Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**
UNIVERSITÉ DE FRANCHE-COMTÉ

Algorithmes rapides pour le traitement des images bruitées sur GPU

Gilles PERROT

SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**

UNIVERSITÉ DE FRANCHE-COMTÉ

N° | X | X | X |

THÈSE présentée par

Gilles PERROT

pour obtenir le

Grade de Docteur de
l'Université de Franche-Comté

Spécialité : **Informatique**

Algorithmes rapides pour le traitement des images bruitées sur GPU

Unité de Recherche :
Institut FEMTO-ST, département DISC

Soutenue le 17 novembre 2013 devant le Jury :

Incroyable HULK	Rapporteur	Professeur à l'Université de Gotham City Commentaire secondaire
Super MAN	Examinateur	Professeur à l'Université de Gotham City
Bat MAN	Directeur de thèse	Professeur à l'Université de Gotham City

Sommaire

1 Introduction **9**

2 Les processeurs graphiques (GPU) NVidia® **13**

2.1 Pourquoi ? 13

2.2 Comment ? 14

2.2.1 Le matériel 14

2.2.2 Le logiciel 16

2.2.3 L'occupancy 17

3 Modèles d'image et de bruits - notations **23**

3.1 Modèle d'image bruitée 23

3.2 Modèles de bruit 23

3.2.1 Le bruit gaussien 23

3.2.2 Le speckle 24

3.2.3 Le bruit « sel et poivre » 24

3.2.4 Le bruit de Poisson 25

4 Les techniques de réduction de bruit **27**

4.1 Les techniques de réduction de bruit 27

4.1.1 Les opérateurs de base 28

4.1.1.1 Le filtre de convolution 28

4.1.1.2 Le filtre médian 29

4.1.1.3 Le filtre bilatéral 30

4.1.1.4 Les algorithmes de filtrage par dictionnaire 30

4.1.2 Les algorithmes de filtrage par patches 32

4.2 Les implémentations sur GPU des algorithmes de filtrage 33

4.2.1 Le filtrage par convolution 33

4.2.2 Le filtre médian 34

4.2.3 Le filtre bilatéral 35

4.2.4 Les filtres par patches 37

5 Les techniques de segmentation des images **39**
5.1 Les techniques de segmentation . . . 39
5.1.1 Analyse d'histogramme . . . 40
5.1.2 Partitionnement de graphe . . . 40
5.1.3 kernel-means, mean-shift et apparentés . . . 43
5.1.4 Les contours actifs, ou snakes . . . 45
5.1.5 Méthodes hybrides . . . 47
5.2 Les implémentations des techniques de segmentation sur GPU . . . 47
5.2.1 Calcul d'histogramme . . . 48
5.2.2 Partitionnement de graphe . . . 48
5.2.3 K-means, mean-shift et apparentés . . . 49
5.2.4 Level set et snakes . . . 52
5.2.5 Algorithmes hybrides . . . 54
5.3 Conclusion . . . 55

6 La segmentation par snake polygonal orienté régions **59**
6.1 Introduction . . . 59
6.2 Présentation de l'algorithme . . . 59
6.2.1 Formulation . . . 59
6.2.2 Optimisation des calculs . . . 60
6.2.3 Implémentation séquentielle . . . 62
6.2.4 Performances . . . 65
6.3 Implémentation parallèle GPU du snake polygonal . . . 68
6.3.1 Pré-calculs des images cumulées . . . 69
6.3.2 Calcul des contributions des segments . . . 71
6.3.2.1 Cas particulier des segments dont la pente k vérifie $|k| \leq 1$ 74
6.3.3 Performances . . . 74
6.3.4 Détermination intelligente du contour initial . . . 76
6.3.5 Conclusion . . . 77

7 Réduction de bruit par recherche des lignes de niveaux **81**
7.1 Introduction . . . 81
7.2 Présentation de l'algorithme . . . 82
7.2.1 Formulation . . . 82
7.2.1.1 Isolines à un seul segment . . . 82

7.2.1.2 Isolines composées de plusieurs segments - critère d'allongement . . . 84
7.3 Modélisation des isolines pour l'implémentation parallèle sur GPU . . . 85
7.3.1 Isolines évaluées semi-globalement . . . 86
7.3.2 Isolines à segments pré-évalués - modèle PI-PD . . . 88
7.3.3 Modèle PI-PD hybride . . . 90
7.3.3.1 Le détecteur de zone à faible pente . . . 91
7.4 Résultats . . . 94
7.5 Extension aux images couleurs . . . 96
7.5.1 Expression du critère . . . 96
7.5.2 Résultats . . . 97
7.6 Conclusion . . . 99

8 Le filtre médian sur GPU 103

8.1 Introduction . . . 103
8.2 Les transferts de données . . . 103
8.3 Utilisation des registres . . . 104
8.3.1 La sélection de la valeur médiane . . . 105
8.3.2 Masquage des latences . . . 107
8.4 Résultats . . . 109
8.5 Conclusion . . . 111

9 Les filtres de convolution sur GPU 113

9.1 Introduction . . . 113
9.2 Implémentation générique de la convolution non séparable sur GPU . . . 113
9.3 Implémentation optimisée de la convolution non séparable sur GPU . . . 115
9.4 Cas de la convolution séparable . . . 119
9.5 Conclusion . . . 124

10 Conclusion générale 127

1

INTRODUCTION

Le traitement d'image n'est pas issu des technologies numériques. C'est pourtant l'idée qui peut venir à l'esprit lorsqu'on évoque cette discipline, tant elle a bénéficié du développement récent de ce que l'on nomme les « nouvelles technologies ». C'est aussi dans ce cadre que s'inscrivent les travaux présentés ici et qui effectuent des traitements rapides d'images numériques.

N'oublions pas, toutefois, que l'origine en est notre propre perception visuelle, qui elle-même procède à une interprétation des données brutes recueillies par l'œil. Au delà de l'œil lui-même, notre cerveau génère ou extrait de précieuses informations sur ce qui nous entoure : couleur, forme, volume, texture, mais aussi position, vitesse et accélération relatives à notre propre mouvement. Ces images de la réalité donnent à leur tour naissance à des représentations simplifiées, des modèles, de cette perception immédiate et qui ont permis de faire naître la lecture et l'écriture, au travers de la reconnaissance des signes qui la composent et donnent le sens au texte écrit. Mais ces représentations interviennent aussi dans les diverses expressions picturales dont la pratique est presque aussi ancienne que l'Homme.

Les avènements de la photographie, puis de l'électricité avec la radio et télévision ont toutefois ouvert des voies nouvelles au traitement technique de l'information, dont les applications à l'image forment un des domaines aux contours et méthodes particuliers.

Durant une période que l'on peut juger, *a posteriori*, assez courte, la chaîne d'imagerie électronique fut entièrement analogique, de la caméra vidéo à l'écran cathodique par exemple, la télévision étant un des moteurs des progrès techniques dans le domaine. Peut-être qu'aujourd'hui, à l'époque du numérique roi, les jeunes gens ne se le figurent pas, mais de nombreuses opérations étaient déjà appliquées à ces signaux analogiques qui véhiculaient les images. Parmi les tous premiers furent les traitements visant à améliorer la qualité visuelle, c'est-à-dire à réduire l'effet des perturbations imputables aux dispositifs d'acquisition ou bien ajuster les niveaux de luminosité et contraste d'une prise de vue effectuée dans des conditions difficiles. Bien d'autres, plus complexes, furent aussi maîtrisés comme l'incrustation d'éléments synthétiques ou le codage permettant de réserver la visualisation des images aux détenteurs d'un décodeur adapté.

Naturellement, la représentation numérique des signaux a fait émerger de nouvelles perspectives que l'on ne cerne vraisemblablement que partiellement, pour en vivre les évolutions au jour le jour. Au cœur de ces techniques récentes, l'informatique a déjà permis le développement d'opérations très complexes, quasi irréalisables en analogique, mais aujourd'hui mises à la portée de tout possesseur d'ordinateur personnel : citons en exemple la segmentation, capacité à distinguer différentes zones d'une image, ou bien la recon-

naissance de formes qui intervient, entre autres, dans l'interprétation de texte manuscrit, ou encore la poursuite (*tracking*) permettant de suivre l'évolution d'objets en mouvement dans une séquence d'images.

Malgré tout, même si certains algorithmes sont capables d'opérations impossibles à la vision humaine, comme par exemple l'extraction d'information dans des images très fortement bruitées, d'autres, comme la segmentation, demeurent très difficiles à automatiser alors que notre cerveau semble les effectuer sans effort.

Les images numérisées, par opposition aux images numériques de synthèse, partagent la caractéristique d'être naturellement altérées par des *bruits* de nature et d'intensité variables trouvant leurs origines dans les dispositifs d'acquisition et l'éclairage des scènes. La technologie des capteurs (CMOS), l'accroissement de leur densité en pixels et l'augmentation des fréquences de balayage concourent d'ailleurs à en intensifier les effets, justifiant la recherche de méthodes de réduction de bruit adaptées.

De nombreuses solutions ont été proposées et expérimentées au fil des années, mais à ce jour, aucune ne s'est imposée comme universelle par ses propriétés ou son domaine d'application, tant les caractéristiques des images et des perturbations sont variées.

Par ailleurs, aucune ne fait sauter le verrou du compromis qualité - vitesse et ainsi les méthodes qui génèrent des résultats de grande qualité ne peuvent pas être utilisées de manière interactive du fait de leur temps de calcul trop long. Paradoxalement, ce type de limitation ne se résout pas automatiquement et systématiquement avec l'accroissement régulier des capacités de calcul des ordinateurs car la résolution des capteurs grandit en suivant une courbe quasi identique. La recherche de performance dans le traitement des images doit donc continuer de diversifier les voies qu'elle emprunte pour faire émerger des structures adaptées. Dans ce contexte, répondant à la demande du grand public pour un rendu toujours plus réaliste des jeux vidéos, les fabricants d'adaptateurs graphiques (cartes graphiques) ont progressivement étendu les capacités de leurs produits pour les doter de moyens de traitement très spécialisés mais pouvant être exécutés à grande vitesse, permettant d'envisager la gestion en temps réel de flux vidéos en haute définition, ce que le processeur central d'un ordinateur personnel ne peut, le plus souvent, pas garantir. Ces cartes graphiques GPU (Graphical Processing Unit) sont ainsi devenues de véritables assistants processeurs graphiques, ou coprocesseurs.

La technologie de fabrication des GPUs est identique à celle utilisée pour la fabrication des microprocesseurs classiques qui pilotent nos ordinateurs (CPU) : il s'agit de la photolithogravure de motifs sur un substrat de silicium. Les éléments fonctionnels de base de ces motifs sont des transistors CMOS (Complementary Metal Oxyde semiconductor) mesurant aujourd'hui $22\ nm$. Pour parvenir à traiter les flux d'images plus rapidement que ne peuvent le faire les CPUs, les constructeurs de GPU ont organisé différemment les transistors de sorte à disposer d'un grand nombre de petites unités de calcul, chacune avec une faible capacité de stockage individuelle. Ce faisant, les GPUs ne disposent plus de toutes les fonctionnalités de gestion et de contrôle que possèdent les CPUs et doivent donc être employés différemment pour tirer parti de cette structure particulière.

Pour tenter de comprendre l'organisation d'un GPU actuel, il faut s'imaginer des millions d'exécutants (les fils d'exécution ou threads) regroupés en équipes de travail (les blocs), comportant jusqu'à un millier d'exécutant. Au sein d'une même équipe, les exécutants peuvent communiquer entre eux et échanger des données grâce à un espace de stockage proche et rapide. En revanche, la communication inter équipe est difficile et très lente, associée à un espace de stockage, certes volumineux mais dont l'accès est diffi-

cile et dont l'usage s'avère donc pénalisant. Même si cette organisation très particulière est naturellement adaptée aux opérations d'affichage, pour lesquelles elle a été conçue, les chercheurs et développeurs, considérant les débits permis par ces processeurs graphiques, ont très vite voulu profiter de ces performances pour traduire des algorithmes dont les implémentations étaient jusqu'alors trop lentes. L'adaptation à l'architecture des GPUs s'est toutefois avérée souvent délicate et n'a parfois apporté aucune amélioration des performances.

C'est dans ce cadre que s'inscrivent les travaux présentés ici, axés sur la recherche de méthodes performantes pour le traitement sur GPU d'images numériques bruitées. Nos premiers travaux ont porté sur la segmentation, en parallélisant un algorithme existant pour CPU, tout en lui conférant la capacité de traiter des images de plus grande taille (jusqu'à 16 millions de pixels). La présentation détaillée en est faite au chapitre 6. Nous nous sommes ensuite intéressés à la réduction de bruit et avons proposé un algorithme original et adaptable à une grande variété de perturbations, du bruit additif gaussien au bruit multiplicatif gamma. La conception de cet algorithme a été guidée par la volonté de fournir un élément fonctionnel performant sur GPU, permettant une utilisation temps réel tout en apportant une amélioration qualitative par rapport aux filtres rapides simples, ainsi que le montre le chapitre 7 qui lui est consacré. Dans le même esprit, le chapitre 8 décrit notre contribution à l'implémentation du très employé et très étudié filtre médian pour en proposer l'implémentation la plus performante connue à ce jour avec un débit de plus de 1.85 milliards de pixels à la seconde. À cette occasion, nous avons appliqué à l'utilisation des différents types de mémoires du GPU, des principes qui semblent être transposables avec succès à d'autres classes d'algorithmes. Nous l'avons ainsi montré au chapitre 9 pour les filtres de convolutions auxquels nous avons également apporté des performances inégalées sur GPU avec plus de 2 milliards de pixels traités à la seconde (plus de 7 milliards pour le calcul seul).

Ces techniques permettent des gains substantiels en terme de vitesse, mais imposent en contrepartie un style de programmation très inhabituel et fastidieux, ôtant parfois toute versatilité aux blocs fonctionnels ainsi conçus. C'est pourquoi enfin, nous avons développé une application en ligne, accessible à tous et permettant de générer simplement le code source des fonctions souhaitées (médians, convolutions) en sélectionnant les options et valeurs des paramètres adaptés au besoin.

Nous nous attacherons dans la suite à effectuer un survol des techniques de traitement d'images numériques et des principales classes d'algorithmes employés. Ce chapitre sera également l'occasion de définir les notations générales applicables à tout le manuscrit et de présenter les principaux travaux de recherche en lien avec nos contributions. Celles-ci seront détaillées, en commençant par la segmentation orientée régions, suivie par la réduction de bruit sous les formes du filtre contraint par lignes de niveaux, du filtre médian, et enfin des filtres de convolution.

2

LES PROCESSEURS GRAPHIQUES (GPU) NVIDIA®

2.1/ POURQUOI ?

