8×8, 8×4, 4×8, 4×4}	4

Les auteurs ont déterminé expérimentalement les deux formules suivantes pour calculer la complexité des modes à tester selon la valeur de *g* et la valeur de CHK_{intra} :

$$C(K, CHK_{\text{intra}} = 0)(g) = \begin{cases} 1, & g = 1 \\ 2.42, & g = 2 \\ 3.12, & g = 3 \\ 5.28, & g = 4 \end{cases} \tag{3.5}$$

$$C(K, CHK_{\text{intra}} = 1)(g) = \begin{cases} 4.97, & g = 1 \\ 6.48, & g = 2 \\ 7.23, & g = 3 \\ 9.48, & g = 4 \end{cases} \tag{3.6}$$

Le modèle de la complexité globale d'une trame *t* se calcule alors de la manière suivante :

$$C^{\sim t} = \sum_{k=0}^{M-1} C(K, CHK_{\text{int}\,ra})(g) \tag{3.7}$$

Enfin, la complexité maximale accordée à une tranche sur une machine ayant N unités d'exécution est la suivante :

$$C_{tranche}^{\sim t} = \frac{C^{\sim t}}{N} \tag{3.8}$$

La distribution des macroblocs dans les tranches est basée sur la complexité maximale des tranches $C_{tranche}^{\sim t}$ et la valeur estimée de la complexité du macrobloc courant $C(K, CHK_{\text{int}\,ra} = 1)(g)$. L'algorithme de partitionnement des tranches ajoute, pour chaque tranche, des macroblocs jusqu'à ce que la valeur totale de la complexité estimée de chaque macrobloc composant la tranche se rapproche de la valeur de $C_{tranche}^{\sim t}$ sans la dépasser. Les auteurs ne présentent, dans leur article, que les résultats théoriques de cette approche. Ces résultats montrent que leur approche obtient un gain théorique de vitesse allant jusqu'à environ 30 % pour certaines séquences vidéo.

3.5 Approches systèmes à mémoire distribuée pour l'encodage parallèle

La majorité des approches à mémoire distribuée, dont (Luo, Sun et Tao, 2008; Rodriguez, Gonzalez et Malumbres, 2006), utilisent une décomposition effectuée au niveau des groupes d'images. Ce type de décomposition a l'avantage de réduire la synchronisation entre les unités d'exécution de ces systèmes qui a un coût de communication plus élevé que les systèmes à mémoire partagée. Cependant, ce type de décomposition a le désavantage de ne pas réduire le délai d'encodage.

3.5.1 Approches parallèles pour les processeurs massivement parallèles

Dans (Qiang et Yulin, 2003), les auteurs étudient quatre approches parallèles pour paralléliser un encodeur H.264 sur une architecture massivement parallèle. La première approche consiste à diviser chaque trame en plusieurs tranches de même taille en vue de distribuer ces tranches sur des unités d'exécution. La seconde approche, située au niveau des macroblocs, associe chacune des sept structures de partitionnement inter possibles (voir section 2.3.4) à une unité d'exécution. La troisième approche, située au niveau des trames, associe chaque trame de référence, pour un maximum de cinq trames, à une unité d'exécution dans le but d'effectuer un encodage vidéo multi-références. La dernière approche, située au niveau des macroblocs, associe chaque mode de prédiction intra à une unité d'exécution.

Parmi ces quatre approches, les auteurs ont conclu que seule la première est potentiellement intéressante, car elle permet une bonne mise à l'échelle et un balancement de charge potentiellement bien équilibré. La mise à l'échelle de la deuxième approche est limitée par le nombre de partitions inter possibles par macrobloc. De plus, la différence de complexité entre les sept structures de partitionnement empêche un balancement de charge équilibré. Par exemple, les simulations des auteurs montrent que la septième structure de partitionnement (16 blocs 4×4) est au moins deux fois plus complexe que la première structure de partitionnement (1 bloc 16×16). Pour sa part, la troisième approche limite la mise à l'échelle à cinq unités d'exécution. De plus, elle force le système à utiliser plusieurs trames de

référence, ce qui n'est pas toujours désiré. Cependant, le balancement de charge de travail est généralement assez bien équilibré quand le schéma d'encodage sépare les trames I d'au moins 15 trames. La dernière approche présente tout simplement une proportion de parallélisme trop faible pour être utilisée seule.

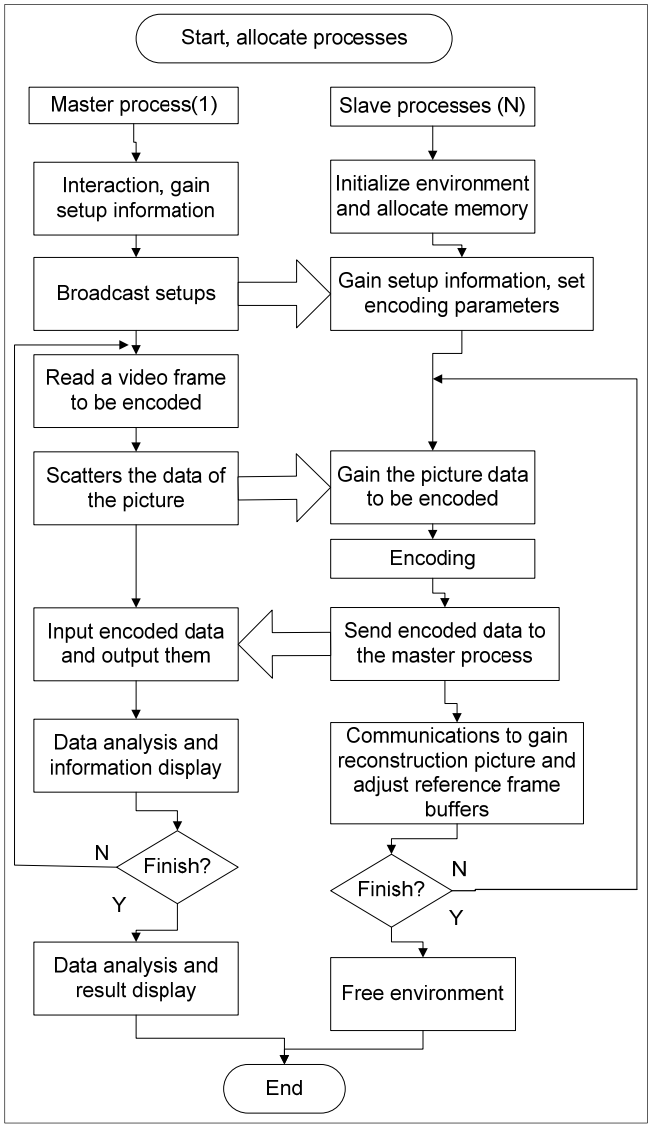


Figure 3.4 Diagramme de flux de l'approche parallèle de (Qiang et Yulin, 2003).
Tirée de (Qiang et Yulin, 2003)

Pour implémenter la première approche, les auteurs ont utilisé une conception de type maître/travailleur, illustrée à la figure 3.4, et l'interface de programmation parallèle MPI (Foster, 1995), une interface spécialisée dans le passage de messages. Les auteurs ont testé

leur approche sur une machine Dawning-2000 massivement parallèle avec quatre séquences vidéo QCIF (*miss_am, suzie, foreman, carphone*)[5] (Xiph.org, 2010). L'encodeur a été configuré pour encoder 115 trames, incluant 3 trames I, 36 trames P, et 76 trames B. Le tableau 3.2 montre les accélérations obtenues pour ces 4 séquences avec 2, 4, 6, 8 et 10 unités d'exécution (UE).

Tableau 3.2 Accélération de l'approche parallèle

Séquence	**Accélération**				
	2 UE	**4 UE**	**6 UE**	**8 UE**	**10 UE**
miss_am	1,95	3,65	4,85	6,08	6,59
suzie	1,93	3,53	4,92	6,09	6,54
foreman	1,82	3,31	4,53	5,64	6,5
carphone	1,95	3,54	4,97	6,11	6,32

(Yung et Chu, 1998) proposent de paralléliser un encodeur H.261 au niveau des tranches et d'utiliser une méthode prédictive pour balancer équitablement la charge de travail entre les unités d'exécution. Les auteurs présentent d'abord leur algorithme de balancement de charge avec des temps d'exécution connus. L'objectif de cet algorithme est de distribuer les macroblocs entre les unités d'exécution de manière à ce que chacune d'elles obtienne un temps d'encodage $t_{balancée}^{tranche}$ proche de :

$$t_{balancée}^{tranche} = \frac{1}{P}\sum_{k=1}^{M} t_k^{macrobloc} \tag{3.9}$$

où $t_k^{macrobloc}$ représente le temps d'encodage du macrobloc k de la trame courante, M le nombre de macroblocs par trame, et P le nombre d'unités d'exécution. L'algorithme de balancement assigne deux indices à chacune des unités d'exécution : *indiceDepart[j]* et *indiceFin[j]* qui représentent respectivement les numéros du premier et du dernier macrobloc

[5] Ces séquences sont régulièrement utilisées dans les articles scientifiques portant sur le codage vidéo. L'emploi de ses séquences contribue à la reproductivité des expériences.

à traiter par l'unité d'exécution *j*. Après l'assignation des indices de *j* unités d'exécution, le temps d'encodage restant à allouer en moyenne par unité d'exécution correspond à :

$$t_{restant} = \frac{1}{P-j} \sum_{i=indiceFin[j]+1}^{P} t_i \tag{3.10}$$

Au départ, l'algorithme assigne respectivement les valeurs de 1 et de M/P aux indices *indiceDepart[1]* et *indiceFin[1]*. Cette assignation produit un $t_1^{tranche}$ et un $t_{restant}$ respectivement égal à :

$$t_1^{tranche} = \sum_{i=1}^{1+(M/P)} t_i^{macrobloc} \tag{3.11}$$

$$t_{restant} = \frac{1}{P-1} \sum_{i=(M/P)+2}^{M} t_i^{macrobloc} \tag{3.12}$$

Si $t_1^{tranche}$ est plus grand que $t_{restant}$, alors l'algorithme retranche le dernier macrobloc de la tranche 1. Si $t_1^{tranche}$ est plus petit que $t_{restant}$, alors l'algorithme rajoute le prochain macrobloc à la tranche 1. L'algorithme poursuit ainsi jusqu'à ce que $t_1^{tranche}$ soit près de $t_{restant}$. Par la suite, l'algorithme applique le même genre de routine aux autres unités d'exécution. Malheureusement, les temps d'exécution de chacun des macroblocs ne peuvent pas être connus avant leur encodage. Pour contourner ce problème, les auteurs proposent la formule suivante pour prédire les temps d'encodage des macroblocs de la trame courante en fonction des valeurs de la trame précédente :

$$\tilde{t}_{(k,f)} = \tilde{t}_{(k,f-1)} + a \times (t_{(k,f-1)} - \tilde{t}_{(k,f-1)}) \tag{3.13}$$

où *a* est un facteur de mise à l'échelle déterminé empiriquement. Les valeurs prédites et les indices *indiceDepart[j]* et *indiceFin[j]* seront, par la suite, utilisés pour distribuer les macroblocs, sous la forme de tranche, à chacun des processeurs. Les auteurs ont testé cette approche sur un système IBM SP2 composé de 24 processeurs avec les 39 premières trames de la séquence CIF de *table-tennis (Xiph.org, 2010)*. Leurs résultats indiquent que le temps d'exécution de leur approche balancée représente environ 80% du temps d'exécution de la même approche non-balancée.

3.5.2 Approches parallèles pour les grappes d'ordinateurs

Dans (Luo, Sun et Tao, 2008), les auteurs présentent un algorithme parallèle pour un encodeur H.264 et pour une architecture parallèle basée sur une grappe d'ordinateurs. Les auteurs mentionnent que 4 facteurs doivent être tenus en compte pour développer un algorithme efficace : le débit de l'encodage, le balancement de charge, l'architecture du système et l'état des nœuds. La ligne directrice de ces auteurs est la suivante : un algorithme ne doit pas avoir d'impact sur le débit d'encodage et sur la qualité vidéo. Comme dans plusieurs autres articles portant sur les grappes d'ordinateurs, les auteurs suggèrent et utilisent une décomposition au niveau des groupes d'images. Ce type de décomposition a pour avantages de minimiser la communication inter-nœuds, ce qui est important pour ce type d'architecture, et de permettre une mise à l'échelle importante (on peut rajouter des nœuds pour augmenter les performances.) Pour contrôler l'échange entre les nœuds, les auteurs utilisent un modèle maître/travailleur et un balancement de charge qui attribue dynamiquement un groupe d'images à chaque nœud inoccupé. Dans ce contexte, le modèle maître/travailleur est approprié pour modifier rapidement le nombre de nœuds du système. Grâce à cet assemblage de méthodes, les auteurs obtiennent notamment une accélération de 9.44 pour une grappe de 11 nœuds, ce qui correspond à une efficience de 85.6 %.

Les auteurs de (Rodriguez, Gonzalez et Malumbres, 2006) vont un peu plus loin et proposent une implémentation basée sur la combinaison de deux décompositions du domaine : la première décomposition utilise la librairie MPI pour distribuer des groupes d'images sur des

sous-nœuds d'ordinateurs. Cette approche permet d'accélérer le temps de traitement au prix d'une latence élevée. La seconde approche a pour but de réduire cette latence et consiste à diviser les trames en tranches et à distribuer ces tranches aux ordinateurs appartenant à un sous-groupe d'ordinateurs.

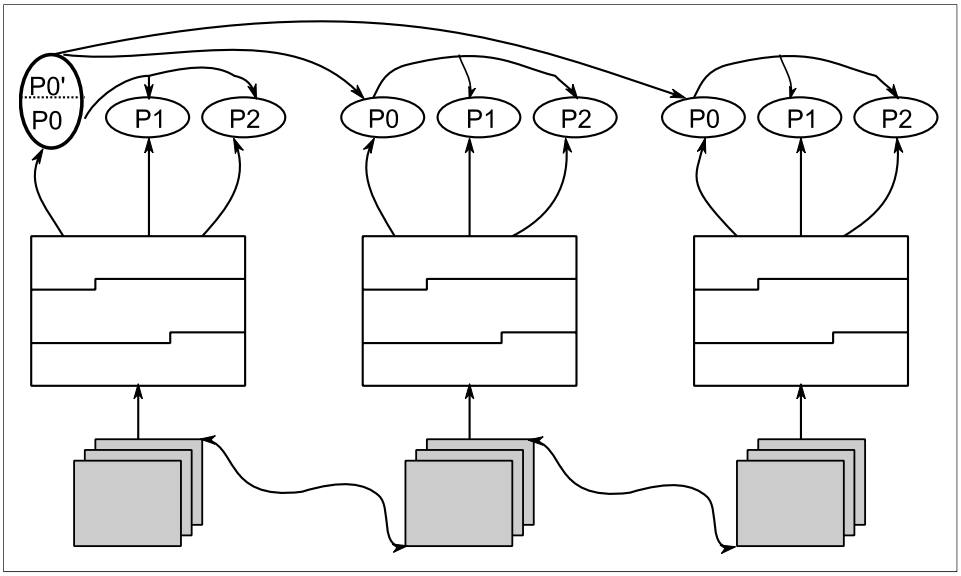


Figure 3.5 Encodage parallèle hiérarchique.
Tirée de(Rodriguez, Gonzalez et Malumbres, 2006)

La figure 3.4 montre l'exemple de cet algorithme appliqué à trois groupes d'ordinateurs, chacun composé de trois processeurs (*P0*, *P1*, *P2*) et traitant un groupe d'images. Cette approche utilise un modèle parallèle maître/travailleur hiérarchique. Le processeur *P0* du premier groupe d'images, noté aussi *P0'*, correspond au processeur maître de l'application et gère la distribution des groupes d'images. Les processeurs *P0* de chaque groupe d'images sont des maîtres locaux qui gèrent la distribution des tranches. Cette approche a pour avantage de réduire le temps de réponse du système mais, comme toute décomposition effectuée au niveau des tranches, a l'inconvénient d'augmenter le débit. Les auteurs ont notamment testé cette approche sur un nœud composé de 4 bi-processeurs AMD Opteron, connectés par un commutateur réseau Ethernet Gigabit et cadencés à 2 GHz et. Les auteurs ont testé 4 configurations et obtiennent respectivement, pour une séquence d'une résolution de 720 × 480, une accélération d'environ 5.9 pour 8 groupes d'une unité d'exécution, de 6.4

pour 4 groupes de 2 unités d'exécution, de 6.7 pour 2 groupes de 4 unités d'exécution, et de 6.8 pour un groupe de 8 unités d'exécution. Malheureusement, les auteurs ne mentionnent pas les latences respectives de chacune de ces configurations. De plus, cette méthode n'obtient pas une efficience supérieure à celle de la méthode précédente, puisque les efficiences des 4 configurations sont respectivement égales à 73.8, 80, 83.8 et 85 %.

3.6 Conclusion

Dans ce chapitre, nous avons brièvement comparé les approches basées sur une décomposition fonctionnelle avec les approches basées sur une décomposition du domaine. Nous avons mentionné que le premier type de décomposition est surtout utilisé par des processeurs spécialisés, puisqu'il ne permet pas de produire des approches suffisamment génériques pour supporter un nombre variable de cœur. Par la suite, nous avons présenté plus en détails quelques approches appartenant à la seconde catégorie de décomposition. Plus spécifiquement, nous avons vu qu'il existe des approches au niveau des groupe d'images, au niveau des trames, au niveau des tranches et au niveau des macroblocs, ainsi que des approches hybrides.

Dans le prochain chapitre, nous présenterons l'approche parallèle de l'encodeur H.264 d'Intel, qui est une approche effectuée au niveau des tranches. Nous avons choisi d'analyser cette approche puisque nos travaux sont basés sur cet encodeur d'Intel. De plus, nous croyons que cette approche est une bonne approche de base puisqu'elle implique peu de changement sur la structure de l'application; qu'elle est flexible aux changements de code, par exemple, à l'emploi de différents algorithmes d'estimation de mouvement; qu'elle ne limite pas le choix des paramètres d'encodage disponibles, sauf le nombre de tranches, qui doit être égal aux nombre d'unités d'exécution que l'utilisateur désire employer. Enfin, l'approche d'Intel permet d'obtenir des accélérations raisonnables à un faible coût de conception et de maintenance.

CHAPITRE 4

APPROCHE MULTI-TRANCHES D'INTEL

Dans ce chapitre, nous analyserons l'approche multi-tranches implémentée par Intel dans son encodeur H.264 livré en code d'exemple avec sa librairie *Integrated Performance Primitives* (IPP) (Intel, 2010b). Puis, nous proposerons une modification simple, au niveau du filtre de déblocage de H.264, pour améliorer l'accélération de cette approche. Nous y présenterons aussi les résultats de nos analyses et de cette proposition. Des propositions plus complexes, basées sur un parallélisme au niveau des tranches et des trames, seront abordées au prochain chapitre.

4.1 Introduction

L'encodeur H.264 d'Intel utilise une approche parallèle qui se situe au niveau des tranches. Cette approche divise la trame courante en un nombre de tranches égal au nombre de fils utilisé par l'encodeur. Chacune de ces tranches est associée, dans une relation un-à-un, à un fil par l'intermédiaire de l'interface de programmation parallèle OpenMP. Par la suite, ces tranches sont encodées en parallèle par l'encodeur. Cet encodeur atteint, habituellement, son accélération maximale quand le nombre de fils spécifié est égal au nombre d'unités d'exécution disponibles sur le système. Si nous n'avons qu'un seul fil, l'application sera exécutée en mode séquentiel. La figure 4.1 illustre l'application de l'approche d'Intel pour un encodage parallèle d'une trame effectué à l'aide de quatre fils.

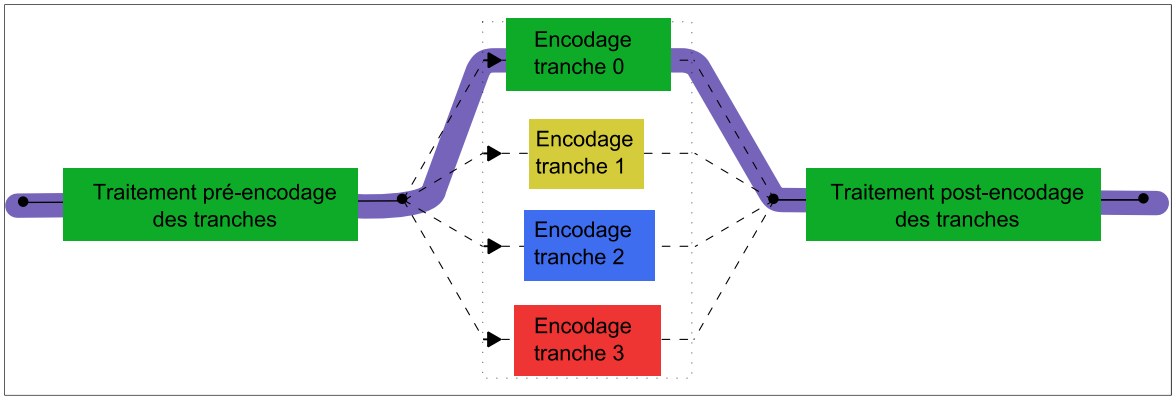


Figure 4.1 Schéma de l'encodage parallèle d'Intel effectué sur une trame et utilisant quatre fils.

Dans cette figure, le bloc *traitement pré-encodage des tranches* représente le code exécuté, par un seul fil d'exécution, avant l'encodage parallèle des tranches. Cette section a notamment pour responsabilités : de lire la prochaine trame; de déterminer son type; d'allouer, si nécessaire, un tampon mémoire pour conserver cette trame; d'identifier et de sélectionner la prochaine trame à encoder (nécessaire seulement dans un contexte où l'encodeur utilise des trames B) et d'encoder l'entête de cette trame. Les blocs d'encodage de tranches, correspondant ici à une exécution parallèle à quatre fils, ont pour responsabilité d'encoder, en parallèle, les tranches de la trame et de produire, pour chacune de ces tranches, un train de bit intermédiaire. Notons, que les blocs responsables de l'encodage des tranches ont, sur la figure 4.1, des tailles différentes. Ces tailles montrent, à titre d'exemple, que le temps d'encodage des tranches peut varier. L'étape *traitement post-encodage des tranches* a pour rôles : de récupérer les trains de bits intermédiaires, produits précédemment, pour les écrire dans le train de bit principal, celui du fichier généré à la sortie; d'appliquer le filtre de déblocage, si nécessaire, sur les macroblocs de la trame encodée, et de mettre à jour la liste des trames de référence. Soulignons aussi la présence d'une large ligne grise qui représente l'ordre d'exécution des différentes sections de l'approche d'Intel. Ainsi, le traitement de l'étape *post-encodage des tranche*s peut débuter seulement après la fin de l'encodage de la tranche la plus longue à encoder, soit la tranche 0 dans l'exemple illustré à la figure 4.1.

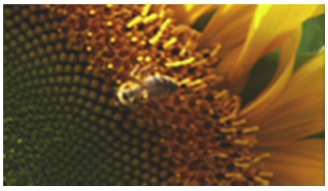
L'approche d'Intel permet de réduire significativement le temps d'exécution de l'encodage, en plus de réduire légèrement sa latence. Par exemple, nous verrons plus loin que l'accélération est habituellement comprise entre 2.5 et 3.0 sur un processeur quad-cœur d'Intel. De plus, l'implémentation d'Intel est presque transparente aux yeux de l'utilisateur. En effet, les modes d'exécution séquentiel et parallèle utilisent la même interface et les mêmes paramètres d'encodage, à l'exception du nombre de tranches qui doit être égal au nombre de fils à utiliser pour le mode parallèle. Malgré les avantages et la simplicité de cette approche multi-tranches, deux inconvénients majeurs sont à noter. En premier lieu, l'accélération obtenue est encore loin de l'accélération maximale théorique qui est, par exemple, de 4.0 pour un quad-cœur. Cette accélération est limitée pour trois raisons : la proportion non négligeable du code séquentiel, qui représente environ 5 % du temps d'exécution pour le mode séquentiel; la distribution de charge de travail inéquitable pour l'encodage des tranches et la hausse du temps d'encodage occasionnée par l'augmentation du nombre de tranches. En second lieu, la qualité vidéo, mesurée par le PSNR, se dégrade au fur et à mesure qu'on rajoute des tranches. C'est que l'encodeur ne profite plus alors pleinement de la corrélation qui existe entre les macroblocs situés aux frontières de deux tranches voisines et que des entêtes supplémentaires doivent être rajoutés au train de bits produit pour représenter l'entête des tranches et les unités NAL affiliées.

4.2 Description des tests d'encodage

Dans ce chapitre, ainsi que dans le chapitre suivant, nous présenterons les résultats de nos différentes simulations. Ces simulations ont été exécutées sur les 300 premières images de six séquences vidéo YUV de format 4:2:0, présentées dans le tableau 4.1, d'une résolution spatiale de 1280×720 pixels (Xiph.org, 2010). D'une part, nous avons sélectionné ces séquences pour améliorer la reproductivité de nos expériences. En effet, ces séquences sont gratuitement distribuées dans Internet à titre de séquences de test. De fait, d'autres chercheurs peuvent les télécharger en vue, par exemple, de reproduire nos expériences. D'autre part, nous avons sélectionné ces séquences car elles contiennent, pour la plupart,

beaucoup d'objets en mouvement ainsi que des déplacements de caméra. Ces deux caractéristiques permettent de tester l'impact des approches sur l'estimation de mouvement.

Tableau 4.1 Séquences vidéo utilisées pour les tests

Séquence	**Première image**	**Caractéristiques**
Train-station		• Zoom arrière de la caméra • Seulement quelques petits objets en déplacement.
Horse-cab		• Mouvement panoramique de la caméra de droite à gauche. • Quelques gros objets en déplacement.
Sunflower		• Mouvement saccadé de la caméra de gauche à droite et de haut en bas. • Mouvement lent d'un objet.
Waterskiing		• Séquence très complexe • Beaucoup d'effets de vagues sur l'eau • Mouvement panoramique de la caméra vers la droite.
Rally		• Mouvement saccadé de la caméra à l'intérieur d'une voiture en déplacement. • Beaucoup de poussière. • Déplacement d'un objet moyen.

Séquence	Première image	Caractéristiques
Tractor		• Mouvement panoramique de la caméra de droite à gauche • Mouvement moyen d'un gros objet.

Ces simulations ont été exécutées en utilisant, notamment, les paramètres suivants :

- Un taux d'échantillonnage temporel de 50 images par seconde.
- Un débit variable (*Variable Bit Rate*, en anglais.)
- Encodage entropique CABAC.
- Un ratio de trames I de 1/20 et un ratio de trames P de 19/20.
- Une étendue de l'estimation de mouvement de huit pixels qui forment une zone de recherche de 16 × 16 pixels.
- Une estimation de mouvement, précis au quart de pixel, effectué par l'algorithme de recherche logarithmique (Jain et Jain, 2002).
- Une trame de référence pour l'estimation de trames P.
- Le profil principal (*main profile*).
- Des débits respectifs de 1, 5, 15 et 20 Mbit/s. (Voir l'annexe I pour une comparaison visuelle entre ces débits).

L'annexe II présente plus en détail les paramètres utilisés dans l'encodeur

4.3 Analyse des performances de l'approche d'Intel

Dans les prochaines sous-sections, nous présenterons l'impact de l'utilisation des tranches sur le temps d'exécution et la qualité vidéo. Nous examinerons aussi la proportion approximative du code séquentiel de l'encodeur et présenterons son impact sur l'accélération de l'encodeur en mode parallèle. Finalement, nous discuterons de la distribution inégale de charge de travail de l'approche d'Intel.

4.3.1 Impacts du nombre de tranches sur les performances

Dans notre contexte d'encodage, l'utilisation de plus d'une tranche par trame a pour conséquence de réduire la qualité pour un débit donné de la séquence encodée et d'augmenter le temps nécessaire pour l'encodage. En premier lieu, l'inter-indépendance des tranches d'une même trame empêche les macroblocs situés à la frontière supérieure d'une tranche *n* d'accéder à leurs voisins qui sont situés dans la tranche *n-1*. Par exemple, la figure 4.2 illustre en gris les macroblocs qui n'ont pas accès à l'ensemble de leurs voisins. Ces macroblocs appartiennent à une trame de 80×45 macroblocs divisée en quatre tranches de 900 macroblocs (c'est ainsi que procède l'encodeur d'Intel).

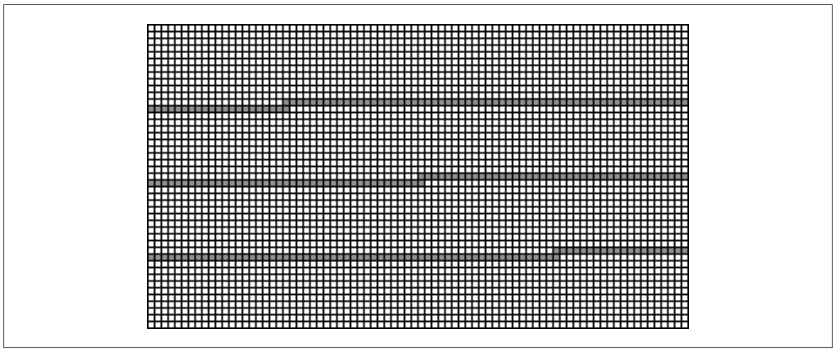


Figure 4.2 Macroblocs, d'une trame à quatre tranches, n'ayant pas accès à l'ensemble de leurs voisins.

Partant de ce fait, l'encodeur est dans l'impossibilité d'appliquer, sur chacun de ces macroblocs, les neufs modes de la prédiction intra (voir section 2.4.1), quand il s'agit d'un macrobloc intra, et la prédiction du vecteur de mouvement basé sur trois voisins (voir section 2.4.2), quand il s'agit d'un macrobloc inter. Dans le premier cas, seuls les modes ayant accès aux données nécessaires seront appliqués (ce qui élimine l'ensemble des modes utilisant le voisin supérieur). Dans le second cas, la prédication sera effectuée seulement avec les voisins disponibles. Le nombre de macroblocs n'ayant pas accès à tous leurs voisins dépend du nombre de tranches par trame et de la résolution de la séquence à encoder. Si on exclut les macroblocs qui n'ont pas accès à leur voisin supérieur de gauche et que le nombre de

tranches est inférieur ou égal au nombre de lignes de macroblocs, alors cette valeur est obtenue en appliquant l'équation suivante :

$$nbMbs(nbTranches) = nbMBsLigne \times (nbTranches - 1) \quad (4.1)$$

où *nbMBsLigne* représente le nombre de macroblocs par ligne, Pour une séquence d'une résolution de 1280 × 720 pixels, on remplace *nbMbsLigne* par 80, et on obtient :

$$nbMbs(nbTranches) = 80 \times (nbTranches - 1) \quad (4.2)$$

Quand une tranche ne débute pas au premier macrobloc, comme illustré à la figure 4.2, l'équivalent d'une ligne de macroblocs n'a pas accès à leurs voisins supérieurs, en plus d'un macrobloc supplémentaire qui est situé directement sous le premier macrobloc de la tranche courante qui n'a pas accès à son voisin supérieur de gauche. La prédiction de ce dernier macrobloc sera seulement affectée pour les modes 4, 5 et 6 (voir figure 2.7) de la prédiction intra 4 × 4, et cela uniquement pour le premier bloc du macrobloc. C'est pourquoi nous ne tenons pas compte de ce type de macrobloc dans les équations 4.1 et 4.2.

Sur le plan de la compression, l'encodeur réduit ses probabilités d'utiliser un mode de prédiction optimal, puisque certains modes ne sont pas disponibles. Les macroblocs affectés consomment donc plus de bits qu'à la normale, ce qui a pour effet de réduire le nombre de bits disponibles pour les autres macroblocs, et par extension, de réduire la qualité globale de la séquence vidéo compressée. Notons que les séquences vidéo ayant beaucoup de mouvements verticaux sont davantage affectées par l'utilisation restreinte des modes de prédictions. De plus, l'encodeur doit rajouter au train de bits, pour chaque tranche, un entête à taille variable contenant de l'information sur la tranche, et un entête NAL de 40 bits. Cela réduit davantage le nombre de bits disponibles pour la compression.

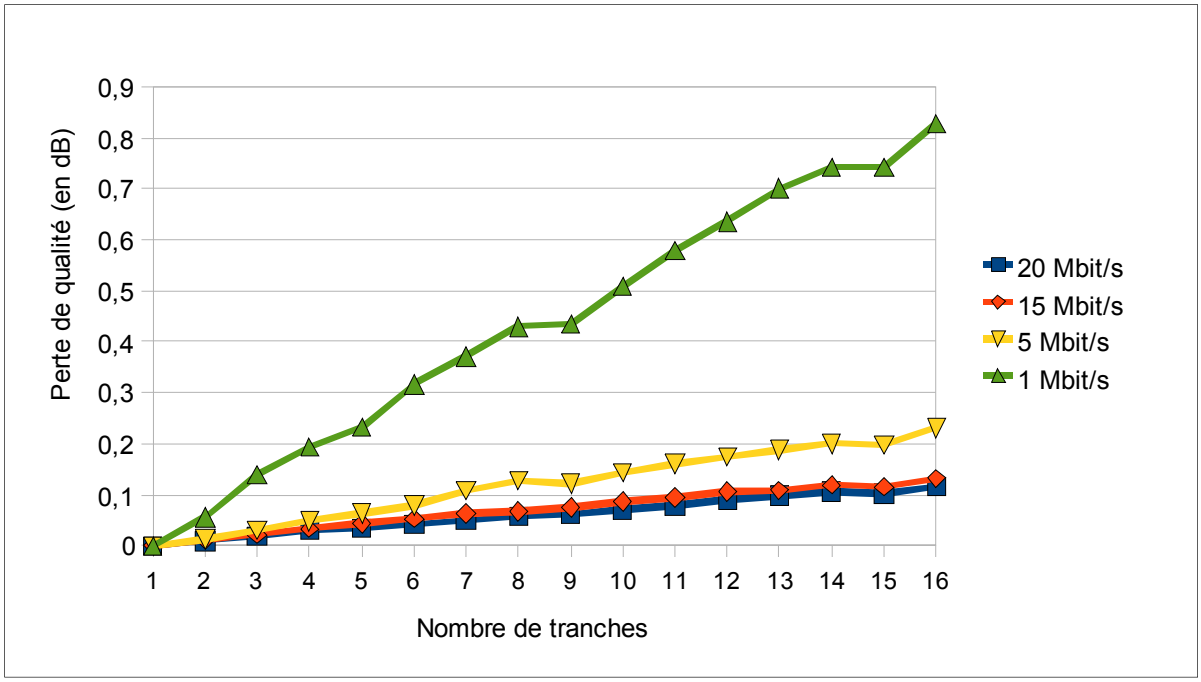


Figure 4.3 Perte de qualité, en dB, de l'approche originale d'Intel en fonction du nombre de tranches pour des débits de 1, 5, 15 et 20 Mbit/s.

La figure 4.3 illustre la perte de qualité moyenne, en dB, après l'application du PSNR, des six séquences vidéo décrites précédemment. Ces séquences ont été encodées avec des débits de 1, 5, 15 et 20 Mbit/s et un nombre de tranches allant de 1 à 16. La perte de qualité obtenue par un encodage de *t* tranches est calculée, pour chaque séquence et chaque débit, par rapport à un encodage effectuée avec une seule tranche. On observe d'abord sur la figure que cette perte augmente avec le nombre de tranches et cela, d'une manière presque linéaire. On remarque aussi que la perte de qualité est beaucoup plus grande pour un encodage effectué à faible débit. Par exemple, un encodage effectué avec huit tranches produit une perte d'environ 0.06 dB pour un débit de 20 Mbit/s, comparativement à 0.42 dB pour un débit de 1 Mbit/s. Cette différence est notamment causée par la proportion des bits générés par l'encodeur pour représenter les entêtes et les unités NAL des tranches. Cette proportion est inversement proportionnelle au débit. Ainsi, l'encodage à 20 Mbit/s utilisera, pour huit tranches, environ 0.04% de son train de bits pour représenter les entêtes NAL des tranches, alors que l'encodage à 1 Mbit/s utilisera environ 0.8 % de son train de bits pour le faire.

(L'annexe III présente des graphiques détaillant la perte de qualité de chacune des six séquences utilisées.)

Sur le plan de la complexité d'encodage, l'ajout de tranches a pour effet d'augmenter le temps d'encodage. Cette augmentation est calculée à l'aide de l'équation suivante :

$$ATET_{seq(n)} = \frac{T_{seq(n)}^{encodage}}{T_{seq(1)}^{encodage}} \tag{4.3}$$

Où $T_{seq(n)}^{encodage}$ représente le temps moyen pour encoder séquentiellement une trame composée de *n* tranches et $T_{seq(1)}^{encodage}$, le temps moyen pour encoder séquentiellement une trame d'une seule tranche. L'augmentation de la complexité d'encodage est essentiellement attribuable, pour les macroblocs n'ayant pas accès à l'ensemble de leurs voisins, à la réduction du nombre de voisins utilisés pour la prédiction du vecteur de mouvement et à la désactivation de certains modes pour la prédiction intra. D'abord, la réduction du nombre de voisins utilisés pour la prédiction de vecteur de mouvement, jumelée à des trames de moins bonne qualité (résultat de la perte de qualité), a pour conséquence de réduire le nombre de macroblocs de type sauté (voir annexe IV), qui sont très rapides à coder, et d'augmenter les macroblocs de type inter, puisque la prédiction est moins précise. L'encodage de ce dernier type de macrobloc est plus long parce qu'il implique, notamment, des appels à des méthodes d'estimation de mouvement. Ces méthodes ne sont pas appelées quand on applique une prédiction de vecteur de mouvement et que cette prédiction donne un résultat acceptable (c'est-à-dire que la différence entre le bloc à encoder et le bloc de référence obtenu après prédiction est suffisamment petite.) Par ailleurs, la réduction du nombre de modes utilisés pour la prédiction intra augmente, elle aussi, le temps d'encodage. En effet, la prédiction intra permet généralement d'identifier plus rapidement le mode d'encodage, soit le mode 16 × 16 ou le mode 4 × 4, du macrobloc, en plus de réduire la quantité de données à encoder. L'emploi de moins de modes produit donc le comportement inverse.

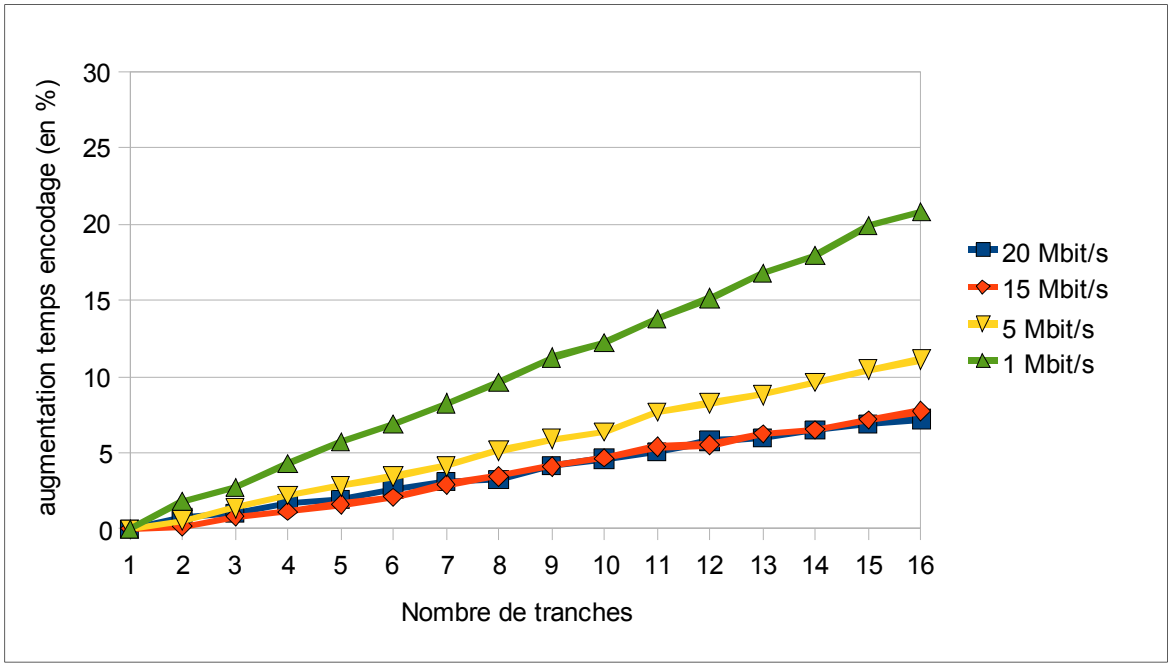


Figure 4.4 Augmentation du temps d'encodage de l'approche originale d'Intel en fonction du nombre de tranches pour des débits de 1, 5, 15 et 20 Mbit/s.

La figure 4.4 illustre l'augmentation moyenne du temps d'encodage en fonction du nombre de tranches et du débit utilisé. Les paramètres d'encodage utilisés pour produire cette figure sont les mêmes que ceux utilisés pour produire la figure 4.3. L'augmentation du temps obtenue par un encodage de *t* tranches est calculée, pour chaque séquence et chaque débit, par rapport à un encodage effectuée avec une seule tranche. De plus, l'encodage a été effectué en mode séquentiel. Par conséquent, l'augmentation du temps d'encodage est seulement attribuable à l'augmentation du nombre de tranches. Dans un premier temps, on remarque que le débit a encore un effet significatif sur le comportement de l'encodeur. En effet, un encodage avec un débit de 1 Mbit/s subit une augmentation du temps d'encodage d'environ 20 % pour un encodage avec 16 tranches par trame, par rapport à un encodage avec une tranche par trame. Alors qu'un encodage avec un débit de 20 Mbit/s subit seulement une augmentation d'environ 8 %. Cette différence est, d'une part, explicable par le concept de proportionnalité (décrit à la section 4.3.1), c'est-à-dire que les fonctions touchées par un encodage à faible débit représentent, en relatif, une quantité de code plus faible que celle touchée par un encodage à haut-débit. D'autre part, l'augmentation du débit a tendance à réduire la croissance du nombre de macroblocs de type inter lors de l'ajout de tranches (voir annexe IV). (Les annexes

V, VI, VII présentent des graphiques plus détaillés sur le temps d'encodage et l'augmentation du temps d'encodage. Ces graphiques montrent différente vues, regroupés par séquences ou par débits, des mêmes résultats .)

4.3.2 Proportion du code séquentiel de l'approche d'Intel

Nous avons vu à la figure 4.1 que l'approche d'Intel contient une section parallèle, qui a pour responsabilité d'encoder les tranches, et deux sections séquentielles, l'une précédant la section parallèle et l'autre, la succédant. Selon la loi d'Amdahl (voir la section 1 à ce sujet), la proportion du code séquentiel a un impact important sur l'accélération d'une application parallèle. De ce fait, il est donc essentiel d'estimer cette proportion pour l'encodeur d'Intel. La figure 4.5 illustre les proportions de code séquentiel mesurées lors de l'exécution des tests précédemment décrits. Cette figure montre que les sections séquentielles de l'application représentent environ 5 % (cette proportion oscille entre 3 % et 7 %) du temps d'exécution de l'application en mode séquentiel. Cette proportion de 5 % se traduit, après l'application de la loi d'Amdahl sur des systèmes à quatre et huit unités d'exécution, par des accélérations théoriques maximales de 3,47 et 5,92 respectivement. Ces accélérations maximales sont encore plus faibles quand on intègre l'augmentation du temps d'encodage résultant de l'augmentation des tranches à la loi d'Amdahl. Aussi, la figure 4.5 montre que le débit a encore une fois, mais plus subtilement, une influence sur les performances parallèles de l'application. La proportion de code séquentiel plus grande pour un débit plus faible est conforme aux attentes de la loi de Gustafson. En effet, la complexité de la section parallèle augmente en même temps que le débit utilisé, tandis que la complexité de la section séquentielle est relativement constante.

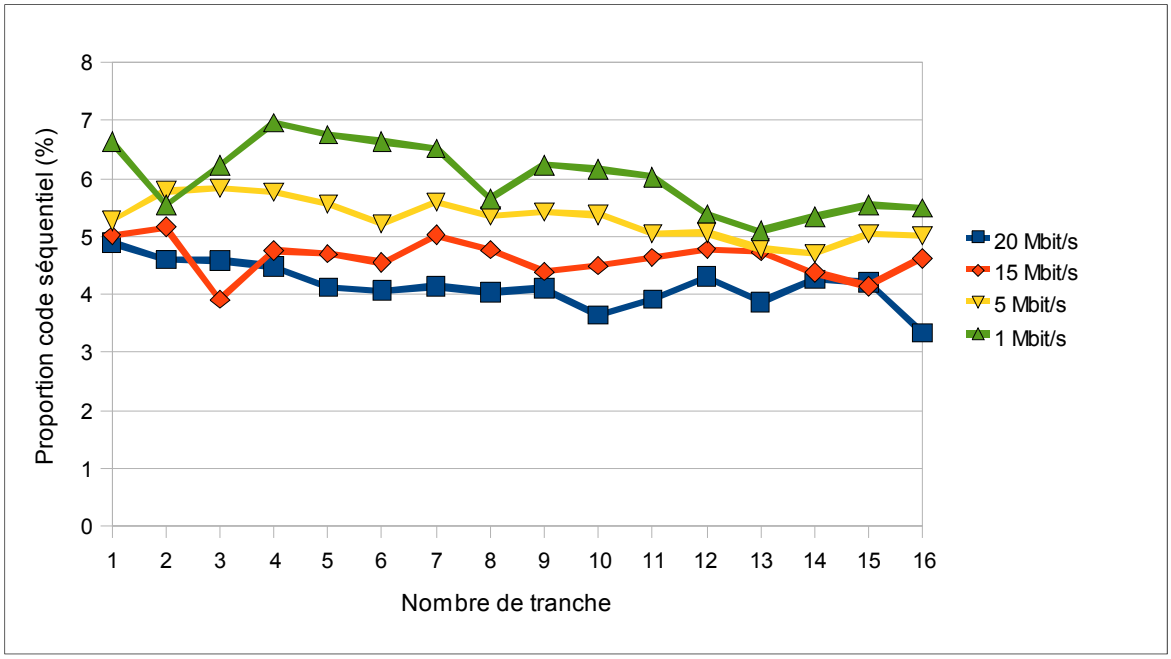


Figure 4.5 Proportion du code séquentiel (en pourcentage) de l'approche originale d'Intel en fonction du nombre de tranches pour des débits de 1, 5, 15 et 20 Mbit/s.

4.3.3 Distribution de la charge de travail de l'approche d'Intel

Nous savons qu'une application parallèle doit avoir une forte proportion de code parallèle pour être efficiente. Cependant, nous savons aussi que cette condition n'est pas suffisante à elle seule. Une application parallèle doit, en plus, permettre une distribution équitable de la charge de travail entre les unités d'exécution. Dans notre contexte, cela signifie qu'on doit attribuer, à chacun des fils, des tranches de même complexité. À ce sujet, Intel a choisi de découper ses trames en un groupe de tranches composé, à plus ou moins un, du même nombre de macroblocs. Par exemple, l'encodage parallèle d'une séquence vidéo d'une résolution de 1280×720 pixels (3600 macroblocs) utilisant quatre fils va générer quatre tranches composées chacune de 900 macroblocs. Dans un contexte idéal, où chaque macrobloc prend le même temps d'encodage, ce type de décomposition produit un balancement de charge équitable (les fils débutent et terminent leur traitement simultanément). Dans un contexte réel, le temps de traitement d'un macrobloc dépend, notamment, du temps nécessaire pour effectuer l'estimation de mouvement, de son vecteur de mouvement et de ses données résiduelles, si données il y a. Par exemple, un macrobloc

qui représente une région sans changement nécessitera un temps d'encodage significativement plus bas qu'un macrobloc contenant une région en déplacement. En effet, l'encodeur d'Intel n'appliquera pas d'estimation de mouvement pour le premier macrobloc, puisque le vecteur (0,0) minimise suffisamment l'erreur, et n'aura pas besoin d'encoder de données résiduelles, alors que le second macrobloc nécessitera ce type de traitement. Le temps de traitement variable des macroblocs a pour effet de produire une distribution de charge inégale entre les fils. Ainsi, le fil qui encode la tranche qui a globalement les macroblocs les plus simples terminera son traitement avant les autres et tombera alors en état d'inactivité. Et inversement, le fil qui encode la tranche la plus complexe terminera son traitement après les autres et deviendra alors le goulot d'étranglement de la section parallèle.

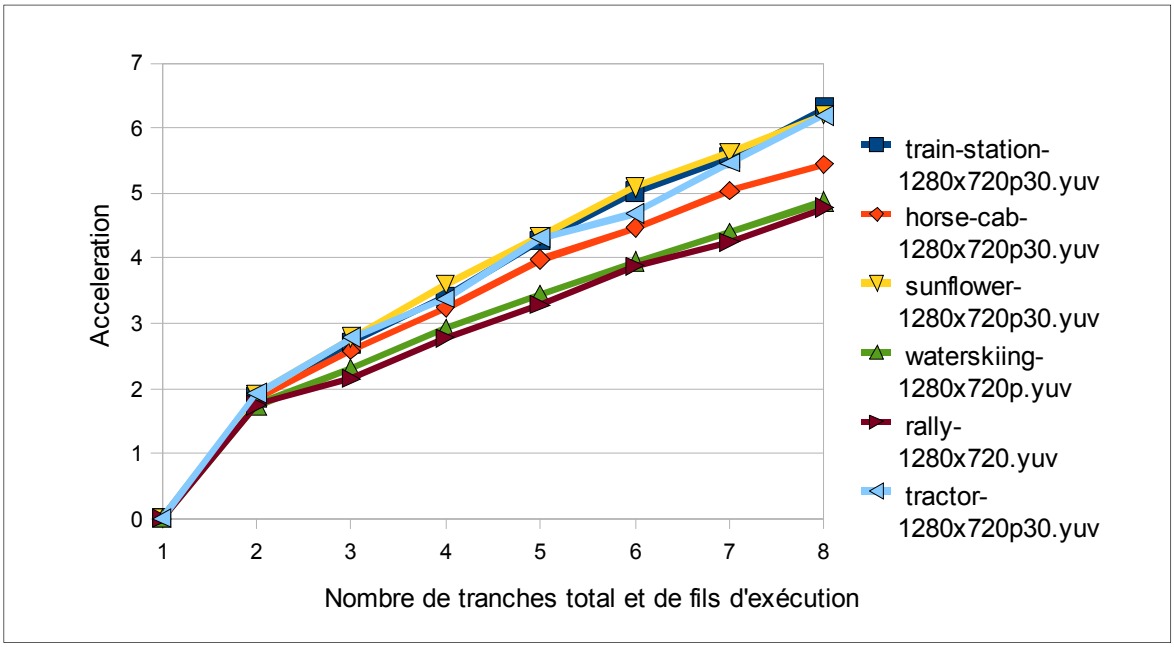


Figure 4.6 Accélération de l'encodage parallèle des tranches pour six séquences encodées avec un débit de 20 Mbit/s.

La figure 4.6 illustre les répercussions de la distribution de charge inégale sur l'accélération de la section parallèle qui traite l'encodage des tranches d'une trame. Cette figure a été produite pour des débits d'encodage de 20 Mbit/s. L'abscisse indique le nombre de tranches, de fils et de cœurs utilisés et l'ordonnée indique l'accélération. Une distribution de charge

égale aurait pour conséquence de produire une accélération très proche du nombre de cœurs (et de fils) utilisé. Ici, les accélérations obtenues sont nettement en dessous du nombre de cœurs utilisés. Les séquences ayant les distributions de charges les plus inégales, telles que rally et waterskiing, obtiennent les accélérations les plus faibles. Pour produire la figure 4.6, nous avons calculé l'accélération obtenue par la section qui encodage les tranches uniquement. L'équation utilisée pour calculer cette accélération est la suivante :

$$A_{(n)}^{tranches} = \frac{T_{par(n)}^{tranches}}{T_{séq(n)}^{tranches}} \tag{4.4}$$

où $A_{(n)}^{tranches}$ représente l'accélération obtenue par la section qui encodage *n* tranches en parallèle en employant *n* fils, $T_{(n)}^{tranches}$ le temps d'encodage nécessaire pour encoder en parallèle *n* tranches, et $T_{seq(n)}^{tranches}$ le temps nécessaire pour encoder *n* tranches en séquentiel. Pour obtenir l'augmentation du temps d'encodage occasionné par la mauvaise distribution de la charge de travail, par rapport à une distribution idéal, on doit appliquer l'équation suivante :

$$ATET_{par(n)} = \frac{n}{A_{(n)}^{tranches}} \tag{4.5}$$

4.3.4 Modèle de prédiction de l'accélération de l'approche d'Intel

En combinant les trois causes de ralentissement dans la loi d'Amdahl, on obtient l'équation de prédiction suivante de l'accélération de l'approche Intel utilisant *n* tranches *n* fils par rapport à la solution séquentiel (une tranche et un fil) :

$$A^{total\sim}_{(n,d,v)} = \frac{1}{\left(S^{\sim}_{(n,d)} + \left(\frac{(1 - S^{\sim}_{(n,d)}) * ATET^{\sim}_{(n,d,v)}}{n} \right) \right)} \tag{4.6}$$

où $A^{total\sim}_{(n,d,v)}$ représente l'accélération prédite en fonction du nombre de fils *n* utilisés, du débit d'encodage *d* et de la séquence vidéo *v* à encoder; $S^{\sim}_{(n,d)}$ la proportion prédite du code séquentiel; et $ATET^{\sim}_{(n,d,v)}$ l'augmentation du temps d'encodage des tranches. Cette dernière fonction combine les deux causes de l'augmentation du temps d'encodage des tranches. Elle se formule ainsi :

$$ATET^{\sim}_{(n,d,v)} = ATET^{\sim}_{seq(n,d)} * ATET^{\sim}_{par(n,d,v)} \tag{4.7}$$

Les valeurs de prédiction des fonctions $S^{\sim}_{(n,d)}$ et $ATET^{\sim}_{seq(n,d)}$ peuvent être calculés en effectuant la moyenne des valeurs obtenues sur plusieurs séquences vidéo d'une même résolution et encodés avec les mêmes paramètres. En ce qui concerne la fonction $ATET^{\sim}_{par(n,d,v)}$, l'augmentation du temps d'encodage des tranches produit par la complexité inégale des tranches est trop variable pour être prédite à l'aide de la moyenne de plusieurs séquences vidéo.

Le tableau 4.2 présente les accélérations prédites et les accélérations réelles des séquences *train-station* et *rally* encodées avec un débit de 20 Mbit/s. Dans ce tableau, les valeurs de $S^{\wedge}$ et de $ATET^{\sim}_{seq(n,d)}$ sont nos valeurs calculées expérimentalement, et les valeurs de $ATET^{\sim}_{par(n,d,v)}$ sont les valeurs réelles obtenues lors de nos tests. Les quatre dernières colonnes affichent respectivement les valeurs de l'accélération prédite par le modèle, de l'accélération réelle, de l'accélération prédite de $\boldsymbol{S}^{\sim}$ seulement (c'est-à-dire que $ATET^{\sim}_{(n,d,v)}$ est remplacée par 1 dans l'équation 4.7) et de l'accélération produite par $ATET^{\sim}_{par(n,d,v)}$ seulement

(c'est-à-dire que les fonctions $\tilde{S}$ et $\tilde{ATET}_{seq(n,d)}$ sont remplacées par 1 dans l'équation 4.7). Ces deux dernières colonnes montrent que la proportion du code séquentiel réduit davantage l'accélération que la distribution inégale de la charge de travail sur les fils, quand les tranches d'une vidéo ont des complexités similaires. Par exemple, si on tient respectivement compte seulement de l'effet de la proportion de code séquentiel $\tilde{S}$ sur l'accélération et de la distribution de charge inégale de travail inégale, nous obtenons les accélérations 6.24 et 6.83 pour la séquence train-station encodé avec huit fils et huit cœurs. Inversement, quand les tranches d'une séquence vidéo ont des écarts de complexités importants, la distribution de la charge inégale sur les fils réduit davantage l'accélération que la proportion du code séquentiel.

Tableau 4.2 Accélérations réelles et prédites des séquences train-station et rally encodées avec un débit de 20 Mbit/s avec l'approche d'Intel

Séq. vidéo	**Nb. de fils**	$\tilde{S}$ *(en %)*	**$\tilde{ATET}_{seq(n,d)}$**	**$\tilde{ATET}_{par(n,d,v)}$**	**Acc. prédite**	**Acc. réelle**	**Acc. S^**	**Acc. $\tilde{ATET}_{par(n,d,v)}$**
Train-station	2	4.6	1.080	1.032	1.84	1.84	1.91	1.94
	4	4.5	1.017	1.122	3.5	3.27	3.53	3.56
	8	4	1.033	1.171	5.39	5.28	6.24	6.83
Rally	2	4.6	1.008	1.122	1.71	1.7	1.91	1.78
	4	4.5	1.017	1.406	2.59	2.57	3.53	2.84
	8	4	1.033	1.578	4.24	4.15	6.24	5.07

4.4 Parallélisation du filtre de déblocage de H.264

L'analyse de la section précédente montre que trois causes réduise l'efficacité de l'encodeur H.264 parallèle d'Intel : (1) la restriction, pour certains macroblocs, de méthodes de prédiction lors de l'ajout de tranches; (2) la proportion du code séquentiel de l'encodeur; (3) la distribution de la charge de travail inégale sur les fils lors du traitement des tranches. Nous

avons vu que la contribution de la première cause du ralentissement de l'accélération est beaucoup moins importante que les deux dernières causes, ce qui rend son amélioration moins importante. L'amélioration de la distribution de la charge de travail inégale implique la création d'un modèle de prédiction qui sera utilisé pour partitionner les tranches en fonction de la complexité de traitement prédite de chacun des macroblocs de la trame courante. Un tel modèle est difficile à développer, surtout dans un contexte, comme le nôtre, où l'encodeur est appelé à évoluer et à être utilisé avec différents paramètres et profils. Enfin, la diminution de la proportion séquentielle du code est une option envisageable, puisque Intel ne parallélise pas le filtre de déblocage. Ce filtre, qui est une méthode séquentielle gourmande en calculs, est utilisé pour réduire les effets de blocs, donc pour améliorer la qualité des images. Ce sera cette option que nous étudierons dans cette section. Nous donnerons d'abord une description sommaire du filtre de déblocage de H.264. Le lecteur qui désire en savoir plus sur ce filtre est invité à lire les articles (List *et al.*, 2003; Luthra et Topiwala, 2003). Puis nous présenterons les impacts de sa parallélisation sur le PSNR et l'accélération. Notons que la prochaine section présentera plus en détail les résultats obtenus pour les comparer avec l'approche parallèle d'Intel.

4.4.1 Description du filtre de déblocage de H.264

Le filtre de déblocage de H.264 a pour objectif d'analyser et de réduire les artefacts du type *effet de bloc*, produits aux frontières des blocs, en vue d'améliorer la qualité vidéo de la séquence. Pour les macroblocs intra, ces artefacts sont le résultat de l'application, sur chacun des blocs, d'une transformée et d'une quantification qui produisent, conjointement, une distorsion de continuité aux frontières des blocs. Pour chacun des macroblocs inter, ces discontinuités sont créées pour une raison similaire ou parce que l'estimation de mouvement n'a pas donné un vecteur de mouvement qui pointe exactement sur le bon bloc.

Le filtre de déblocage de H.264 est appliqué à l'encodage et au décodage, et cela, de la même manière pour produire des résultats cohérents. À cet égard, les macroblocs sont filtrés selon

leur ordre d'encodage, c'est-à-dire, de gauche à droite et de haut en bas. Les données résultantes de l'encodage d'un bloc sont utilisées comme données d'entrée pour les prochains blocs. Pour chaque macrobloc, un filtre horizontal est d'abord appliqué, sur une longueur de deux pixels, aux frontières verticales des blocs. Puis un filtre vertical est appliqué, sur une largeur de deux pixels, aux frontières horizontales des blocs. Le filtrage complet (le filtrage horizontal et vertical) d'un macrobloc doit être complété avant de passer au prochain macrobloc. Le filtre de déblocage de H.264 est optionnel et inclus deux modes de filtrage. Le premier mode applique le filtre de déblocage au niveau de la trame courante. Le second mode l'applique, de manière indépendante, au niveau des tranches composant la trame. C'est-à-dire, que dans ce cas, le filtrage des macroblocs situés à la frontière supérieure d'une tranche ne dépend pas du filtrage des macroblocs situés à la frontière inférieure de la tranche précédente.

Le filtre de déblocage est adaptatif : il effectue d'abord, pour chaque macrobloc, une analyse des données, puis il applique les noyaux de filtrage appropriés. L'analyse permet d'identifier, parfois à tort, les discontinuités produites par la compression, et de rejeter les discontinuités naturelles, par exemple, les bordures et les textures, qui ne doivent pas être filtrées. L'analyse sélectionne aussi un noyau de filtrage adapté au contexte. Par exemple, un bloc intra aura un noyau de filtrage plus agressif, parce que ce type de macrobloc est reconnu pour avoir des artefacts plus apparents.

Plus spécifiquement, le déblocage s'effectue sur trois niveaux :

- Au niveau des tranches : L'encodeur spécifie dans l'entête de chacune des tranches les champs $Offset_A$ et $Offset_B$. Ces deux champs, qui acceptent des valeurs comprises entre -6 et 6, sont utilisés pour ajuster la force du filtrage. Des valeurs négatives réduisent la force du filtrage et, inversement, des valeurs positives l'augmentent. Le choix de ces valeurs, qui a un impact sur la qualité subjective visuelle, est laissé à la discrétion des concepteurs. Typiquement, la réduction de la force du filtrage est utilisée pour conserver les détails des séquences hautes-résolution, où les artefacts sont généralement moins visibles. Et à l'opposé, l'augmentation de la force de filtrage est habituellement utilisée

pour réduire davantage les artefacts, qui sont plus visibles, sur des séquences vidéo de faibles résolutions (Luthra et Topiwala, 2003).

- An niveau des blocs : Chaque bloc est associé à une classe de filtrage qui sera utilisée par le prochain niveau. La classe de filtrage est sélectionnée en fonction du type du bloc (intra ou inter) et en fonction, pour les blocs de type inter seulement, des vecteurs de mouvement et des données résiduelles. Le tableau 4.3 présente les cinq classes BS (*Boundary-Strength*) de filtrage et les caractéristiques des blocs qui leurs sont associés. Il existe en tout trois modes de filtrage : un mode utilisé par la classe BS 4, un autre mode utilisé par les classes BS 1 à 3 et un dernier mode utilisé par la classe BS 0. Le premier mode est plus agressif que le second et le troisième mode est utilisé pour signifier qu'aucun filtre ne sera appliqué sur le bloc.
- Au niveau des échantillons (pixels) : À cette étape, le module de filtrage utilise la classe BS et le facteur de quantification du bloc pour traiter, individuellement, chaque pixel en deux étapes. À la première étape, le module de filtrage détermine si le pixel remplit les conditions nécessaires pour être filtré. À la seconde étape, si le pixel doit effectivement être filtré, le module de filtrage sélectionne alors le noyau de filtrage approprié et l'applique au pixel.

Tableau 4.3 Classe de filtrage Boundary-Strength (BS) en fonction des caractéristiques des blocs

Caractéristiques des blocs	**Bs**
Un des blocs est de type intra, et la frontière entre les blocs sépare deux macroblocs	4
Un des blocs est de type intra	3
Un des blocs contient des données résiduelles	2
La différence du mouvement du bloc est plus grande ou égale à 1 en échantillon luma	1
La compensation de mouvement est effectuée avec différentes trames de référence	1
Sinon	0

Typiquement, l'application du filtre de déblocage de H.264 permet une réduction du débit de 5 à 10 %, pour une même qualité visuelle objective (List *et al.*, 2003). Cette réduction dépend, notamment, du ratio des trames intra sur l'ensemble des trames (le filtre de déblocage est plus agressif sur les trames intra); du débit d'encodage (un débit élevé produit moins d'artefacts qu'un débit faible, ce qui implique moins de filtrage); des caractéristiques de la séquence vidéo (par exemple, une séquence avec peu de mouvement nécessite généralement moins de filtrage).

4.4.2 Parallélisation du filtre de déblocage de H.264

Comme nous l'avons dit plus haut, la syntaxe de H.264 permet de filtrer, de manière indépendante, les tranches d'une même trame. Nous utiliserons donc cette syntaxe pour paralléliser le filtrage des tranches. Pour cela, il suffit de spécifier, dans le champ *disable_deblocking_filter_idc* de chaque tranche, la valeur 2, qui indique que le filtre est seulement désactivé aux frontières des tranches, et d'indiquer, à l'aide d'OpenMP, que la boucle qui filtre les tranches doit être parallèle. Cette amélioration s'implémente donc rapidement (voir annexe VIII). Cependant, la désactivation du filtrage entre les bordures des tranches va produire, comme nous le verrons, une perte de qualité.

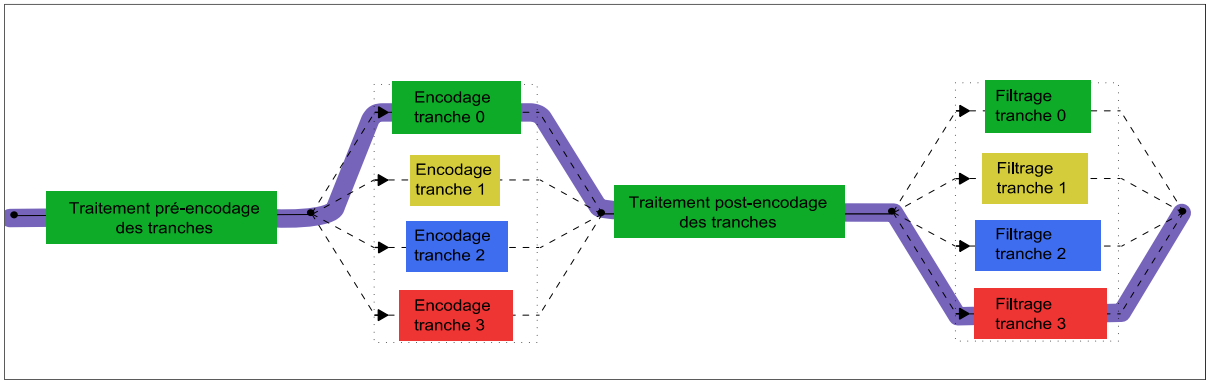


Figure 4.7 Encodage parallèle d'Intel, effectué sur une trame et utilisant quatre fils, avec un filtre de déblocage parallèle.

4.4.3 Impacts de la désactivation du filtrage des bordures des tranches sur la qualité

Pour évaluer la perte de qualité produite par la désactivation du filtrage entre les bordures des tranches, nous avons effectué des tests sur les six séquences précédemment décrites. Nous avons utilisé des débits de 1, 5, 10, 15 et 20 Mbit/s et un nombre de tranches allant de 1 à 8. Nous avons effectués nos tests sur les deux modes de filtrages actifs : celui qui filtre les frontières de deux tranches voisines, et celui qui ne les filtre pas. Nos résultats nous indiquent que le second type de filtrage ne produit pas une perte de qualité significative (i.e. en deçà de 0.01 dB) pour les encodages effectués avec les séquences et les débits mentionnés. Par contre, nos résultats indiquent des pertes moyennes, en dB, de 0.02, 0.01, 0.02, 0.04, 0.04 et 0.03, respectivement pour un nombre de tranches allant de 3 à 8 et pour des séquences encodées avec un débit de 1 Mbit/s. Ce qui corrobore le principe que le filtre est appliqué plus sévèrement sur les séquences encodées avec un faible débit.

4.5 Résultats

Dans cette section nous étudions les accélérations obtenues par l'approche d'Intel originale et l'approche Intel ayant été modifiée pour filtrer en parallèle les tranches. Les tests ont été effectués avec les paramètres et les séquences présentés à la section 4.2. Ces tests ont été exécutés sur un système HP ProLiant ML350 G6 Performance équipé de deux processeurs quad-cœurs Xeon E5530 de 2.4 GHz; d'une mémoire totale de 16 GO (12 Go RAM SAS, de type remplacement à chaud, et 4 Go DIMM DDR3 1333 MHz); deux disques durs, de type remplacement à chaud, totalisant 300 Go.

La figure 4.8 montre les accélérations moyennes obtenues, en fonction du nombre de fils, avec l'approche d'Intel originale. Nous observons d'abord que la progression de l'accélération est relativement constante et que les pentes s'amenuisent avec l'augmentation du nombre de tranches. Nous voyons aussi que l'accélération obtenue pour un débit donné est généralement supérieure à l'accélération obtenue pour un plus petit débit, notamment parce que la

proportion de code séquentiel est plus grande pour les petits débits. De plus, l'écart entre les accélérations de différents débits a tendance à s'élargir avec l'augmentation du nombre de fils. Par exemple, l'accélération moyenne sur huit fils pour un débit de 20 Mbit/s est de 4.8, alors que cette accélération moyenne est de 4.1 pour un débit de 1 Mbits/sec.

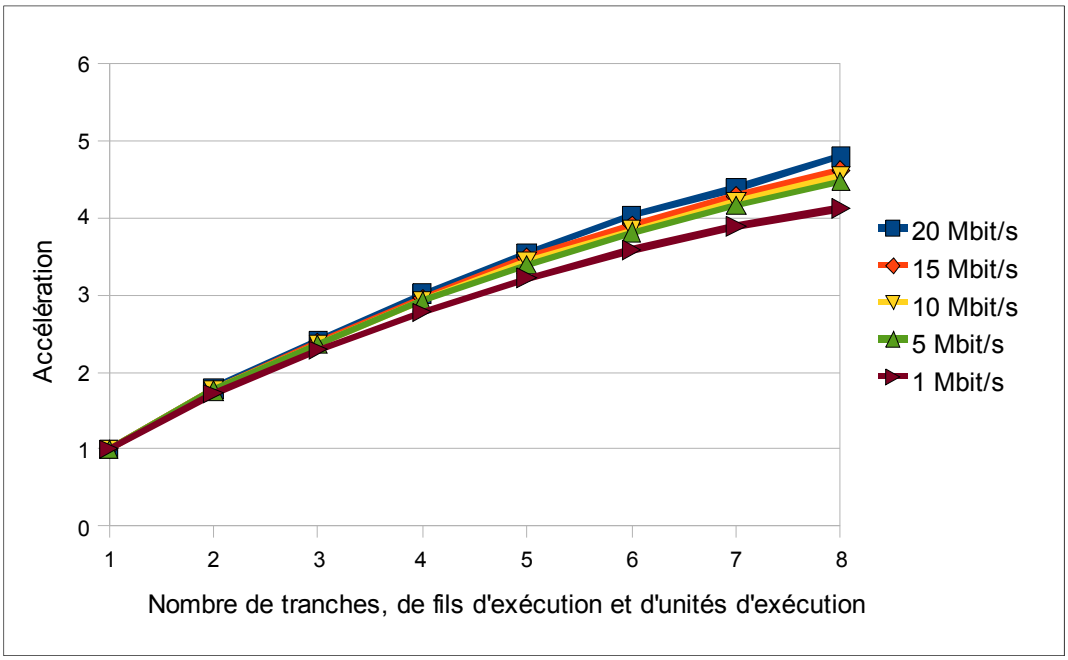


Figure 4.8 Accélération moyenne de l'approche originale d'Intel en fonction du nombre de fils pour des débits de 1, 5, 10, 15 et 20 Mbit/s.