S'il fallait en réduire les raisons à une seule, c'est vraisemblablement la concurrence commerciale et la croissance du marché des jeux vidéos qui a poussé les fabricants de cartes graphiques à une innovation permanente qui a donnée naissance aux GPUs. Le rendu graphique dans ce cadre est une opération exigeante en calcul mais intrinsèquement parallèle car chaque pixel y est traité individuellement. L'amélioration des résolutions a aussi contribué à faire évoluer la nature de ces besoins de *parallèle* à *massivement parallèle*, un écran actuel pouvant comporter plus de 2,5 millions de pixels.

La technologie de fabrication des GPUs étant identique à celle des CPUs, c'est donc au niveau de la répartition des fonctionnalités que les GPUs se distinguent : là où un CPU comporte quelques cœurs de calcul et beaucoup de transistors dédiés à la réalisation de mémoire cache et de contrôle de flux, un GPU présente plusieurs unités comportant chacune une grande quantité de cœurs de calcul ne disposant que de très peu de mémoire cache et des capacités de contrôle rudimentaires, comme l'illustrent les schémas de la figure 2.1.

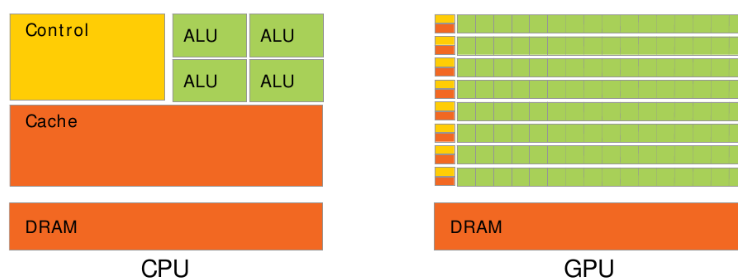
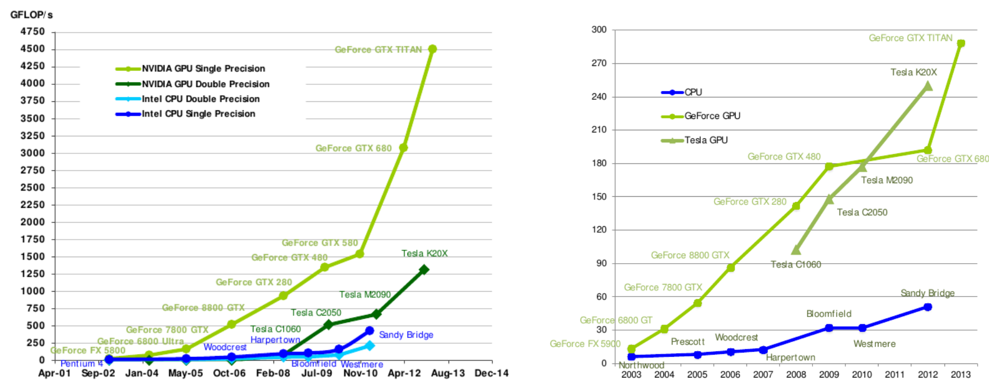


FIGURE 2.1 – Comparaison des structures d'un cœur de GPU et d'un cœur de CPU (d'après [66]). ALU = Arithmetical & Logical Unit.

Cette spécialisation des circuits GPU a permis d'en améliorer les performances brutes beaucoup plus rapidement que pour les CPUs, au fil des évolutions de la technologie. Il en est allé de même pour les débits mémoire théoriques. Les graphiques de la figure 2.2 comparent les rythmes de ces évolutions pour les GPUs Nvidia®et pour les CPUs Intel®.

Les problèmes requérant les capacités de calcul spécifiques des GPUs ne sont cependant pas limités aux questions de rendu graphique, aussi les scientifiques ont-ils très vite

cherché à tirer parti de la puissance de calcul croissante des GPUs pour traiter d'autres types de problèmes, faisant sens à l'acronyme GPGPU (General Purpose Graphical Processing Unit).



(a) Nombre maximum théorique d'opérations en virgule flottante par seconde en fonction de l'année et de l'architecture.

(b) Bande passante théorique maximale des diverses architectures.

FIGURE 2.2 – Comparaison des performances des GPUs Nvidia et des CPU Intel (d'après [66]).

Malgré des caractéristiques prometteuses, les GPGPUs n'ont pas immédiatement déclenché une vague d'expérimentations scientifiques dans des domaines variés, l'essentiel des travaux étant liés à la construction et la visualisation des données issues des instruments d'imagerie médicale.

C'est la parution de l'extension de langage CUDA qui a réellement démocratisé l'emploi des GPGPUs et favorisé l'émergence de travaux variés. CUDA est une extension de haut niveau du langage C permettant d'écrire facilement des fonctions s'exécutant en parallèle sur le GPU, que Nvidia nomme *kernels*. L'extension CUDA permet également de gérer de manière transparente les changements d'échelle du parallélisme d'une architecture à une autre ou tout simplement les dimensions de la grille de calcul.

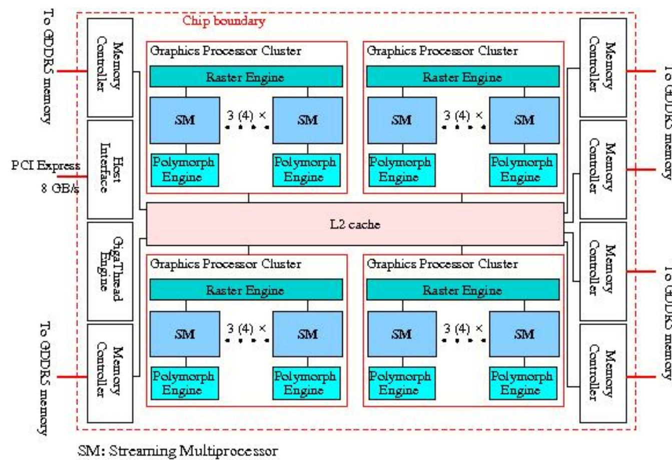
2.2/ COMMENT ?

2.2.1/ LE MATÉRIEL

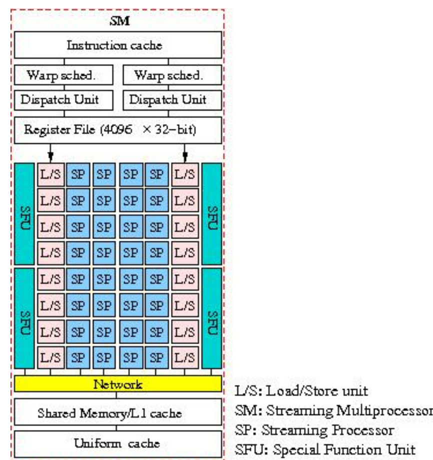
Pour bien tirer parti des capacités des GPUs, il est important d'en comprendre l'organisation matérielle. Nous limiterons cette présentation à l'architecture dite *Fermi* à laquelle appartient le GPU C2070 qui a servi à l'essentiel de nos expérimentations et dont la constitution est détaillée par les diagrammes de la figure 2.3.

Le circuit se divise en 4 groupes de processeurs de flux (les SMs = *Streaming Multiprocessors*), chaque groupe en comprenant 3 ou 4 pour un total de 14. Chaque SM héberge à son tour 32 cœurs de calcul (les SPs = *Streaming Processors*). Les threads sont exécutés par *warps* de 2×16, soit un par cœur de calcul.

Les ressources mémoire sont de nature diverse, tant du point de vue du volume disponible, que de la portée ou du débit. Retenons que les registres, peu nombreux et



(a) Organisation en groupes de SMs



(b) Constitution d'un SM.

FIGURE 2.3 – Organisation des GPUs d'architecture Fermi, comme le C2070 (d'après www.hpcresearch.nl).

embarqués sur le circuit (*on-chip*), présentent les meilleures performances alors que la mémoire principale, dite globale, externe au circuit (*off-chip*) a une grande capacité mais présente des latences importantes de plusieurs centaines de cycles d'horloge. Le fabricant fournit des indications essentiellement qualitatives concernant les latences d'accès aux mémoires, mais ne fournit pas de chiffres sur les latences réelles en fonction des accès et de la proximité du cache lorsqu'un des 3 niveaux est sollicité. Le tableau 2.1 présente une synthèse des caractéristiques du modèle C2070, issue d'expérimentations menées par nos soins à l'aide des micro-tests présentés dans [104].

Une petite quantité de mémoire on-chip est présente sur chaque SM et permet la communication entre les threads s'exécutant sur ce SM. Cette mémoire est appelée *mémoire partagée* et permet des débits bien supérieurs à la mémoire globale.

	Propriétés				
	Emplacement (on/off-chip)	Portée	Latence (cycles)	débit (Go/s)	Taille (octets)
Registres	on-chip	thread	1	8000	32K par SM
Partagée	on-chip	bloc	38	1300	48K
Constante	off-chip	grille	370/46/140	8000	64K
Texture	off-chip	grille	500/260/372	N/C	6G
Locale	off-chip	thread	550	N/C	512K
Globale	off-chip	grille	580/80/350	144	6G

TABLE 2.1 – Caractéristiques des différents types de mémoire disponibles sur le GPU. Pour les mémoires cachées, les latences sont données selon l'accès *sans-cache/L1/L2*. Les mesures ont été obtenues à l'aide des microprogrammes de test de [104].

2.2.2/ LE LOGICIEL

Dans le modèle CUDA, chaque *kernel* est exécuté par un certain nombre de threads. Chaque thread possède un identifiant unique, accessible à l'intérieur du kernel, ce qui permet d'en individualiser le traitement. L'ensemble des threads est organisé en plusieurs blocs indépendants, eux-même rassemblés en une grille. Pour faciliter la représentation de modèles variés, les blocs de threads ainsi que la grille de blocs peuvent être décrits par des tableaux à une, deux ou trois dimensions. Le nombre total de threads appartenant à un même bloc est cependant limité, selon la version de GPU, à 512 ou 1024. Le diagramme de la figure 2.4 illustre l'organisation d'une grille de calcul à 2 dimensions et de ses blocs, également à 2 dimensions, situation qui correspond à la majorité des modèles d'exécution de nos *kernels* de traitement d'image.

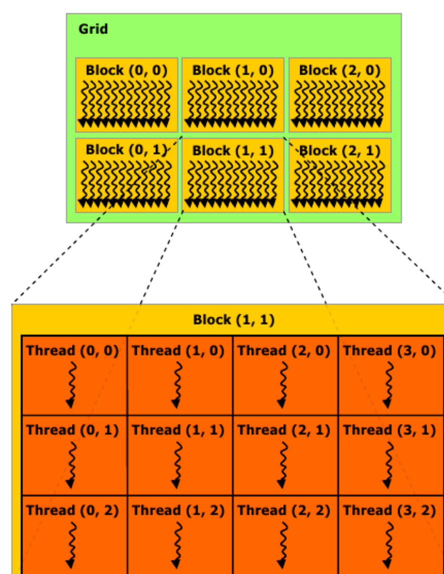


FIGURE 2.4 – Représentation d'une grille de calcul en 2D et des blocs de threads, à 2 dimensions, qui la composent.

Au sein d'un même bloc, les threads peuvent communiquer efficacement entre eux grâce à la mémoire partagée, rapide et présentant une faible latence. En revanche, la communication inter-blocs passe nécessairement par la mémoire globale dont l'emploi est pénalisant.

L'emploi de la mémoire partagée n'est cependant pas transparent comme peut l'être un cache de niveau 1 (L1) et les motifs d'accès doivent respecter un certain nombre de conditions pour obtenir les performances attendues. Le non-respect de ces contraintes conduit le plus souvent à des fragments de code dont l'exécution s'avère plus lente que leurs équivalents CPU.

2.2.3/ L'OCCUPANCY

Pour atteindre les meilleures performances possible, le fabricant Nvidia recommande d'avoir toujours suffisamment de threads dans chaque bloc et de blocs dans la grille et ce, pour masquer les latences des accès aux mémoires, mais aussi celles des instructions arithmétiques. Il définit un indice nommé *occupancy*, que l'on pourrait franciser par *charge* qui représente, à chaque instant, le rapport du nombre de threads actifs par SM sur le nombre maximum de threads actifs par SM (1536 sur C2070).

La valeur de l'*occupancy* peut se trouver limitée par un usage trop intensif des ressources mémoire par les threads (registres, mémoire partagée) ou bien par une grille de calcul mal dimensionnée. L'ensemble des paramètres intervenant dans le calcul de l'*occupancy* est pris en compte dans l'*occupancy calculator*, feuille de calcul fournie par le fabricant pour aider les développeurs à bien employer les ressources des divers modèles de GPU.

Les limitations de l'*occupancy* ont pour origine :

1. **l'usage des registres**. Si chaque thread utilise le maximum de 64 registres possible (63 pour l'utilisateur $+1$ pour le processeur), le bloc de threads affecté au SM ne peut donc activer simultanément que $32K/64 = 512$ threads, soit une *occupancy* de $512/1536 = 0.33$.
2. **l'usage de la mémoire partagée**. L'architecture Fermi permet de choisir la répartition entre cache L1 et mémoire partagée, soit 16K/48K, soit 48K/16K. En configuration 48K de mémoire partagée, si chaque thread en emploie 48 octets, le GPU ne peut activer que $48K/48 = 1024$ threads, soit une *occupancy* de $1024/1536 = 0.66$.
3. **la taille des blocs**. Un SM ne pouvant activer que 8 blocs simultanément, la taille des blocs limite donc potentiellement l'*occupancy*. Si on exécute un *kernel* sur une grille de calcul dont les blocs ont le minimum de 32 threads, les 8 blocs actifs représenteront alors 256 threads, soit une *occupancy* de $256/1536 = 0.16$.

Nous verrons que cette notion d'*occupancy*, si elle conserve du sens, peut toutefois être remise en question en optimisant d'autres aspects permettant d'arriver à une réduction de l'effet des latences, comme le parallélisme d'instructions ou l'augmentation du volume des transactions. En effet, ces techniques, et surtout l'utilisation avisée des différents types de mémoire du GPU permettent d'obtenir des performances élevées, parfois inenvisageables en suivant les prescriptions du constructeur.

Le traitement des images

Introduction

L'étendue des techniques applicables aux images numériques est aujourd'hui si vaste qu'il serait illusoire de chercher à les décrire ici de manière exhaustive. Ce chapitre présente plus spécifiquement les algorithmes utilisés en présence d'images (fortement) bruitées, c'est-à-dire présentant des distorsions par rapport à la réalité « absolue » qu'elles représentent.