La figure 4.9 montre les accélérations moyennes obtenues, en fonction du nombre de fils, en utilisant l'approche d'Intel avec un filtrage parallèle. Nous observons d'abord que les accélérations produites sont généralement plus élevées que les accélérations obtenues sans le filtrage parallèle. Par exemple, l'accélération moyenne pour un débit de 20 Mbit/s pour huit fils est de 5.7 comparativement à 4,8 avec l'approche sans filtrage parallèle.

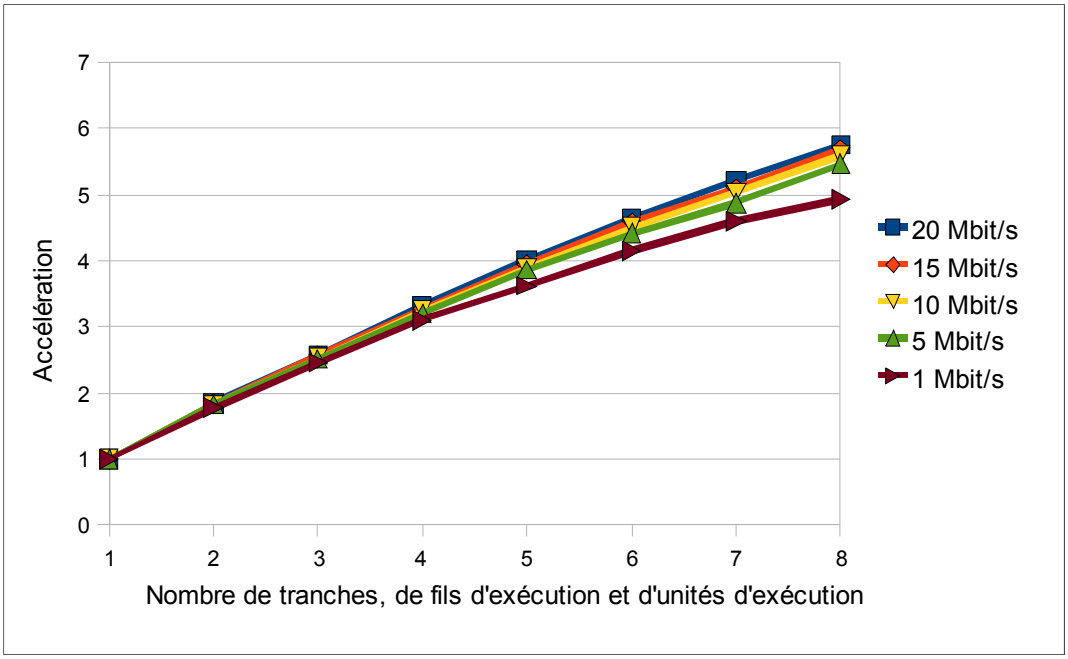


Figure 4.9 Accélération de l'approche d'Intel (modifié) avec filtre parallèle en fonction du nombre de fils pour des débits de 1, 5, 10, 15 et 20 Mbit/s.

À la vue de ces deux graphiques, nous pouvons conclure que l'approche d'Intel avec filtre de déblocage parallèle offre une meilleure accélération que l'approche d'Intel avec un filtre de déblocage séquentiel. Cependant, cette parallélisation supplémentaire ne permet pas de régler l'accélération sous-optimale occasionnée par la distribution de charge inégale et par la proportion de code séquentiel. Nous rappelons aussi que la perte supplémentaire de qualité produit par la désactivation du filtrage entre les frontières de deux tranches est en moyenne en dessous de 0.05 dB (une perte négligeable).

4.6 Conclusion

Dans ce chapitre, nous avons présenté l'approche parallèle de l'encodeur H.264 d'Intel. Nous avons vu que cette approche est basée sur une décomposition située au niveau des tranches et qu'elle offre des accélérations raisonnables, au prix d'une perte en qualité. Cependant, elle présente une accélération encore loin de l'accélération théorique maximale. Nous avons discuté des trois principales causes de cette accélération sous-optimale : la désactivation de méthodes de prédiction lors de l'ajout de tranches pour certains macroblocs; la proportion du

code séquentiel de l'application; la distribution de la charge de travail inégale sur les fils lors du traitement des tranches. Par la suite, nous avons étudié le filtre de déblocage de H.264 et nous avons conclu que sa parallélisation, qui est facile d'implémentation, permettra d'augmenter l'accélération de l'encodeur au prix d'une faible perte en qualité. Enfin, nous avons étudié plus spécifiquement les accélérations obtenues par nos tests sur l'approche d'Intel originale et sur sa version modifiée. Dans le prochain chapitre, nous présenterons une approche basée sur un encodage parallèle multi-trames et multi-tranches (MTMT). Cet algorithme a pour objectif de trouver une solution au problème de la mauvaise distribution de la charge de travail en codant simultanément, selon certaines restrictions, les tranches de deux trames voisines.

CHAPITRE 5

NOUVELLE APPROCHE MULTI-TRANCHES ET MULTI-TRAMES

Dans ce chapitre, nous présentons notre approche parallèle multi-trames et multi-tranches (MTMT) et ses trois modes, qui offrent des compromis au niveau de l'accélération et de la perte de qualité. Dans un premier temps, nous fournirons une description de notre approche et de ses trois modes. Dans un second temps, nous présenterons et comparerons les résultats de notre approche MTMT avec les résultats de l'approche originale d'Intel, ainsi qu'avec les résultats de l'approche d'Intel dont nous avons transformé le filtre de déblocage séquentiel en un filtre parallèle.

Nous rappelons que nous avons pour objectifs de développer une approche offrant une bonne mise à l'échelle, une accélération se rapprochant de l'accélération théorique et une perte de qualité limitée.

5.1 Introduction

Dans le chapitre précédent, nous avons montré que l'approche parallèle d'Intel offre une bonne accélération, mais que cette accélération n'est pas optimale. De plus, nous avons montré que cette approche produisait une perte de qualité visuelle qui augmentait avec l'ajout de tranches. Nous avons proposé de paralléliser le filtrage de déblocage pour améliorer les performances. Cette parallélisation supplémentaire augmente l'accélération, mais celle-ci est encore loin de l'accélération théorique, puisque la charge de travail inégale, combinée avec des sections de code séquentiel, empêche l'utilisation continue des fils disponibles.

Pour réduire l'impact de ce problème, nous proposons dans ce chapitre une approche MTMT qui réduit, par rapport à l'approche d'Intel, le temps d'inactivité des fils. L'idée centrale de ce type d'approche est d'augmenter le nombre de tranches prêtes à être encodées en permettant l'encodage simultané de deux trames. Ainsi, un fil peut encoder, sous certaines conditions,

une tranche appartenant à la prochaine trame quand aucune autre tranche n'est disponible dans la trame courante.

Nous avons développé et testé trois sous-approches MTMT, que nous appellerons modes 1, 2 et 3 dans ce document. Les algorithmes de ces modes se distinguent au niveau des conditions de disponibilités des tranches à encoder et des tranches de référence utilisées pour effectuer l'estimation de mouvement. Le premier mode, le plus rapide mais dégradant le plus la qualité, restreint la région de recherche de l'estimation de mouvement à l'intérieur des limites de la tranche courante. En d'autres mots, seule la tranche de la trame précédente, utilisée comme trame de référence, située à la même position que la tranche courante sera utilisée pour effectuer l'estimation de mouvement. Les données des autres tranches seront ignorées. Le second mode, moins rapide mais dégradant moins la qualité que le premier, élargit la région de recherche aux tranches voisines quand les tranches de référence y correspondant ont été traitées. Le troisième mode, moins rapide que le second mais offrant la meilleure qualité, rend une tranche prête à l'encodage seulement quand les tranches de référence couvrant la région de recherche ont été traitées.

Nous verrons plus loin que notre approche, quel que soit le mode utilisé, obtient une meilleure accélération que l'approche d'Intel au prix d'une perte de qualité plus grande mais souvent acceptable. De plus, nous verrons que notre approche a l'avantage de pouvoir cohabiter, sans trop d'effort de maintenance, avec la version originale de l'encodeur H.264 d’Intel.

5.2 Description de notre approche de type multi-trames et multi-tranches

Dans cette section nous présentons d'abord une vue sommaire de notre approche parallèle MTMT. Puis nous présentons quelques détails concernant, notamment, la division d'une trame en tranches; la structure parallèle de notre application, qui est divisé en deux sections parallèles; les points et les mécanismes de synchronisation que nous utilisons pour gérer les dépendances et transférer de l'information entre les fils; l'impact des dépendances entre des

trames voisines sur le contrôle de débit ainsi que la conséquence, sur l'estimation de mouvement au demi et quart de pixel, du filtre d'interpolation utilisé pour générer les demi-pixels .

5.2.1 Vue sommaire

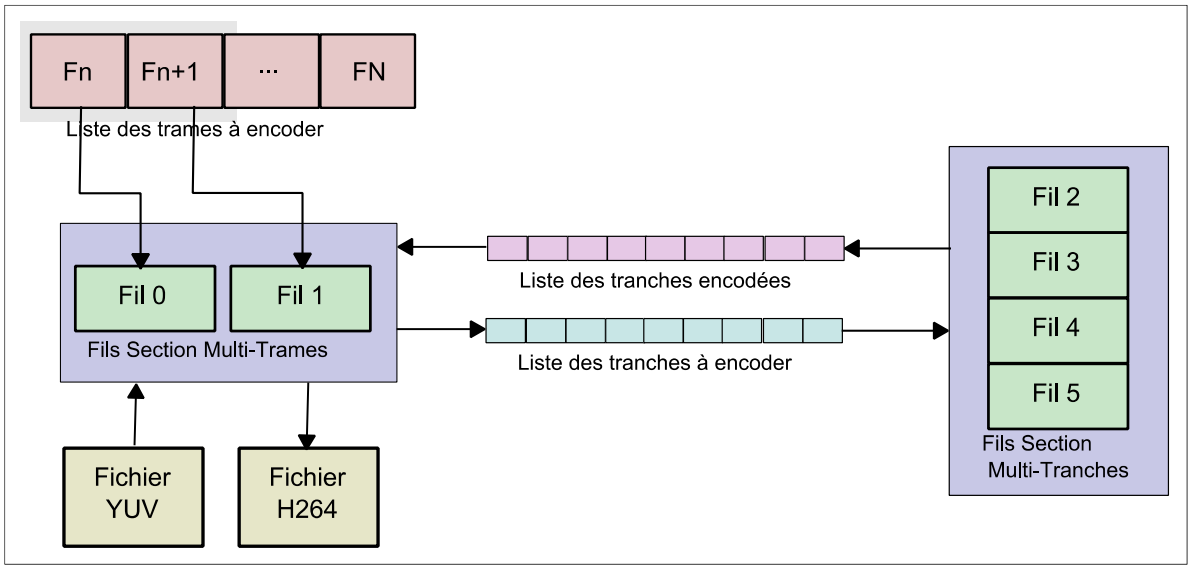


Figure 5.1 Schéma général de notre approche MTMT.

La Figure 5.1 illustre le schéma général de notre approche MTMT. Cette approche utilise deux sections parallèles, respectivement appelées *Section Multi-Trames* et *Section Multi-Tranches*, pour effectuer son traitement parallèle. Ces deux sections, activées après l'initialisation de l'encodeur, s'exécutent simultanément et se terminent uniquement une fois la dernière trame encodée. La première section est composée de deux fils, soient les fils 0 et 1. Ces fils ont pour responsabilité d'encoder en parallèle la trame courante, notée F_n, et la trame suivante, notée F_{n+1}. À cette fin, chaque fil est associé à un encodeur, soit à l'encodeur 1 ou à l'encodeur 2. Le premier encodeur traite les trames paires (0, 2, 4, etc.) et le second, les trames impaires (1, 3, 5, etc.). Comme pour l'approche d'Intel, chaque trame est divisée en trois étapes : l'étape *pré-encodage des tranches*, l'étape *encodage des tranches*, et l'étape *post-encodage des tranches*. Pour chaque trame à encoder, chaque fil effectue le traitement suivant : d'abord, il lit la trame YUV entrante; puis, il effectue le traitement de l'étape *pré-*

encodage des tranches; ensuite, il s'endort et attend que la deuxième section parallèle termine l'encodage de l'ensemble des tranches de la trame qu'il traite et lui envoie un message pour le réveiller; puis, il effectue le traitement de l'étape *post-encodage des tranches* et, finalement, il attend la disponibilité de la prochaine trame avant de recommencer ce processus pour la prochaine trame qu'il aura à gérer. La figure 5.2 illustre, à l'aide de flèches pointillées, les relations de dépendance qui existent entre les trames. Ces dépendances forcent les deux encodeurs à se synchroniser et empêchent un encodeur de commencer le traitement d'une trame F_{n+2} alors que l'autre encodeur traite encore une trame F_n.

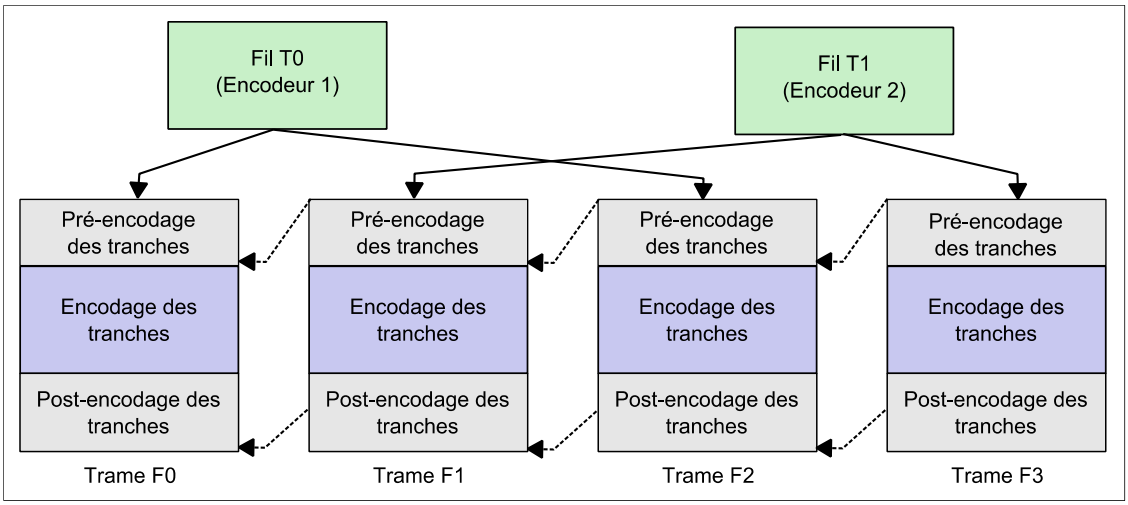


Figure 5.2 Relations de dépendance entre les trames et les encodeurs.

La seconde section (la section multi-tranches) traite en parallèle, à l'aide de *n* fils, l'encodage des tranches appartenant à l'une ou l'autre des deux trames traitées. Le nombre *n* de fils utilisés par la deuxième section est déterminé par l'utilisateur. Typiquement, ce nombre doit se situer entre deux et le nombre d'unités d'exécution disponibles sur le système. En dehors de cette plage de valeurs, l'algorithme devient généralement contre-productif, car il oblige l'ordonnanceur à effectuer régulièrement des changements de contexte, ce qui réduit l'efficacité de l'encodeur. Plus le nombre de fils est élevé et plus l'encodeur obtient des accélérations plus élevées. Néanmoins, un utilisateur pourrait choisir un nombre de fils inférieur au nombre de cœurs présents sur le système, par exemple, pour dédier un cœur à l'exécution d'une autre application.

Cette deuxième section parallèle utilise un gestionnaire de la file des tranches, que nous avons appelé le *SlicesPoolManager*, pour distribuer le traitement des tranches aux fils. Ces derniers demandent à tour de rôle à ce gestionnaire de leur fournir une tranche. Si la file des tranches à traiter n'est pas vide, ce gestionnaire réserve et assigne la tranche la plus prioritaire au fil demandeur. Ce dernier encode la tranche et redemande, par la suite, au *SlicesPoolManager,* une nouvelle tranche à encoder.

Le *SlicesPoolManager* sélectionne la prochaine tranche à encoder en deux étapes : d'abord, il identifie la trame non-traitée la plus ancienne, c'est-à-dire, la trame non-traitée ayant le numéro d'identification le plus bas; puis, il sélectionne, parmi les tranches prêtes à être encodées dans cette trame, la tranche ayant le numéro d'identification le plus bas. Dans le premier et le second mode de notre approche MTMT, une tranche est prête à être encodée quand l'encodeur qui la traite a reçu, de l'autre encodeur, la tranche qui servira de référence. Dans le troisième mode, une tranche est prête à être encodée seulement quand l'encodeur qui la traite a reçu l'ensemble des tranches de référence qui lui sera nécessaire lors de l'estimation de mouvement.

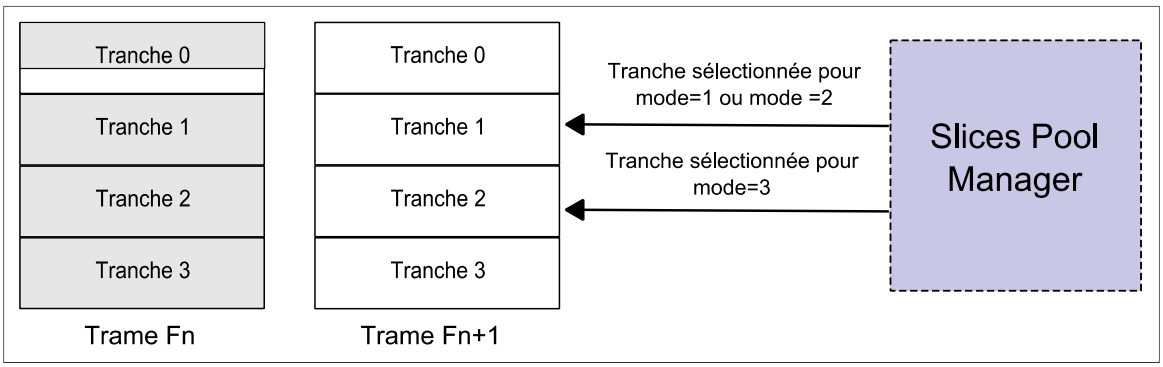


Figure 5.3 Sélection de la prochaine tranche en fonction du mode utilisé et des tranches disponibles.

La Figure 5.3 illustre un exemple de cette méthode de sélection. Dans cet exemple les tranches 1, 2, 3 de la trame F_n ont été entièrement encodées; la tranche 0, de la trame F_n, est en cours d'encodage et aucune tranche de la trame F_{n+1} n'est en court d'encodage. Dans ce

contexte, le *SlicesPoolManager* doit sélectionner la prochaine tranche, une tranche de la trame F_{n+1}, qui sera encodée. Si l'encodeur utilise le mode 1 ou le mode 2, le *SlicesPoolManager* sélectionnera alors la tranche 1, puisque la tranche 0 n'a pas accès à sa tranche de référence. Si l'encodeur utilise le mode 3, le *slicesPoolManager* sélectionnera alors la tranche 2, puisque c'est la seule tranche qui a accès à l'ensemble de ses tranches de références, c'est-à-dire aux tranches 1, 2 et 3 de la trame F_n.

Enfin, notons que la *Section Multi-Tranches* a aussi la responsabilité de vérifier, entre l'encodage de tranches, si des tranches encodées, qui serviront de références, doivent être transférées d'un encodeur à un autre. Si c'est le cas, la *Section Multi-Tranches* effectue la copie.

5.2.2 Division d'une trame

Tel que mentionné à la section 4.3.1, Intel divise les trames en tranche au niveau des macroblocs en attribuant à chaque tranche un nombre égal, plus ou moins un, de macroblocs. Dans notre approche, nous divisions les trames en tranches au niveau des lignes, c'est-à-dire que chaque tranche est composée d'une ou de plusieurs lignes entières de macroblocs. Cette division a l'avantage de faciliter la gestion de la mémoire et de l'estimation de mouvement, puisque nous n'avons pas de cas spéciaux à traiter comme des cas de zones de recherche non rectangulaires lors de l'estimation de mouvement.

Le nombre de lignes de macroblocs attribué à chaque tranche dépend de la résolution $H \times V$ pixels des trames à encoder et du nombre N de tranches désiré. Le nombre L de lignes de macroblocs par trame est égal à $V / 16$. Soient $S = L / N$ et $R = L$ modulo N, alors nous aurons $N - \mathrm{R}$ tranches avec S lignes et R tranches avec $S + 1$ lignes. L'annexe IX présente les structures exactes utilisées pour des séquences vidéo de 1280×720 pour un nombre de tranches compris entre 1 et 20 inclusivement.

5.2.3 Points de synchronisation

L'usage de deux encodeurs parallèles et d'une liste des tranches prêtes à être encodées implique l'utilisation de différents points de synchronisation et de mécanismes de synchronisation. Un point de synchronisation représente une ligne dans le code où un fil doit attendre la fin du traitement d'une autre section de code ou l'arrivée des données d'un autre fil avant de continuer son traitement. Un mécanisme de synchronisation est un mécanisme utilisé pour rendre applicable un point de synchronisation.

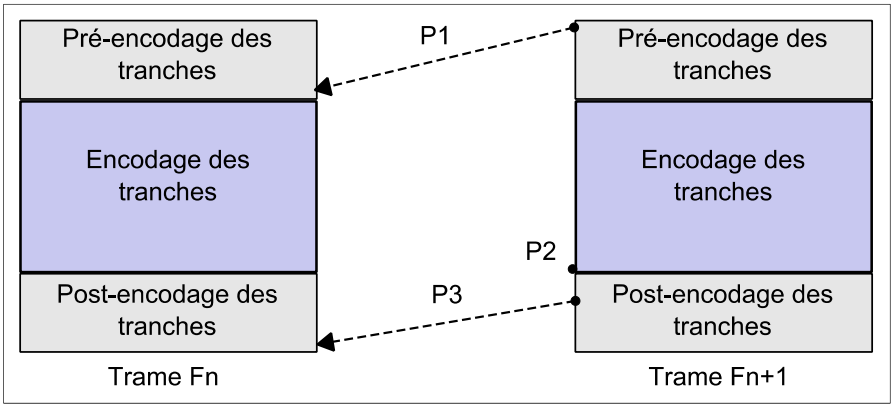


Figure 5.4 Points de synchronisation entre deux trames voisines.

Dans un premier temps, nous avons ajouté trois points de synchronisation, les points P_1, P_2, et P_3 qui sont illustrés, sous la forme de dépendances pour les points P_1 et P_3, sur la figure 5.3. Ces points sont implémentés à l'intérieur de la *Section Multi-Trames* seulement. Les points de synchronisation P_1 et P_3 servent, d'une part à bloquer le traitement de la trame F_{n+1} tant et aussi longtemps que le traitement de la trame F_n n'a pas atteint un certain point. D'autre part, ces points servent à transférer de l'information, depuis l'encodeur qui traite la trame F_n vers l'encodeur qui traite la trame F_{n+1}, une fois le blocage du point de synchronisation terminé. Le point de synchronisation P_2 est utilisé pour bloquer le démarrage du traitement de l'étape *post-encodage des tranches* tant et aussi longtemps que la *Section Multi-Tranches* n'a pas terminé l'encodage des tranches appartenant à la trame F_{n+1}.

Le point de synchronisation P_1 est situé au début de l'étape *pré-encodage des tranches* de la trame F_{n+1}, et le point de synchronisation P_3, au milieu de l'étape *post-encodage des tranches* de cette même trame. Le premier point attend la fin du traitement de l'étape *pré-encodage des tranches* de la trame F_n avant de transférer, notamment, le numéro de la trame courante, un compteur pour identifier l'ordre de l'image et un compteur utilisé pour insérer à intervalle régulier des trames IDR. Le second point attend la fin du traitement de l'étape *post-encodage des tranches* de la trame F_n avant de transférer la taille courante du fichier et des variables relatives au contrôle de débit. L'annexe X présente plus en détail, sous la forme de méthodes, les données échangées entre les deux encodeurs. Nous avons identifié ces données grâce à une analyse sur les changements de valeurs qui ont lieu entre deux trames voisines. Cette analyse a été effectuée pour un ensemble de paramètres spécifiques. Par conséquent, les méthodes de synchronisation implémentées ne garantissent pas un bon fonctionnement pour des paramètres d'encodage différents de ceux que nous avons utilisés.

Par ailleurs, l'utilisation de deux encodeurs parallèles nécessite aussi des points de synchronisation pour les tranches, puisque les tranches de l'encodeur de la trame F_n seront utilisées à titre de référence par l'encodeur de la trame F_{n+1}. Deux conditions sont nécessaires pour transférer une tranche d'un encodeur à un autre : (1) Le traitement d'une tranche à transférer et appartenant à une trame F_n doit être terminé; (2) l'encodeur de la trame F_{n+1} doit avoir terminé le traitement de l'étape *pré-encodage des tranches*, puisque c'est cette section qui alloue la mémoire et qui prépare la trame qui servira de trame de référence et qui, elle, recevra la tranche encodée à transférer. La synchronisation d'une tranche entre deux encodeurs peut avoir lieu à deux endroits : immédiatement après le traitement de la tranche à transférer, si l'encodeur destinataire est prêt à la recevoir; ou à l'intérieure de la *Section Multi-Tranches*, quand celle-ci vérifie les tranches à transférer.

5.2.4 Mécanismes de synchronisation

Nous avons implémenté deux types de mécanismes de synchronisation pour gérer les points de synchronisation P_1 à P_3 : un mécanisme de type *wait-and-signal* et un mécanisme de type

sleep-and- wakeup. Un seul de ces mécanismes est utilisé à l'exécution de notre encodeur. Le mécanisme utilisé dépend de la configuration qui a été utilisée lors de la compilation du code. Nous avons implémenté ces deux mécanismes pour comparer les différences de performances.

Pour le premier mécanisme, nous utilisons trois fonctions : *vm_event_init, vm_event_signal, et vm_event_wait*. Ces trois fonctions appartiennent à la machine virtuelle (*Virtual Machine* (VM)) d'IPP. En fait, cette machine correspond à un ensemble d'interfaces de fonctions communes aux systèmes d'exploitation Windows et Linux, où chaque interface a son implémentation spécifique sur chacun de ces systèmes. L'utilisation de ces trois fonctions est basée sur le concept d'événement (*event*). Dans ce contexte, chaque événement est associé à une variable événement spécifique. La fonction *vm_event_init* est utilisée pour initialiser une variable événement *e* avec une valeur égale à zéro. Pour sa part, la fonction *vm_event_wait* est employée pour attendre l'arrivé d'un signal *s*, quand *e* est égal à zéro; puis restaurer cette variable à zéro quand la fonction reçoit le signal *s*. Enfin, la fonction *vm_event_signal* est utilisée pour attribuer la valeur un à la variable *e*, quand celle-ci est égale à zéro, et envoyer un signal *s*, celui qui sera capturé par la méthode *vm_event_wait*, signalisant le changement de valeur de la variable *e*. Dans notre implémentation, chacun des deux fils de la *Section Multi-Trames*, les fils 0 et 1 sur la Figure 5.1, associe une variable distincte à leurs trois points de synchronisation. Chacune de ces variables est initialisée à zéro lors de l'initialisation de chacun des encodeurs. Puis, chaque point de synchronisation appelle la méthode *vm_event_wait,* en lui passant en paramètre la variable événement associée au point de synchronisation, et attend l'arrivée du signal qui est envoyé par l'encodeur de la trame précédente, via *vm_event_signal*, quand celui-ci a terminé le traitement et généré les données nécessaires pour débloquer le point de synchronisation.

Le second mécanisme de synchronisation implémenté fonctionne d'une manière similaire au premier mécanisme. Ce second mécanisme emploie des variables booléennes plutôt que des variables événements et une boucle qui vérifie si le point de synchronisation est bloqué à la place du mécanisme *wait-and-signal*. Si le point de synchronisation est débloqué, la boucle et

le point de synchronisation se terminent. Dans le cas contraire, la boucle appelle la méthode *vm_time_sleep*, une fonction de la VM d'IPP, pour endormir le fil en cours d'exécution pour une milliseconde. Par la suite, le fil se réveille pour vérifier à nouveau le statut du point de synchronisation et ainsi de suite jusqu'au déblocage du point de synchronisation.

Dans le cadre de nos travaux, nous avons testé ces deux mécanismes et nous n'avons constaté aucune différence de performance significative entre ces mécanismes. Les résultats présentés dans ce chapitre sont basés sur des simulations produites avec le second mécanisme.

5.2.5 Contrôle du débit avec délai

Pour atteindre un débit cible, un encodeur vidéo doit utiliser un mécanisme de contrôle de débit, appelé *Rate Control* (RC) en anglais, qui ajuste le facteur de quantification (ou *quantization parameter* dénoté QP) pendant l'encodage en fonction des bits utilisés et disponibles. L'implémentation de l'encodeur H.264 d'IPP 6.0 contient un contrôle de débit offrant cinq modes pour gérer le débit d'encodage :

- Deux modes d'encodage à débit constant (ou *Constant Bit Rate* noté CBR), l'un au niveau des trames et l'autre au niveau des tranches.
- Deux modes d'encodage à débit variable (ou *Variable Bit Rate* noté VBR), l'un au niveau des trames et l'autre au niveau des tranches.
- Un mode d'encodage avec un QP fixe pour l'ensemble des macroblocs de la séquence.

Les modes d'encodage de type CBR permettent d'ajuster le QP à la fin du traitement de chaque trame ou tranche, selon le mode activé, pour obtenir un débit d'encodage relativement constant au prix d'une qualité variable. Inversement, un encodage VBR permet d'ajuster le QP progressivement pour avoir une qualité relativement constante au prix d'un débit variable. Un contrôle de débit appliqué au niveau des tranches ajuste le QP à chaque tranche, alors qu'un contrôle de débit effectué au niveau des trames ajuste le QP à chaque trame.

Dans l'encodeur H.264 d'Intel la mise-à-jour des variables du contrôle de débit a lieu à la fin du traitement des trames dans la section que nous avons appelée l'étape *post-encodage des tranches*. Cette mise-à-jour est effectuée par l'entremise de la fonction *H264_AVBR_PostFrame* qui modifie la valeur du QP en fonction du QP courant et des bits actuellement utilisés.

Pour notre approche, nous avons opté pour un encodage de type CBR appliqué au niveau des trames, pour trois raisons : (1) le contrôle de débit de type QP n'est pas représentatif des cas d'utilisation réels; (2) l'encodage CBR représente mieux un contexte d'utilisation temps réel sur des réseaux; (3) l'encodage au niveau des tranches rend difficile la synchronisation de l'état du contrôle de débit entre les fils, en plus de rendre l'approche potentiellement non-déterministe, puisque les tranches ne sont pas toujours encodées dans le même ordre d'une exécution à l'autre.

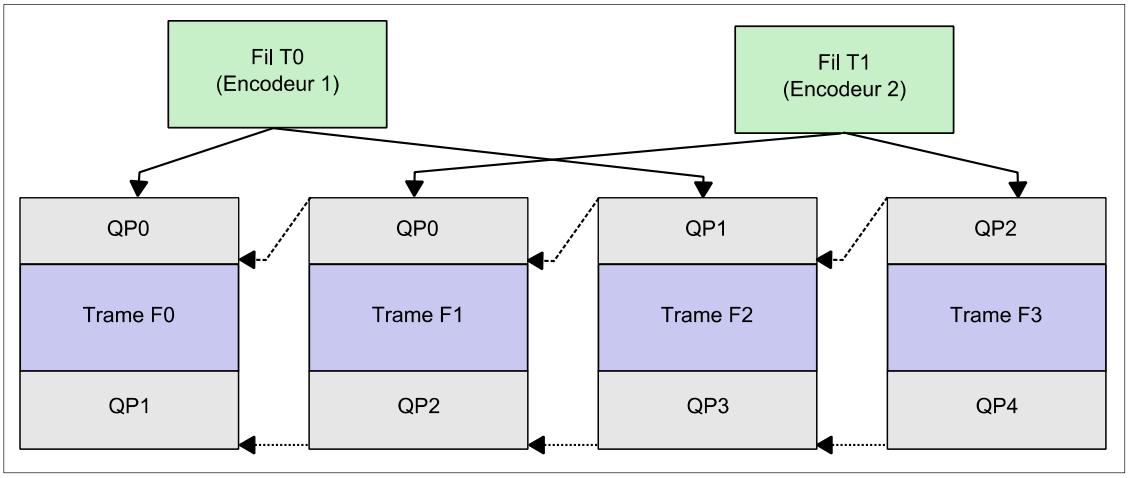


Figure 5.5 Délai d'une trame du QP généré par le contrôle de débit.

La Figure 5.5 montre que l'utilisation de deux encodeurs parallèles qui effectuent un encodage entrelacé, c'est-à-dire que l'un encode les trames paires et l'autre, les trames impaires, entraîne, pour notre approche, un retard d'une trame pour l'utilisation du QP calculé par le contrôle de débit à la fin d'une trame. Dans cette figure, les lignes pointillées illustrent les dépendances, au niveau des étapes *pré-encodage des tranches* et *post-encodage des tranche*s, de deux trames voisines. Les lignes continues associent chaque trame à l'un des

deux encodeurs utilisés. Les QP_x, où x est compris entre 0 et 4, représentent les valeurs des QP calculés par le contrôle de débit à la fin de l'encodage d'une trame, sauf pour QP_0 qui est le QP calculé à l'initialisation des encodeurs en fonction des paramètres d'encodage. Le calcul d'un QP_x dépend de QP_{x-1} qui dépend lui-même de ses prédécesseurs. De ce fait, les deux encodeurs parallèles ont une gestion commune du débit.

Le délai d'une trame du QP est causé par les relations de dépendances qui existent entre deux trames voisines. Par exemple, l'encodage de la trame F_1 dépend seulement de l'*étape pré-encodage des tranches* et des tranches de la trame F_0. L'encodage parallèle de deux trames serait impossible dans un contexte où F_1 dépendrait entièrement de F_0. De ce fait, la valeur de QP_1, qui est calculée à la fin de la trame F_0 par le contrôle de débit de l'encodeur 1, sera indisponible lors de l'encodage des tranches de la trame F_1. Cependant, cette valeur sera disponible et utilisée à la fin du traitement de la trame F_1 pour calculer la valeur de QP_2. De plus, elle sera utilisée lors de l'encodage des tranches de la trame F_2.

5.2.6 Estimation de mouvement et interpolation

Comme nous l'avons mentionné au chapitre 2, l'estimation de mouvement par bloc est une technique utilisée pour réduire les redondances temporelles qui existent entre des trames rapprochées dans le temps. Nous avons aussi vu que cette technique consiste à rechercher, dans une fenêtre de recherche limitée, pour chacun des blocs de la trame courante, le bloc de la trame de référence le plus similaire. Enfin, nous avons mentionné que le standard de compression H.264 a été conçu pour supporter des estimations de mouvement précises au demi et au quart de pixel grâce à l'interpolation de l'image.