Le bruit rend potentiellement délicate l'extraction des informations utiles contenues dans les images perturbées ou en complique l'interprétation, automatisée ou humaine. L'intuition incite donc à chercher des méthodes efficaces de pré-traitement réduisant la puissance du bruit et permettant ainsi aux traitements de plus haut niveau (comme la segmentation), d'opérer dans de meilleures conditions.

Toutefois, il faut également considérer que les opérations préalables de réduction de bruit génèrent des modifications statistiques et peuvent altérer les caractéristiques que l'on cherche à mettre en évidence grâce au traitement principal. En ce sens, il peut être préférable de chercher à employer des algorithmes de haut niveau travaillant directement sur les images bruitées pour en préserver toute l'information, ce qui est le cadre de notre contribution portant sur l'algorithme du *snake* (chapitre 6).

De plus, toute opération supplémentaire si basique soit elle, réduit le temps de traitement disponible pour l'opération de haut niveau. En effet, lorsque les images à analyser sont de grande taille, procéder à un débruitage préalable peut s'avérer incompatible avec les contraintes de débit.

Les images auxquelles nous nous intéressons sont généralement les images numériques allant des images naturelles telles que définies par Caselles [15] aux images d'amplitude issues de l'imagerie radar à ouverture synthétique (ROS ou en anglais SAR) [29], de l'imagerie médicale à ultrasons (échographie) ou de la microscopie électronique. Ces dispositifs d'acquisition sont, par essence, générateurs de bruits divers, inhérents aux technologies mises en œuvre et qui viennent dégrader l'image idéale de la scène que l'on cherche à représenter ou analyser. On sait aujourd'hui caractériser de manière assez précise ces bruits et la section 3.2 en détaille les origines physiques ainsi que les propriétés statistiques qui en découlent. On peut d'ores et déjà avancer que la connaissance de l'origine d'une image et donc des propriétés des bruits associés qui en corrompent les informations, est un atout permettant de concevoir des techniques de filtrage adaptées à chaque situation. Quant à la recherche d'un filtre universel, bien qu'encore illusoire, elle n'est pas abandonnée, tant les besoins sont nombreux, divers et souvent complexes.

3

Modèles d'image et de bruits - notations

3.1/ Modèle d'image bruitée

On considère qu'une image observée, de largeur L pixels et de hauteur H pixels, est un ensemble de $N = LH$ observations sur un domaine Ω à deux dimensions ($\Omega \subset \mathbb{Z}^2$). À chaque élément de Ω, aussi appelé *pixel*, est associé un indice unique $k \in [\![1; N]\!]$, une position $x_k = (i, j)_k \in \Omega$ et une valeur observée $v_k = v(i, j)_k$. La valeur observée peut, selon les cas, être de dimension 1 pour les images représentées en niveaux de gris ou de dimension 3 pour les images couleur représentées au format RVB. Les dimensions supérieures, pour la représentation des images hyperspectrales ne sont pas abordées dans ce manuscrit. L'image observée peut ainsi être considérée comme un vecteur à N éléments $\bar{v} = (v_k)_{k\in[\![1;N]\!]}$. Les divers traitements appliqués aux images observées ont souvent pour but d'accéder aux informations contenues dans une image sous-jacente, débarrassée de toute perturbation, dont nous faisons l'hypothèse qu'elle partage le même support Ω et que nous notons $\bar{u}$. L'estimation de $\bar{u}$ réalisée par ces traitements est notée $\widehat{\bar{u}} = (\widehat{u_k})_{k\in[\![1;N]\!]}$. Le lien entre $\bar{u}$ et $\bar{v}$ peut être exprimé par la relation $\bar{v} = \bar{u} + \sigma\epsilon$, où $\epsilon \in \mathbb{R}^N$ représente le modèle de perturbation appliquée à $\bar{u}$ et σ représente la puissance de cette perturbation qui a mené à l'observation de $\bar{v}$. Dans le cas général, ϵ dépend de $\bar{u}$ et est caractérisé par la densité de probabilité (PDF pour Probability Density Function) $p(v|u)$.

3.2/ Modèles de bruit

3.2.1/ Le bruit gaussien

Le bruit gaussien est historiquement le plus étudié et celui auquel sont dédiées le plus de techniques de débruitage. La génération des images numériques au travers des capteurs CMOS et CCD est le siège de nombreuses perturbations dues à la technologie de fabrication et à la nature du rayonnement dont ils mesurent l'intensité en différents zones de leur surface, appelées *photosites* [62, 94]. On distingue en particulier les bruits suivants selon leur origine physique :

- la non uniformité de réponse des photosites.
- le bruit de photon
- le bruit de courant d'obscurité
- le bruit de lecture

– le bruit de non uniformité d'amplification des photosites.
On trouve des descriptions détaillées des mécanismes concourant à la génération de ces bruits, entre autres dans [45] et [2]. Dans un certain intervalle usuel d'intensité lumineuse, il est toutefois admis que l'ensemble de ces perturbations peut être représenté par un seul bruit blanc gaussien, de type *additif* (AWGN, Additive White Gaussian Noise), dont la densité de probabilité suit une loi normale de moyenne nulle et de variance σ^2. On a alors l'expression suivante, où $\sigma > 0$

$$p(v|u) = \frac{1}{\sqrt{2}\pi\sigma} e^{-\frac{(v-u)^2}{2\sigma^2}}$$

3.2.2/ Le speckle

En imagerie radar, sonar ou médicale, les surfaces que l'on veut observer sont « éclairées » par des sources cohérentes. Les propriétés locales de ces surfaces sont le siège de réflexions multiples qui interfèrent entre elles pour générer un bruit de tavelures, ou speckle, dont l'intensité dépend de l'information contenue dans le signal observé. Le speckle est ainsi un bruit de type *multiplicatif* qui confère aux observations une très grande variance, laquelle peut être réduite, pour une scène donnée, par moyennage de plusieurs observations, ou vues. Si L est le nombre de vues, le speckle est traditionnellement modélisé par la PDF suivante :

$$p(v \mid u) = \frac{L^2 v^{(L-1)} e^{-L\frac{v}{u}}}{\Gamma(L) u^L}$$

L'espérance vaut $\mathrm{E}[v] = u$ et la variance $\sigma^2 = \frac{u^2}{L}$ est effectivement inversement proportionnelle à L, mais pour le cas mono vue où $L = 1$, la variance vaut u^2, soit un écart type du signal v égal à sa moyenne.

3.2.3/ Le bruit « sel et poivre »

Le bruit *sel et poivre*, ou bruit *impulsionnel* trouve son origine dans les pixels défectueux des capteurs ou dans les erreurs de transmission. Il tire son nom de l'aspect visuel de la dégradation qu'il produit : des pixels noirs et blancs répartis dans l'image. Le bruit impulsionnel se caractérise par la probabilité P d'un pixel d'être corrompu. La PDF peut alors être exprimée par parties comme suit, pour le cas d'images en 256 niveaux de gris (8 bits) :

$$p(v \mid u) = \begin{cases} \frac{P}{2} + (1-P) & \text{si } v = 0 \text{ et } u = 0 \\ \frac{P}{2} + (1-P) & \text{si } v = 255 \text{ et } u = 255 \\ \frac{P}{2} & \text{si } v = 0 \text{ et } u \neq 0 \\ \frac{P}{2} & \text{si } v = 255 \text{ et } u \neq 255 \\ (1-P) & \text{si } v = u \text{ et } u \notin \{0, 255\} \\ 0 & \textit{sinon} \end{cases}$$

3.2.4/ LE BRUIT DE POISSON

Aussi appelé *bruit de grenaille* (shot noise), ce type de bruit est inhérent aux dispositifs de détection des photons. Il devient prépondérant dans des conditions de faible éclairement, lorsque la variabilité naturelle du nombre de photons reçus par un photosite par intervalle d'intégration influe sur les propriétés statistiques du signal. Le bruit de grenaille est de type multiplicatif et suit une loi de Poisson. La PDF peut s'écrire comme suit :

$$p(v \mid u) = \mathrm{e}\frac{u^{v}}{v!}$$

4

LES TECHNIQUES DE RÉDUCTION DE BRUIT

4.1/ LES TECHNIQUES DE RÉDUCTION DE BRUIT

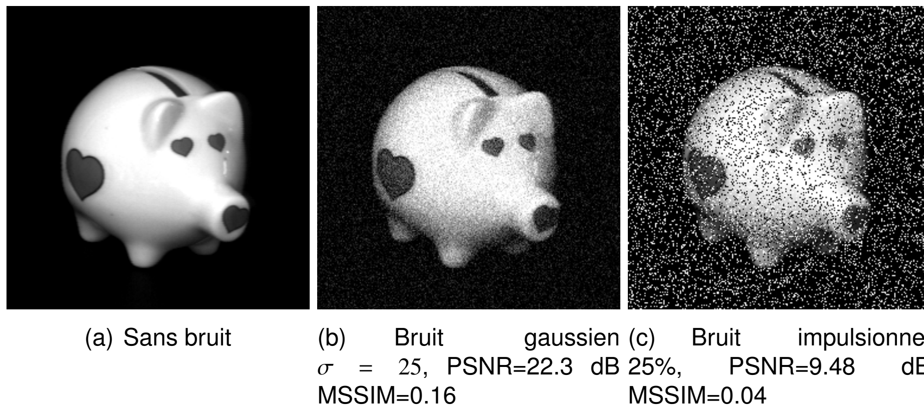
La très grande majorité des algorithmes de réduction de bruit fait l'hypothèse que la perturbation est de type gaussien, même si le développement des systèmes d'imagerie radar et médicale a favorisé l'étude des bruits multiplicatifs du type *speckle* ou *Poisson*. Un très grand nombre de travaux proposant des méthodes de réduction de ces bruits ont été menés, ainsi que beaucoup d'états de l'art et d'études comparatives de ces diverses techniques. Aussi nous focaliserons nous sur les techniques en lien avec les travaux que nous avons menés et qui ont donné lieu à des implémentations efficaces susceptibles de fournir des éléments opérationnels rapides pour le pré-traitement des images.

La figure 4.1 montre une image de synthèse issue de la base de test COIL [65], supposée sans bruit et qui sera considérée comme référence, ainsi que deux versions bruitées, respectivement avec un bruit gaussien d'écart type 25 et un bruit impulsionnel affectant 25% des pixels. L'indice de qualité le plus employé pour mesurer la similarité entre deux images est le PSNR (pour Peak Signal to Noise Ratio). Il est exprimé en décibels (dB) et se calcule en appliquant la formule

$$PSNR = 10log_{10}\left(\frac{D^2}{\frac{1}{N}\sum_{k<N}(v_k - u_k)^2}\right)$$

si l'on cherche à évaluer le PSNR de l'image observée $\bar{v}$ par rapport à l'image de référence $\bar{u}$. Le nombre D représente la dynamique maximale des images, e.g 255 pour des images en niveaux de gris codés sur 8 bits.

Cet indicateur seul est cependant insuffisant pour caractériser convenablement la qualité de débruitage d'un filtre, mesure hautement subjective. Un indice global de similarité structurelle (MSSIM pour Mean Structural Similarity Index) a été proposé par Wang *et al.* [103] et permet, en conjonction avec le PSNR, de garantir une mesure de qualité plus en rapport avec la perception visuelle. Le MSSIM prend ses valeurs dans l'intervalle $[0;1]$ avec une similarité d'autant plus grande que la valeur est proche de 1.



(a) Sans bruit (b) Bruit gaussien $\sigma = 25$, PSNR=22.3 dB MSSIM=0.16 (c) Bruit impulsionnel 25%, PSNR=9.48 dB MSSIM=0.04

FIGURE 4.1 – Images 256×256 en niveau de gris 8 bits utilisées pour l'illustration des propriétés des filtres. a) l'image de référence non bruitée. b) l'image corrompue par un bruit gaussien d'écart type $\sigma = 25$. c) l'image corrompue par un bruit impulsionnel à 25%.

4.1.1/ LES OPÉRATEURS DE BASE

4.1.1.1/ LE FILTRE DE CONVOLUTION

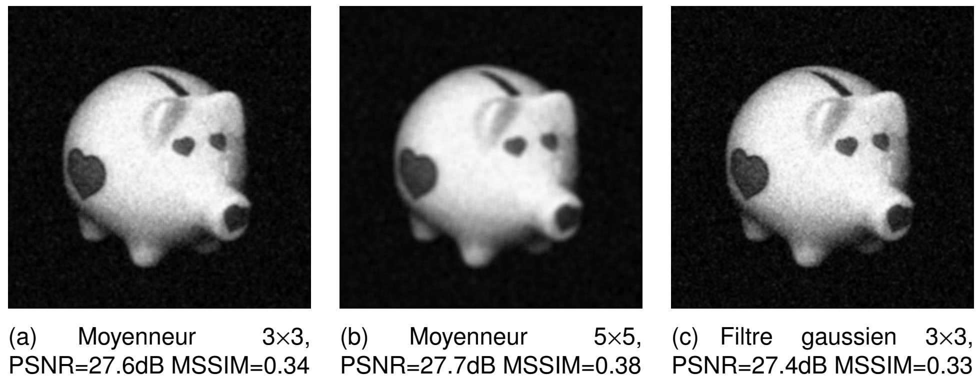
L'opération la plus employée dans les procédés de traitement d'image est sans doute la convolution. Selon les valeurs affectées aux coefficients du masque, le filtrage par convolution permet de réaliser bon nombre de traitements comme la réduction de bruit par moyennage ou noyau gaussien ou encore la détection de contours. Si la fonction définissant le masque de convolution est notée h, l'expression générale de la valeur estimée de pixel de coordonnées (i, j) est donnée par

$$\widehat{u}(x, y) = (\overline{v} * h) = \sum_{(i<H)} \sum_{(j<L)} v(x - j, y - i)h(j, i) \tag{4.1}$$

Dans les applications les plus courantes, h est à support borné et de forme carrée et l'on parle alors de la taille du masque pour évoquer la dimension du support. La figure 4.2 présente les résultats de la convolution par deux masques débruiteurs *moyenneurs* h_3 et h_5 de taille différentes, appliqués à l'image corrompue par un bruit gaussien : on voit la diminution des fluctuations mais aussi le flou apporté et qui rend les contours d'autant moins définis que la taille du masque est grande. La troisième image montre l'effet d'un masque gaussien h_{g3}. Les matrices définissant les masques sont les suivantes :

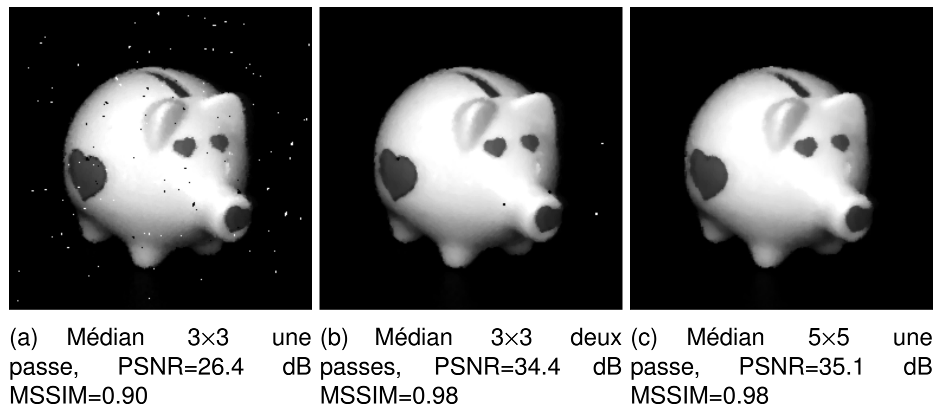
$$h_3 = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}, h_5 = \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}, h_{g3} = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Lorsque la matrice h définissant le masque peut s'écrire comme le produit de deux vecteurs à une dimension h_v et h_h, on parle alors de convolution séparable, qui peut être effectuée en deux étapes distinctes de convolution à une dimension, l'une verticale, l'autre horizontale.



(a) Moyenneur 3×3, PSNR=27.6dB MSSIM=0.34 (b) Moyenneur 5×5, PSNR=27.7dB MSSIM=0.38 (c) Filtre gaussien 3×3, PSNR=27.4dB MSSIM=0.33

FIGURE 4.2 – Filtrage par convolution.



(a) Médian 3×3 une passe, PSNR=26.4 dB MSSIM=0.90 (b) Médian 3×3 deux passes, PSNR=34.4 dB MSSIM=0.98 (c) Médian 5×5 une passe, PSNR=35.1 dB MSSIM=0.98

FIGURE 4.3 – Réduction du bruit impulsionnel par filtre médian.

4.1.1.2/ LE FILTRE MÉDIAN

Le filtrage médian [96] est également une opération très employée en pré-traitement pour sa simplicité et ses propriétés de préservation des contours alliées à une capacité de réduction du bruit gaussien importante. La valeur du niveau de gris de chaque pixel est remplacée par la médiane des niveaux de gris des pixels voisins. Un des intérêts de ce filtre réside dans le fait que la valeur filtrée est une des valeurs du voisinage, contrairement à ce qui se produit lors d'une convolution. Un autre est de bien filtrer les valeurs extrêmes et par conséquent de trouver naturellement son application dans la réduction du bruit impulsionnel. Toutefois, la non-linéraité de cette technique et sa complexité n'en ont pas fait un filtre très utilisé jusqu'à ce que des implémentation efficaces soient proposées, en particulier le filtre à temps de calcul « constant » décrit par Perreault et Hebert [72]. Il est à noter que le filtrage médian est souvent appliqué en plusieurs passes de voisinage restreint. La figure 4.3 montre la réduction de bruit impulsionnel obtenu sur l'image 4.1(c) grâce au filtre médian, dans trois conditions distinctes : médian 3×3 en une ou deux passes, puis médian 5×5.

4.1.1.3/ LE FILTRE BILATÉRAL

Le filtre bilatéral [95] est une composition d'opérations que l'on peut voir comme un filtre de convolution dont les coefficients ne dépendraient pas uniquement de la position du pixel courant par rapport au pixel central, mais également de la différence de leurs intensités (cas des images en niveaux de gris). Si l'on note Ω_k le voisinage du pixel d'indice k, l'expression générale du niveau de gris estimé est donnée par

$$\widehat{u_k} = \frac{\sum_{p \in \Omega_k} \left(F_S(x_p, x_k) F_I(v_p, v_k) v_p \right)}{\sum_{p \in \Omega_k} \left(F_S(x_p, x_k) F_I(v_p, v_k) \right)}$$

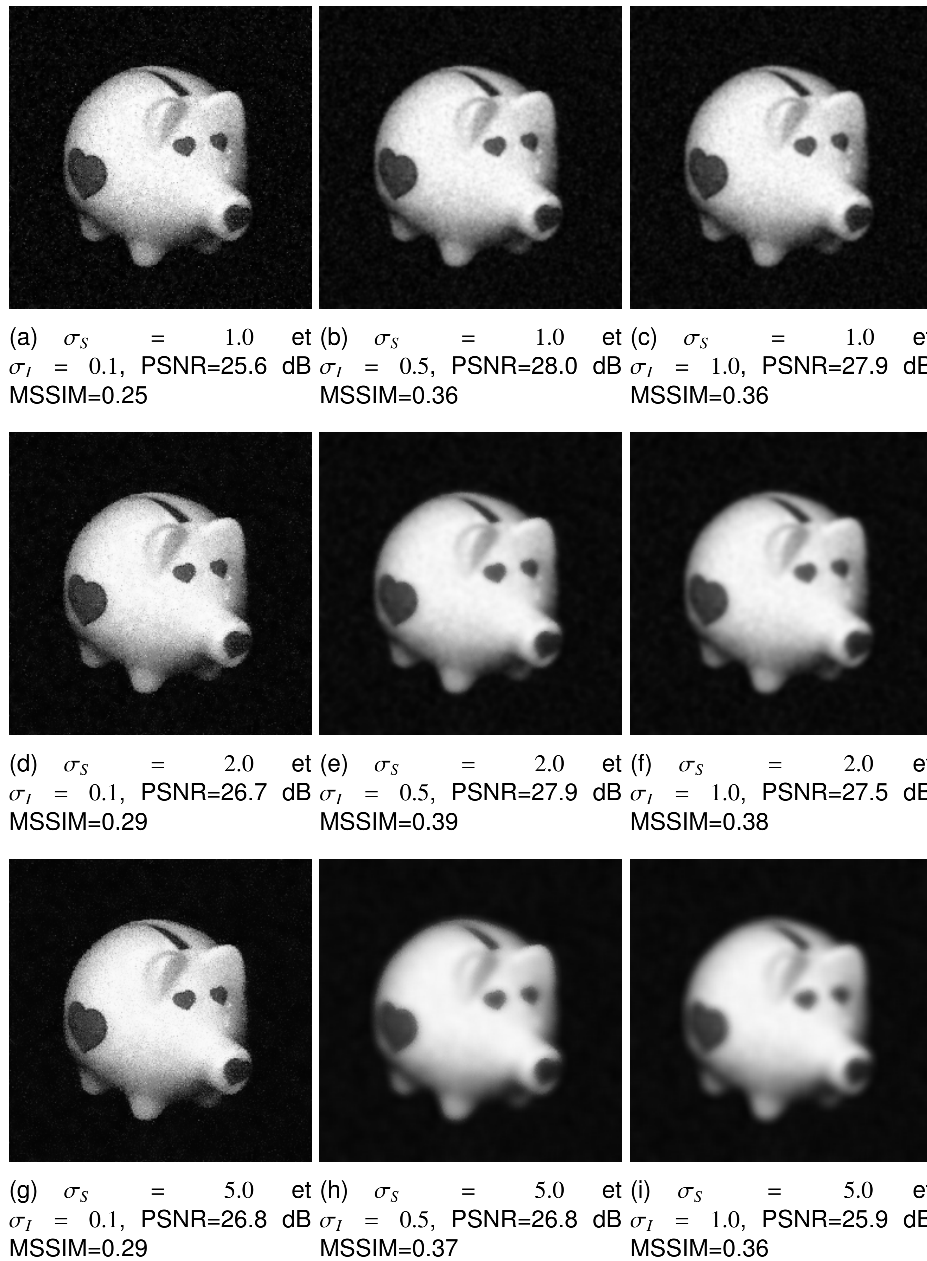
où F_S et F_I sont les fonctions de pondération spatiale et d'intensité. Classiquement, F_S et F_I sont des gaussiennes de moyennes nulles et d'écarts type σ_S et σ_I. Ce filtre se prête également bien à une utilisation en plusieurs passes sans flouter les contours. Des approximations séparables du filtre bilatéral, comme celle proposée dans [77], permettent d'obtenir des vitesses d'exécution plus élevées que les versions standards. Une variante à temps de calcul constant a même été proposée en 2008 par Porikli [80]. Ce filtre atteint un bon niveau de réduction de bruit gaussien, mais au prix d'un nombre de paramètres plus élevé à régler, ce qu'illustre la figure 4.4 où le filtrage de la même image a été réalisé avec 9 combinaisons de σ_S et σ_I. On y constate que la qualité s'obtient par compromis entre la valeur de σ_S et celle de σ_I et que le « meilleur » résultat parmi nos 9 combinaisons est à la figure 4.4(e).

Il existe beaucoup de variantes d'algorithmes basés sur des moyennes ou médianes locales effectuées sur des voisinages de formes diverses, variables et/ou adaptatives afin de sélectionner le plus finement possible les pixels pris en compte dans le calcul de la valeur filtrée. Le principal défaut de ces techniques est de générer des aplats dans les zones homogènes et des marches d'escalier dans les zones de transition douce (staircase effect), ces dernières pouvant être considérablement atténuées comme il a été montré dans [14].

L'un de ces algorithmes cherche dans l'image bruitée, une portion de la ligne de niveau de chaque pixel et l'utilise ensuite comme voisinage pour le moyennage. Cette technique a été présentée dans [9] et employée pour réduire le bruit de speckle. Nous y reviendrons en détail dans le chapitre 7.

4.1.1.4/ LES ALGORITHMES DE FILTRAGE PAR DICTIONNAIRE

Ces algorithmes font l'hypothèse qu'il est possible de décrire l'image à débruiter en utilisant une base de fonctions permettant de décomposer l'image en une combinaison linéaire des éléments de cette base. Les bases les plus employées sont les ondelettes [61, 32] ainsi que les fonctions sinusoïdales (DCT [22, 92]). Les éléments de la base peuvent être prédéterminés ou bien calculés à partir des données de l'image, par exemple en s'appuyant sur une analyse en composantes principales ou après apprentissage [34]. Le principe du débruitage est de considérer que le bruit est décorellé des fonctions de la base et donc représenté par les petits coefficients de la décomposition, que l'on peut annuler. Diverses politiques de seuillage peuvent alors être appliquées selon le type d'image et le modèle de bruit ayant chacune ses propres avantages et inconvénients. L'intérêt principal de ces méthodes est de bien restituer les transitions rapides (grande énergie), mais elles génèrent en revanche des artefacts dus aux possibles



(a) $\sigma_S = 1.0$ et $\sigma_I = 0.1$, PSNR=25.6 dB MSSIM=0.25

(b) $\sigma_S = 1.0$ et $\sigma_I = 0.5$, PSNR=28.0 dB MSSIM=0.36

(c) $\sigma_S = 1.0$ et $\sigma_I = 1.0$, PSNR=27.9 dB MSSIM=0.36

(d) $\sigma_S = 2.0$ et $\sigma_I = 0.1$, PSNR=26.7 dB MSSIM=0.29

(e) $\sigma_S = 2.0$ et $\sigma_I = 0.5$, PSNR=27.9 dB MSSIM=0.39

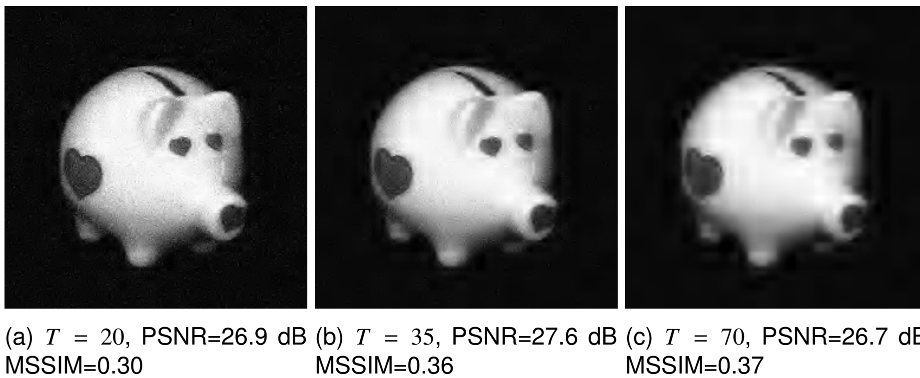
(f) $\sigma_S = 2.0$ et $\sigma_I = 1.0$, PSNR=27.5 dB MSSIM=0.38

(g) $\sigma_S = 5.0$ et $\sigma_I = 0.1$, PSNR=26.8 dB MSSIM=0.29

(h) $\sigma_S = 5.0$ et $\sigma_I = 0.5$, PSNR=26.8 dB MSSIM=0.37

(i) $\sigma_S = 5.0$ et $\sigma_I = 1.0$, PSNR=25.9 dB MSSIM=0.36

FIGURE 4.4 – Réduction de bruit gaussien par filtrage bilatéral de voisinage 5×5. σ_S et σ_I sont les écarts type des fonctions gaussiennes de pondération spatiale et d'intensité.



(a) $T = 20$, PSNR=26.9 dB MSSIM=0.30 (b) $T = 35$, PSNR=27.6 dB MSSIM=0.36 (c) $T = 70$, PSNR=26.7 dB MSSIM=0.37

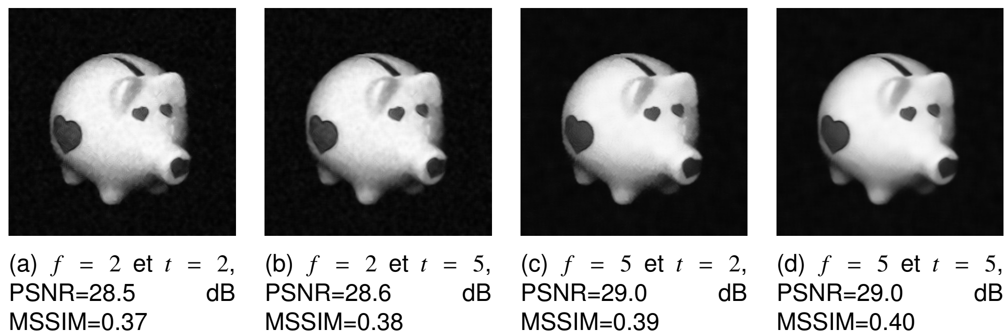
FIGURE 4.5 – Filtrage par décomposition en ondelettes et seuillage dur des coefficients inférieurs au seuil T.

grands coefficients de bruit. La figure 4.5 illustre cela en montrant le résultat du débruitage obtenu par décomposition en ondelettes et seuillage « dur » : lorsque la valeur du seuil croît, des aplats apparaissent et sont visuellement génants bien que les valeurs des indices de qualité n'aient pas diminué significativement. Certains algorithmes récents, en particulier ceux utilisant une base d'ondelettes adaptative, comme dans [34] sont proches, en terme de qualité, de l'état de l'art du domaine, avec souvent un avantage lié à des vitesses d'exécution assez rapides.