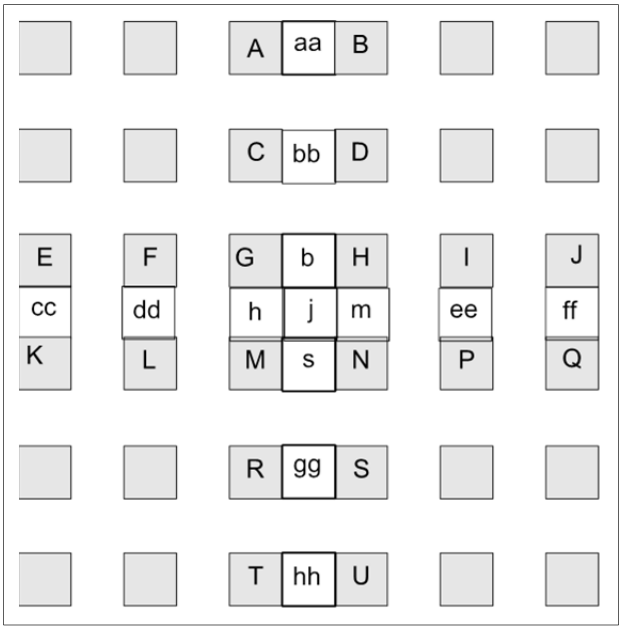


Figure 5.6 Interpolation au demi-pixel.
Tirée de (Richardson, 2003)

Pour l'interpolation au demi-pixel, chaque demi-pixel (pixel à interpoler) adjacent à deux pixels entiers (pixels originaux) est interpolé à l'aide d'un filtre à réponse impulsionnelle finie utilisant six échantillons à l'entrée avec des poids de 1/32, -5/32, 5/8, 5/8, -5/32, 1/32. Par exemple, le pixel *b* sur la figure 5.5 est interpolé de la manière suivante :

$$b = Arrondir((E - 5F + 20G + 20H - 5I + J)/32) \tag{5.1}$$

Une fois l'interpolation de l'ensemble de ces demi-pixels (ceux adjacents à deux pixels entiers) terminée, les pixels restants sont interpolés ou bien verticalement, ou bien horizontalement, puisque les deux filtres donnent le même résultat. Par exemple, *j* peut être interpolé horizontalement avec les valeurs de *cc*, *dd*, *h*, *m*, *ee*, et *ff*; ou verticalement avec les valeurs de *aa*, *bb*, *b*, *s*, *gg*, et *hh*.

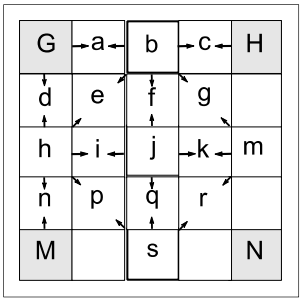


Figure 5.7 Interpolation au quart de pixel.

Par la suite, les échantillons générés par l'interpolation au demi-pixel sont utilisés pour effectuer l'interpolation au quart de pixel. Ce dernier type d'interpolation consiste à appliquer un filtre moyenneur utilisant 2 échantillons pour générer les quarts de pixels. La figure 5.7 montre que ce filtre est appliqué verticalement, horizontalement, ou en diagonale, selon la localisation des demi-pixels voisins.

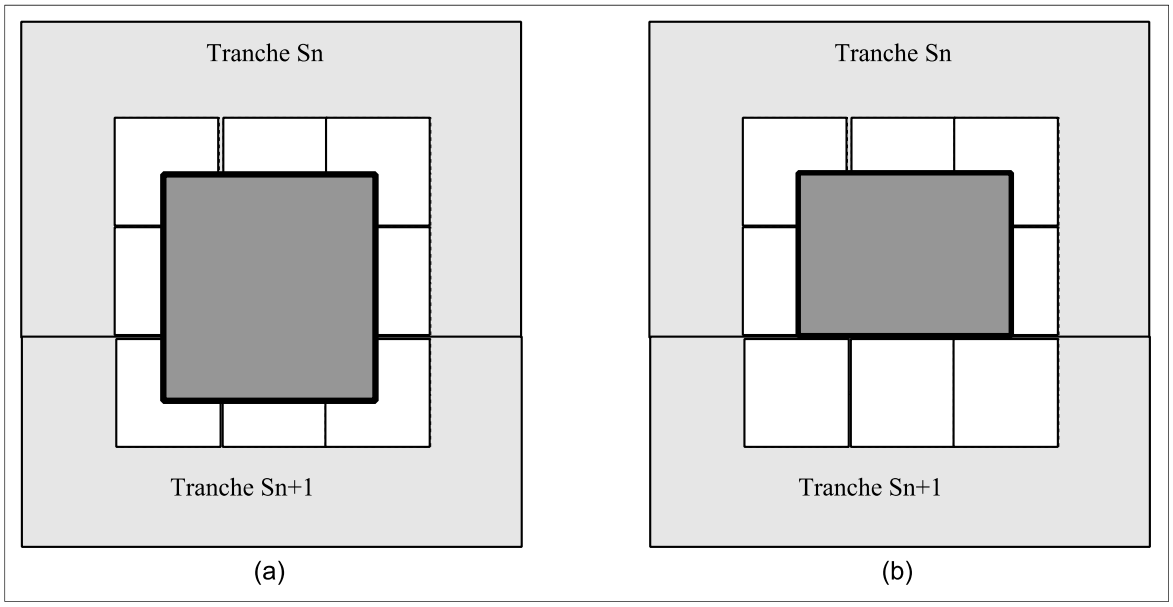


Figure 5.8 Fenêtre de recherche utilisée pour un macrobloc situé à la frontière d'une tranche Sn, où (a) l'encodeur a accès à la tranche voisine inférieure et (b) l'encodeur n'a pas accès à cette tranche.

L'encodeur H.264 d'Intel estime le mouvement en deux étapes. À la première étape, l'encodeur effectue, pour un macrobloc donné, une estimation de mouvement uniquement sur les pixels entiers de l'image. À la seconde étape, le vecteur de mouvement estimé à la

première étape est raffiné, dans une fenêtre de recherche de sept quarts de pixels par sept quarts de pixels, pour atteindre une précision au quart de pixel. Comme nous l'avons vu dans l'introduction de ce chapitre, notre approche inclut des modes restreignant l'estimation de mouvement à l'intérieur d'une tranche. Dans un tel cas, la fenêtre de recherche utilisée à la première étape de l'estimation de mouvement est tronquée verticalement, si nécessaire, aux frontières de la tranche courante (voir figure 5.8 (b)). Quant à la fenêtre de recherche de la méthode de raffinage au quart de pixel, celle-ci est tronquée verticalement aux frontières de la tranche courante moins deux rangées de pixels. Ces rangées représentent la zone d'une tranche qui a besoin d'accéder aux pixels de la tranche voisine, qui n'est pas accessible, pour effectuer une interpolation au demi et au quart de pixels. La figure 5.9 illustre, à l'aide d'une zone hachurée, les pixels des macroblocs situés à la frontière inférieure de la tranche S_n,. Ces pixels ne peuvent pas être interpolés ni utilisés lors du raffinage au quart de pixels.

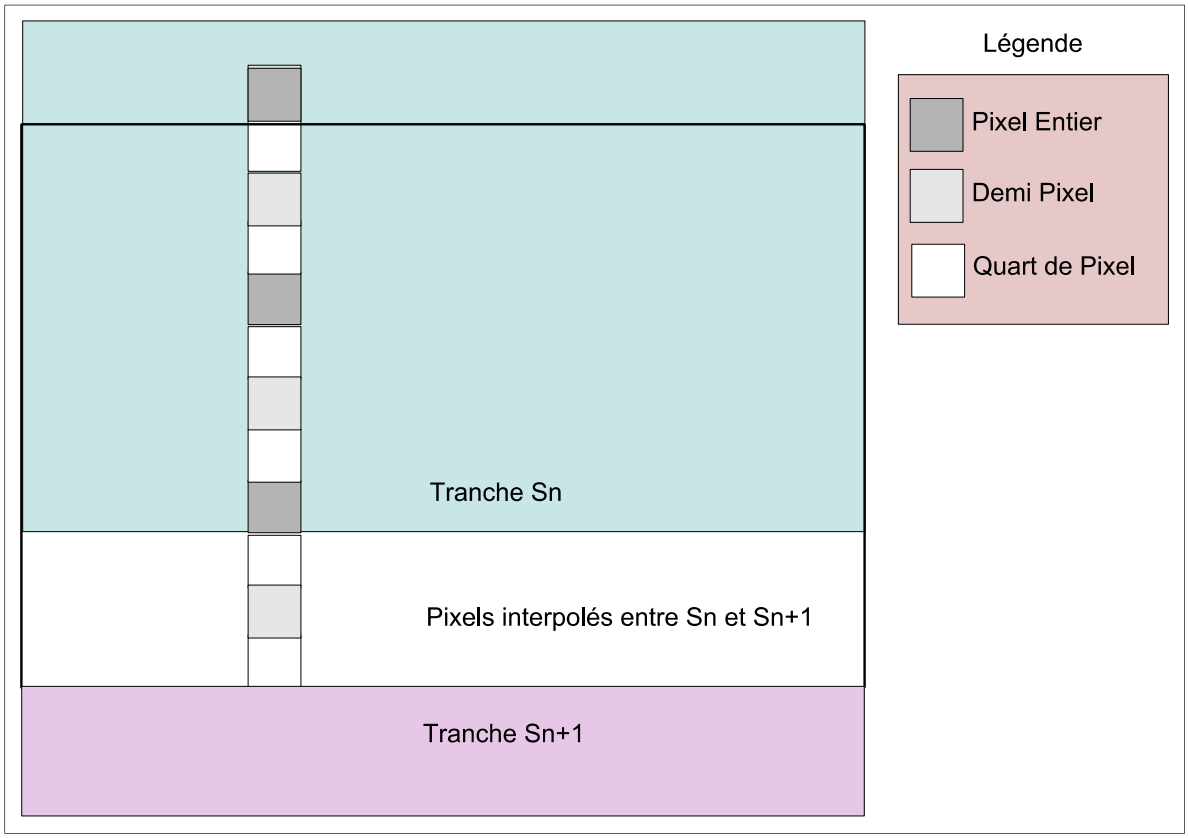


Figure 5.9 Région de recherche désactivée pour la tranche Sn durant le raffinage au quart de pixel du vecteur de mouvement trouvé pendant l'estimation de mouvement entière.

5.2.7 Utilisation de tranches supplémentaires

Nous avons vu précédemment que l'approche d'Intel utilise un nombre de tranches égal au nombre de fils spécifié par l'utilisateur à l'encodeur. Dans un contexte où ce nombre de fils est égal ou inférieur au nombre d'unités d'exécution disponibles sur le système, cette configuration produit généralement les meilleurs résultats atteignables, en terme d'efficience (l'accélération sur le nombre de cœurs utilisé), pour cette approche, puisque l'utilisation d'un nombre de tranches qui est supérieur au nombre de fils utilisés a tendance à nuire au balancement de charges (plus particulièrement quand le nombre de tranches n'est pas un multiple du nombre de fils utilisés) et, par extension, à augmenter le temps d'inoccupation global des fils. De plus, quelle que soit l'approche utilisée, l'utilisation de tranches supplémentaires, qui se définit comme la différence entre le nombre de tranches par trame et le nombre de fils utilisé pour encoder les tranches, augmente la perte de qualité (voir la discussion sur l'impact des tranches sur la qualité à la section 4.3.1).

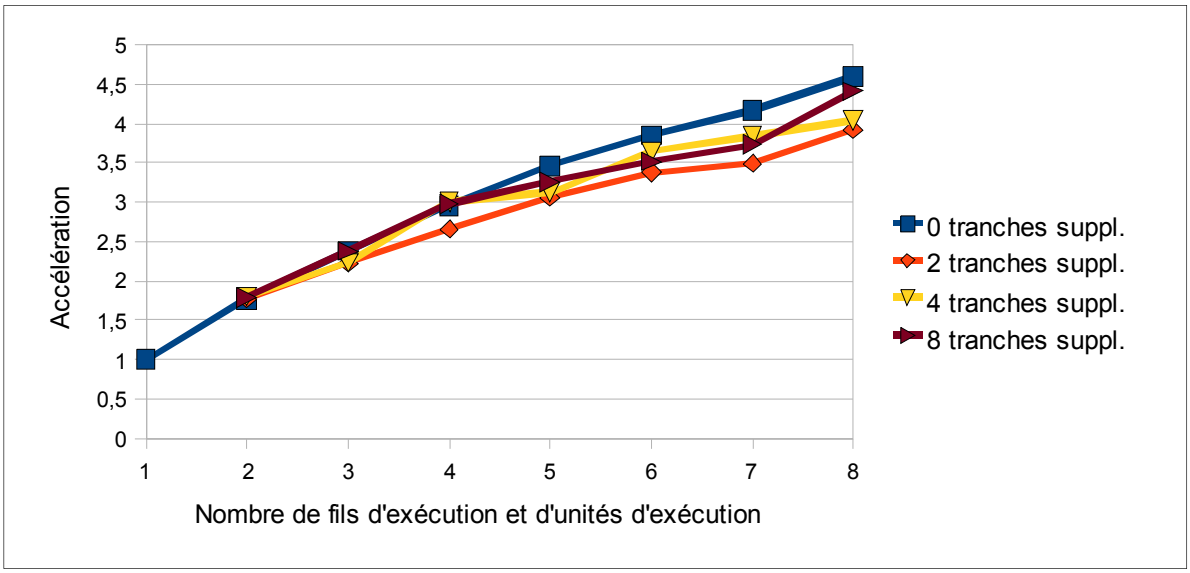


Figure 5.10 Accélération moyenne en fonction du nombre de tranches supplémentaires et du nombre de fils (cœurs) utilisés par l'approche d'Intel pour un débit de 15 Mbit/s.

La figure 5.10 illustre l'accélération moyenne que nous avons obtenue avec l'approche d'Intel pour un nombre de fils allant de 2 à 8, avec 0, 2, 4, 8 tranches supplémentaires, pour les

séquences *train-station*, *horse-cab*, *sunflower*, *waterskiing*, *rally* et *tractor* encodés avec un débit de 15 Mbit/s. Cette figure montre que les meilleurs résultats moyens sont généralement obtenus avec un nombre de tranches égal au nombre de fils. Notons cependant que certaines séquences, notamment, les séquences ayant une distribution de la complexité significativement inégale entre les tranches obtiennent de meilleurs résultats avec l'ajout d'une ou de plusieurs tranches supplémentaires.

Dans notre approche, l'utilisation d'un nombre de tranches égal au nombre de fils produit des résultats sous-optimaux, puisque cette configuration ne permet pas toujours de remplir suffisamment la file d'attente des tranches à encoder pour tenir occupé l'ensemble des fils. La figure 5.11 montre deux exemples de blocage dans un contexte où nous utilisons quatre fils d'exécution et quatre tranches par trame. L'exemple (a) illustre un cas de blocage pouvant se produire à l'exécution du premier ou du second mode proposés (voir section 5.3). Dans cet exemple, le fil 4, qui a complété l'encodage de la première tranche de la trame F_{n+1} n'est pas en mesure de démarrer l'encodage de la dernière tranche de la même trame, puisque le fil 1 traite toujours cette tranche pour la trame F_n. L'exemple (b) illustre un cas de blocage pouvant avoir lieu à l'exécution du troisième mode. Dans cet exemple, les fils 3 et 4 sont bloqués puisque la première et la troisième tranche de la trame F_{n+1} ont besoin d'accéder respectivement à la seconde et à la dernière tranche de la trame F_n, qui, elles, sont toujours en cours de traitement par les fils 1 et 2. L'ajout de tranches dans notre approche pourrait permettre de réduire ce nombre de cas de blocage, puisque la file d'attente des tranches à encoder a généralement plus de tranches prêtes à être encodées. La section 5.4 présente de façon plus détaillée l'impact de l'ajout des tranches sur les performances de notre approche.

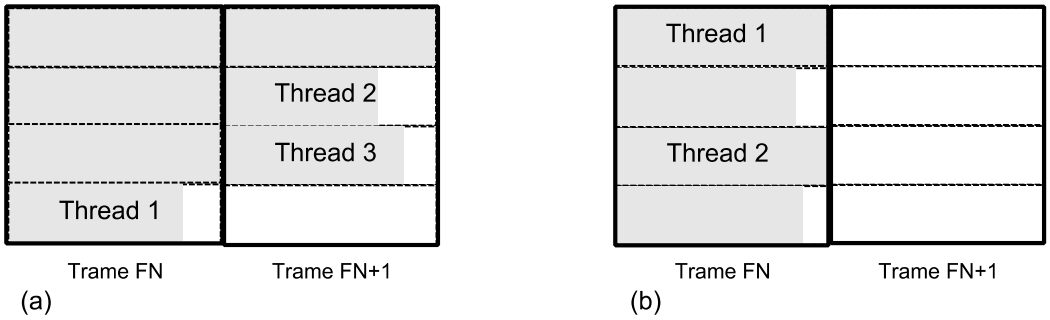


Figure 5.11 Deux exemples de situations provoquant le blocage de fils
(a) exemple de blocage pour les deux premiers modes
(b) exemple de blocage pour le troisième mode.

5.3 Modes d'estimation de mouvement

Notre implémentation est constituée de trois modes d'encodage parallèle de type MTMT. Le premier mode restreint la recherche des vecteurs de mouvements à l'intérieur des limites de la tranche traitée; c'est-à-dire qu'une tranche S_m d'une trame F_n peut débuter son traitement aussitôt que la tranche S_m de la trame F_{n-1} a terminé le sien. Le second mode permet lui aussi à la tranche S_m d'une trame F_n de démarrer son traitement une fois le traitement de la tranche S_m de la trame F_{n-1} terminé. Cependant, ce mode élargit la recherche des vecteurs de mouvement aux tranches voisines de S_m qui ont déjà été encodées dans la trame F_{n-1}. Enfin, le troisième mode traite une tranche S_m d'une trame F_n seulement après que la trame F_{n-1} a traité les tranches voisines qui font partie de la région de recherche de la tranche S_m.

Globalement, le premier mode obtient la meilleure accélération au prix d'une perte de qualité relativement élevée. Le second mode offre une accélération plus faible que le premier mode, mais produit une perte de qualité significativement plus faible. Enfin, le dernier mode produit une perte de qualité inférieure aux deux autres modes, mais au prix d'une accélération encore plus faible. Il est important de noter que ces modes modifient le comportement, et par le fait même les performances, de l'application de référence (l'application qui servira à calculer l'accélération de notre approche). De ce fait, il est possible d'obtenir une accélération supérieure à l'accélération théorique pour les deux premiers modes. En effet, ces deux modes

restreignent l'estimation de mouvement et, par conséquent, réduisent le nombre de calculs nécessaires à cette tâche.

5.3.1 Le mode restreint aux frontières de la tranche courante

Nous avons développé le premier mode d'estimation de mouvement, le mode restreint aux frontières de la tranche courante, dans l'optique de maximiser l'accélération au prix d'une dégradation de qualité présente aux frontières des tranches. Quelles que soient les tranches de référence disponibles, ce mode restreint la fenêtre de recherche de l'estimation de mouvement à l'intérieur de la tranche courante. De plus, ce mode limite le raffinage pour l'estimation de mouvement au demi et au quart de pixel, tel que vu précédemment. Ces deux restrictions empêchent l'encodeur de suivre des mouvements qui ont lieu à la jonction de deux tranches, et empêchent le raffinement du vecteur de mouvement au demi et au quart de pixel quand le vecteur de mouvement trouvé à l'estimation entière se situe à la frontière ou a un pixel de la frontière d'une tranche. Cette estimation et ce raffinage incomplets génèrent des macroblocs de moindre qualité aux frontières des tranches. Le filtre de déblocage parallèle, qui n'est pas appliqué aux macroblocs situés aux frontières d'une tranche, mais qui est appliqué aux autres macroblocs, augmente davantage cette différence de qualité. De même, un faible débit occasionne une dégradation aux frontières, par rapport à la dégradation des autres macroblocs, plus importante que celle produite par un débit élevé.

La figure 5.12 illustre les performances de ce mode pour des encodages parallèles configurés avec un débit de 10 Mbit/s, un nombre de tranches supplémentaires allant de 0 à 4, et un nombre de cœurs allant de 2 à 8. Cette figure montre que l'ajout de tranches pour ce mode a tendance à augmenter l'accélération et la perte de qualité. Lorsque nous utilisons quatre tranches supplémentaires, l'accélération augmente d'une manière presque linéaire, par exemple, elle se situe aux alentours de 8 lorsque nous utilisons huit cœurs. Néanmoins, pour cette configuration, ce mode atteint une perte de qualité moyenne d'environ 2 dB. Cette perte est beaucoup plus élevée que la perte moyenne, qui est de 0.08 dB, obtenue par l'approche d'Intel. Nous obtenons des comportements similaires avec les autres débits testés : 1, 5, 15 et

20 Mbit/s (voir annexe XI). Typiquement, une réduction du débit provoque une diminution de l'accélération et une augmentation de la perte de qualité, et vice-versa. Ainsi, la méthode proposée est plus performante à haut débit, tant en accélération qu'en qualité.

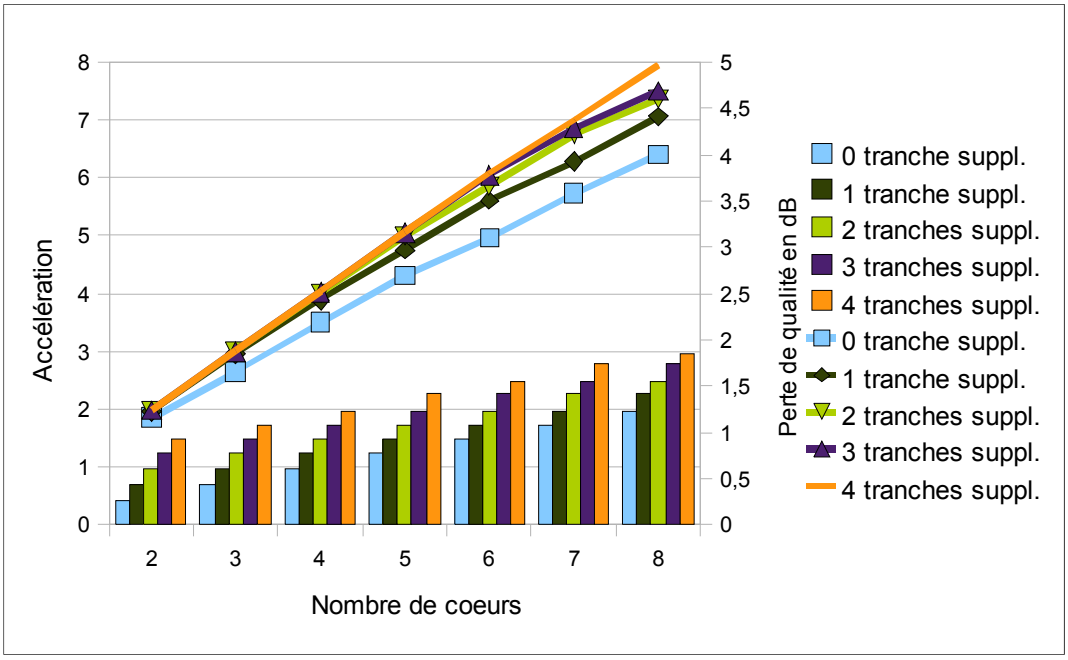


Figure 5.12 Influence du nombre de tranches sur les performances du mode 1 MTMT pour un débit de 10 Mbit/s.

5.3.2 Le mode ouvert aux tranches disponibles

Après avoir constaté que la perte de qualité produite par le premier mode était très élevée, nous avons développé le mode ouvert aux tranches disponibles. Ce mode a pour objectif de réduire la perte de qualité du mode précédent en élargissant la fenêtre de l'estimation de mouvement aux tranches voisines quand celles-ci sont disponibles dans la trame de référence. Ce mode a pour avantage de réduire les artefacts situés aux frontières des tranches, mais a l'inconvénient de produire des résultats non-déterministes, puisque les tranches disponibles dépendent de l'exécution de l'application. De plus, ce mode a le désavantage d'obtenir une accélération moyenne significativement plus faible que celle obtenue par le premier mode, car l'augmentation de l'étendue de la fenêtre de l'estimation augmente la complexité de l'estimation.

La figure 5.13 illustre les performances de ce deuxième mode pour des encodages parallèles configurés avec un débit de 10 Mbit/s, un nombre de tranches supplémentaires allant de 0 à 4, et un nombre de cœurs allant de 2 à 8. Nous constatons sur la figure que l'accélération est significativement plus faible que celle du mode précédent, et que l'ajout de tranches apporte un gain en accélération plus petit que le gain du mode précédent. De plus, nous observons que l'ajout de tranches dans ce mode n'augmente pas toujours la perte de qualité. Tout comme le premier mode, la réduction du débit diminue l'accélération et augmente la perte de qualité, et vice-versa (voir l'annexe XII).

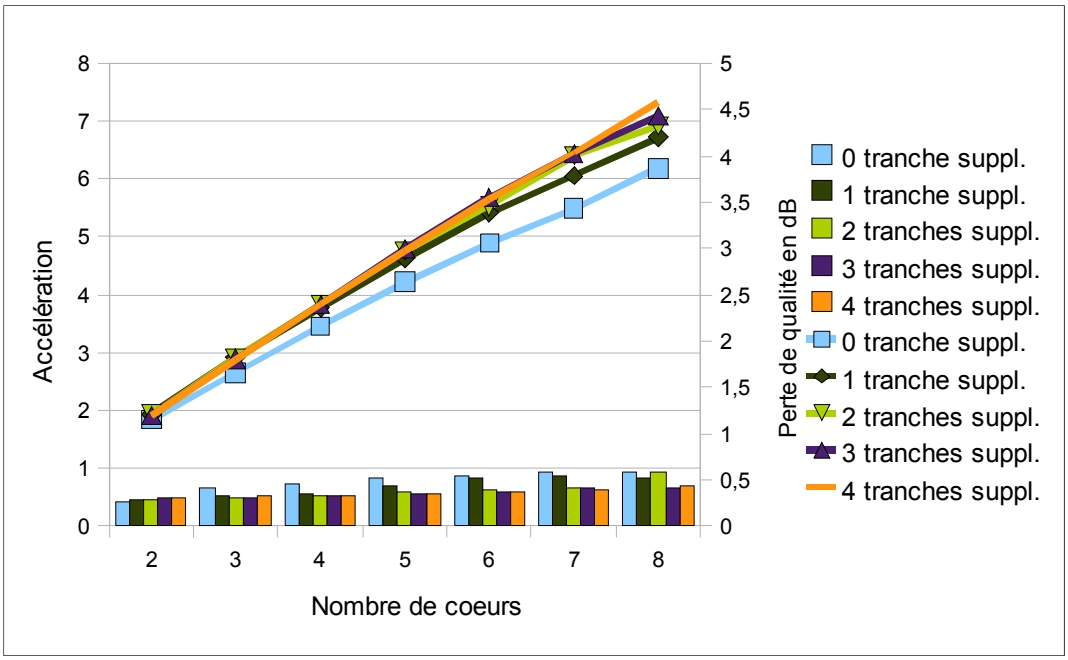


Figure 5.13 Influence du nombre de tranches sur les performances du mode 2 MTMT pour un débit de 10 Mbit/s.

5.3.3 Le mode complet avec blocage

Nous avons développé le dernier mode, le mode complet avec blocage, avec l'objectif de réduire la perte de qualité par rapport aux deux modes précédents. Ce mode a pour avantage d'atteindre une qualité très proche de celle obtenue par l'approche originale d'Intel, avec une

accélération légèrement plus faible que celle du second mode. Par rapport à ce dernier mode, le mode complet a aussi l'avantage d'être déterministe, puisqu'il attend toujours la disponibilité de l'ensemble des tranches nécessaires à l'estimation de mouvement pour une tranche donnée.

La figure 5.14 illustre les performances de ce troisième mode pour des encodages parallèles configurés avec un débit de 10 Mbit/s, un nombre de tranches supplémentaires allant de 0 à 4, et un nombre de cœurs allant de 2 à 8. Nous constatons sur la figure que l'accélération est similaire, légèrement inférieure, à celle du mode précédent. De plus, nous observons que l'ajout de tranches dans ce mode augmente graduellement la perte de qualité. Tout comme les deux modes précédents, la réduction du débit diminue l'accélération et augmente la perte de qualité, et vice-versa (voir l'annexe XIII).

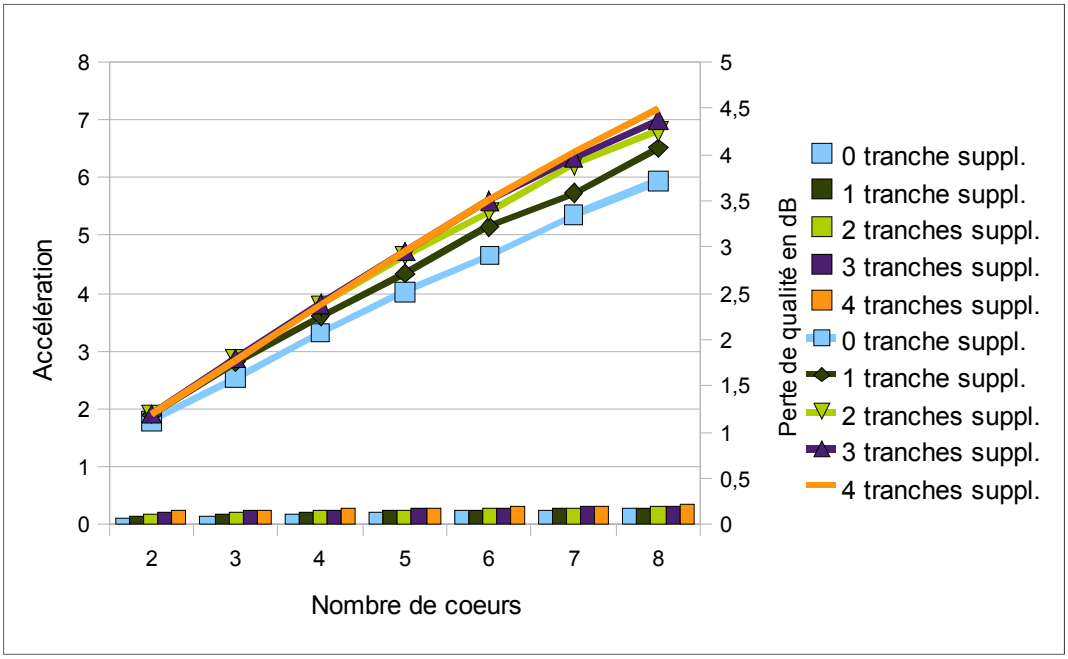


Figure 5.14 Influence du nombre de tranches sur les performances du mode 3 MTMT pour un débit de 10 Mbit/s.

5.4 Analyse des résultats

Dans un premier temps, nous comparons dans cette section les accélérations et les pertes de qualité que nous avons obtenues pour les trois modes MTMT que nous avons conçus et simulés. Dans un second temps, nous comparons nos résultats avec les résultats de l'approche d'Intel original et l'approche d'Intel que nous avons modifiée pour paralléliser le filtre de déblocage.

5.4.1 Comparaison entre les trois modes proposés

Les figures 5.15 et 5.16 illustrent les accélérations, sous forme de lignes, et les pertes de qualité, sous forme de barres, que nous avons obtenues pour nos trois modes MTMT avec un nombre de tranches supplémentaires égal à zéro et égal à quatre. Ces deux graphiques contiennent les résultats obtenus pour un débit de 10 Mbit/s et pour de 2 à 8 cœurs. L'annexe XIV contient les résultats pour les débits de 1, 5, 15 et 20 Mbit/s. Nous remarquons sur ces deux figures que l'ajout de cœurs augmente l'accélération et la perte de qualité d'une manière relativement approximativement linéaire. Nous observons aussi, tel que discuté précédemment, que le premier mode obtient une perte de qualité beaucoup plus grande que les deux autres modes.

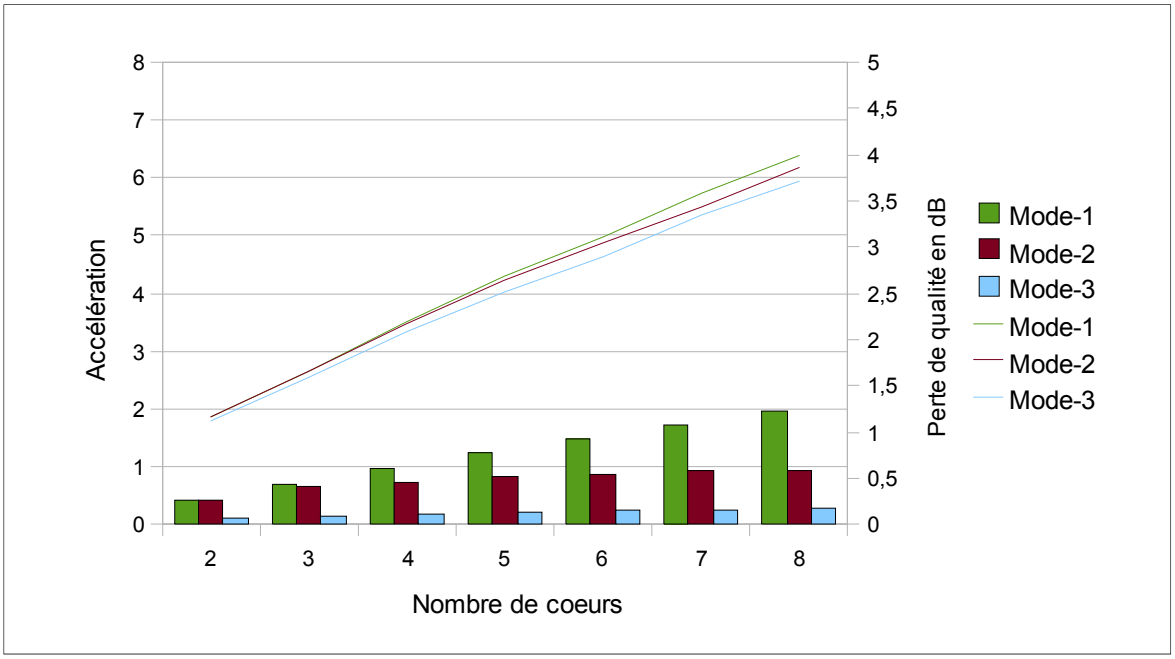


Figure 5.15 Comparaison entre nos trois modes MTMT pour un débit de 10 Mbit/s et zéro tranche supplémentaire.

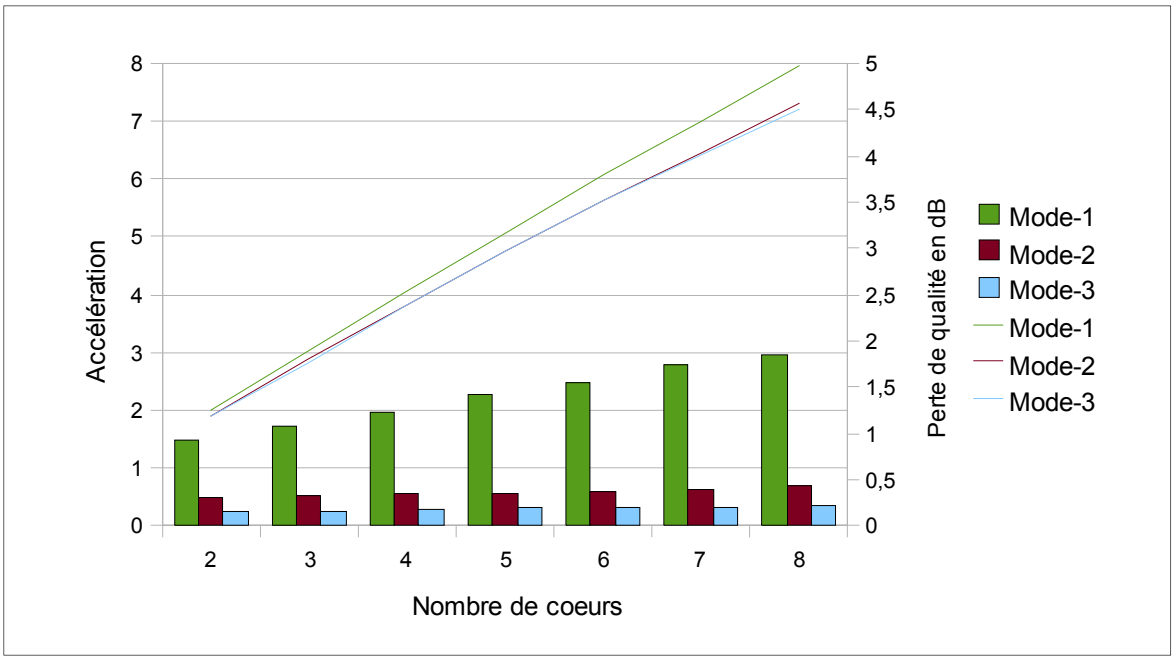


Figure 5.16 Comparaison entre nos trois modes MTMT pour un débit de 10 Mbit/s et quatre tranches supplémentaires.

Par ailleurs, nous observons que l'exécution sans tranche supplémentaire produit des accélérations (celles affichées à la figure 5.15) nettement inférieures aux accélérations obtenues avec l'utilisation de quatre tranches supplémentaires (celles affichées à la figure 5.16). Nous constatons aussi sur la figure 5.16 que l'accélération du second mode est légèrement supérieure à l'accélération du troisième mode pour une perte de qualité qui est environ deux fois plus grande que la perte du troisième mode.

Sur la base de ces observations, nous proposons quatre recommandations : (1) préférer l'utilisation de quatre tranches supplémentaires à l'utilisation de zéro tranche supplémentaire; (2) ne jamais exploiter le deuxième mode, puisque la perte de qualité est beaucoup trop élevé pour un gain d'accélération peu significatif par rapport à celui du troisième mode; (3) utiliser le premier mode pour augmenter l'accélération de l'encodage sans tenir compte de la perte de qualité; (4) utiliser le troisième mode pour augmenter l'accélération sans provoquer une perte de qualité importante. Nous croyons que pour la plupart des applications, le troisième mode sera à privilégier puisque le gain supplémentaire en vitesse offert par le mode 1 n'est pas assez significatif pour justifier son aussi grande perte de qualité.

5.4.2 Comparaison avec l'approche d'Intel

La figure 5.17 montre une comparaison entre l'approche originale d'Intel, l'approche d'Intel avec filtre de déblocage parallèle, le premier mode de notre approche avec respectivement zéro et quatre tranches supplémentaires, et le troisième mode de notre approche avec quatre tranches supplémentaires. Nous remarquons que les deux premières approches présentent les plus faibles pertes en qualité et les plus faibles accélérations. Nous observons aussi que le premier mode produit une perte de qualité largement supérieure à la perte obtenue par les deux autres modes. Enfin, nous constatons que le troisième mode avec quatre tranches supplémentaires offre une excellente accélération, approximativement égale à 7.2, par rapport à 5.5 pour l'approche d'Intel avec filtre parallèle, pour un encodage parallèle utilisant huit cœurs, pour une perte de qualité relativement basse.

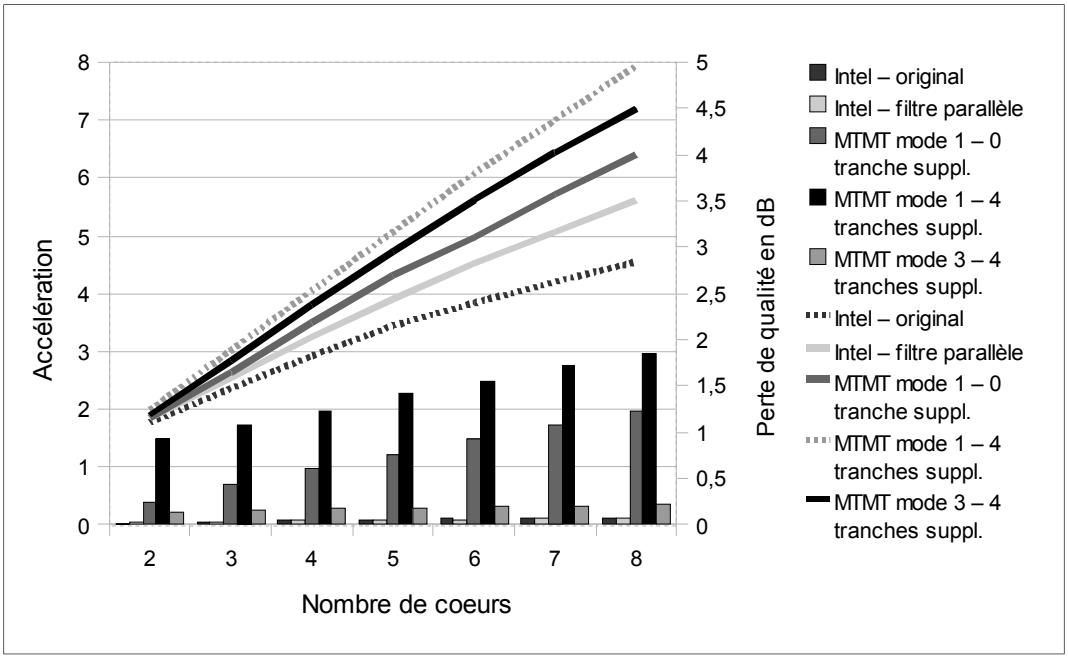


Figure 5.17 Comparaison de performance entre l'approche d'Intel et notre approche MTMT pour un débit de 10 Mbit/s.

5.5 Discussion

Nos simulations expérimentales montrent que le troisième mode de notre approche offre un meilleur compromis accélération/perte de qualité que l'approche d'Intel. Par conséquent, ce mode de notre approche pourrait remplacer l'approche utilisée par Intel dans son encodeur. De plus, notre approche offre une plus grande flexibilité sur le choix des tranches, puisque l'ajout de tranches augmente généralement l'accélération de l'encodage. Quant au premier mode, celui-ci pourrait être utilisé dans certains contextes où l'accélération prévaut sur la perte de qualité.

Notons cependant que notre implémentation actuelle comprend plusieurs limites. D'abord, notre implémentation ne supporte pas les trames de type B. Ce type de trame exige une synchronisation différente au niveau des fils que les trames de types I ou P. Nous prévoyons cependant que l'ajout de trames B permettrait d'accélérer davantage notre application. En effet, les tranches des trames B pourraient être placées dans un tampon secondaire de tranches à encoder. La *Section Multi-Tranches* pourrait alors accéder à ce tampon lorsque le

tampon primaire (celui qui contient les tranches I et P) est vide ou lorsque l'encodage de ces tranches deviennent prioritaires (avant que leur délai d'encodage soit expiré) de manière à réduire le temps d'inoccupation des fils. Ensuite, notre implémentation ne supporte pas le ré-encodage d'une trame qui, par exemple, a consommé trop de bits lors d'un premier encodage. Il serait important pour une implémentation commerciale d'ajouter le support du ré-encodage afin d'éviter les cas de surpassement et de soupassement de la mémoire tampon alloué aux trames compressées.

Nous rappelons aussi que notre approche insère un délai d'une trame dans le contrôle de débit. Ce délai peut contribuer à augmenter les cas de ré-encodage, puisque le contrôle de débit n'est plus en mesure de suivre convenablement les changements de complexité ayant lieu d'une trame à l'autre. Par exemple, si la complexité d'une trame n augmente significativement par rapport à la trame précédente, l'encodage de la trame $n+1$ ne s'ajustera pas à ce changement et aura, par conséquent, tendance à conserver une qualité trop élevée par rapport au débit réellement disponible. Pour contourner ce problème, il serait envisageable d'utiliser un contrôle de débit précis au niveau des tranches. Ce type de stratégie permettrait d'éliminer le délai en ajustant le QP en fonction de l'ensemble des tranches qui ont été encodés. Cependant, cette approche aura pour conséquence de produire des résultats non-déterminisme, puisque les tranches peuvent être traitées, donc influencer le débit, dans un ordre différent d'une exécution à l'autre.

Par ailleurs, le partitionnement d'une trame en tranche, effectué au niveau des lignes de macroblocs, de notre approche a l'inconvénient de réduire le nombre maximal de cœurs qui peuvent être utilisés pour effectuer un encodage parallèle. Par exemple, notre approche offre un maximum de neuf cœurs pour une séquence QCIF (176×144 pixels ou 11×9 macroblocs). Pour améliorer la mise à l'échelle de notre approche, il serait possible d'effectuer, tout comme l'approche d'Intel, un partitionnement au niveau des macroblocs. Cependant, ce partitionnement aura pour conséquence de rendre l'estimation de mouvement moins efficace pour les modes 1 et 2 puisque ces derniers modes devront travailler avec certaines régions de recherche non-rectangulaires.

En ce qui concerne les performances de notre approche, nous prévoyons obtenir, pour un même nombre de cœurs, des accélérations inférieures avec des résolutions plus petites. En effet, nous croyons que les séquences de plus faibles résolutions augmentent la quantité de code séquentiel, génèrent, en relatif, une charge de gestion des fils plus élevée que celle générée par l'encodage séquences de haute-résolutions, en plus de produire davantage de tranches de taille inégale. Aussi, l'impact de l'ajout de tranches supplémentaires de notre approche dépend du nombre de cœurs utilisés. Empiriquement, nos résultats indiquent que le nombre de tranches supplémentaires doit être environ égal à la moitié du nombre de cœurs utilisés. Ainsi, nous recommandons d'utiliser deux tranches supplémentaires pour une exécution parallèle avec quatre cœurs, quatre tranches supplémentaires pour une exécution parallèle avec huit cœurs, et ainsi de suite. Nous rappelons cependant, que l'ajout de tranches augmente aussi la perte de qualité visuelle.

Enfin, nous avons comparé l'accélération obtenue avec de notre approche avec l'approche MTMT de Steven *et al.* (Steven, Xinmin et Yen-Kuang, 2003) et avec l'approche onde de choc de Chen *et al.* (Chen *et al.*, 2006) (voir les section 3.4.1 et 3.4.2). Les auteurs de ces deux études ont testé leur implémentation avec des séquences CIF sur des systèmes multiprocesseur Xeon d'Intel composés de quatre processeurs. Steven *et al.* ont obtenu une accélération de 3.95 et Chen *et al.* une accélération de 3.80. À titre de comparaison, nous obtenons des accélérations moyennes de 4.02 pour le mode 1, 3.78 pour le mode 2 et 3.77 pour le mode 3 pour l'encodage de séquences vidéo de 1280 × 720 pixels encodées avec un débit de 5 Mbit/s et quatre cœurs. Nous rappelons, tel que discuté à la section 5.3, que l'emploi des deux premiers modes réduit la complexité de l'estimation de mouvement par rapport à l'application de référence. De ce fait, il est normal que l'accélération du mode 1 soit supérieure à l'accélération théorique maximale, qui elle est égale à 4. Par rapport à l'approche de Steven *et al.,* notre approche a l'avantage de ne pas nécessiter l'emploi de trames B pour être fonctionnelle. Aussi, notre approche obtient des accélérations comparables à celles de l'approche de Chen *et al.*, pour des pertes de qualité peu significative. Par rapport à cette dernière approche, nous avons l'avantage d'offrir une

approche qui réutilise l'architecture de base de l'approche parallèle d'Intel, soit une architecture parallèle multi-tranches. Cependant, notons que Steven *et al.* ainsi que Chen *et al.* utilisent l'implémentation de référence de H.264 (sans spécifier la version exacte). Par conséquent, notre méthode est difficilement comparable aux méthodes de ces deux groupes d'auteurs.

5.6 Conclusion

Dans ce chapitre nous avons décrit les aspects les plus importants de notre approche d'encodage parallèle MTMT. Nous avons présenté les trois modes d'estimation de mouvement que nous avons conçus spécifiquement pour cette approche. Le premier mode limite la région de recherche d'un vecteur de mouvement à l'intérieur de la tranche courante. Le second mode étend la région de recherche d'un vecteur de mouvement aux tranches voisines disponibles seulement. Le troisième mode attend la disponibilité des tranches voisines avant de démarrer l'encodage d'une tranche. Nos simulations expérimentales ont montré que notre approche, quel que soit le mode, obtient une meilleure accélération que l'approche d'Intel et l'approche d'Intel dont nous avons parallélisé le filtre de déblocage. Notre premier mode obtient les meilleures accélérations, mais produit des pertes de qualité visuelle excessives. Le troisième mode affecte peu la qualité tout en offrant une très bonne accélération. Enfin, le deuxième mode offre une accélération légèrement meilleure que celle du troisième mode mais au prix d'une perte de qualité beaucoup plus grande. Notons que le troisième mode offre une accélération moyenne environ 25% plus élevée, pour huit fils d'exécution, que l'accélération obtenue avec l'approche d'Intel avec filtre parallèle, et 55% plus élevée que l'approche d'Intel de base.

CONCLUSION

Le travail que nous avons présenté dans le cadre de ce mémoire a été divisé en quatre parties. Dans la première partie nous avons présenté, en deux chapitres distinctifs, une introduction au traitement parallèle et une introduction au standard de codage vidéo H.264. Nous avons notamment montré que la parallélisation d'un encodeur vidéo peut s'effectuer au niveau des données, on parle alors d'une décomposition du domaine, ou encore au niveau des fonctions, on parle alors d'une décomposition fonctionnelle. Nous avons décrit la structure hiérarchique d'une séquence vidéo qui est composée de groupes d'images, de trames, de tranches, de macroblocs et de blocs. Nous avons aussi présenté le schéma d'encodage de H.264.

Dans la seconde partie du travail, nous avons présenté l'état de l'art de l'encodage vidéo parallèle en mettant l'accent sur les approches parallèles développées pour des systèmes à mémoire partagée. Nous avons montré que les approches basées sur une décomposition fonctionnelle étaient surtout utilisées par des processeurs spécialisés et que les approches basées sur une décomposition du domaine étaient davantage utilisées par les processeurs généraux, puisque ce type d'approche s'adapte particulièrement bien à des architectures ayant un nombre d'unités d'exécution différents.

Dans la troisième partie, nous avons analysé l'approche parallèle d'Intel, une approche au niveau des tranches. Nous avons montré que cette approche obtenait une accélération sous-optimales pour trois raisons principales : d'abord, parce qu'une partie significative du code n'est pas parallèle, ensuite, parce que la distribution de la charge de travail entre les fils d'exécution est inégale, et finalement, parce l'ajout de tranches augmente le temps d'encodage. Pour améliorer les performances de cette approche, nous avons proposé de paralléliser l'application du filtre de déblocage. Cette contribution a permis d'améliorer l'accélération de l'approche d'Intel au prix d'une perte de qualité non-significative.

Dans une quatrième partie, nous avons présenté notre approche parallèle MTMT et ses trois modes d'estimation de mouvement qui offrent un compromis entre l'accélération et la perte

de qualité. Le premier mode, le plus rapide, mais le moins efficace, restreint la région de recherche de l'estimation de mouvement à l'intérieur des limites de la tranche courante. Le second mode, moins rapide, mais offrant une meilleure qualité que le premier, élargit la région de recherche aux tranches voisines, quand les tranches de référence y correspondant ont été traitées. Le troisième mode, moins rapide que le second, mais offrant une meilleure qualité, rend une tranche prête à l'encodage seulement quand les tranches de référence couvrant la région de recherche ont été traitées. Nous avons montré que le second mode est le moins intéressant des trois puisqu'il n'obtient pas une accélération significativement supérieure au troisième mode, malgré sa perte de qualité élevée, et que son accélération est nettement trop petite par rapport au premier mode. Nous avons aussi indiqué, que le troisième mode de notre approche offre une accélération moyenne environ 25% plus élevée, pour huit fils d'exécution, que l'accélération obtenue avec l'approche d'Intel avec filtre parallèle, et 55% plus élevée que l'approche d'Intel de base, dans le premier cas, pour une perte de qualité supérieure à l'approche d'Intel, et dans le second cas, pour une perte de qualité supplémentaire non-significative. Nous avons aussi vu que notre approche a l'inconvénient d'ajouter un délai d'une trame au contrôle de débit.

Par rapport aux travaux présentés dans la littérature, notre travail comporte essentiellement trois éléments contributifs importants. Premièrement, dans notre approche MTMT l'encodage d'une tranche appartenant à une trame T_n dépend seulement des tranches de référence de la trame T_{n-1} qui seront utilisées lors de l'estimation de mouvement, alors, qu'à notre connaissance, les autres approches présentées dans la littérature dépendent de la trame de référence complète. Deuxièmement, notre approche permet d'utiliser trois modes lors de l'estimation de mouvement. Comme nous l'avons vu, ces modes offrent un compromis entre l'accélération et la perte de qualité. Troisièmement, nous avons implémenté notre approche à l'intérieur de l'encodeur H.264 livré en code d'exemple avec la librairie *Integrated Performance Pritmitives* (IPP) d'Intel, contrairement aux autres approches présentées dans ce mémoire, qui sont implémentées avec le code de référence d'H.264.

Suite à notre travail, nous recommandons d'ajouter un support aux trames B dans notre approche dans le but d'augmenter davantage l'accélération en réduisant l'inactivité des fils. En effet, les tranches B pourraient être mises dans un tampon secondaire de tranches à être encodées et être encodées seulement quand leur expiration approche ou encore quand le tampon qui contient les tranches I et P est vide. Nous recommandons aussi d'explorer une autre approche parallèle basée sur l'utilisation, pour chaque trame, d'une trame de référence autre que la trame précédente. Par exemple, un fil pourrait encoder les trames paires en utilisant une trame de référence n-2 pour encoder une trame n, un autre fil pourrait encoder parallèlement les trames impaires en utilisant une trame de référence n-1 pour encoder une trame n+1.

ANNEXE I

COMPARAISON ENTRE LA QUALITÉ VISUELLE DES SÉQUENCES VIDÉO EN FONCTION DES DÉBITS UTILISÉS

Tableau-A I-1 Comparaison entre la qualité visuelle des séquences vidéo en fonction des débits utilisés

	PSNR (en dB)				
Séquence	**à 1 Mbit/s**	**à 5 Mbit/s**	**à 10 Mbit/s**	**à 15 Mbit/s**	**à 20 Mbit/s**
Train-station	36,47	41,59	43,32	44,42	45,32
Horse-cab	28,18	35,04	38,59	40,98	42,84
Sun-flower	37,08	43,66	45,59	46,56	47,32
Waterskiing	29,32	36,78	40,64	43,42	45,46
Rally	33,59	40,08	43,45	45,95	47,81
Tractor	30,2	37,35	40,33	42,02	43,18

ANNEXE II

PARAMÈTRES D'ENCODAGE

Les paramètres d'encodage définis dans la fonction H264EncoderParams::H264EncoderParams() sont les suivants :

```
key_frame_controls.method    = H264_KFCM_INTERVAL;
key_frame_controls.interval  = 20;
key_frame_controls.idr_interval = 1;
B_frame_rate = 0;
treat_B_as_reference = 1;
num_ref_frames = 1;
num_ref_to_start_code_B_slice = 1;
num_slices = 0;  // Autoselect
profile_idc = H264_MAIN_PROFILE;
level_idc = 0;  //Autoselect
chroma_format_idc = 1; // YUV 420.
bit_depth_luma = 8;
bit_depth_chroma = 8;
aux_format_idc = 0;
bit_depth_aux = 8;
alpha_incr_flag = 0; alpha_opaque_value = 0;
alpha_transparent_value = 0;
rate_controls.method = H264_RCM_VBR;
rate_controls.quantI = 20;
rate_controls.quantP = 20;
rate_controls.quantB = 20;
mv_search_method = 2;
me_split_mode = 0;
me_search_x = 8; me_search_y = 8;
use_weighted_pred = 0;
use_weighted_bipred = 0;
use_implicit_weighted_bipred = 0;
direct_pred_mode = 0;
use_direct_inference = 1;
deblocking_filter_idc          = 2;
deblocking_filter_alpha        = 2;
deblocking_filter_beta         = 2;
transform_8x8_mode_flag = 1;
use_default_scaling_matrix = 0;
qpprime_y_zero_transform_bypass_flag =0;
entropy_coding_mode = 1; // 1 = CABAC
cabac_init_idc = 1;
coding_type = 0;
m_do_weak_forced_key_frames = false;
write_access_unit_delimiters = 0;
use_transform_for_intra_decision = true;
numFramesToEncode = 0;
m_QualitySpeed = 0;
quant_opt_level = 0;
```

ANNEXE III

PERTE DE QUALITÉ, EN FONCTION DU NOMBRE DE TRANCHE, POUR CHACUNE DES SÉQUENCES TESTÉES

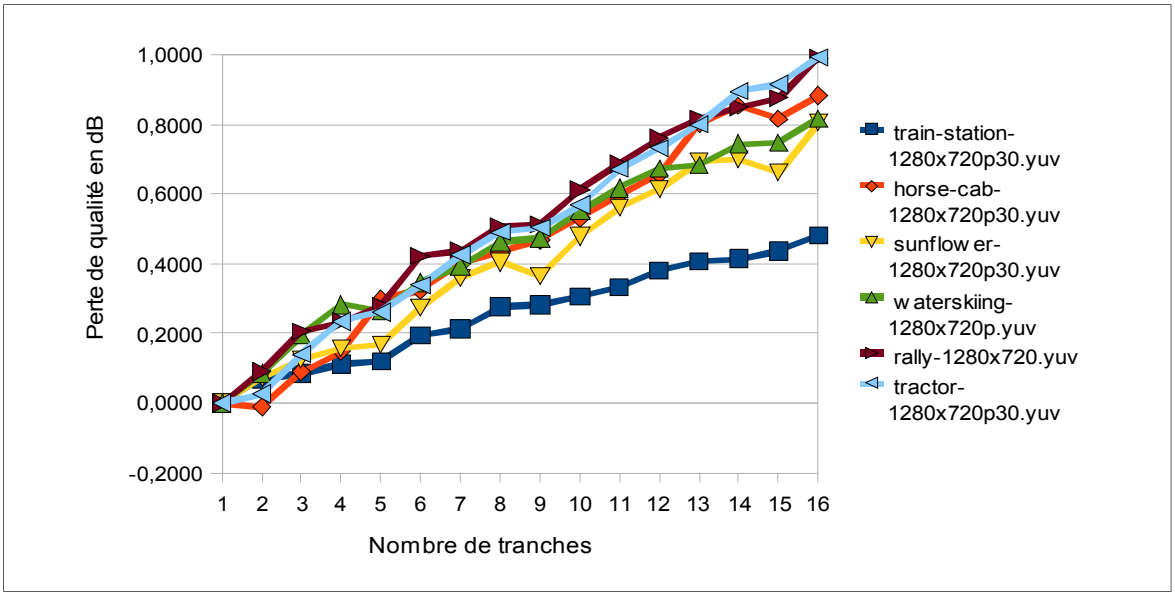


Figure-A III-1 Perte de qualité, en fonction du nombre de tranche, pour un débit de 1 Mbit/s.

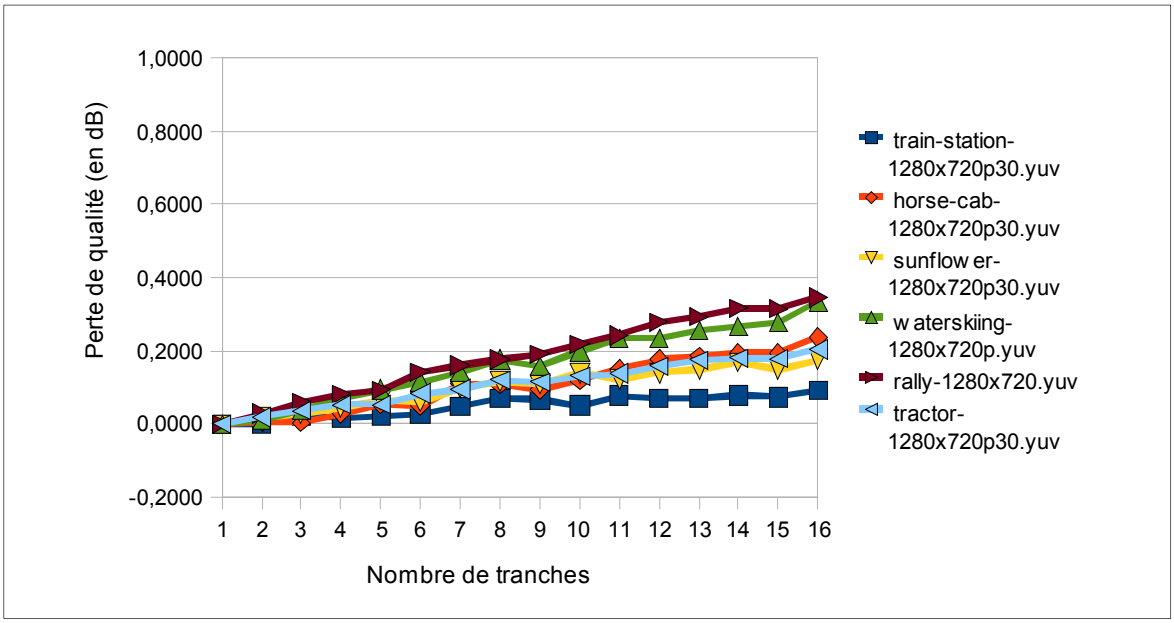


Figure-A III-2 Perte de qualité, en fonction du nombre de tranche, pour un débit de 5 Mbit/s.

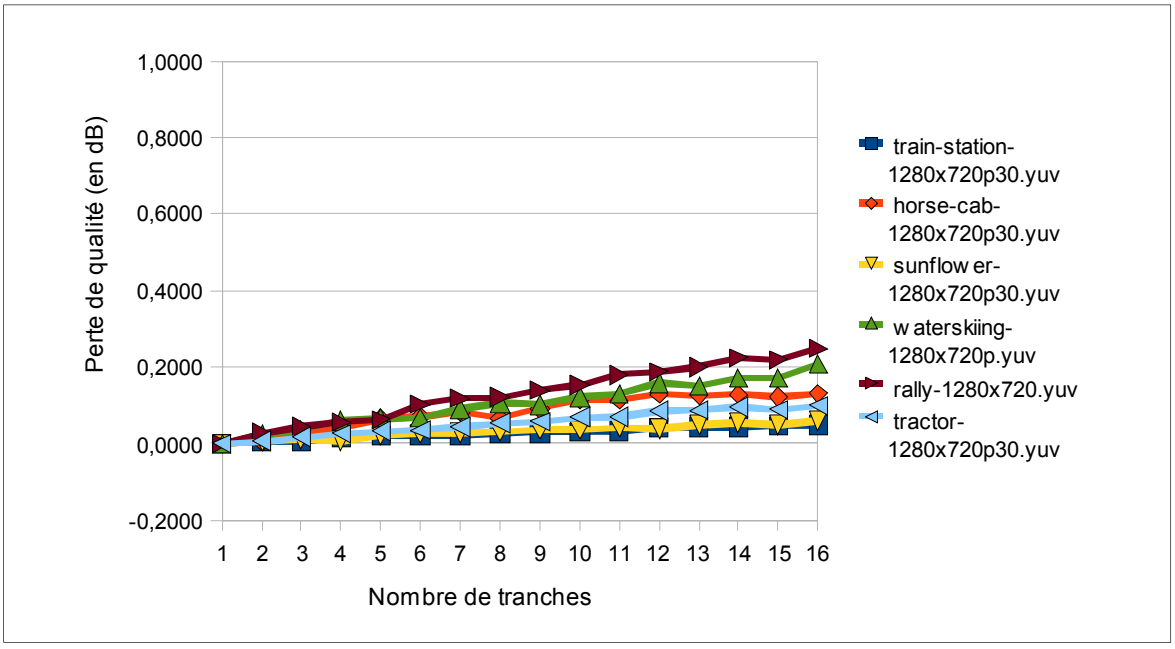


Figure-A III-3 Perte de qualité, en fonction du nombre de tranche, pour un débit de 15 Mbit/s.

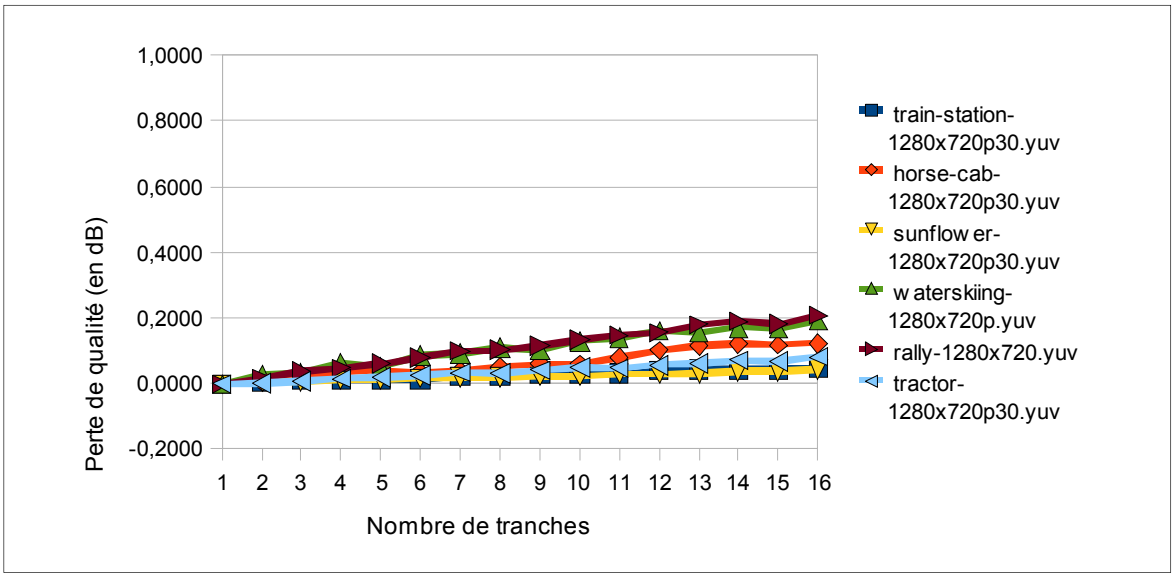


Figure-A III-4 Perte de qualité, en fonction du nombre de tranche, pour un débit de 20 Mbit/s.

ANNEXE IV

DISTRIBUTION DES TYPES DE MACROBLOC EN FONCTION DU NOMBRE DE TRANCHES ET DU DÉBIT

Tel que discuté à la section 4.3.1, l'ajout de tranches a pour effet d'augmenter le nombre de macroblocs n'ayant pas accès à l'ensemble de leurs voisins. Ceci a pour conséquence de réduire l'efficacité de la prédiction des vecteurs de mouvement et de la prédiction intra. Cette efficacité réduite, comme le montre le tableau suivant, a pour impact de réduire le nombre de macroblocs de type sauté et d'augmenter les macroblocs de type inter.

Tableau-A IV-1 Distribution des différents types de macrobloc en fonction du nombre de tranches

Nombre de tranches	Débit (Mbit/s)	Macroblocs intra (%)	Macroblocs inter (%)	Macroblocs sauté (%)
1	1	13.31	23.39	63.3
	5	13.89	59.39	26.72
	10	13.99	70.09	15.92
	15	14.4	73.84	11.76
	20	14.86	75.46	9.68
16	1	13.99	31.78	54.24
	5	14.04	64.59	21.37
	10	13.99	72.85	13.16
	15	14.29	75.74	9.96
	20	14.7	76.93	8.37

ANNEXE V

TEMPS D'ENCODAGE SÉQUENTIEL DE L'APPROCHE D'INTEL POUR DIFFÉRENTES SÉQUENCES EN FONCTION DU NOMBRE DE TRANCHES

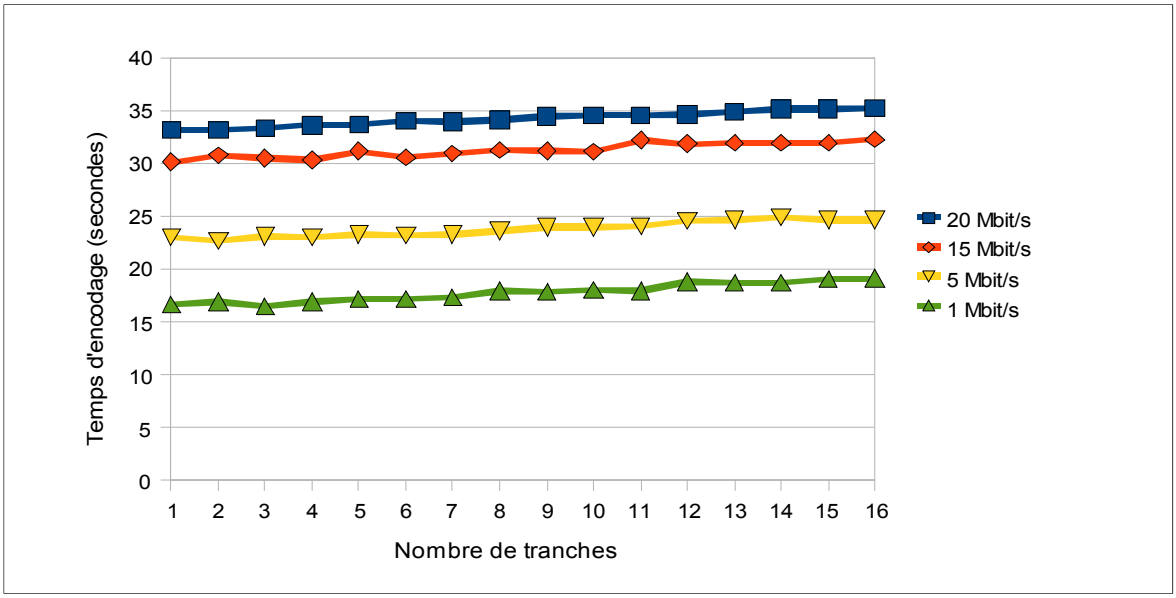


Figure-A V-1 Temps d'encodage séquentiel de l'approche d'Intel, en fonction du nombre de tranches, pour la séquence train-station.

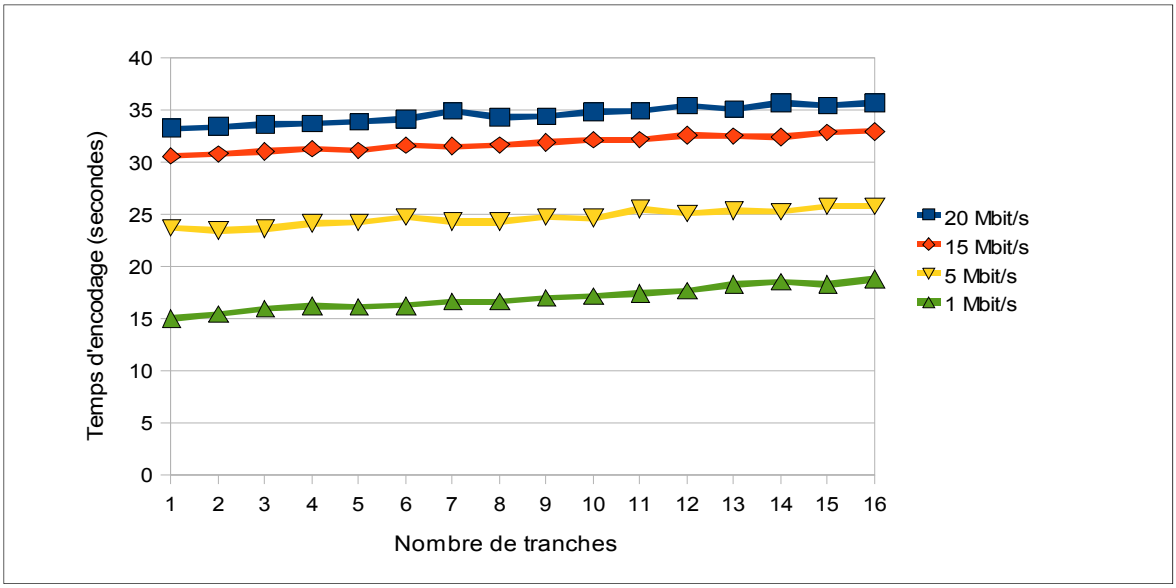


Figure-A V-2 Temps d'encodage séquentiel de l'approche d'Intel, en fonction du nombre de tranches, pour la séquence sun-flower.