4.1.2/ LES ALGORITHMES DE FILTRAGE PAR PATCHES

Les techniques de réduction de bruit les plus efficaces sont aujourd'hui celles qui reposent sur les propriétés d'auto-similarité des images, on les appelle aussi les techniques par patches. L'idée principale est, comme pour les techniques classiques à base de voisinage, de rechercher un ensemble de pixels pertinents et comparables afin d'en faire une moyenne. Cependant, dans le cas des techniques à patches, la recherche de cet ensemble ne se limite pas à un voisinage du pixel central, mais fait l'hypothèse qu'il existe des zones semblables au voisinage du pixel central, réparties dans l'image et pas nécessairement immédiatement contiguës. Le moyennage s'effectue alors sur l'ensemble de ces zones identifiées. L'algorithme des moyennes non locales (NL-means, [13]), parmi les premiers de cette lignée à être proposé, bien qu'ayant représenté un progrès notable dans la qualité de débruitage, fut rapidement suivi, en particulier par le BM3D et ses variantes qui représentent actuellement ce qui se fait de mieux en terme de qualité de débruitage [30, 31].

Les différences entre ces algorithmes résident essentiellement dans la méthode de recherche et d'identification des patches similaires, incluant la possiblité de forme et taille variables. Une telle recherche est d'autant plus coûteuse en temps de calcul qu'elle est effectuée sur une zone étendue autour du patch central et cela représente le principal inconvénient de ces techniques qui peuvent présenter des temps d'exécution prohibitifs dans l'optique d'un traitement en temps réel. La figure 4.6 montre des résultats de débruitage obtenus par la méthode des NL-means avec plusieurs combinaisons des paramètres de similarité des patches et de non localité du voisinage, notés f et t. La figure 4.7 montre quant à elle le résultat du débruitage par BM3D. Les points forts de ces deux techniques sont, comme on le voit, la qualité du débruitage avec pour l'implémentation



(a) $f = 2$ et $t = 2$, PSNR=28.5 dB MSSIM=0.37

(b) $f = 2$ et $t = 5$, PSNR=28.6 dB MSSIM=0.38

(c) $f = 5$ et $t = 2$, PSNR=29.0 dB MSSIM=0.39

(d) $f = 5$ et $t = 5$, PSNR=29.0 dB MSSIM=0.40

FIGURE 4.6 – Filtrage par NL-means pour différentes combinaisons des paramètres de similarité f et de non localité t.

FIGURE 4.7 – Filtrage par BM3D, PSNR=29.3 dB MSSIM=0.41

BM3D l'avantage de ne nécessiter aucun réglage de paramètres.

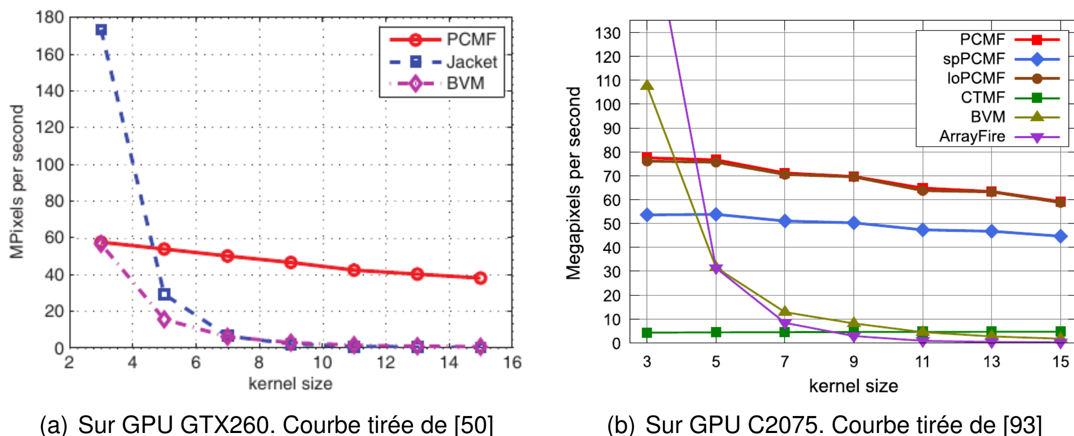
4.2/ LES IMPLÉMENTATIONS SUR GPU DES ALGORITHMES DE FILTRAGE

Le fabricant de processeurs graphiques Nvidia, seul type d'équipements dont nous disposons, fournit des implémentations performantes de certains pré-traitements et algorithmes de filtrage. C'est le cas des transformées de fourrier (FFT, DCT), qui sont par exemple utilisées dans l'implémentation d'un algorithme d'*inpainting* [53] et de l'opération de convolution dont l'étude est présentée ci-dessous.

4.2.1/ LE FILTRAGE PAR CONVOLUTION

L'opération de convolution a fait l'objet d'une étude et d'une optimisation poussées par le fabricant Nvidia pour déterminer la combinaison de solutions apportant la plus grande vitesse d'exécution [90]. L'étude a testé 16 versions distinctes, chacune présentant une optimisation particulière quant à l'organisation de la grille de calcul, aux types de transferts de données entre l'hôte et le GPU ainsi qu'aux types de mémoire employés pour le calcul sur le GPU.

Les résultats montrent que l'emploi de texture comme mémoire principale pour le stockage des images à traiter apporte un gain d'environ 50% par rapport à l'utilisation de la mémoire globale. Par ailleurs, les transactions par paquets de 128 bits apportent également une amélioration sensible, ainsi que l'emploi de la mémoire partagée comme zone



(a) Sur GPU GTX260. Courbe tirée de [50]

(b) Sur GPU C2075. Courbe tirée de [93]

FIGURE 4.8 – Performances relatives des filtres médians implémentés sur GPU dans libJacket/ArrayFire, PCMF et BVM et exécutés sur deux modèles de générations différentes.

de travail pour le calcul des valeurs de sortie. Le traitement de référence effectué pour les mesures est la convolution générique (non séparable) d'une image 8 bits de 2048×2048 pixels par un masque de convolution de 5×5 pixels, expression que l'on raccourcira dorénavant en *convolution 5×5*.

Le meilleur résultat obtenu dans les conditions détaillées précédemment, sur architecture GT200 (carte GTX280) est de 1,4 ms pour le calcul, ce qui représente un débit global de 945 MP/s lorsque l'on prend en compte les temps de transfert aller et retour des images (1,5 ms d'après nos mesures). Nous continuerons d'utiliser cette mesure de débit en *pixels par seconde* pour toutes les évaluations à venir ; elle permet en particulier de fournir des valeurs de performance indépendantes de la taille des images soumises au traitement.

4.2.2/ LE FILTRE MÉDIAN

On connaît peu de versions GPU du filtre médian, peut-être en raison des implémentations CPU performantes et génériques que l'on a déjà évoquées (voir par exemple [72]) et dont le portage sur GPU ne laisse pas entrevoir de potentiel, ou bien reste à inventer. Néanmoins, une bibliothèque commerciale (LibJacket et ArrayFire) en propose une implémentation GPU dont nous avons pu mesurer les performances pour un masque de 3×3 et qui est également prise comme référence par Sanchez *et al.* pour évaluer les performances de leur propre implémentation appelée PCMF [85].

Sur architecture GT200 (GTX260), les performances maximales de ces deux versions sont obtenues pour un masque de 3×3 pixels avec respectivement 175 MP/s pour libJacket et 60 MP/s pour PCMF (transferts de données compris). Une précédente implémentation avait été réalisée, basée sur l'algorithme BVM décrit dans [50]. Elle prouve son efficacité dans l'élimination des artefacts générés par les dispositifs d'imagerie médicale magnétique en 3D [21], mais ne permet pas d'exploiter véritablement le parallélisme des GPU en filtrage d'image en 2D.

La figure 4.8(a), tirée de [50], compare ces trois implémentations et montre que le débit permis par la libJacket décroit très vite avec la taille du masque pour passer à 30 MP/s

dès la taille 5×5, alors que le PCMF décroit linéairement jusqu'à la taille 11×11 où il permet encore de traiter quelque 40 MP/s. Ceci s'explique simplement par le fait que libJacket utilise un tri simple pour la sélection de la valeur médiane alors que le PCMF exploite les propriétés des histogrammes cumulés et n'est ainsi que très peu dépendant de la taille du masque.

Plus récemment, Sanchez *et al.* ont actualisé dans [93] leurs mesures sur architecture Fermi (GPU C2075) en comparant leur PCMF à la version ré-écrite en C de libJacket, nommée ArrayFire. Les courbes sont celles de la figure 4.8(b), où l'on constate que les variations selon la taille du masque demeurent comparables, avec toutefois des valeurs de débit augmentées, avec près de 185 MP/s pour ArrayFire et 82 MP/s pour PCMF.

Parallèlement, on trouve aussi des implémentations de filtre médian dans des traitements plus complexes comme dans [5] où les auteurs décrivent la plus récente évolution de leur technique itérative de réduction de bruit impulsionnel, sans qu'il soit possible d'évaluer le débit du médian seul.

Il faut noter enfin que certains codes sont plus performants sur l'ancienne architecture GT200/Tesla que sur la plus récente Fermi. C'est le cas pour l'implémentation du médian incluse dans la bibliothèque ArrayFire et nous reviendrons sur les raisons de cette perte de performances constatée au passage à une architecture plus récente dans le chapitre 8.

4.2.3/ Le filtre bilatéral

Le filtre bilatéral a été plus étudié et un certain nombre de publications font état d'implémentations rapides. Une implémentation à temps constant en est proposée par Yang *et al.* [107] et s'exécute entre 3,7 ms et 15 ms pour une image de 1 MP. Cela ne constitue pas une référence de vitesse pour les masques de petite taille, mais devient compétitif pour des masques de grande taille (plus de 400 pixels dans le voisinage). Une autre plus classique, employée dans la génération des images médicales tomographiques, annonce 16 ms pour un masque de 11×11 sur une image de 0,25 MP. Il demeure souvent difficile de comparer les implémentations sans disposer des codes sources, en raison de conditions de test très variables, en particulier en ce qui concerne le modèle de GPU et la taille du masque. Ceci étant précisé, on peut prendre comme première référence la version proposée par Nvidia dans le SDK CUDA et nommée « ImageDenoising ». Elle permet d'exécuter sur GPU GTX480 un filtre bilatéral 7×7 sur une image, déjà en mémoire GPU, de 1 MP en 0,411 ms, pour un débit global de 133 MP/s.

Dans [109], les auteurs présentent un cadre général pour optimiser l'accès aux données par les différents *kernels* en utilisant la mémoire partagée pour les threads d'un même bloc. Le principe est de pré-charger les valeurs utiles au bloc de threads dans la mémoire partagée, c'est-à-dire les valeurs (niveaux de gris) des pixels associés aux threads ainsi que le halo correspondant aux voisinages des pixels de la bande périphérique. On appelle communément cet ensemble la *region of interest* ou ROI. La figure 4.9 illustre la mise en œuvre de cette technique en montrant comment les threads d'un bloc se répartissent les pré-chargements en mémoire partagée des valeurs des pixels de la ROI. Dans cet exemple, chaque thread (en vert) gère le chargement de 4 pixels (blocs en gris) en mémoire partagée. On parvient ainsi à éviter les conflits de banques de mémoire partagée et à répartir la charge des threads de sorte à générer le moins de perte de performances. La géométrie des blocs de threads est ici choisie carrée, mais elle s'applique

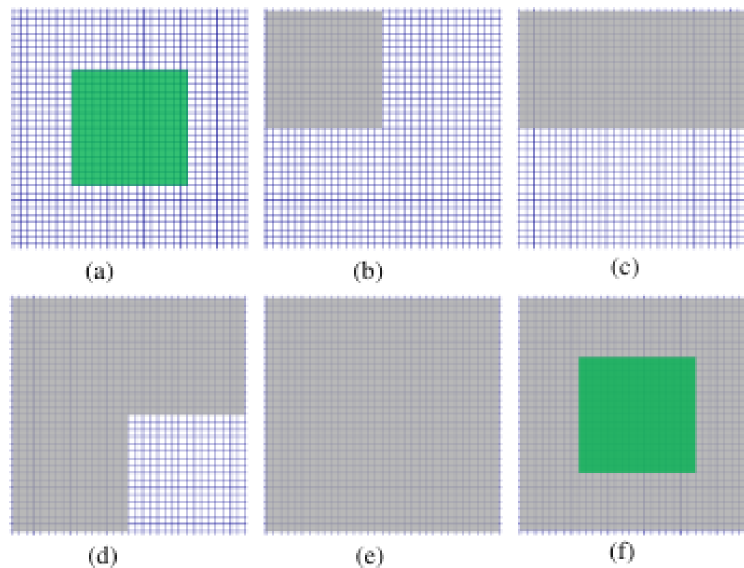


FIGURE 4.9 – Illustration du pré-chargement en mémoire partagée mis en œuvre dans [109] pour l'implémentation, entre autres, du filtre bilatéral. a) en vert le bloc de threads associé aux pixels centraux. b-e) les blocs de pixels successivement pré-chargés en mémoire partagée. f) la configuration finale de la ROI en mémoire partagée.

aisément à d'autres proportions comme nous le verrons plus loin. Les limites de cette méthode sont :

– la taille de la mémoire partagée qui doit pouvoir stocker l'ensemble des valeurs des pixels de la ROI, ce qui peut imposer une limite sur la taille des blocs de threads.
– l'étendue du voisinage qui ne peut être pré-chargé de cette façon (4 pixels par thread) que si la surface de la ROI demeure inférieure à 4 fois le nombre de threads par bloc.

Cette recette est ensuite appliquée dans l'implémentation d'un filtre bilatéral et d'un filtre à moyennes non locales (NL-means). Concernant le filtre bilatéral, les auteurs pré-calculent aussi les coefficients de la pondération spatiale, alors que ceux de la pondération d'intensité restent calculés à la volée. Ces deux optimisations permettent un gain de 20% sur le temps de calcul du filtre bilatéral pour arriver à 0.326 ms dans les mêmes conditions que ci-dessus (bilatéral 7×7 sur image 1 MP). Toutefois, le débit global ne progresse pas (132 MP/s) en raison de la prépondérance des temps de transfert annoncés à 7,5 ms pour l'image de 1 MP, alors qu'ils n'étaient que de 7,1 ms dans les conditions décrites par Nvidia.

Ce travail d'optimisation ne perd toutefois pas son intérêt dans la mesure où, si le filtre fait partie d'une chaîne de traitement entièrement exécutée par le GPU, le transfert des données n'a besoin d'être effectué qu'une seule fois en tout début et en toute fin de traitement.

Enfin, l'implémentation qui semble à ce jour la plus performante s'attache à réduire les redondances de calculs et parvient à filtrer une image de 9 MP avec un masque de 21×21 en seulement 200 ms, soir un débit de 47 MP/s hors transferts.

4.2.4/ LES FILTRES PAR PATCHES

Intuitivement, les algorithmes à base de patches paraissent moins adaptés au parallélisme des GPUs, du fait de la nécessité d'accéder à un voisinage étendu autour de chaque pixel. On en recense tout de même quelques implémentations, comme celle présente dans le SDK CUDA qui fait l'hypothèse que les coefficients de pondération spatiale sont localement constants. Dans [70], le modèle de bruit employé vise une adaptation aux images échographiques présentant du bruit proche du speckle. Dans cette implémentation, aucune approximation des coefficients n'est faite, mais la taille maximale du patch est limitée par la quantité de mémoire partagée disponible pour chaque bloc de threads. Une version plus récente implémente exactement l'algorithme original [43] en proposant des optimisations algorithmiques exploitant la symétrie des coefficients spatiaux ainsi que l'interprétation du calcul de la similarité comme une convolution séparable, opération aisément parallélisable sur GPU, comme nous le détaillerons plus loin. Les auteurs parviennent ainsi à filtrer des séquences vidéo couleur de dimension 720×480 à plus de 30 fps en améliorant le PSNR de 16 dB (la séquence bruitée présentant un PSNR de 20 dB).

5

LES TECHNIQUES DE SEGMENTATION DES IMAGES

5.1/ LES TECHNIQUES DE SEGMENTATION

La segmentation représente un enjeu particulièrement important dans le domaine du traitement d'image et a ainsi fait l'objet d'abondants travaux et publications touchant les nombreux cas d'analyse dans lesquels une segmentation est utilisée. On peut citer la reconnaissance de formes, la détections et/ou la poursuite de cibles, la cartographie, le diagnostic médical, l'interaction Homme-machine, la discrimination d'arrière plan, etc.

On pourrait donner de la segmentation une définition spécifique par type d'usage, mais dans un souci d'unification, nous proposons la formulation générique suivante :
« La segmentation consiste à distinguer des zones homogènes au sein d'une image ».
Le caractère *homogène* s'entend au sens d'un critère pré-établi, adapté aux contraintes particulières de traitement comme le type de bruit corrompant les images, le modèle d'image ou bien la dimension du signal observé $\bar{\nu}$ selon que l'image est en couleur ou non. Un tel critère peut ainsi être un simple seuil de niveau de gris ou bien nécessiter de coûteux calculs statistiques dont certains seront détaillés dans les chapitres suivants.

Devant la diversité des cas à traiter et des objectifs à atteindre, on sait aujourd'hui qu'à l'instar du filtre unique, la méthode universelle de segmentation n'existe pas et qu'une bonne segmentation est celle qui conduit effectivement à l'extraction des structures pertinentes d'une image selon l'interprétation qui doit en être faite.

Les éléments constitutifs de la segmentation sont soit des régions (les segments), soit des contours. Les deux notions sont complémentaires étant donné que les contours délimitent des régions, mais les techniques de calcul basées sur l'un ou l'autre de ces éléments relèvent d'analyses différentes.

Les algorithmes de segmentation orientés régions s'appuient pour beaucoup sur des techniques de regroupement, ou *clustering*, pour l'identification et le peuplement des régions. Ce lien trouve son origine dans la théorie du *gestalt* [48] où l'on considère que la perception conceptuelle s'élabore au travers de regroupements visuels d'éléments.

La plupart des approches proposées jusqu'à très récemment consistent à minimiser une fonction d'énergie qui n'a pas de solution formelle et que l'on résout donc à l'aide de techniques numériques souvent itératives.

5.1.1/ Analyse d'histogramme

Les techniques de segmentation les plus simples à mettre en œuvre sont dites « de seuillage » et basées sur une analyse de l'histogramme des niveaux de gris (ou de couleurs). Elles cherchent à distinguer les différentes classes comme autant d'occurrences représentant des *régions* homogènes. Différents critères peuvent être appliqués pour cette analyse, visant par exemple à maximiser la variance [68] ou encore à maximiser le contraste pour déterminer les valeurs pertinentes des seuils.

Malgré la multitude de variantes proposées, ces méthodes demeurent peu robustes et présentent l'inconvénient majeur de ne pas garantir la connexité des régions déterminées. On les réserve à des applications très spécifiques où, par exemple, on dispose d'une image de référence dont l'histogramme peut être comparé à celui des images à traiter. C'est le cas de certaines applications de contrôle industriel où la simplicité algorithmique permet de surcroît des implémentations très rapides, voire câblées.

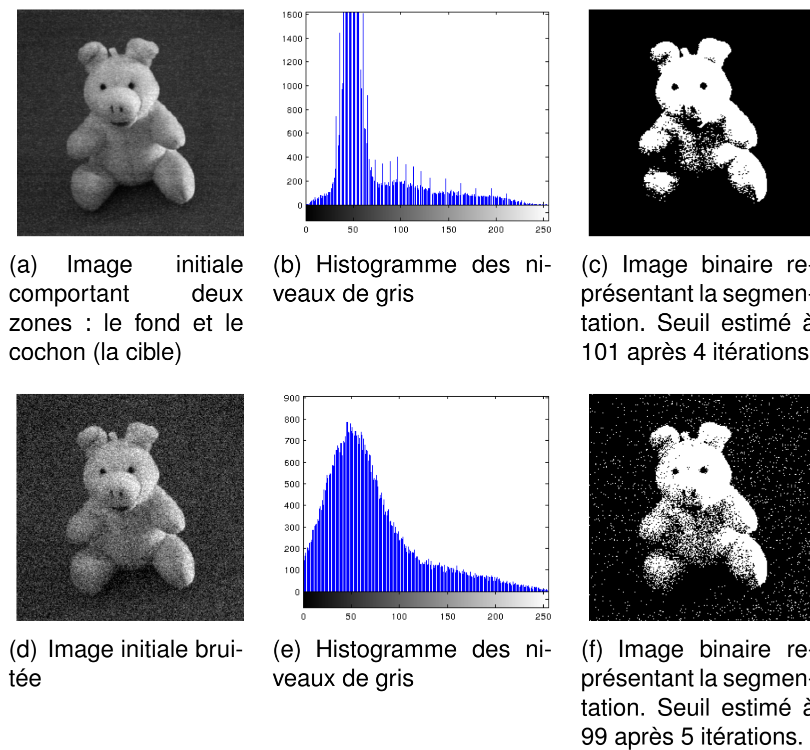
Ces techniques peuvent aujourd'hui être considérées comme rudimentaires mais les calculs d'histogrammes et les analyses associées interviennent dans beaucoup d'algorithmes récents parmi les plus évolués et performants. La figure 5.1 illustre le traitement typique de l'histogramme de l'image d'entrée 5.1(a) dans le but de distinguer les deux régions du fond et du cochon (la cible). La première étape consiste à dresser l'histogramme des niveaux de gris sur tout le domaine de l'image 5.1(b). Il faut ensuite identifier le seuil de séparation des deux régions supposées, ici, homogènes au sens des valeurs de niveau de gris. Une estimation visuelle peut-être faite, mais on voit immédiatement que même dans une situation aussi claire, le choix du seuil n'est pas évident. Pour un traitement automatique, on peut par exemple proposer la technique itérative présentée par l'Algorithme 1 qui conduit à la segmentation de la figure 5.1(c). Le principe mis en œuvre est de fixer arbitrairement une valeur initiale au seuil (ici 128), puis de calculer les « poids » respectifs des valeurs situées au-dessus et en-dessous de ce seuil (ligne 6). Cette évaluation de la répartition énergétique permet d'ajuster la valeur du seuil (ligne 8), puis de recommencer jusqu'à convergence, lorsque l'écart entre deux valeurs successives du seuil devient inférieur à une limite fixée à l'avance (ϵ, ligne 9).

L'image 5.1(d) est l'image initiale, corrompue par un bruit gaussien de moyenne nulle et d'écart type 25. Les résultats de la segmentation (5.1(c) et 5.1(f)) de cette image sont clairement insuffisants : les régions identifiées présentent des discontinuités et dans le cas de l'image bruitée, quantité de pixels orphelins subsistent en quantité. Cette technique nécessiterait une étape supplémentaire pour obtenir une segmentation pertinente.

5.1.2/ Partitionnement de graphe

Un autre formalisme qui a généré une vaste classe d'algorithmes de segmentation est celui des graphes. Il repose sur l'idée que les régions de l'image sont représentées par les nœuds du graphe, alors que les liens traduisent les relations de voisinage existant entre les régions, l'idée de base étant d'initialiser le graphe avec un nœud pour chaque pixel. La segmentation est obtenue par partitionnement itératif du graphe, en évaluant les liens et en déterminant ceux à supprimer, et ce jusqu'à convergence.

L'essentiel de la problématique réside donc dans la métrique retenue pour évaluer les liens ainsi que dans le critère de sélection et, là encore, la littérature regorge d'une grande variété de propositions. Les premières d'entre elles, qui n'étaient pas spécifiquement dé-



(a) Image initiale comportant deux zones : le fond et le cochon (la cible)

(b) Histogramme des niveaux de gris

(c) Image binaire représentant la segmentation. Seuil estimé à 101 après 4 itérations.

(d) Image initiale bruitée

(e) Histogramme des niveaux de gris

(f) Image binaire représentant la segmentation. Seuil estimé à 99 après 5 itérations.

FIGURE 5.1 – Segmentation d'une image en niveaux de gris de 128 × 128 pixels par analyse simple d'histogramme. Colonne de gauche : image d'entrée. Colonne centrale : histogramme des niveaux de gris. Colonne de droite : résultat de la segmentation.

Algorithme 1 : Calcul du seuil de séparation des segments de l'histogramme.

1 $\overline{h} \leftarrow$ histogramme sur l'image ;
2 $S_{init} \leftarrow 128$;
3 $S_k \leftarrow S_{init}$;
4 $\epsilon \leftarrow 1$;
5 **repeat**
6 $\quad \mu_{inf} = \dfrac{\sum_{i<S_k} h_i i}{\sum_{i<S_k} h_i}$;
7 $\quad \mu_{sup} = \dfrac{\sum_{i \geq S_k} h_i i}{\sum_{i \geq S_k} h_i}$;
8 $\quad S_k = \frac{1}{2}(\mu_{inf} + \mu_{sup})$;
9 **until** $||S_k - \frac{1}{2}(\mu_{inf} + \mu_{sup})|| < \epsilon$;

diées à la segmentation d'images numériques mais au regroupement d'éléments répartis sur un domaine (1D ou 2D), ont été élaborées autour d'une mesure locale des liens basée sur la distance entre les éléments. La réduction du graphe est ensuite effectuée en utilisant un algorithme spécifique, comme le *minimum spanning tree*, dont l'application a été décrite dès 1970 dans [108] et où il s'agit simplement de supprimer les liens *inconsistants*, c'est à dire ceux dont le poids est significativement plus élevé que la moyenne des voisins se trouvant de chaque coté du lien en question.

Le principe en a rapidement été étendu aux images numériques en ajoutant l'intensité des pixels au vecteur des paramètres pris en compte dans l'évaluation du poids des liens. D'autres critères de partitionnement ont été élaborés, avec pour ambition de toujours mieux prendre en compte les caractéristiques structurelles globales des images pour parvenir à une segmentation conduisant à une meilleure perception conceptuelle. Le principe général des solutions actuelles repose sur la construction d'une matrice de similarité qui traduit les liens entre les segments et représente le graphe à partitionner.

Pour des images en niveaux de gris, l'expression générale des éléments w_{ij} de la matrice de similarité W est :

$$w_{ij} = \begin{cases} e^{\|v_i - v_j\|^2/\sigma_v^2} e^{\|x_i - x_j\|^2/\sigma_x^2} & \text{si } \|x_i - x_j\| < r \\ 0 & \text{sinon} \end{cases}$$

On construit également la matrice de connectivité D, diagonale et dont les éléments sont :

$$d_i = \sum_j w_{ij}$$

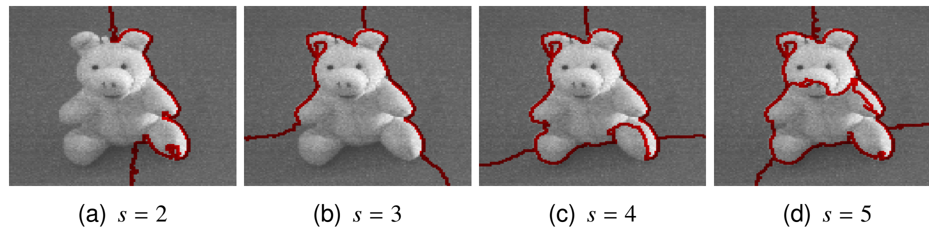
Une famille de méthodes, inspirée par le *graphe optimal* de Wu et Leahy [105], réalise le partitionnement sur la base des valeurs propres λ_k et vecteurs propres Y_k du système

$$(D - W))\, Y = \lambda DY$$

Certains algorithmes proposés plus récemment s'inscrivent dans cette veine [101, 102, 35, 88], avec comme principal point faible, une difficulté à trouver un compromis acceptable entre identification de structures globales et préservation des éléments de détails. Cela se traduit dans la pratique par un ensemble de paramètres à régler pour chaque type de segmentation à effectuer.

La figure 5.2 montre un exemple de l'application de l'algorithme *normalized cuts* décrit dans [88] et implémenté par Cour, Yu et Shi en 2004. Cette implémentation utilise des valeurs pré-établies des paramètres de calcul de la matrice de similarité produisant de bonnes segmentations de sujets dans les images naturelles, mais requiert de prédéterminer le nombre de segments. Les images de la figure représentent les résultats obtenus avec un nombre de segments variant de 2 à 5 et montrent qu'il est difficile de trouver un compromis acceptable. De plus, les temps d'exécution peuvent devenir très rapidement prohibitifs, même avec des implémentations plus optimisées. Les résultats de la figure 5.2 ont été obtenus en 1,5 s environ (Matlab R2010 sur CPU intel core i5-2520M 2.50GHz - linux 3.2.0)

Un autre procédé de partitionnement de graphe, reposant sur le théorème dit du *maximum flow-minimum cut* énoncé par Ford et Fulkerson [38] a fait l'objet de beaucoup de travaux, dont les résultats sont comparés dans [11, 19]. Ce procédé est mis en œuvre avec de bons résultats dans plusieurs algorithmes, comme le *push-relabel* [24] ou le *pseudoflow* [46] qui semble aujourd'hui le plus performant.