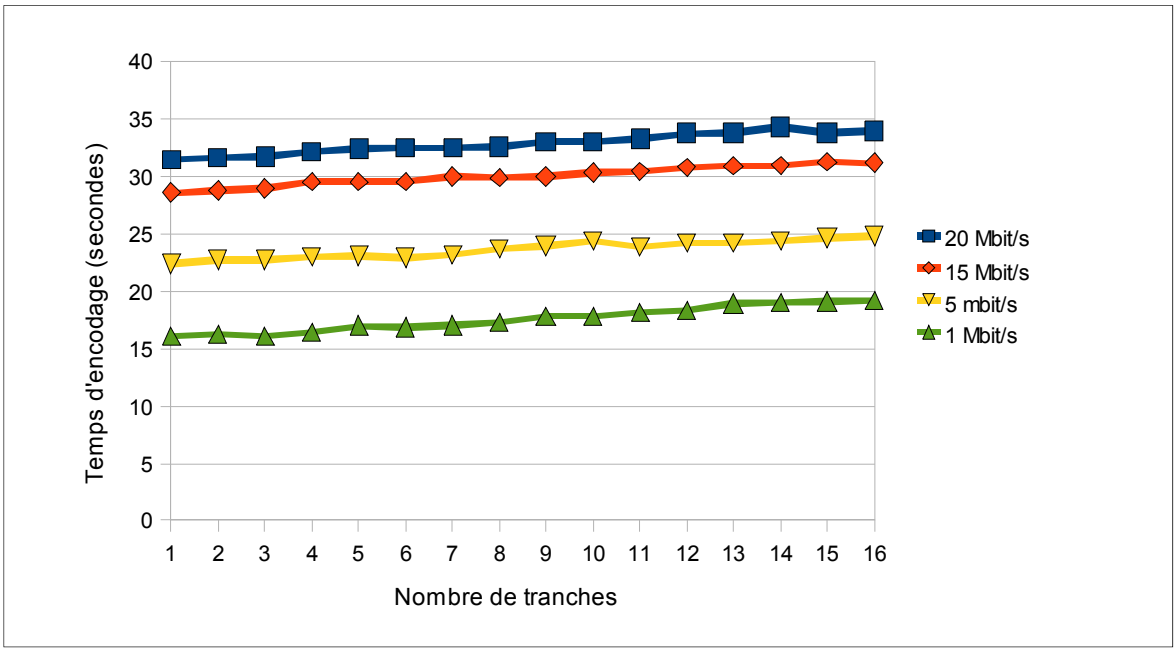


Figure-A V-3 Temps d'encodage séquentiel de l'approche d'Intel, en fonction du nombre de tranches, pour la séquence horse-cab.

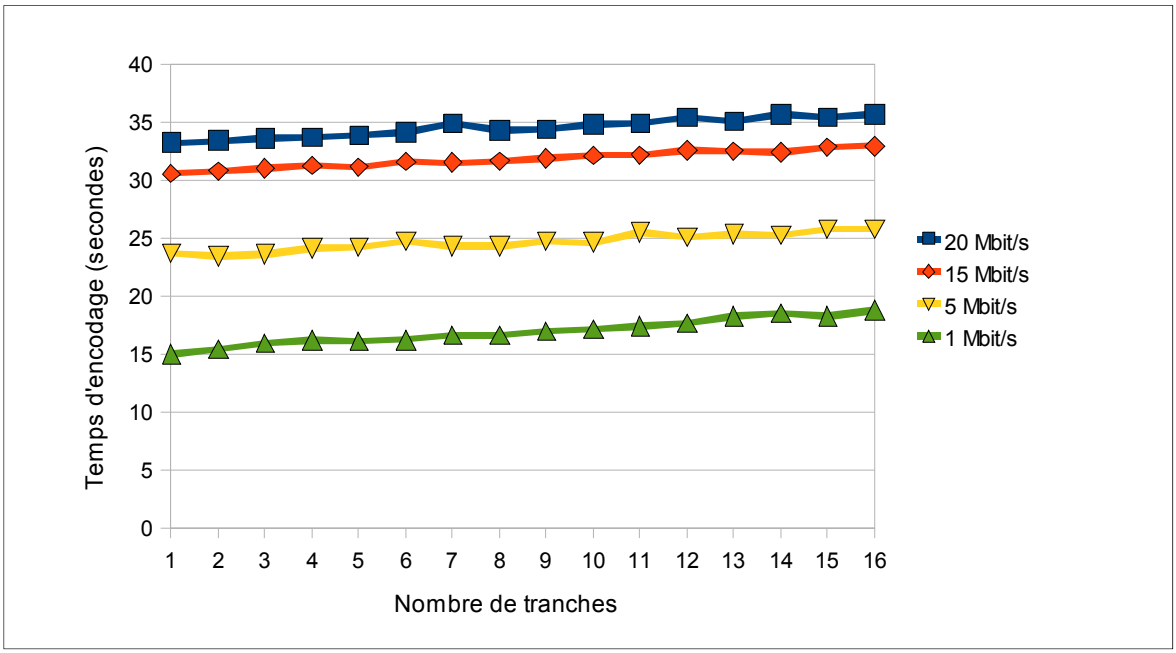


Figure-A V-4 Temps d'encodage séquentiel de l'approche d'Intel, en fonction du nombre de tranches, pour la séquence waterskiing.

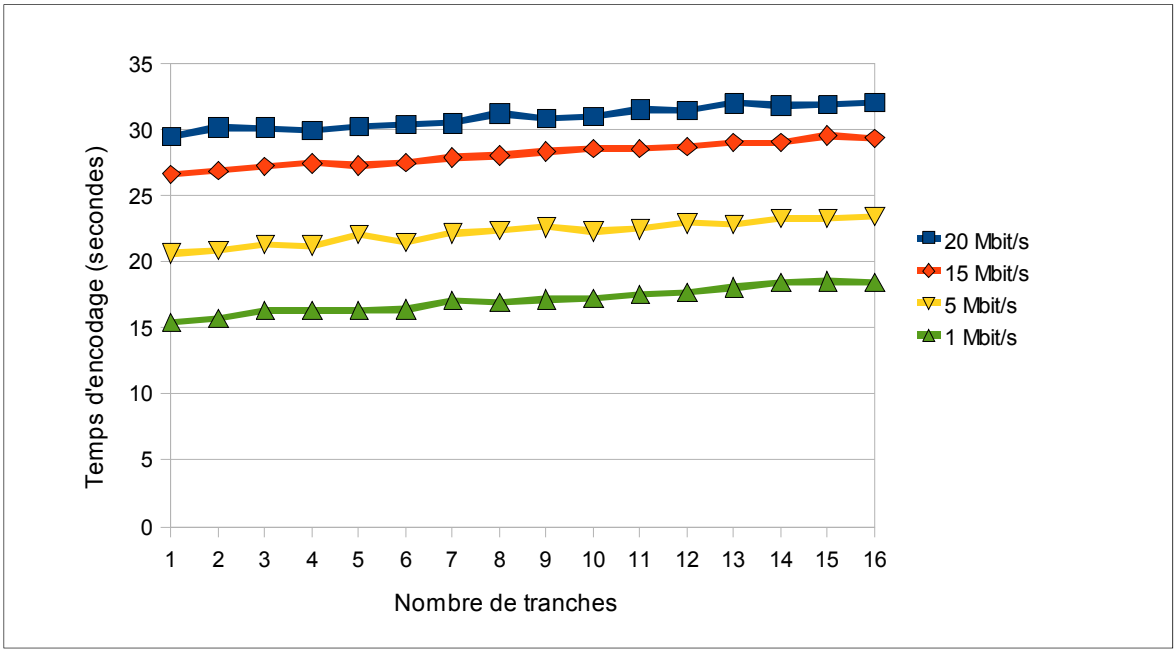


Figure-A V-5 Temps d'encodage séquentiel de l'approche d'Intel, en fonction du nombre de tranches, pour la séquence rally.

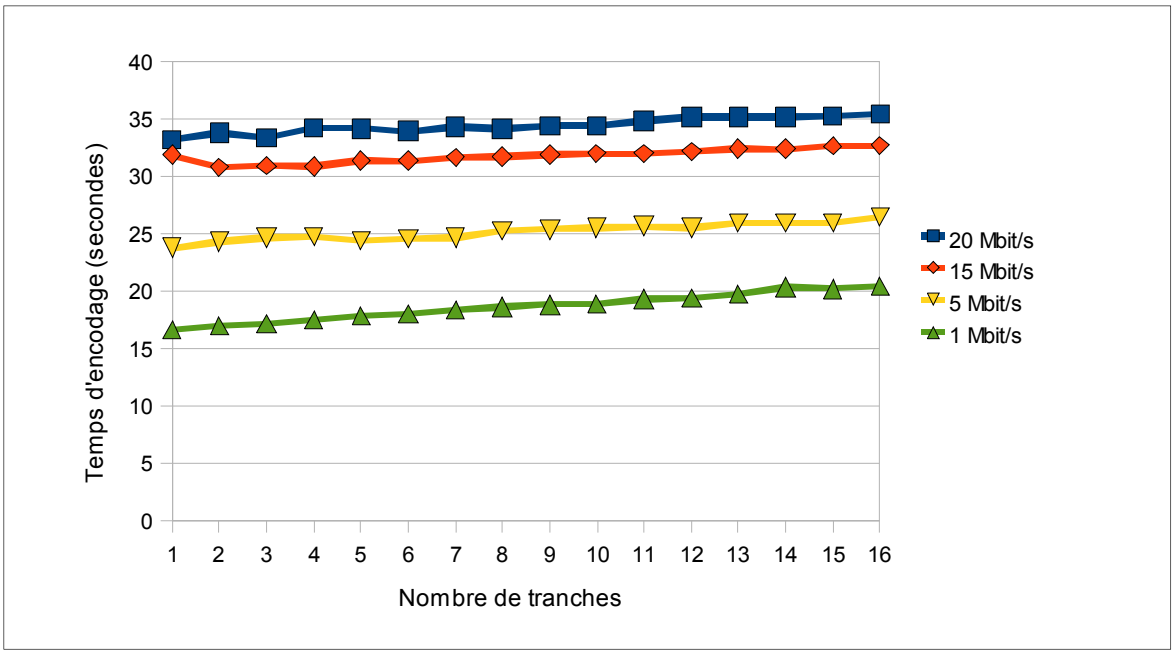


Figure-A V-6 Temps d'encodage séquentiel de l'approche d'Intel, en fonction du nombre de tranches, pour la séquence tractor.

ANNEXE VI

AUGMENTATION DU TEMPS D'ENCODAGE SÉQUENTIEL DE L'APPROCHE D'INTEL POUR DIFFÉRENTS DÉBITS EN FONCTION DU NOMBRE DE TRANCHES

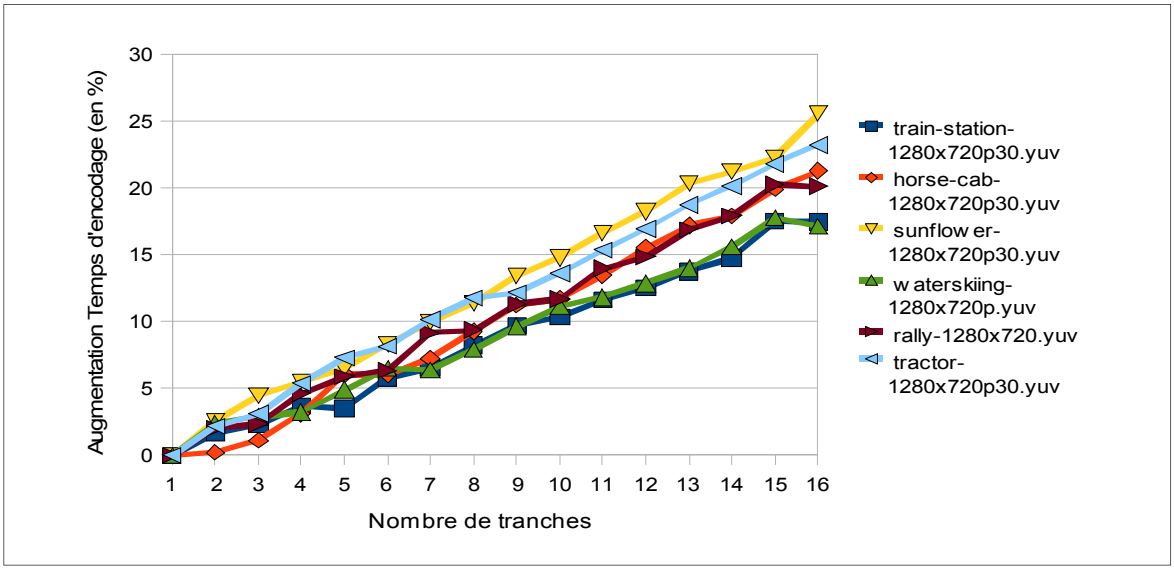


Figure-A VI-1 Augmentation du temps d'encodage séquentiel de l'approche d'Intel, en fonction du nombre de tranches, pour un débit de 1 Mbit/s.

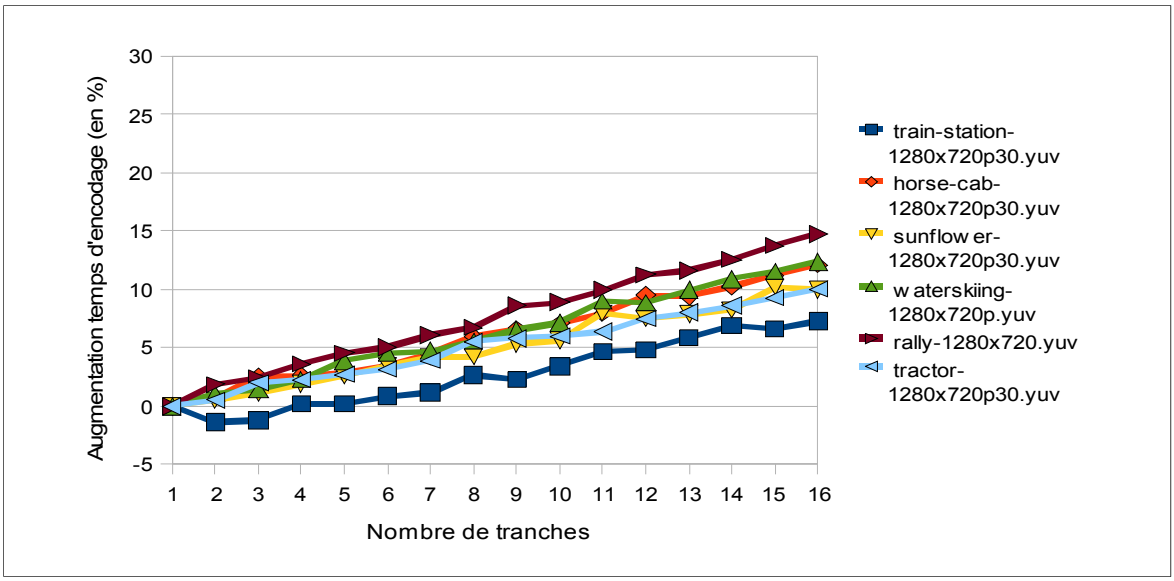


Figure-A VI-2 Augmentation du temps d'encodage séquentiel de l'approche d'Intel, en fonction du nombre de tranches, pour un débit de 5 Mbit/s.

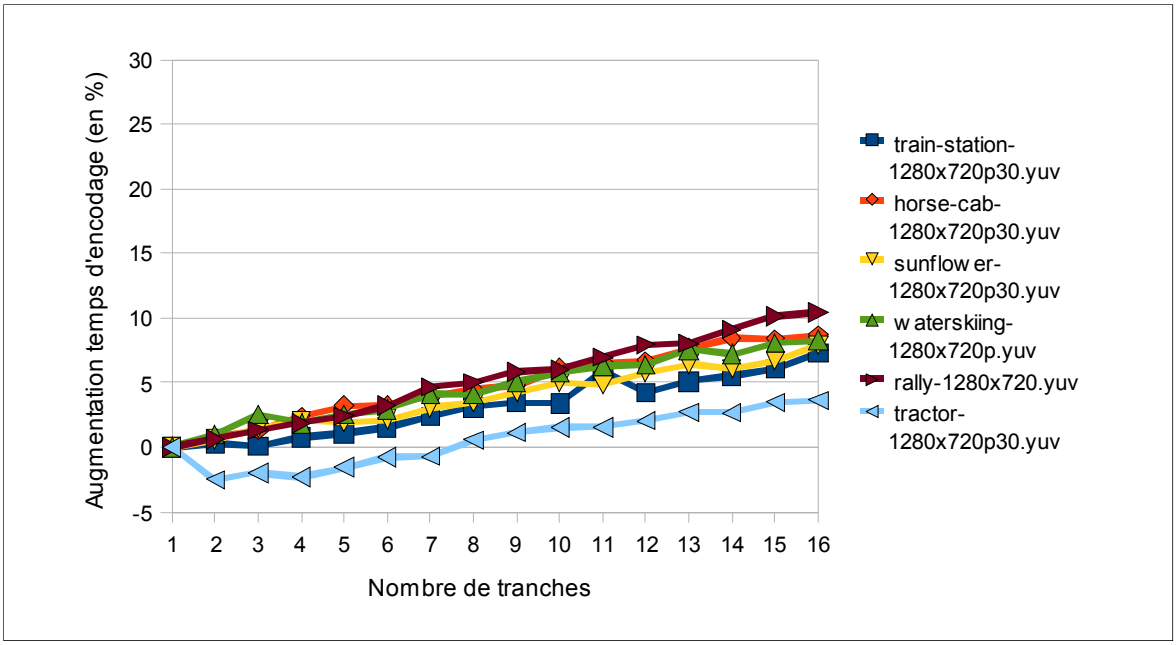


Figure-A VI-3 Augmentation du temps d'encodage séquentiel de l'approche d'Intel, en fonction du nombre de tranches, pour un débit de 15 Mbit/s.

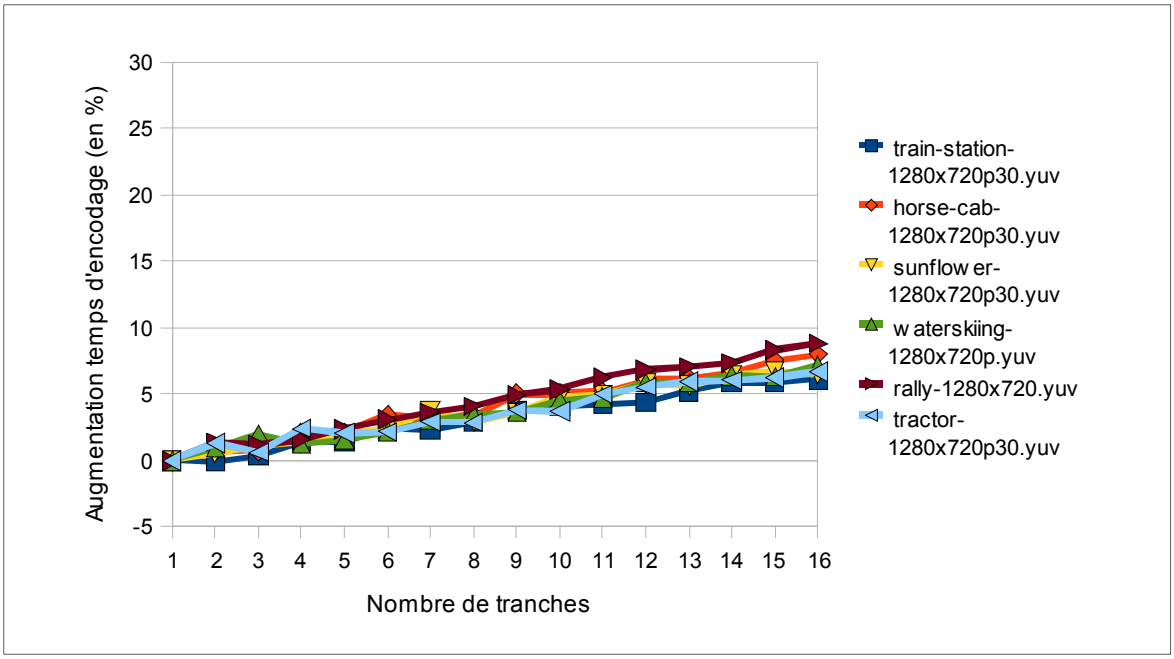


Figure-A VI-4 Augmentation du temps d'encodage séquentiel de l'approche d'Intel, en fonction du nombre de tranches, pour un débit de 20 Mbit/s.

ANNEXE VII

TEMPS D'ENCODAGE EN FONCTION DU NOMBRE DE TRANCHES POUR DIFFÉRENTS DÉBITS EN FONCTION DU NOMBRE DE TRANCHES

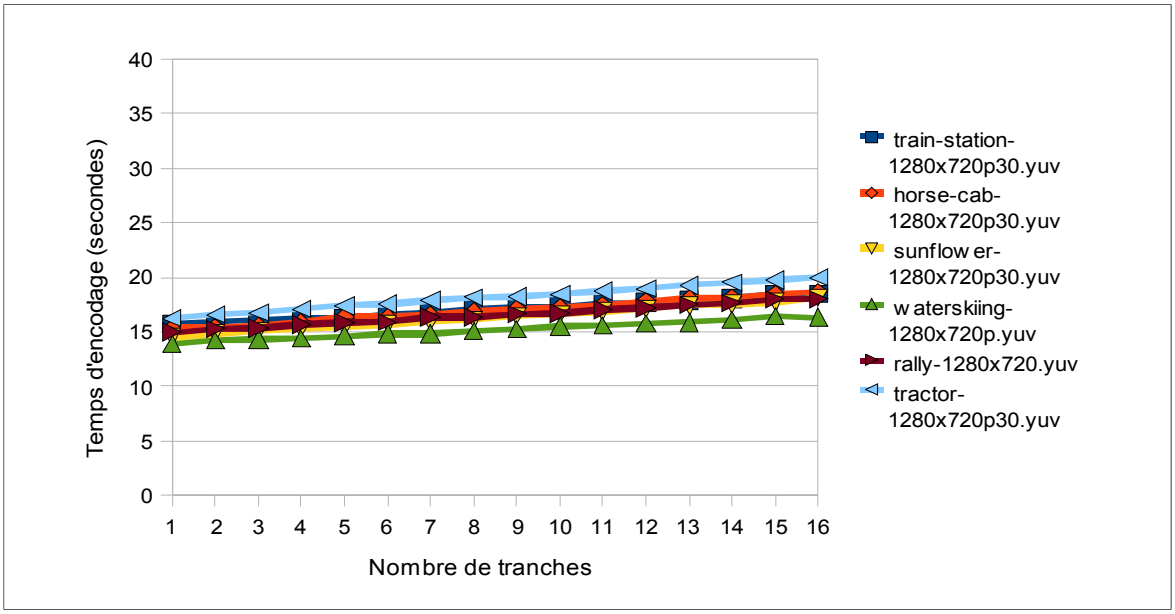


Figure-A VII-1 Temps d'encodage séquentiel de l'approche d'Intel, en fonction du nombre de tranches, pour un débit de 1 Mbit/s.

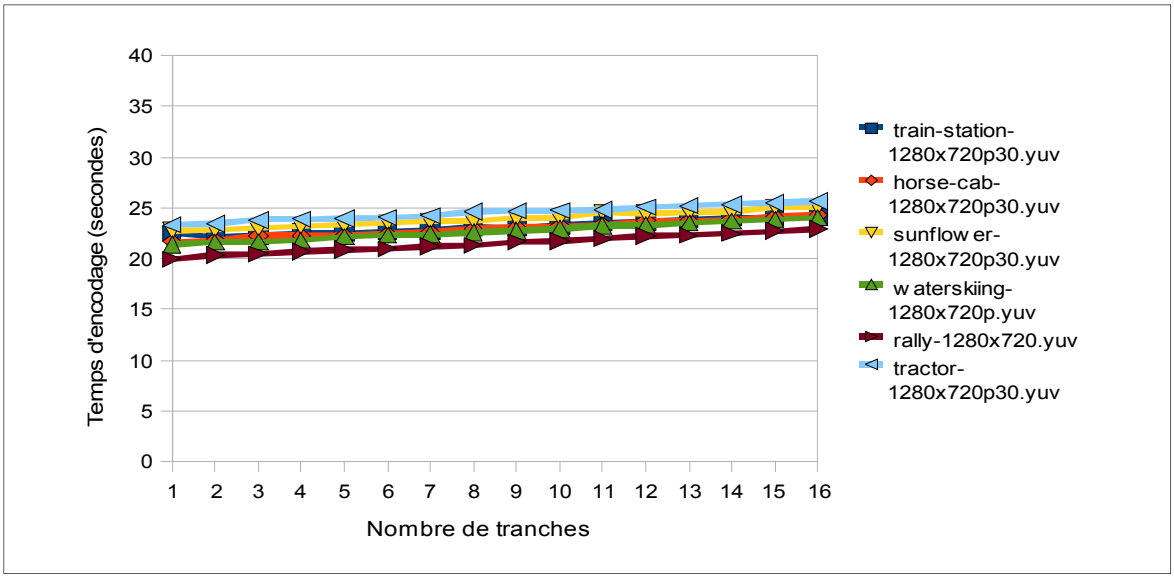


Figure-A VII-2 Temps d'encodage séquentiel de l'approche d'Intel, en fonction du nombre de tranches, pour un débit de 5 Mbit/s.

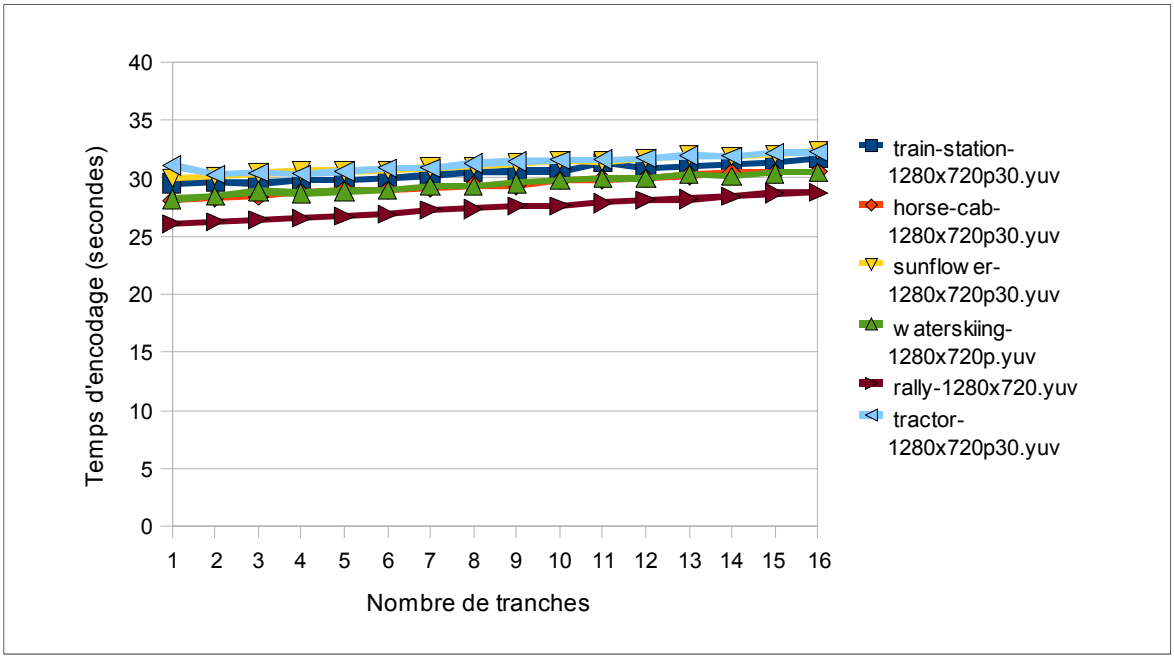


Figure-A VII-3 emps d'encodage séquentiel de l'approche d'Intel, en fonction du nombre de tranches, pour un débit de 15 Mbit/s.

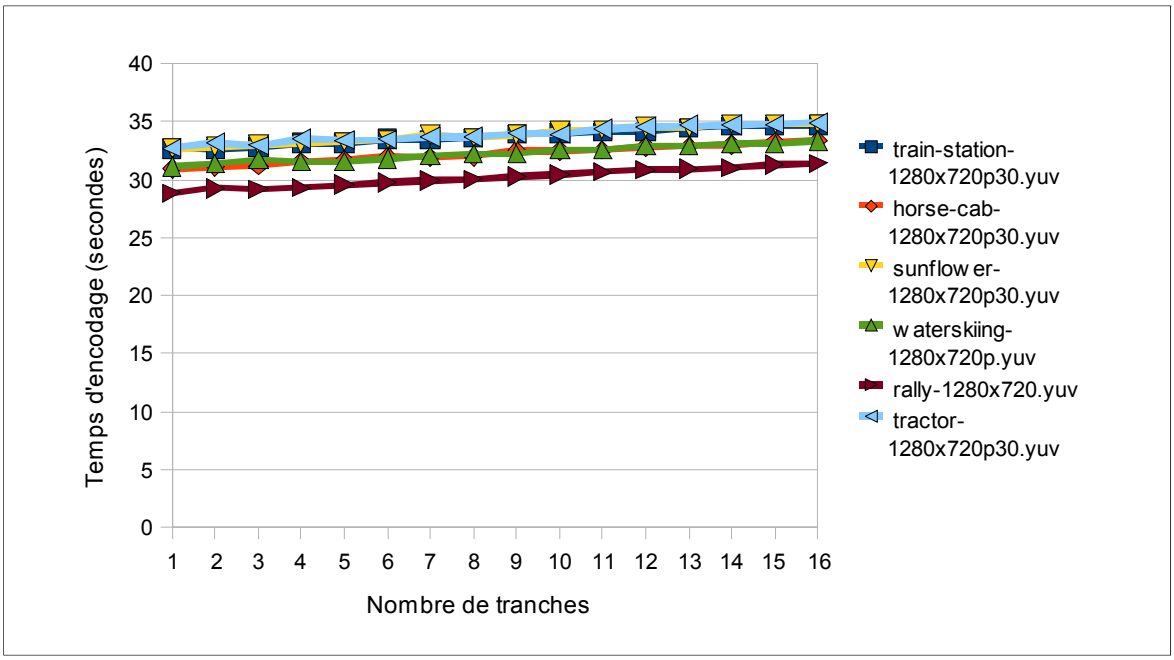


Figure-A VII-4 Temps d'encodage séquentiel de l'approche d'Intel, en fonction du nombre de tranches, pour un débit de 20 Mbit/s.

ANNEXE VIII

PARALLÉLISATION DU FILTRE DE DÉBLOCAGE

La parralélisation du filtre de déblocage de H.264 nécessite dans un premier temps la sélection du bon mode de filtrage. Dans l'encodeur H.264 du code d'exemple fourni avec la librairie IPP version d'Intel, le mode doit être changé dans le fichier umc_h264_video_encoder.cpp dans la fonction H264EncoderParams::H264EncoderParams() en remplaçant la ligne suivante :

```
deblocking_filter_idc          = 0;
```

Par :

```
deblocking_filter_idc          = 2;
```

Dans un second temps, nous devons paralléliser la boucle qui applique le filtre de déblocage en ajoutant, dans la fonction H264CoreEncoder_CompressFrame du fichier umc_h264_gen_enc_tmp.cpp.h, la ligne suivante :

```
#pragma omp parallel for private(slice) schedule(dynamic,1)
```

Au dessus de :

```
for (slice = (Ipp32s)core_enc->m_info.num_slices*core_enc->m_field_index;
slice < core_enc->m_info.num_slices*(core_enc->m_field_index+1); slice++)
{
      if (core_enc->m_Slices[slice].status != UMC_OK){
            core_enc->m_bMakeNextFrameKey = true;
            VM_ASSERT(0);// goto done;
      }
      else if(ePic_Class != DISPOSABLE_PIC)
      {
            H264ENC_MAKE_NAME(H264CoreEncoder_DeblockSlice)(state,
            core_enc->m_Slices + slice, core_enc-
            >m_Slices[slice].m_first_mb_in_slice + core_enc-
            >m_WidthInMBs*core_enc->m_HeightInMBs*core_enc->m_field_index,
            core_enc->m_Slices[slice].m_MB_Counter, slice == core_enc-
            >m_info.num_slices - 1);
      }
}
```

ANNEXE IX

DIVISION D'UNE TRAME EN TRANCHES POUR UNE RÉSOLUTION DE 1280 × 720

Tel que vu précédemment, le nombre de lignes de macroblocs attribué à chaque tranche dans notre approche dépend de la résolution $H \times V$ pixels des trames à encoder et du nombre N de tranches désiré. Le nombre L de lignes de macroblocs par trame est égal à V / 16. Si $S = L / N$ et $R = L$ modulo N, alors nous avons une première série T_0 de N – R tranches avec S lignes et une seconde série T_1 de R tranches avec $S + 1$ lignes.

Tableau-A IX-1 Division d'une trame en tranches pour une résolution de 1280 x 720

N *(nombre de tranches)*	***S*** (L / N)	***R*** (L modulo N)	T_0 $((N - R) \times S)$	T_1 $(R \times (S + 1))$
1	45	0	1 × 45	0 × 46
2	22	1	1 × 22	1 × 23
3	15	0	3 × 15	0 × 16
4	11	1	3 × 11	1 × 12
5	9	0	5 × 9	0 × 10
6	7	3	3 × 7	3 × 8
7	6	3	4 × 6	3 × 7
8	5	5	3 × 5	5 × 6
9	5	0	9 × 5	0 × 6
10	4	5	5 × 4	5 × 5
11	4	1	10 × 4	1 × 5
12	3	9	3 × 3	9 × 4

ANNEXE X

FONCTIONS DE SYNCHRONISATION

```
/* ************************************************************
 * ETS_H264CoreEncoder_PreUpdateCoreEncoderGeneralInfo
 * Author: Jean-FranÃ§ois Franche
 * Description: Copy some core_encoder information from the
 *                      previous core_encoder to the current core_encodor.
 *                      This function is called just after
 *                      Pre-Slices-Encoding-Waiting-Point.
 * N.B: Maybe somes (or all) of this information can be predicted.
 * © ETS 2010
************************************************************/
void
H264ENC_MAKE_NAME(ETS_H264CoreEncoder_PreUpdateCoreEncoderGeneralInfo)(
    void* state)
{
    H264CoreEncoderType* core_enc = (H264CoreEncoderType *)state;
      If(core_enc->m_info.parallel_algorithm==
ETS_H264_MFMS_PARALLEL_APPROACH_ID)
      {
#ifdef LOG_SYNCHRO_FUNCTIONS
            printf("Start
ETS_H264CoreEncoder_PreUpdateCoreEncoderGeneralInfo\n");
#endif

            H264CoreEncoderType* pre_core_enc =
H264ENC_MAKE_NAME(ETS_H264CoreEncoder_GetPreviewCoreEncoderPointer)(core_e
nc);

          core_enc->m_uIDRFrameInterval   = pre_core_enc-
>m_uIDRFrameInterval;  // H264CoreEncoder_DetermineFrameType
          core_enc->m_uIntraFrameInterval = pre_core_enc-
>m_uIntraFrameInterval;
          core_enc->m_uFrameCounter             = pre_core_enc-
>m_uFrameCounter;
          core_enc->m_PicOrderCnt_Accu          = pre_core_enc-
>m_PicOrderCnt_Accu;
          core_enc->m_PicOrderCnt               = pre_core_enc-
>m_PicOrderCnt;

          // special case for first frame processed by core_encoder
          if(core_enc->m_uFrames_Num==0)
          {
            core_enc->m_uFrames_Num += core_enc-
>m_MyTabCoreEncodersPosition;
              core_enc->m_l1_cnt_to_start_B           = pre_core_enc-
>m_l1_cnt_to_start_B;
              core_enc->m_bMakeNextFrameKey           = 0; // Force to be
0 for frame after frame 0.
```

```
      H264ENC_MAKE_NAME(H264CoreEncoder_SetPictureParameters)(state);
          }
          else
          {
            core_enc->m_uFrames_Num +=
(NB_CORE_ENCODER_FOR_MULTI_FRAMES_MULTI_SLICES_PAR_ALGO-1);
          }

#ifdef LOG_SYNCHRO_FUNCTIONS
          printf("End
ETS_H264CoreEncoder_PreUpdateCoreEncoderGeneralInfo\n");
#endif
      }
}

/* ************************************************************
 * ETS_H264CoreEncoder_UpdateRateControl
 * Author: Jean-Francois Franche
 * Input: state --> current core_encoder pointer.
 * Description: Copy Rate Control state from the previous core_encoder
 *              into the current core_encoder. This function
 *                      is called in the Post-Slices-Encoding-Part just
after
 *                      the Post-Slices-Encoding-Waiting-Point and just
before
 *                      the call to H264_AVBR_PostFrame function (this
function
 *                      update the rate control in function of number of
used bits
 *                      in the current frame.)
 * N.B: MFMS Algorithm was tested and valided only with
 *      rate_controls.method = H264_RCM_VBR and
 *          rate_controls.method = H264_RCM_CBR.
 * @see H264_AVBR_PostFrame
 * © ETS 2010
 * ************************************************************ */
void H264ENC_MAKE_NAME(ETS_H264CoreEncoder_UpdateRateControl)(
    void* state)
{
    H264CoreEncoderType* core_enc = (H264CoreEncoderType *)state;

    if ((core_enc->m_info.parallel_algorithm ==
ETS_H264_MFMS_PARALLEL_APPROACH_ID) &&(core_enc->m_uFrames_Num!=0))
      {
#ifdef LOG_SYNCHRO_FUNCTIONS
          printf("Start ETS_H264CoreEncoder_UpdateRateControl\n");
#endif

          H264CoreEncoderType* prev_core_enc =
H264ENC_MAKE_NAME(ETS_H264CoreEncoder_GetPreviewCoreEncoderPointer)(core_e
nc);
          H264_AVBR *prev_avbr = &prev_core_enc->avbr;
          H264_AVBR *cur_avbr = &core_enc->avbr;
```

```
          // Update Quantification Value
          cur_avbr->mQuantPrev      = prev_avbr->mQuantPrev;
          cur_avbr->mQuantI         = prev_avbr->mQuantI;
          cur_avbr->mQuantP         = prev_avbr->mQuantP;
          cur_avbr->mQuantB         = prev_avbr->mQuantB;

          // Update Bits encoded Total and Bits desired Total
          cur_avbr->mBitsEncodedTotal           = prev_avbr-
>mBitsEncodedTotal;
          cur_avbr->mBitsDesiredTotal           = prev_avbr-
>mBitsDesiredTotal;

          // Update RateControl variables
          cur_avbr->mRCq                                    = prev_avbr-
>mRCq;
          cur_avbr->mRCqa                               = prev_avbr->mRCqa;
          cur_avbr->mRCfa                                   = prev_avbr-
>mRCfa;
#ifdef LOG_SYNCHRO_FUNCTIONS
            printf("End ETS_H264CoreEncoder_UpdateRateControl\n");
#endif
      }
}

/* ************************************************************
 * ETS_H264CoreEncoder_PostUpdateCoreEncoderGeneralInfo
 * Author: Jean-FranÃ§ois Franche
 * Description: Copy m_total_bits_encoded from the previous core_encoder
 *              into the current core_encoder. This function is called
 *                      just after ETS_H264CoreEncoder_UpdateRateControl.
 * © ETS 2010
 * ************************************************************ */
void
H264ENC_MAKE_NAME(ETS_H264CoreEncoder_PostUpdateCoreEncoderGeneralInfo)(
    void* state)
{
    H264CoreEncoderType* core_enc = (H264CoreEncoderType *)state;
      if (core_enc->m_info.parallel_algorithm ==
ETS_H264_MFMS_PARALLEL_APPROACH_ID)
      {
#ifdef LOG_SYNCHRO_FUNCTIONS
            printf("Start
ETS_H264CoreEncoder_PostUpdateCoreEncoderGeneralInfo\n");
#endif

            H264CoreEncoderType* pre_core_enc =
H264ENC_MAKE_NAME(ETS_H264CoreEncoder_GetPreviewCoreEncoderPointer)(core_e
nc);

          core_enc->m_total_bits_encoded        = pre_core_enc-
>m_total_bits_encoded;

#ifdef LOG_SYNCHRO_FUNCTIONS
      printf("End
ETS_H264CoreEncoder_PostUpdateCoreEncoderGeneralInfo\n");
```

```
#endif
      }
}

/* ************************************************************
 * H264CoreEncoder_WaitToPreSlicesLoopProcess
 * author: Jean-FranÃ§ois Franche
 * Description: Wait (if necessary) the end of Pre-Slice-Encoding-Part of
 *                       the previous core_encoder. This function is used
to block
 *                       the current core_encoder before the call to the
 *                       inter-encoder_core data transfert function
 *
      ETS_H264CoreEncoder_PreUpdateCoreEncoderGeneralInf.
 * @see: ETS_H264CoreEncoder_PreUpdateCoreEncoderGeneralInfo
 * © ETS 2010
 * ************************************************************ */
void H264ENC_MAKE_NAME(ETS_H264CoreEncoder_WaitToPreSlicesLoopProcess)(
    void* state)
{
    H264CoreEncoderType* core_enc = (H264CoreEncoderType *)state;
    if (core_enc->m_info.parallel_algorithm==
ETS_H264_MFMS_PARALLEL_APPROACH_ID)
    {
#ifdef LOG_WAIT_FUNCTIONS
      printf("Start ETS_H264CoreEncoder_WaitToPreSlicesLoopProcess\n");
#endif

          H264CoreEncoderType* pre_core_enc =
H264ENC_MAKE_NAME(ETS_H264CoreEncoder_GetPreviewCoreEncoderPointer)(core_e
nc);

          /* loop and wait until the previous core encoder finish process
his
             Pre-Slices-Encoding-Part */
#ifdef THREADS_SYNC_SLEEP_AND_POOL_MODE
          while(!pre_core_enc->m_PreSlicesLoopSectionCompleted)
          {

      H264LOGGER_LOG_CORE_ENCODER_EVENT(START_WAIT_PREV_FRAME_PRE_SLICES_E
NCODING, core_enc->m_MyTabCoreEncodersPosition,core_enc->m_uFrames_Num);
            vm_time_sleep(DEFAULT_WAIT_TIME);

      H264LOGGER_LOG_CORE_ENCODER_EVENT(END_WAIT_PREV_FRAME_PRE_SLICES_ENC
ODING, core_enc->m_MyTabCoreEncodersPosition,core_enc->m_uFrames_Num);
          }
          pre_core_enc->m_PreSlicesLoopSectionCompleted = false;
#endif
#ifdef THREADS_SYNC_EVENT_MODE
          vm_event_wait(&pre_core_enc->event_PreSlicesEncoding);
#endif
#ifdef LOG_WAIT_FUNCTIONS
      printf("End ETS_H264CoreEncoder_WaitToPreSlicesLoopProcess\n");
#endif
    }
```

```
}

/* ************************************************************
/* H264CoreEncoder_WaitToPostSlicesLoopProcess
 * Input: state --> current core_encoder pointer
 * author: Jean-FranÃ§ois Franche
 * Description: Wait (if necessary) the end of Post-Slice-Encoding-Part of
 *                      the previous core_encoder.
 * © ETS 2010
 * ************************************************************/
void H264ENC_MAKE_NAME(ETS_H264CoreEncoder_WaitToPostSlicesLoopProcess)(
    void* state)
{
    H264CoreEncoderType* core_enc = (H264CoreEncoderType *)state;
      if ((core_enc->m_info.parallel_algorithm
==ETS_H264_MFMS_PARALLEL_APPROACH_ID) &&
            (core_enc->m_uFrames_Num!=0))
      {
#ifdef LOG_WAIT_FUNCTIONS
        printf("Start-ETS_H264CoreEncoder_WaitToPostSlicesLoopProcess\n");
#endif

          H264CoreEncoderType* pre_core_enc =
H264ENC_MAKE_NAME(ETS_H264CoreEncoder_GetPreviewCoreEncoderPointer)(core_e
nc);

      H264LOGGER_LOG_CORE_ENCODER_EVENT(START_WAIT_PREVIOUS_FRAME_POST_SLI
CES_ENCODING, core_enc->m_MyTabCoreEncodersPosition,core_enc-
>m_uFrames_Num);

#ifdef THREADS_SYNC_SLEEP_AND_POOL_MODE
          /* loop and wait until the previous core encoder finish process
his
             Post-Slices-Encoding-Part */
            while(!pre_core_enc->m_PostSlicesLoopSectionCompleted)
          {
            vm_time_sleep(DEFAULT_WAIT_TIME);
          }
            pre_core_enc->m_PostSlicesLoopSectionCompleted = false;
#endif

#ifdef THREADS_SYNC_EVENT_MODE
    vm_event_wait(&pre_core_enc->event_PostSlicesEncoding);
#endif

      H264LOGGER_LOG_CORE_ENCODER_EVENT(END_WAIT_PREVIOUS_FRAME_POST_SLICE
S_ENCODING, core_enc->m_MyTabCoreEncodersPosition,core_enc-
>m_uFrames_Num);
#ifdef LOG_WAIT_FUNCTIONS
        printf("End ETS_H264CoreEncoder_WaitToPostSlicesLoopProcess\n");
#endif
      }
}
```

```
/* ************************************************************
/* H264CoreEncoder_WaitRefFrameIsReadyInNextCoreEncoder
 * author: Jean-FranÃ§ois Franche
 * input: state --> current core_encoder pointer
 * Description: Cette fonction attend que le core_encoder, qui traite la
trame
 * suivante est prÃªt Ã  recevoir les donnÃ©es de la prochaine trame
 * Description: wait frame reference in the next core_encoder is
 *                      ready to receive data. This method is called by
 *                      H264CoreEncoder_CompressFrame
 * @see: H264CoreEncoder_CompressFrame
 * © ETS 2010
 * /* ************************************************************ */
void
H264ENC_MAKE_NAME(ETS_H264CoreEncoder_WaitRefFrameIsReadyInNextCoreEncoder
)(
          void* state)
{
    H264CoreEncoderType* core_enc = (H264CoreEncoderType *)state;
      if (core_enc->m_info.parallel_algorithm ==
ETS_H264_MFMS_PARALLEL_APPROACH_ID)
    {
            if(core_enc->m_uFrames_Num>(core_enc-
>m_info.numFramesToEncode-2))
            {
                  return;
            }
#ifdef LOG_WAIT_FUNCTIONS
        printf("Start-
ETS_H264CoreEncoder_WaitRefFrameIsReadyInNextCoreEncoder\n");
#endif
          H264CoreEncoderType* next_core_enc    =
H264ENC_MAKE_NAME(ETS_H264CoreEncoder_GetNextCoreEncoderPointer)(core_enc)
;

          bool moreSlicesToCopy=false;
            for(Ipp32u i=0;i<core_enc->uNumSlices;i++)
            {
                  if( (core_enc->m_SlicesToCopy[i]) || (!next_core_enc-
>m_SlicesCopied[i]))
                  {
                        moreSlicesToCopy=true;
                  }
            }

            /* loop and wait untel no more slices need to be copy in
             * the next core encoder. If we loop more then
             * DEFAULT_MAX_WAIT_LOOP (5000) times we return a error msg
             * and exit application. */
          Ipp32u iNbLoops = 0;
          while( (iNbLoops<DEFAULT_MAX_WAIT_LOOP) && (moreSlicesToCopy))
          {

            vm_time_sleep(DEFAULT_WAIT_TIME);
            moreSlicesToCopy=false;
```

```
                    for(Ipp32u i=0;i<core_enc->uNumSlices;i++)
                    {
                          if( (core_enc->m_SlicesToCopy[i]) ||
(!next_core_enc->m_SlicesCopied[i]))
                          {
                                moreSlicesToCopy=true;
                          }
                    }
              iNbLoops++;
            }
            if(iNbLoops>(DEFAULT_MAX_WAIT_LOOP-1))
              {
                    printf("Waiting for next ref more than 1 second.\n");
              exit(0);
              }
              #pragma omp flush
#ifdef LOG_WAIT_FUNCTIONS
        printf("End-
ETS_H264CoreEncoder_WaitRefFrameIsReadyInNextCoreEncoder\n");
#endif
    }
}
```

ANNEXE XI

PERFORMANCE DU PREMIER MODE EN FONCTION DU NOMBRE DE CŒURS ET DU NOMBRE DE TRANCHES SUPPLÉMENTAIRES

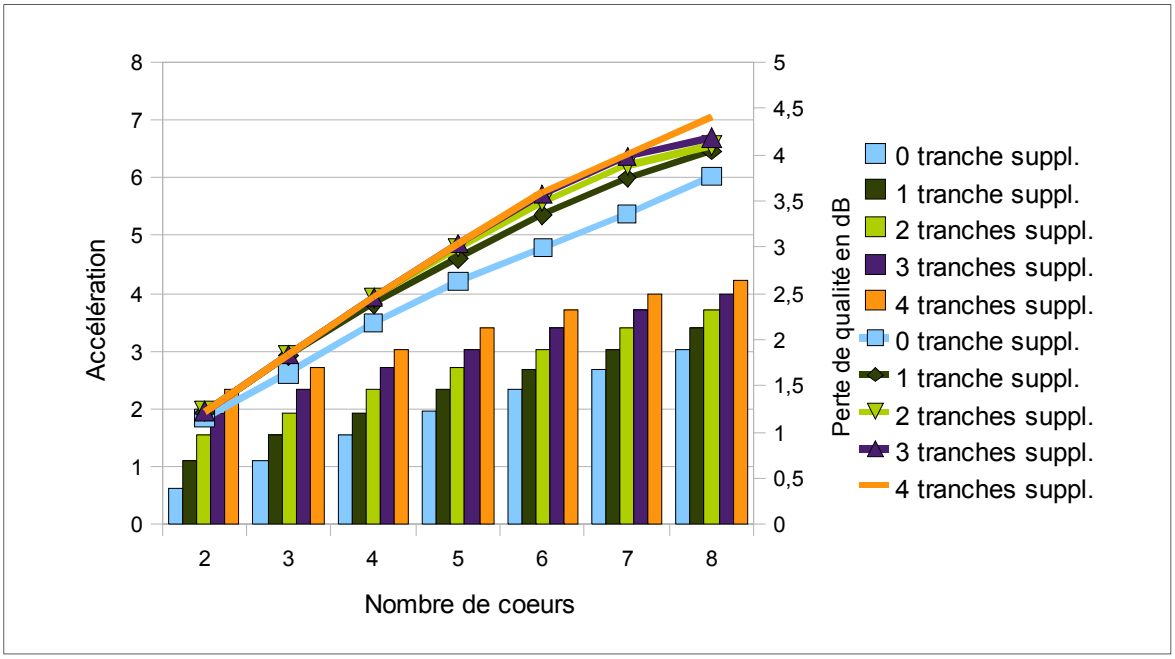


Figure-A XI-1 Accélération et perte de qualité du mode 1 en fonction du nombre de tranches supplémentaires et du nombre de cœurs pour un débit de 1 Mbit/s.

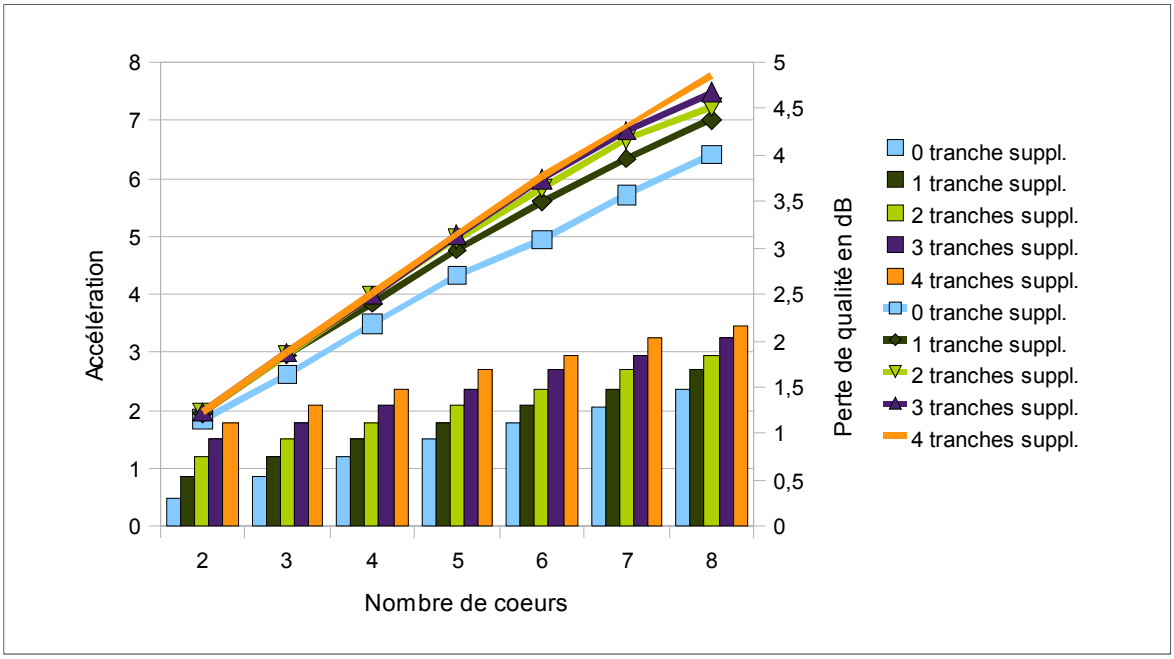


Figure-A XI-2 Accélération et perte de qualité du mode 1 en fonction du nombre de tranches supplémentaires et du nombre de cœurs pour un débit de 5 Mbit/s.

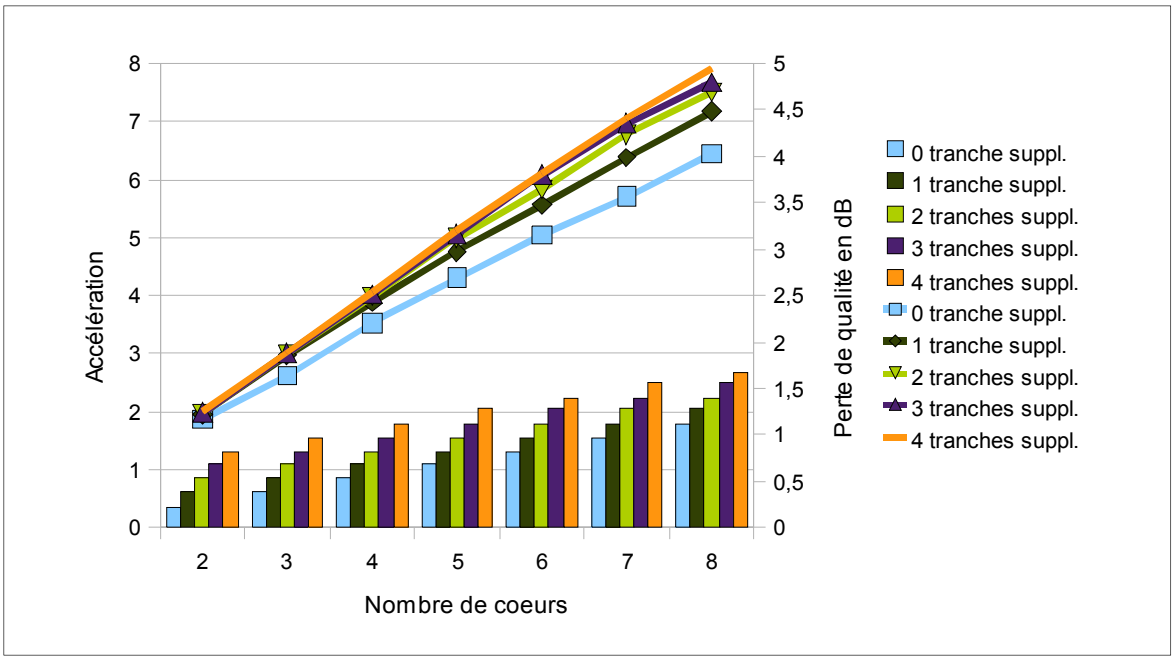


Figure-A XI-3 Accélération et perte de qualité du mode 1 en fonction du nombre de tranches supplémentaires et du nombre de cœurs pour un débit de 15 Mbit/s.

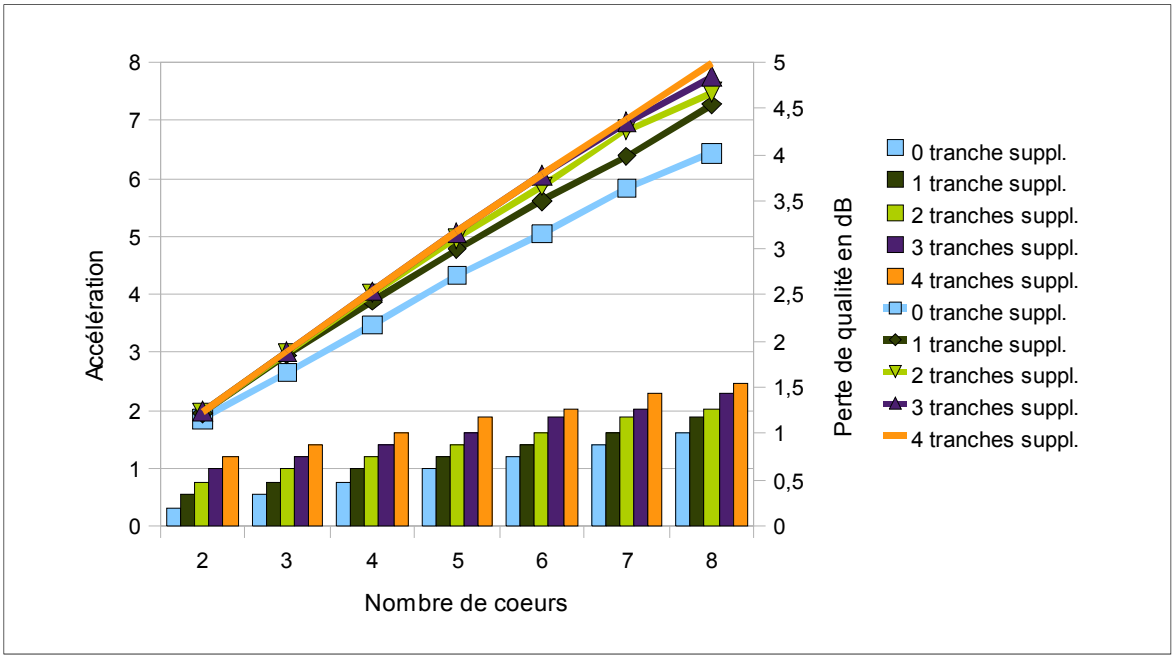


Figure-A XI-4 Accélération et perte de qualité du mode 1 en fonction du nombre de tranches supplémentaires et du nombre de cœurs pour un débit de 20 Mbit/s.

ANNEXE XII

PERFORMANCES DU SECOND MODE EN FONCTION DU NOMBRE DE CŒURS ET DU NOMBRE DE TRANCHES SUPPLÉMENTAIRES

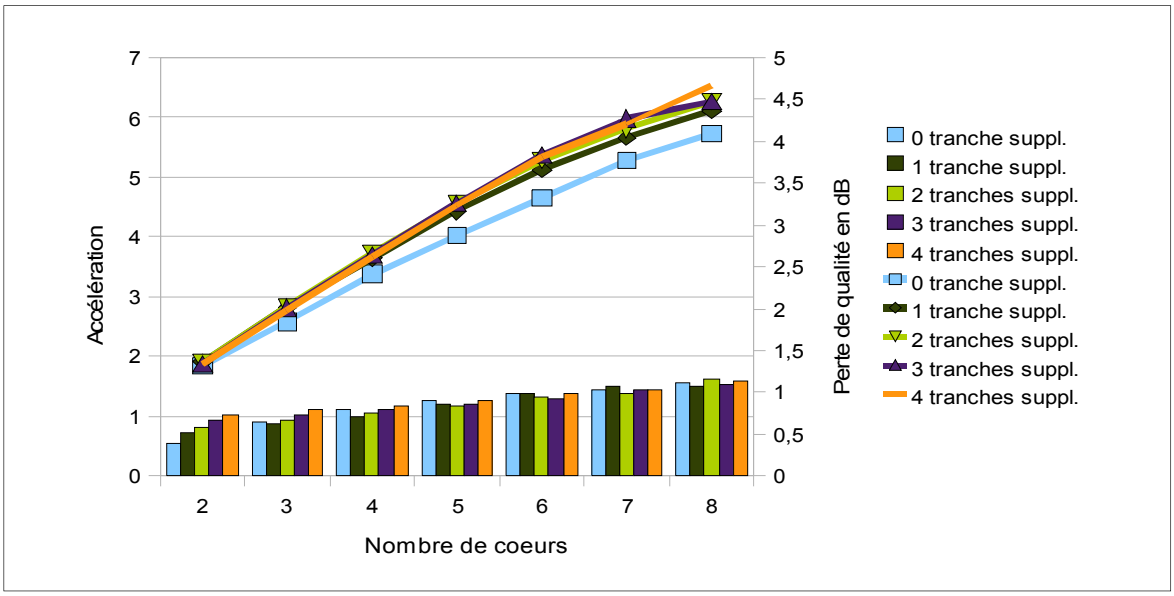


Figure-A XII-1 Accélération et perte de qualité du mode 2 en fonction du nombre de tranches supplémentaires et du nombre de cœurs pour un débit de 1 Mbit/s.

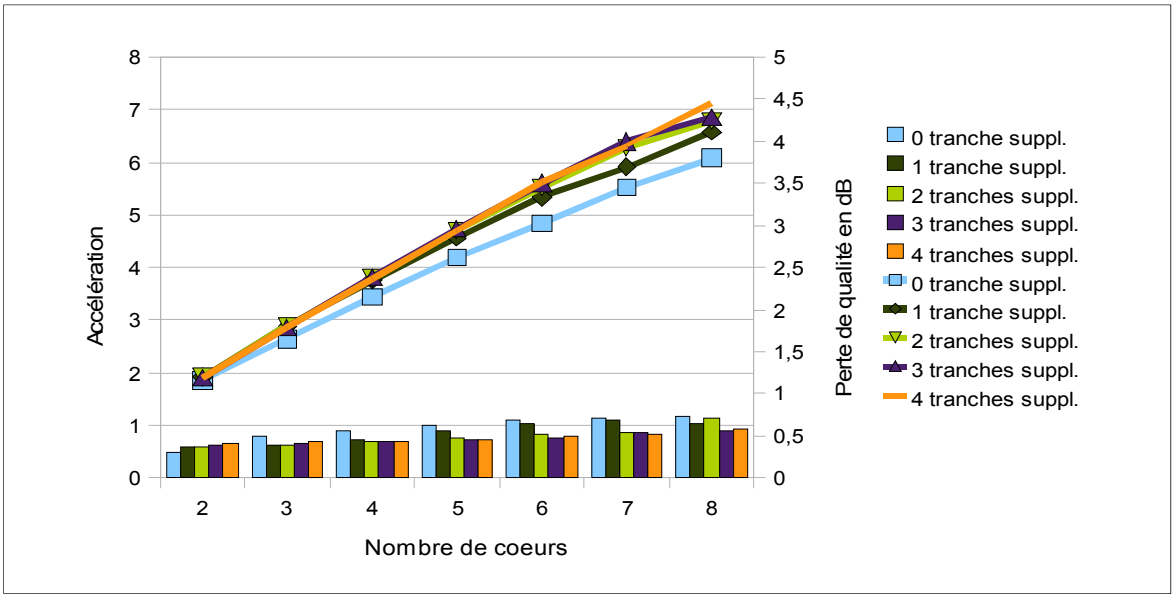


Figure-A XII-2 Accélération et perte de qualité du mode 2 en fonction du nombre de tranches supplémentaires et du nombre de cœurs pour un débit de 5 Mbit/s.

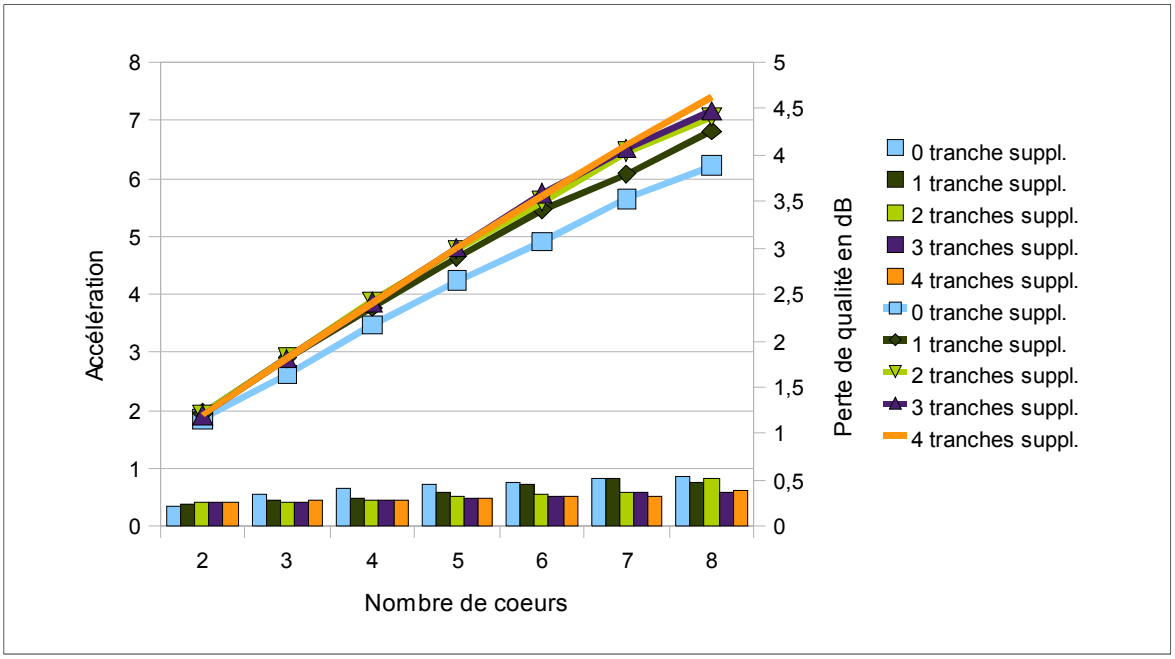


Figure-A XII-3 Accélération et perte de qualité du mode 2 en fonction du nombre de tranches supplémentaires et du nombre de cœurs pour un débit de 15 Mbit/s.

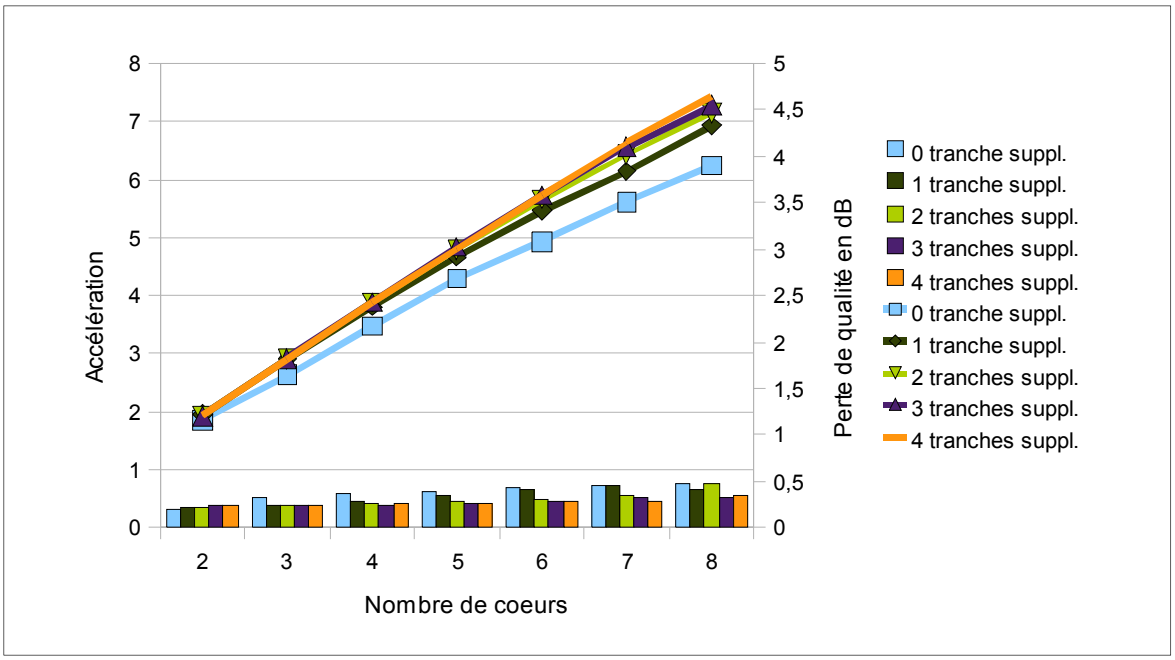


Figure-A XII-4 Accélération et perte de qualité du mode 2 en fonction du nombre de tranches supplémentaires et du nombre de cœurs pour un débit de 20 Mbit/s.

ANNEXE XIII

PERFORMANCES DU TROISIÈME MODE EN FONCTION DU NOMBRE DE CŒURS ET DU NOMBRE DE TRANCHES SUPPLÉMENTAIRES

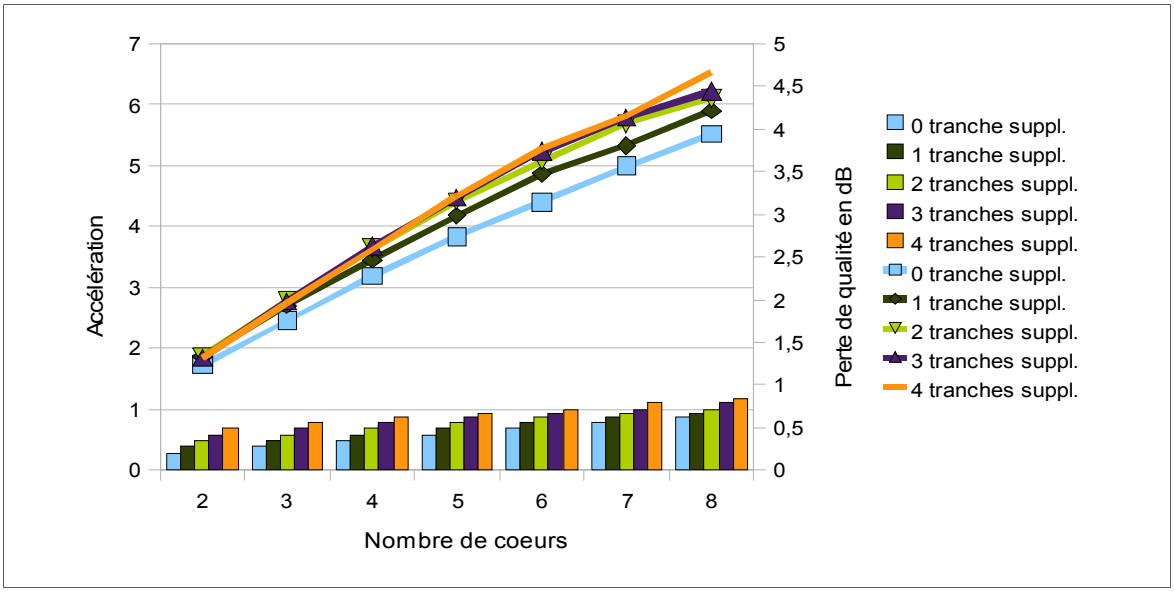


Figure-A XIII-1 Accélération et perte de qualité du mode 3 en fonction du nombre de tranches supplémentaires et du nombre de cœurs pour un débit de 1 Mbit/s.