(a) $s = 2$ (b) $s = 3$ (c) $s = 4$ (d) $s = 5$

FIGURE 5.2 – Segmentation d'une image en niveaux de gris de 128 × 128 pixels par simplification de graphe de type *Normalized cut* pour un nombre s de segments variant de 2 à 5.

5.1.3/ KERNEL-MEANS, MEAN-SHIFT ET APPARENTÉS

Parallèlement à la réduction de graphes, d'autres approches ont donné naissance à une multitude de variantes tournées vers la recherche des moindres carrés. Il s'agit simplement de minimiser l'erreur quadratique totale, ce qui peut se résumer, pour une image de N pixels, en la détermination du nombre C de segments Ω_i et leur contenu, de sorte à minimiser l'expression

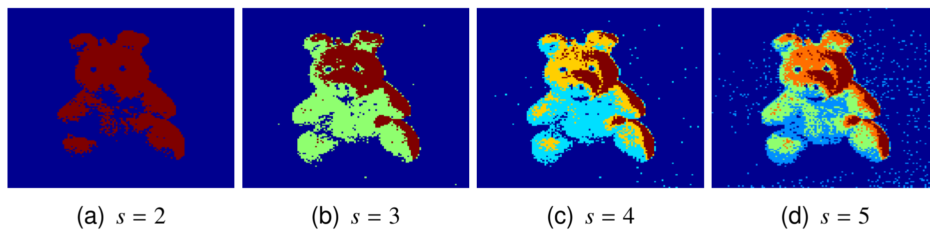
$$\sum_{i\in[1..C]} \sum_{x_k\in\Omega_i} (v_k - \mu_i)^2$$

où μ_i représente la valeur affectée au segment Ω_i, i.e la valeur moyenne des observations v_k sur Ω_i, et $\bigcup_{i\in[1..C]} \Omega_i = \Omega$

Cette idée est très intuitive et simple, mais n'a pas souvent de solution explicite, d'autant que le nombre des segments est *a priori* inconnu. Dès 1965, Mac Queen a proposé l'appellation k-means pour cette procédure itérative de regroupement [60] qui débute avec k groupes d'un seul pixel [1] pris au hasard, puis d'ajouter chaque point au groupe dont la moyenne est la plus proche de la valeur du point à ajouter. La moyenne du groupe nouvellement agrandi doit alors être recalculée avant le prochain ajout. Cette implémentation est extrêmement simple à mettre en œuvre [2] mais elle possède de nombreux défauts dont le principal est qu'elle ne converge pas nécessairement vers le regroupement optimal, même si on connait la « bonne » valeur de k. Un autre est d'être très dépendant du choix des k éléments initiaux, en nombre et en position.

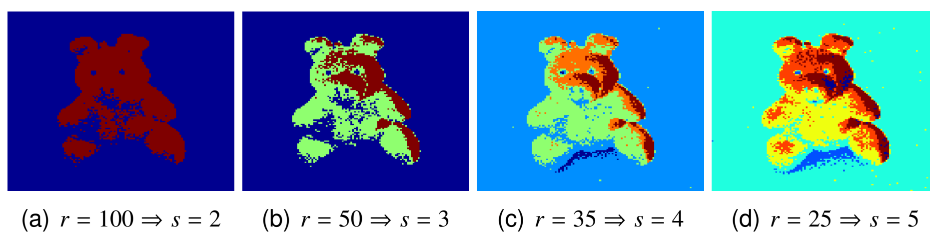
Toutefois, vraisemblablement du fait de sa simplicité d'implémentation et de son temps d'exécution rapide, la communauté scientifique s'est beaucoup penchée sur cette méthode pour en compenser les défauts, jusqu'à en faire une des plus employées, en particulier par les statisticiens. On compte aussi beaucoup de variantes telles les *k-centers* [4] et les *k-médians* [7] qui n'emploient pas la moyenne arithmétique comme expression du « centre » d'un segment. Des solutions ont aussi été apportées pour l'estimation de k en employant, par exemple, un critère de vraisemblance pour choisir la meilleure valeur de k dans un intervalle donné [71]. À titre d'illustration et de comparaison, l'image du cochon a été traitée par une implémentation naïve de l'algorithme original des *k-means* en donnant successivement au nombre de segments les valeurs $s = 2$ à $s = 5$. Les résultats sont reproduits à la figure 5.3 et montrent encore une fois l'influence de s sur la segmentation.

1. Dans son article, MacQueen ne parle pas de pixel mais de point. En effet, la méthode décrite ne visait pas à segmenter des images, mais des données de natures diverses.
2. Même si en 1965, rien n'était simple à programmer



(a) $s = 2$ (b) $s = 3$ (c) $s = 4$ (d) $s = 5$

FIGURE 5.3 – Segmentation d'une image en niveaux de gris de 128 × 128 pixels par algorithme *k-means* pour un nombre s de segments variant de 2 à 5. Chaque couleur est associée à un segment. Les couleurs sont choisies pour une meilleure visualisation des différents segments.



(a) $r = 100 \Rightarrow s = 2$ (b) $r = 50 \Rightarrow s = 3$ (c) $r = 35 \Rightarrow s = 4$ (d) $r = 25 \Rightarrow s = 5$

FIGURE 5.4 – Segmentation d'une image en niveaux de gris de 128 × 128 pixels par algorithme *mean-shift* pour un rayon de voisinage r de 100, 50, 35 et 25 pixels permettant d'obtenir un nombre s de segments variant respectivement de 2 à 5. Le volume minimal admis pour un segment est fixé à 100 pixels. Chaque couleur est associée à un segment. Les couleurs sont choisies pour une meilleure visualisation des différents segments.

Dès 1975, Fukunaga et Hostetler [39] avaient décrit un algorithme générique permettant de déterminer le nombre de segments, ou modes, ainsi que les points, ou pixels, qui les composent. Leur algorithme cherche, pour ce faire, à localiser les k positions où le gradient de densité s'annule. Il utilise à cet effet un voisinage pondéré (ou *kernel*) et détermine le centre de masse des segments en suivant itérativement le gradient de densité dans le voisinage de chaque élément du domaine. Lorsque l'algorithme a convergé, les k segments sont identifiés et contiennent chacun l'ensemble des points qui ont conduit à leurs centres de masse respectifs. Étonnement, malgré ses qualités intrinsèques, cet algorithme du *mean-shift* est resté longtemps sans susciter de grand intérêt, jusqu'à l'étude de Cheng [23] qui en a valorisé les propriétés et établi des liens avec d'autres techniques d'optimisation ou de filtrage, comme la descente/montée de gradient ou le floutage.

Comaniciu et Peer en ont alors étendu l'étude et proposé une application à la segmentation utilisant l'espace colorimétrique CIELUV [37] et ont montré qu'elle permettait une meilleure identification des segments de l'image [27, 28]. Une implémentation de la variante proposée par Keselman et Micheli-Tzanakou dans [52] appliquée à notre image de test, fournit les résultats reproduits à la figure 5.4. Pour se rapprocher des traitements précédents, nous avons identifié, par essais successifs, les tailles de voisinage conduisant à des nombres de segments identiques à ceux des figures précédentes (de 2 à 5), le volume minimal admis pour un segment étant arbitrairement fixé à 100 pixels.

Il est à noter que les segmentations basées sur des algorithmes de *clustering* comme ceux que l'on vient de présenter nécessitent le plus souvent une phase supplémentaire de génération des frontières inter-segments et d'affectation de la valeur de chaque segment

aux éléments qui le composent. Par ailleurs, dans les deux cas du *k-means* et du *mean-shift*, chaque itération génère une réduction de la variance (due au moyennage) et on peut donc rapprocher ces techniques de celles de réduction de bruit par minimisation de variance.

5.1.4/ Les contours actifs, ou snakes

Contrairement aux précédentes techniques et comme leur nom le laisse deviner, les éléments constitutifs de ces méthodes sont cette fois des *contours* et non plus des *régions*. De fait, ils définissent nativement une segmentation de l'image. Le principe général est de superposer une courbe paramétrique S à l'image, le *snake*, puis de lui appliquer des déformations successives destinées à rapprocher le *snake* des contours de l'objet. Les déformations à appliquer sont guidées par l'évaluation d'une fonction d'énergie E_{snake} prenant en compte :

- l'énergie interne E_{int} de la courbe, fonction de son allongement de sa courbure.
- l'énergie externe E_{ext} liée à l'image, fonction de la proximité de la courbe avec les zones de fort gradient et éventuellement une contrainte fixée par l'utilisateur comme des points imposés par exemple.

L'expression générique peut alors s'écrire

$$E_{snake} = E_{int} + E_{ext}$$

où

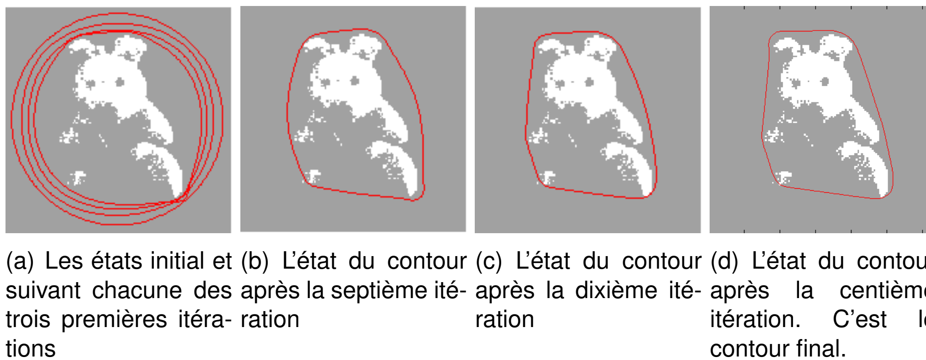
$$E_{int} = \sum_{s \in S} \frac{1}{2} \left(\alpha \left| \frac{\partial x_s}{\partial s} \right|^2 + \beta \left| \frac{\partial^2 x_s}{\partial s^2} \right| \right) ds$$

et

$$E_{ext} = \sum_{s \in S} - |\nabla \left[G_\sigma(x_s) * v_s \right] |^2 \, ds$$

L'objectif de l'algorithme du *snake* est de trouver une courbe S qui minimise l'énergie totale E_{snake}. Ici encore, la résolution du problème revient à minimiser une fonction sous contrainte et les diverses techniques de résolution numérique peuvent s'appliquer comme pour les autres classes d'algorithmes itératifs présentés précédemment. Ici encore, il faut composer avec un nombre de paramètres à régler assez important. Notons également que dans le cas général, les paramètres notés α et β, que l'on qualifie aussi d'élasticité et de raideur, sont des fonctions de l'abscisse curviligne s. La fonction G_σ est la fonction d'attraction aux forts gradients de l'image.

Dans sa version originale proposée par Kass *et al.* en 1988 [51], l'algorithme dit du *snake* présente l'intérêt de converger en un nombre d'itérations assez réduit et permet de suivre naturellement une *cible* en mouvement après une convergence initiale à une position donnée, chaque position de convergence fournissant une position initiale pertinente pour la position suivante. Toutefois, il se montre particulièrement sensible à l'état initial de la courbe et requiert souvent de celle-ci qu'elle soit assez proche de l'objet à « entourer », sous peine de se verrouiller dans un minimum local. La sensibilité au bruit n'est pas non plus très bonne du fait de la formulation locale de l'énergie. Les « concavités » étroites ou présentant un goulot d'étranglement marqué sont par ailleurs mal délimitées. Enfin, la fonction d'énergie étant calculée sur la longueur totale de la courbe, cela pénalise la bonne identification des structures de petite taille vis à vis de la longueur totale de la courbe. La figure 5.5 illustre ces défauts en montrant quelques états intermédiaires ainsi



(a) Les états initial et suivant chacune des trois premières itérations (b) L'état du contour après la septième itération (c) L'état du contour après la dixième itération (d) L'état du contour après la centième itération. C'est le contour final.

FIGURE 5.5 – Segmentation d'une image en niveaux de gris de 128 × 128 pixels par algorithme dit du *snake*, dans sa version originale. Les paramètres d'élasticité, de raideur et d'attraction ont été fixés respectivement aux valeurs 5, 0.1 et 5.

que le résultat final d'une segmentation réalisée à partir d'un contour initial circulaire et des paramètres à valeurs constantes et réglés empiriquement, en employant la méthode du *snake* original. On voit que la convergence est assez rapide mais que le contour ainsi déterminé ne « colle » pas bien à l'objet que l'on s'attend à isoler.

Il est cependant possible de contrôler la finesse de la segmentation mais au prix de temps de calculs qui peuvent devenir très longs. Parmi les variantes élaborées qui tentent de pallier ces défauts, les plus intéressantes sont :

- le *balloon snake*, conçu pour remédier au mauvais suivi des concavités en introduisant une force supplémentaire de pression tendant à *gonfler* le snake jusqu'à ce qu'il rencontre un contour suffisamment marqué. Cela suppose toutefois que l'état initial de la courbe la situe entièrement à l'intérieur de la zone à segmenter. Cette méthode est donc surtout employée dans des applications semi-automatiques où l'utilisateur définit au moins une position et une taille initiales pour la courbe.
- le *snake* GVF (pour Gradient Vector Flow), dont le but est de permettre qu'une initialisation lointaine de la courbe ne pénalise pas la segmentation. Une carte des lignes de gradient est établie sur tout le domaine de l'image et sert à intégrer une force supplémentaire dans l'énergie totale, qui attire la courbe vers la zone de fort gradient.
- les *level-sets*, dont la particularité est de ne pas employer directement une courbe paramétrique plane mais de définir l'évolution des frontières comme l'évolution temporelle de l'ensemble des points d'élévation nulle d'une surface 3D soumise à un champ de force. Les propriétés des contours actifs par *level-sets* se sont révélées intéressantes, en particulier la faculté de se disjoindre ou de fusionner, mais les temps de calcul se sont avérés très pénalisants. Après la formulation initiale de Osher et Sethian en 1988 [67], plusieurs façons de réduire le coût du calcul ont été formulées, dont les plus importantes restent la technique dite *narrow band* [3] (bande étroite) qui ne calcule à chaque itération que les points dans une bande étroite autour du plan $z = 0$ de l'itération courante et celle du *fast marching* [86] qui s'applique dans le cas particulier d'une évolution monotone des fronts.
- les *snake* orientés régions, qui visent essentiellement à mieux caractériser les zones à segmenter et améliorer la robustesse vis-à-vis du bruit en employant une formulation de l'énergie calculée sur le domaine complet de l'image [26, 82]. Les premiers résultats confirment la qualité de cette méthode, mais la nécessité d'effectuer les calculs sur l'image entière générait des temps de traitement prohibitifs jusqu'à ce que Bertaux *et*

al. proposent une amélioration algorithmique exacte permettant à nouveau un calcul en 1D, le long de la courbe, moyennant une simple étape initiale générant un certain nombre d'images intégrales [25, 41, 42]. La section 6.2.2 en donnera une description détaillée.