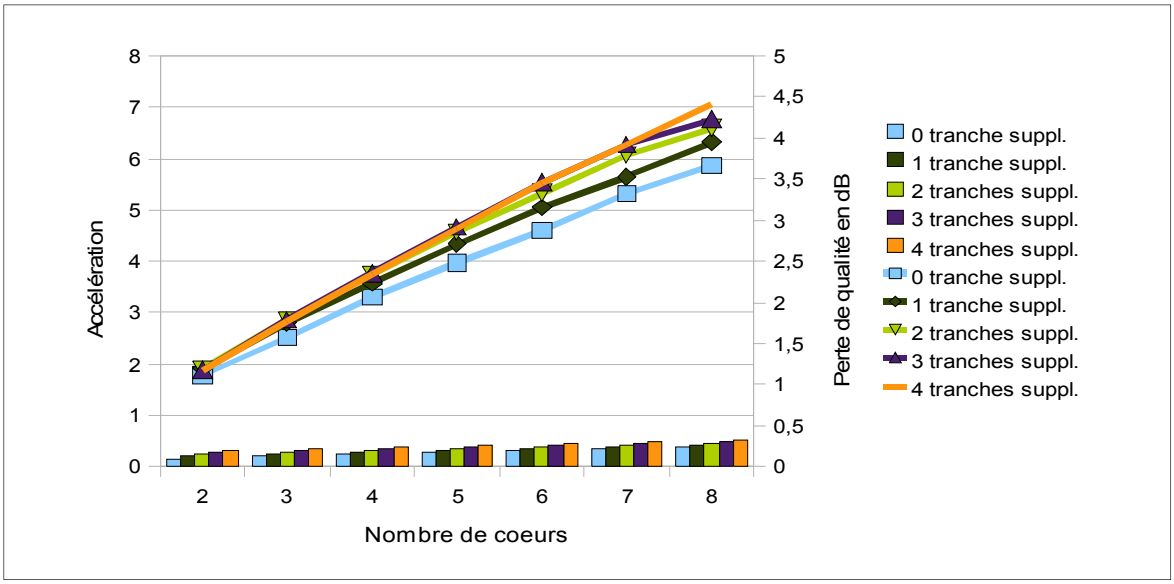


Figure-A XIII-2 Accélération et perte de qualité du mode 3 en fonction du nombre de tranches supplémentaires et du nombre de cœurs pour un débit de 5 Mbit/s.

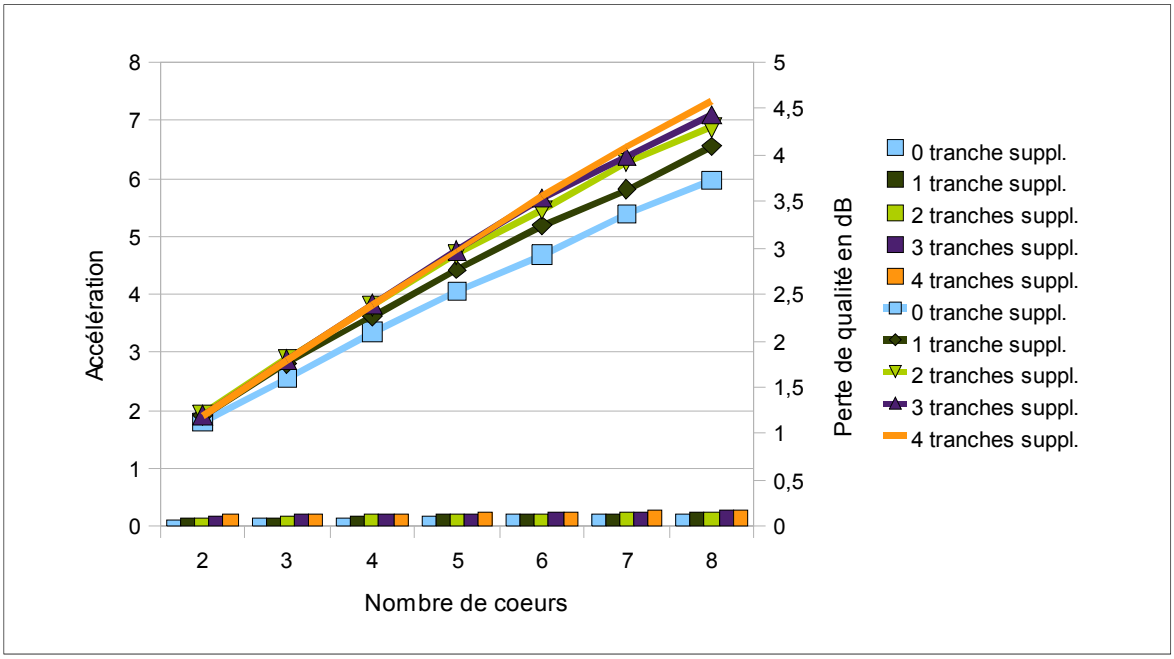


Figure-A XIII-3 Accélération et perte de qualité du mode 3 en fonction du nombre de tranches supplémentaires et du nombre de cœurs pour un débit de 15 Mbit/s.

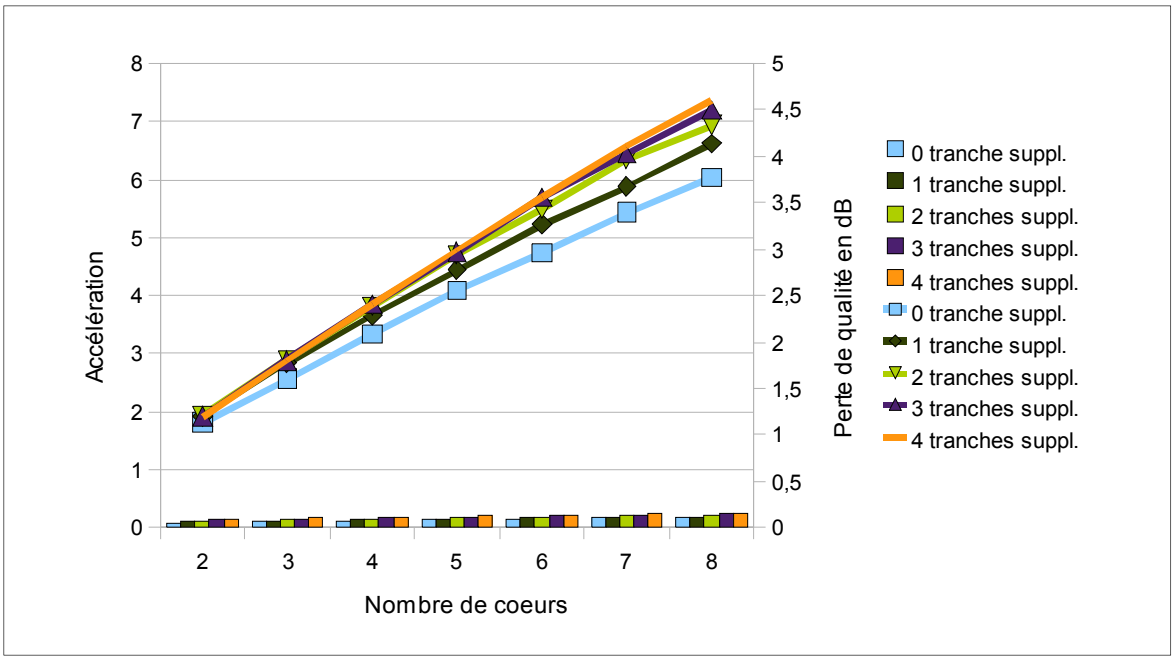


Figure-A XIII-4 Accélération et perte de qualité du mode 3 en fonction du nombre de tranches supplémentaires et du nombre de cœurs pour un débit de 20 Mbit/s.

ANNEXE XIV

COMPARAISON DE PERFORMANCE LES TROIS MODES MTMT PROPOSÉS

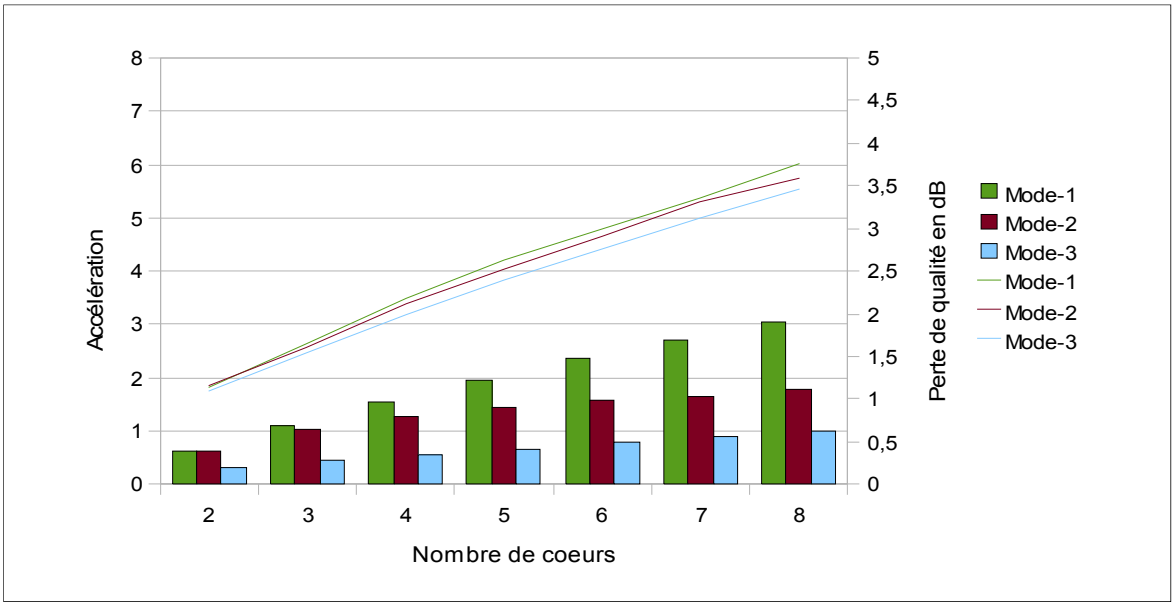


Figure-A XIV-1 Accélération et perte de qualité des modes 1, 2 et 3, en fonction du nombre de cœurs, pour 0 tranches supplémentaires et un débit de 1 Mbit/s.

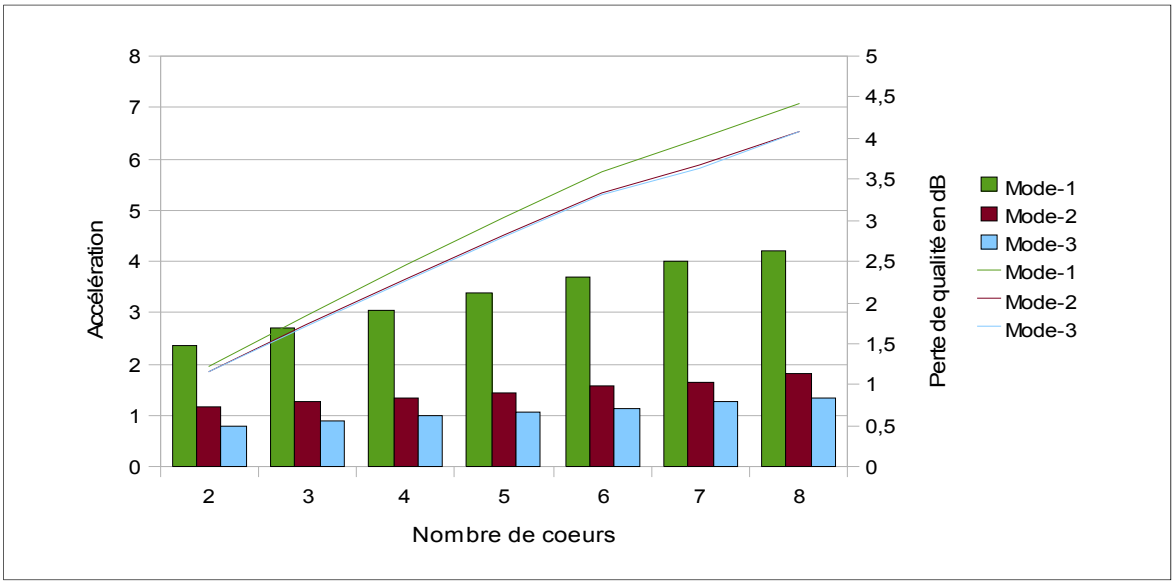


Figure-A XIV-2 Accélération et perte de qualité des modes 1, 2 et 3, en fonction du nombre de cœurs, pour 4 tranches supplémentaires et un débit de 1 Mbit/s.

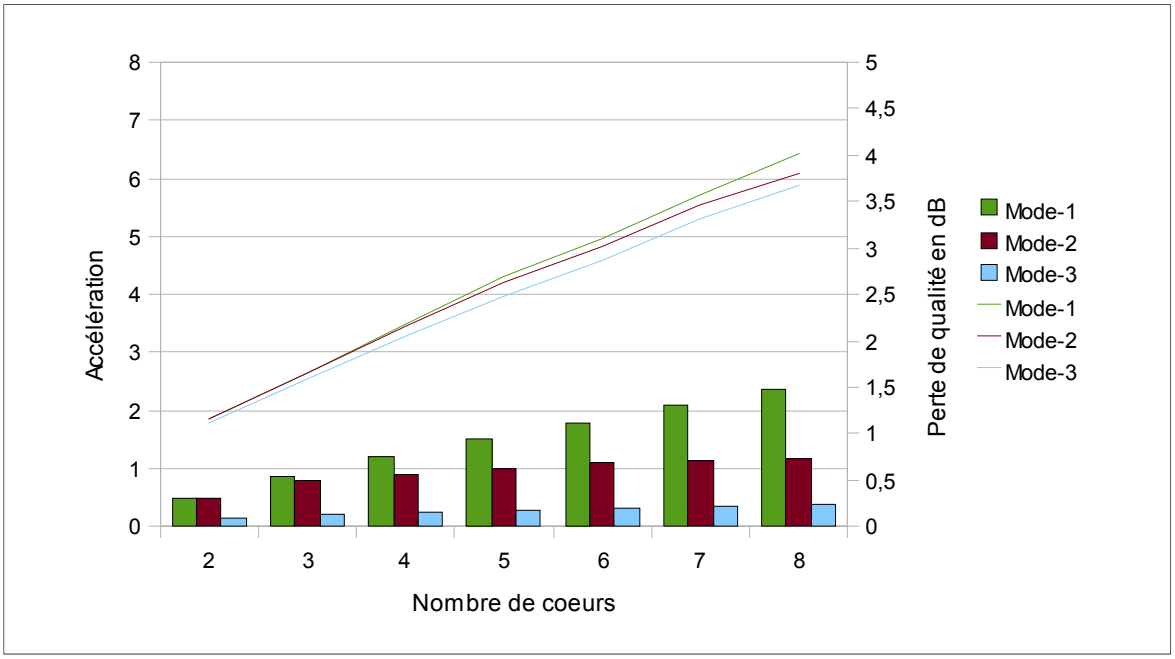


Figure-A XIV-3 Accélération et perte de qualité des modes 1, 2 et 3, en fonction du nombre de cœurs, pour 0 tranches supplémentaires et un débit de 5 Mbit/s.

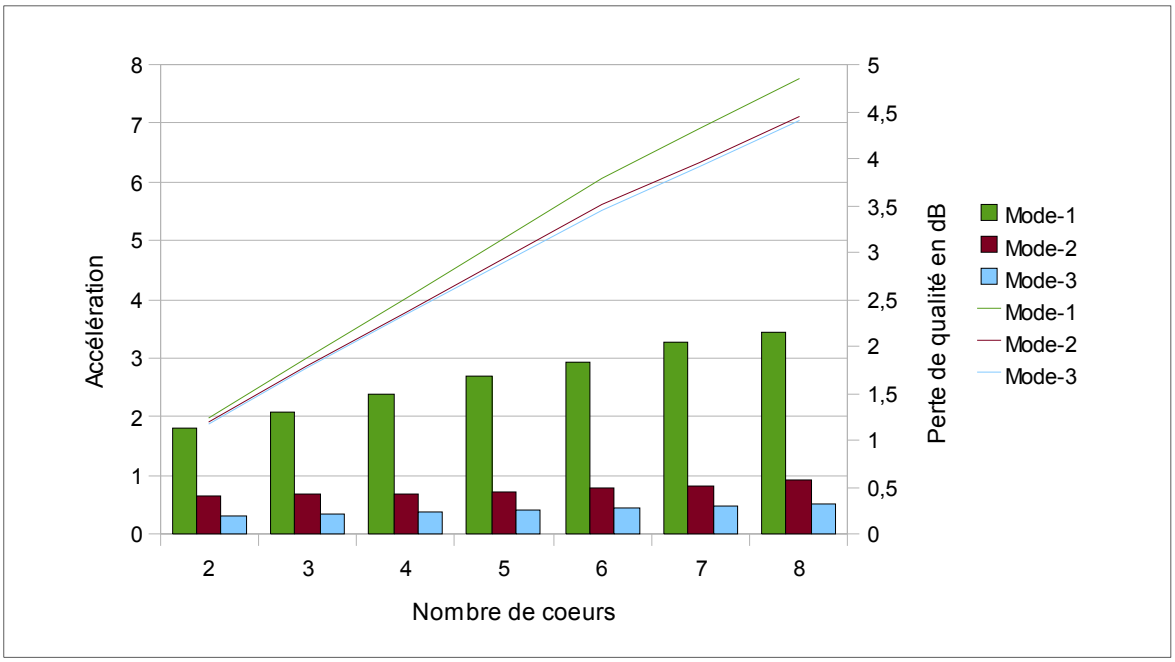


Figure-A XIV-4 Accélération et perte de qualité des modes 1, 2 et 3, en fonction du nombre de cœurs, pour 4 tranches supplémentaires et un débit de 5 Mbit/s.

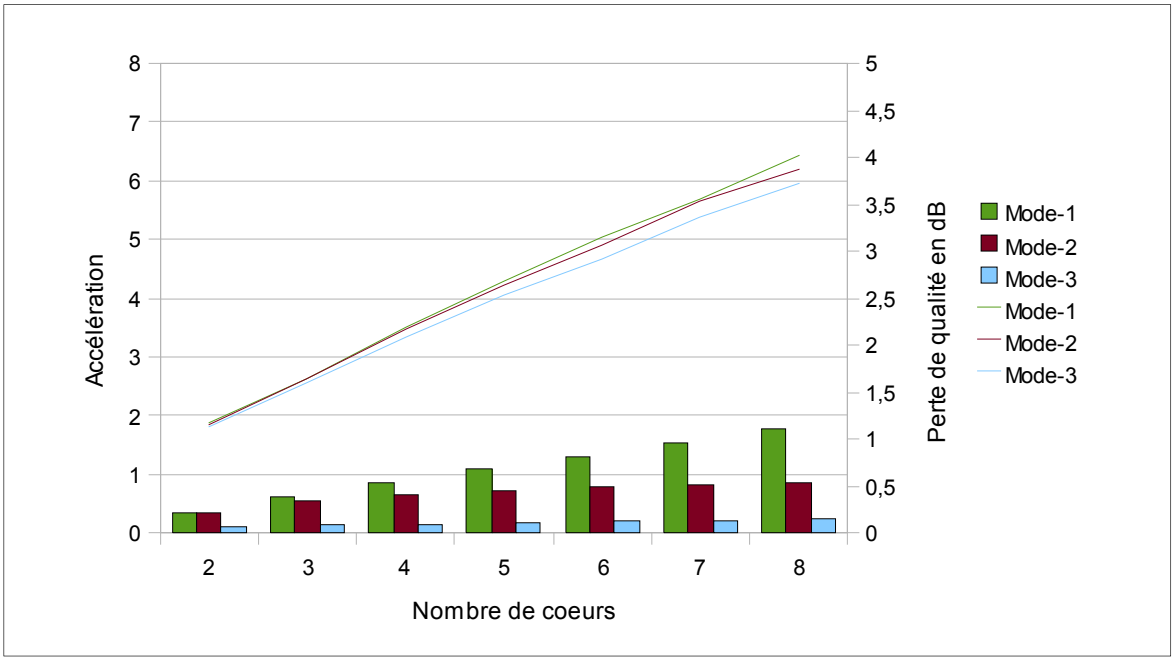


Figure-A XIV-5 Accélération et perte de qualité des modes 1, 2 et 3, en fonction du nombre de cœurs, pour 0 tranches supplémentaires et un débit de 15 Mbit/s.

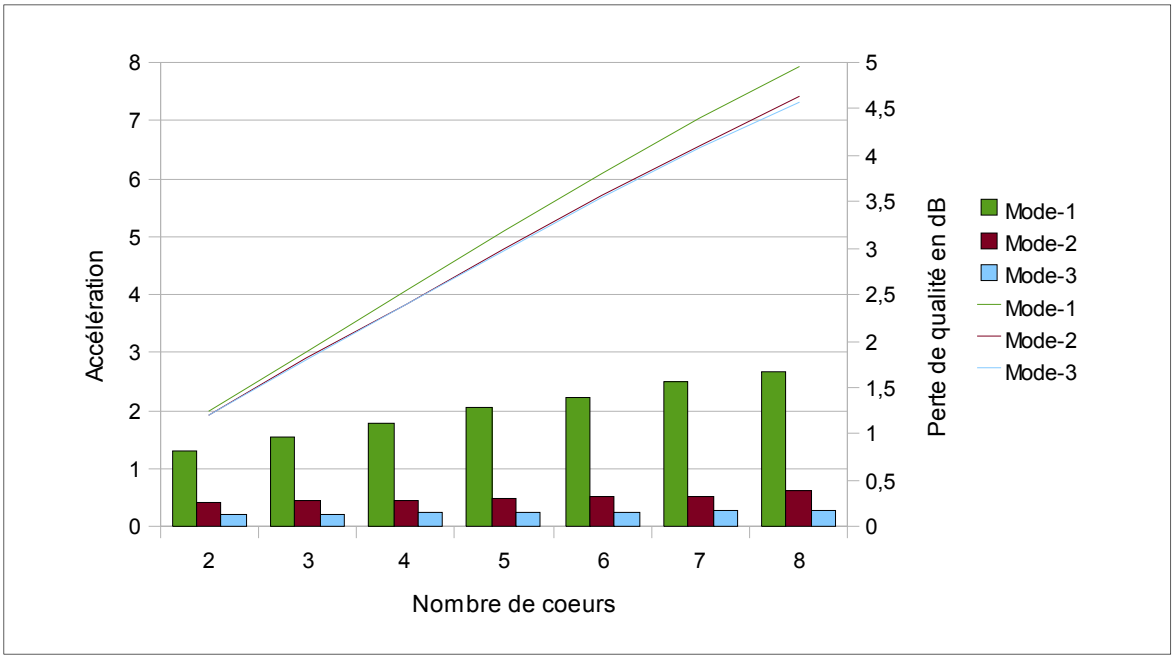


Figure-A XIV-6 Accélération et perte de qualité des modes 1, 2 et 3, en fonction du nombre de cœurs, pour 4 tranches supplémentaires et un débit de 15 Mbit/s.

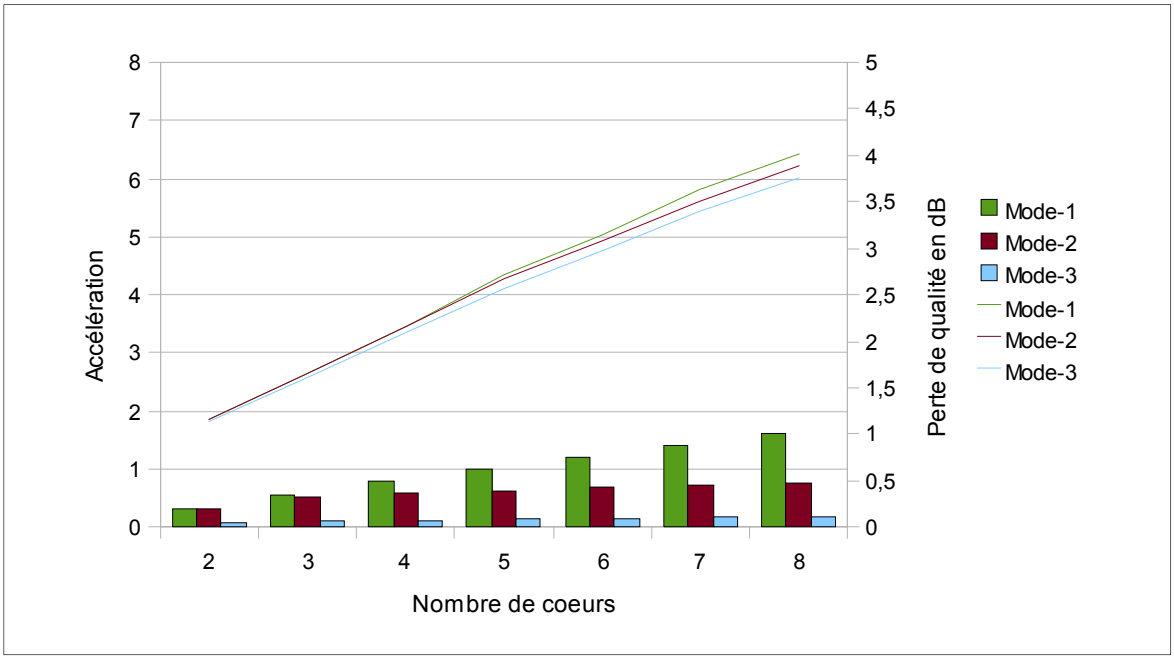


Figure-A XIV-7 Accélération et perte de qualité des modes 1, 2 et 3, en fonction du nombre de cœurs, pour 0 tranches supplémentaires et un débit de 20 Mbit/s.

LISTE DE RÉFÉRENCES BIBLIOGRAPHIQUES

Akhter, S, et J Roberts. 2006. *Multi-core programming: increasing performance through software multi-threading*. Intel Press, 336 p.

Amdahl, GM. 1967. « Validity of the single processor approach to achieving large scale computing capabilities ». In *AFIPS '67 (Spring) Proceedings of the April 18-20, 1967, spring joint computer conference* p. 483-485. ACM.

ARM. 2010. « Neon ». <http://www.arm.com/products/processors/technologies/neon.php>. Consulté le 8 janvier 2011.

Barney, Blaise. 2010. « POSIX threads programming ». https://computing.llnl.gov/tutorials/pthreads/>. Consulté le 8 janvier 2011.

Butenhof, David R. 1997. *Programming with POSIX threads*. Coll. « Addison-Wesley professional computing series ». Reading, Mass.: Addison-Wesley, 381 p.

Chen, YK, EQ Li, X Zhou et S Ge. 2006. « Implementation of H. 264 encoder and decoder on personal computers ». *Journal of Visual Communication and Image Representation,* vol. 17, n^{o} 2, p. 509-532.

Diefendorff, K, PK Dubey, R Hochsprung et H Scales. 2000. « AltiVec extension to PowerPC accelerates media processing ». *IEEE Micro,* vol. 20, n^{o} 2, p. 85-95.

Flynn, M. J. 1966. « Very high-speed computing systems ». *Proceedings of the IEEE,* vol. 54, n^{o} 12, p. 1901-1909.

Foster, Ian. 1995. *Designing and building parallel programs : concepts and tools for parallel software engineering*. Reading, Mass.: Addison-Wesley, 381 p.

Gu, R, JW Janneck, SS Bhattacharyya, M Raulet, M Wipliez et W Plishker. 2009. « Exploring the concurrency of an MPEG RVC decoder based on dataflow program analysis ». *Circuits and Systems for Video Technology, IEEE Transactions on,* vol. 19, n^{o} 11, p. 1646-1657.

Gustafson, JL. 1988. « Reevaluating Amdahl's law ». *Communications of the ACM,* vol. 31, n^{o} 5, p. 532-533.

Hennessy, JL, DA Patterson et D Goldberg. 2003. *Computer architecture: a quantitative approach*. Morgan Kaufmann, 1136 p.

Hennessy, John L., David A. Patterson, David Goldberg et Krste Asanovic. 2003. *Computer architecture : a quantitative approach*, 3rd. San Francisco: Morgan Kaufmann Publishers, 1136 p. <http://www.mkp.com/CA3/>.

Huffman, DA. 1952. « A method for the construction of minimum-redundancy codes ». *Proceedings of the IRE,* vol. 40, n° 9, p. 1098-1101.

IDC. 2006. « Processor data: IDC Worldwide PC Semiconductor 2006-2011 Market forecast ».

Intel. 1999. *Intel Architecture Software Developer's Manual* Volume 2: Instruction Set Reference Manual. Intel. <http://developer.intel.com/design/pentiumii/manuals/243191.htm>.

Intel. 2010a. « Intel Vtune - Intel Software Network ». <http://software.intel.com/en-us/intel-vtune/>. Consulté le 8 janvier 2011.

Intel. 2010b. « Intel® Integrated Performance Primitives 6.1 – Code Samples ». <http://software.intel.com/en-us/articles/intel-integrated-performance-primitives-code-samples/>.

ISO/IEC 11172. 1993. *Information technology – coding of moving pictures and associated audio for digital storage media at up to about 1.5 Mbit/s*.

ISO/IEC 13818. 1995. *Information technology: generic coding of moving pictures and associated audio information*.

ISO/IEC 14496-2. 2001. *Coding of Audio-Visual Objects – Part 2:Visual*.

ITU-T Recommendation H.261. 1993. *Video CODEC for audiovisual services at px64 kbit/s*.

ITU-T Recommendation H.263. 1998. *Video coding for low bit rate communication*.

Jain, J, et A Jain. 2002. « Displacement measurement and its application in interframe image coding ». *Communications, IEEE Transactions on,* vol. 29, n° 12, p. 1799-1808.

Jung, B, et B Jeon. 2008. « Adaptive slice-level parallelism for H. 264/AVC encoding using pre macroblock mode selection ». *Journal of Visual Communication and Image Representation,* vol. 19, n° 8, p. 558-572.

Karczewicz, M, et R Kurceren. 2001. « A proposal for SP-frames ». In *ITU-T. Video Coding Experts Group Meeting, Eibsee, Germany, Jan. 09–12,. 2001*.

Li, ZN, et MS Drew. 2004. *Fundamentals of multimedia*. Prentice hall, 576 p.

List, P., A. Joch, J. Lainema, G. Bjontegaard et M. Karczewicz. 2003. « Adaptive deblocking filter ». *Circuits and Systems for Video Technology, IEEE Transactions on,* vol. 13, n° 7, p. 614-619.

Luo, C, J Sun et Z Tao. 2008. « The Research of H. 264/AVC Video Encoding Parallel Algorithm ». In., p. 201-205. IEEE.

Luthra, A., et P. Topiwala. 2003. « Overview of the H.264/AVC video coding standard ». *Applications of Digital Image Processing Xxvi,* vol. 5203, p. 417-431.

Marr, DT, F Binns, DL Hill, G Hinton, DA Koufaty, JA Miller et M Upton. 2002. « Hyper-threading technology architecture and microarchitecture ». *Intel Technology Journal,* vol. 6, n° 1, p. 4-15.

Mattson, T, B Sanders et B Massingill. 2004. *Patterns for parallel programming*. Addison-Wesley Professional, 384 p.

Merrit, Rick. 2008. « CPU designers debate multi-core future ». *EE Times*, (2/6/2008).

OpenMP Architecture Review Board. 2010. « The OpenMP API specification for parallel programming ». <http://openmp.org/wp/>. Consulté le 8 janvier 2011.

Peleg, A, et U Weiser. 1996. « MMX Technology Extension to the Intel ». *IEEE Micro,* vol. 272, p. 96.

Qiang, Peng, et Zhao Yulin. 2003. « Study on parallel approach in H.26L video encoder ». In *Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference on*. p. 834-837. <10.1109/PDCAT.2003.1236426>.

Ramanathan, RM, P Contributors, R Curry, S Chennupaty, RL Cross, S Kuo et MJ Buxton. 2006. « Extending the world's most popular processor architecture ». *Technology@ Intel Magazine*.

Rec, ISO/IEC 14496-10 and ITU-T. 2003. *H.264, Advanced Video Coding*.

Reinders, James. 2007. *Intel threading building blocks : outfitting C++ for multi-core processor parallelism*, 1st. Beijing ; Sebastopol, CA: O'Reilly, 303 p.

Richardson, Iain E. G. 2003. *H.264 and MPEG-4 video compression : video coding for next generation multimedia*. Chichester ; Hoboken, NJ: Wiley, 281 p.

Rodriguez, A, A Gonzalez et MP Malumbres. 2006. « Hierarchical parallelization of an h. 264/avc video encoder ». In., p. 363-368.

Singhal, R. 2008. « Inside intel next generation nehalem microarchitecture ». In NOTUR2009.

Steven, Ge, Tian Xinmin et Chen Yen-Kuang. 2003. « Efficient multithreading implementation of H.264 encoder on Intel hyper-threading architectures ». In *Information, Communications and Signal Processing, 2003 and the Fourth Pacific Rim Conference on Multimedia. Proceedings of the 2003 Joint Conference of the Fourth International Conference on*. Vol. 1, p. 469-473 Vol.1. <10.1109/ICICS.2003.1292496>.

Stokes, J. 2007. *Inside the machine: An illustrated introduction to microprocessors and computer architecture*. No Starch Press, 320 p.

Tuck, N., et D. M. Tullsen. 2003. « Initial observations of the simultaneous multithreading Pentium 4 processor ». In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*. p. 26-34. <10.1109/PACT.2003.1237999>.

Tullsen, D. M., S. J. Eggers et H. M. Levy. 1995. « Simultaneous multithreading: Maximizing on-chip parallelism ». In *Computer Architecture, 1995. Proceedings. 22nd Annual International Symposium on*. p. 392-403. <10.1109/ISCA.1995.127246>.

Wang, L; Xu, X. . 2007. « Parallel Software Development with Intel® Threading Analysis Tools ». *Intel Technology Journal*.

Wang, Y, J Ostermann et YQ Zhang. 2002. *Video processing and communications*. Prentice Hall New Jersey, 595 p.

Witten, IH, RM Neal et JG Cleary. 1987. « Arithmetic coding for data compression ». *Communications of the ACM,* vol. 30, n^{o} 6, p. 520-540.

Xiao, F. 2000. « DCT-based video quality evaluation ». *Final Project for EE392J*.

Xiph.org. 2010. « Test Media ». <http://media.xiph.org/video/derf/>. Consulté le 8 janvier 2011.

Yen-Kuang, Chen, X. Tian, Ge Steven et M. Girkar. 2004. « Towards efficient multi-level threading of H.264 encoder on Intel hyper-threading architectures ». In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*. p. 63. <10.1109/IPDPS.2004.1302990>.

Yung, N. H. C., et K. C. Chu. 1998. « Fast and parallel video encoding by workload balancing ». In *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on*. Vol. 5, p. 4642-4647 vol.5. <10.1109/ICSMC.1998.727584>.

Zhou, Wang, et A. C. Bovik. 2002. « A universal image quality index ». *Signal Processing Letters, IEEE,* vol. 9, n^{o} 3, p. 81-84.

Zhou, Wang, A. C. Bovik, H. R. Sheikh et E. P. Simoncelli. 2004. « Image quality assessment: from error visibility to structural similarity ». *Image Processing, IEEE Transactions on,* vol. 13, n^{o} 4, p. 600-612.