5.1.5/ Méthodes hybrides

Aujourd'hui, les algorithmes de segmentation les plus performants en termes de qualité emploient des techniques qui tentent de tirer le meilleur parti de plusieurs des méthodes « historiques » décrites précédemment. Le meilleur exemple, et le seul que nous citerons, est le détecteur de contour et l'algorithme de segmentation associé proposé par Arbelaez *et al.* en 2010 [6]. Les auteurs construisent des histogrammes locaux pour générer une matrice de similitude (*affinity matrix*) et appliquent les techniques liées à la théorie des graphes pour réduire la dimension de l'espace de représentation (calcul des valeurs et vecteurs propres). Ils utilisent ensuite une technique adaptée de l'algorithme intitulé *ligne de partage des eaux* pour en regrouper les segments. Les résultats sont très bons et des implémentations efficaces ont d'ores et déjà été écrites (voir section 5.2.5).

5.2/ Les implémentations des techniques de segmentation sur GPU

La problématique tant étudiée de la segmentation n'a pas échappé à l'engouement des chercheurs pour les processeurs graphiques modernes. Un certain nombre de travaux proposent ainsi des implémentations GPU plus ou moins directes de méthodes de segmentation tirant parti de l'architecture massivement parallèle de ces matériels. La majorité d'entre elles cherche à répondre à des besoins liés à l'imagerie médicale allant de la simple extraction des contours d'un organe, d'une tumeur, etc., à la mesure de leur volume ; le traitement en 3D n'étant dans ce cas pas un choix mais une obligation, justifiant d'autant plus l'emploi des GPUs. La nature des tissus et les formes à identifier sont extrêmement variées. Ces images sont souvent très bruitées avec des modèles de bruit qui varient selon l'instrumentation employée. Enfin, le diagnostic médical requérant la plus grande précision possible, aucune solution générique satisfaisante de segmentation n'a encore pu émerger dans ce cadre, laissant place à autant d'implémentations adaptées que de besoins médicaux spécifiques.

Beaucoup d'algorithmes récents destinés à la segmentation comportent plusieurs phases de calcul et mettent en œuvre différents algorithmes réalisant des fonctions élémentaires comme la réduction de bruit ou le calcul d'histogramme. Selon le type de traitement à effectuer sur le GPU, on peut être amené à en concevoir des implémentations parallèles adaptées, ou bien simplement exécuter de multiples instances indépendantes d'une version séquentielle classique du traitement. Dans les deux cas, même si les articles décrivant ces solutions utilisent sans distinction l'expression « implémentation GPU », cela recouvre des réalités et aussi des niveaux de performance souvent très différents.

5.2.1/ CALCUL D'HISTOGRAMME

Étant donné que les segmentations par analyse d'histogramme sont aujourd'hui cantonnées à des applications très particulières et que, dans la pratique, ces traitements sont souvent réalisés par des circuits spécialisés ou programmables de type FPGA, il serait illusoire d'espérer les concurrencer par une solution de type GPU, plus coûteuse, plus volumineuse et vraisemblablement moins robuste.

Le calcul d'histogramme est cependant utilisé de manière intensive dans certains algorithmes de haut-niveau, en particulier le *level-set* et le *gPb*. À ce titre, il faut citer les travaux de Fluck *et al.* [36] qui apportent une réponse efficace au calcul d'histogramme sur GPU, en parvenant à conserver les données dans la mémoire du processeur graphique tout au long de la segmentation par level-set qui était leur motivation initiale [56].

Les résultats annoncés ont été obtenus sur un GPU GeForce 7900 et proclament calculer en 1,6 ms les deux histogrammes nécessaires (64 classes chacun) sur une image de 256×256 pixels en niveau de gris.

5.2.2/ PARTITIONNEMENT DE GRAPHE

Le domaine du traitement des graphes est très actif et peut fournir des éléments pour la segmentation comme l'implémentation du *minimum spanning tree* décrite dans [99] qui annonce la construction du minimum spanning tree d'un graphe de 5 millions de nœuds et 30 millions de liens en moins d'une seconde. En raison de l'indépendance des blocs de threads, la parallélisation GPU des opérations sur les graphes n'est pas simple et peu de travaux font état d'implémentations efficaces mettant en œuvre ces techniques : parmi eux, l'implémentation GPU de l'algorithme *push-relabel* qui effectue le partitionnement selon l'approche *min cut/max flow* a donné lieu aux trois versions remarquables que nous détaillons ci-dessous.

1. Dans [33] une approche assez directe est mise en œuvre et parvient à *binariser* une image de 1 MP en 29 ms (GeForce 6800GT).
2. Les auteurs de [98] remarquent qu'après un nombre réduit d'itérations, très peu de nœuds changent de segment. En conséquence, certains blocs sont activés sans avoir de traitement à effectuer, ce qui retarde le traitement éventuel des blocs en attente. Pour réduire les effets de ce comportement, un indicateur d'activité est calculé à chaque itération et pour chaque bloc, en se basant sur le nombre de changements de segment qui vient d'y être effectué. À l'itération suivante, seuls les blocs considérés comme *probablement* actifs seront activés, réduisant ainsi la latence globale. Un reparamétrage dynamique du graphe après chaque itération est également effectué selon la méthode décrite par Kohli et Torr [54]. Ces optimisations permettent d'atteindre un débit d'environ 30 images de 0.3 MP par seconde sur GTX280, ce qui représente un bond en avant en termes de performance.
3. Enfin, Stitch a proposé dans [91] des optimisations plus étroitement liées à l'architecture des GPUs Nvidia en faisant qu'un même thread mette à jour plusieurs liens du graphe et aussi en compactant la représentation mémoire des indicateurs de changement de segment grâce à l'emploi d'un seul bit par lien, ce qui permet de mémoriser l'état de 32 liens dans une seule variable de type entier. Cela a permis aussi d'accélérer la convergence de l'algorithme, comme le montre la courbe de la

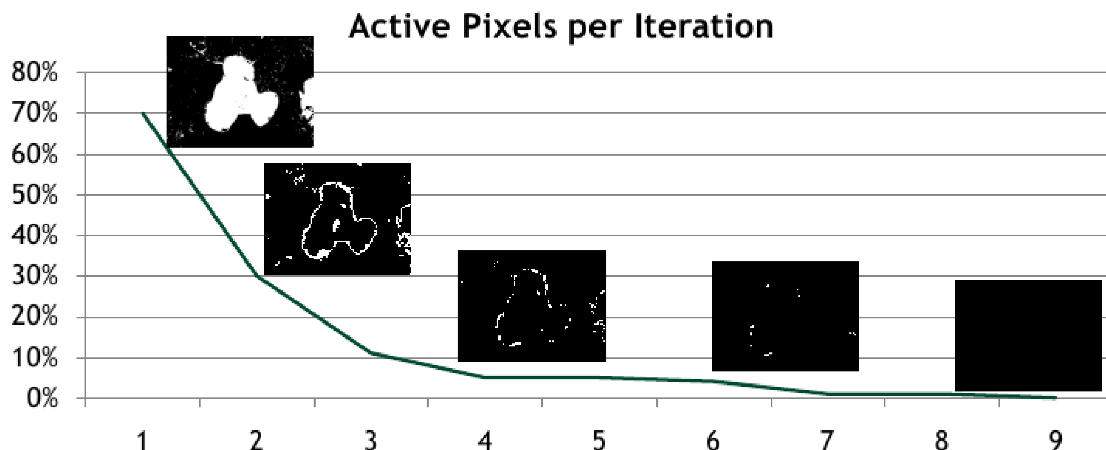


FIGURE 5.6 – Évolution du nombre de pixels actifs pour les itérations successives de l'implémentation de l'algorithme push-relabel de [91]. Les petites images montrent la localisation des pixels actifs après chaque itération, en blanc.

figure 5.6 (tirée de [91]), et d'atteindre les 70 images par seconde dans les mêmes conditions que précédemment (sur C1060).

5.2.3/ K-MEANS, MEAN-SHIFT ET APPARENTÉS

La popularité de l'algorithme des *k-means* a induit des tentatives de portage sur GPU dont [20] qui a implémenté de manière directe l'étiquetage des éléments ainsi qu'une réduction partielle, par bloc, pour la mise à jour des centres ; la réduction finale étant réalisée par le CPU. Cette solution conduit à un transfert des données à chaque itération et ne permet pas d'atteindre des performances élevées. Le temps annoncé pour l'exécution d'une seule itération sur l'ensemble des 819200 éléments de la base de test KDD-Cup-99 [1] comportant 23 segments s'élève à 200 ms. Qui plus est, cette durée n'inclut ni la réduction ni les transferts, l'accélération revendiquée semble alors très discutable.

Dans [47], l'ensemble des tâches d'étiquetage et de mise à jour des centres est réalisé sur le GPU. L'étape de réorganisation des données est exécutée sur le CPU, mais s'avère moins pénalisante que dans la solution précédente, car elle permet de présenter au GPU des données réorganisées pour l'exécution parallèle de la mise à jour des centres (opération de réduction). Les temps d'exécution par itération sont sensiblement les mêmes que pour [20] mais ils incluent cette fois l'ensemble des calculs (hors transferts). Les auteurs fournissent aussi des mesures des temps d'exécution à convergence, qui atteignent la vingtaine de secondes pour le même ensemble de test.

L'implémentation la plus convaincante de *k-means* reste à notre connaissance celle décrite dans [87] où la totalité du traitement est effectuée sur le GPU, moyennant l'emploi d'une texture par segment de données. Les mesures ont montré que cette multiplication du nombre des textures ne constituait pas un facteur de perte de performance, tout du moins jusqu'aux limites des tests, conduits avec un maximum de 32 segments dans des ensembles de 1 million d'éléments. Sur GPU GeForce 8500GT, les temps d'exécution obtenus dans ces conditions sont de 13,8 ms par itération, avec une dépendance très réduite vis-à-vis du nombre de segments.

Un algorithme de *mean-shift* est mis en œuvre dans des travaux à orientation non médicale pour la poursuite de cibles dans des séquences vidéo [58]. L'accélération obtenue par rapport aux implémentations séquentielles existantes n'est que d'un facteur 2. La solution présentée effectue préalablement une réduction de l'espace colorimétrique via un regroupement par la méthode *k-means*, utilisée dans une version séquentielle. Un gain potentiel de performance pourrait être apporté en employant une implémentation GPU du *k-means*, mais serait toutefois limité en raison des itérations nécessaires plus nombreuses pour le traitement *mean-shift*. Par ailleurs, l'implémentation proposée fait un usage intensif de la mémoire partagée et se heurte à sa limite de 16 Ko par bloc, obligeant à réduire la taille des blocs à l'exécution et avec eux, le parallélisme et vraisemblablement aussi la performance de l'application. On peut malgré tout raisonnablement espérer qu'une telle solution présenterait de meilleures performances sur une carte de type Fermi possédant jusqu'à 48 Ko de mémoire partagée par bloc.

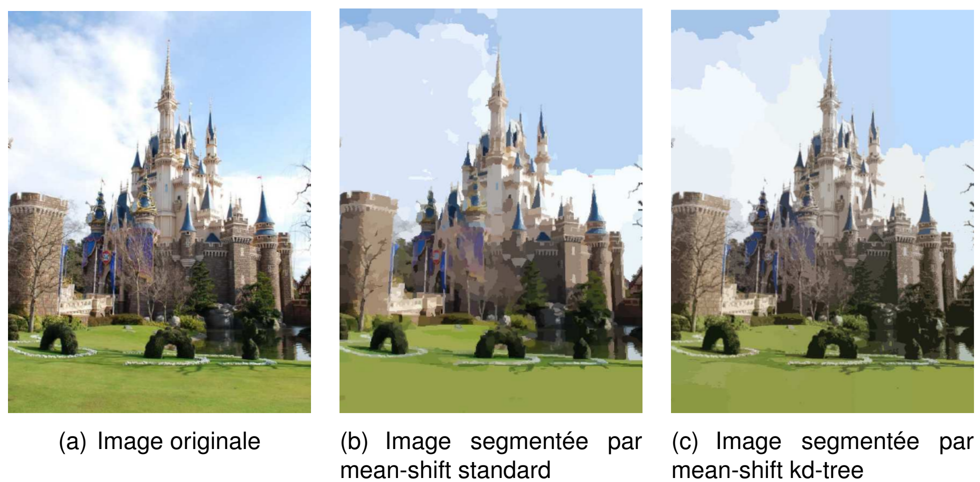
Quick shift est une solution permettant d'obtenir un résultat en une seule passe (sans itérer) par approximation de l'algorithme mean-shift gaussien (dont les masques de pondération sont des gaussiennes). Proposée initialement dans [97], cette technique a été parallélisée sur GPU par ses auteurs et décrite dans [40]. Les performance y sont obtenues grâce à des approximations, parmi lesquelles la restriction des calculs de pondération à des voisinages de rayon 3σ (écart type de la gaussienne définissant les coefficients du masque), considérant qu'au delà, les valeurs en sont négligeables. On construit ensuite un arbre des liens entre les pixels, en limitant la recherche à une distance maximale de σ et en divisant arbitrairement par 2 la dynamique de l'espace colorimétrique. La segmentation est finalement obtenue par simple partionnnement de l'arbre selon un seuil τ.

Pour s'affranchir de la relative petite taille de la mémoire partagée sans pâtir de la grande latence des accès à la mémoire globale du GPU, les auteurs ont choisi d'associer l'image et l'estimation de densité à des textures et ainsi bénéficier du mécanisme de cache 2D. Les expérimentations ont été menées avec différentes valeurs de σ et τ choisies pour les résultats visuels qu'elles induisent et permettent de segmenter une image couleur de 1 MP en environ 1 s avec $\tau = 10$ et $\sigma = 6$. Toutefois, des valeurs plus petites, requérant moins de calculs, permettent des temps d'exécution beaucoup plus courts. Les courbes présentées permettent d'envisager, pour $\tau = 4$ et $\sigma = 2$, une réduction par 30 du temps d'exécution, soit environ 33 ms. Une version améliorée récemment, dans laquelle les positions des centres sont stockées en registres, permet selon les auteurs, de diviser encore par 2 les temps d'exécution pour atteindre une segmentation en environ 16,5 ms. La figure 5.7, tirée de [40], présente quelques segmentations effectuées avec des valeurs différentes, permettant ainsi de juger des effets des variations des paramètres τ et σ.

Récemment, Xiao et Liu ont décrit dans [106] une implémentation de l'algorithme *mean-shift* qui utilise cette fois une construction de *KD-tree* (arbre binaire à K dimensions) pour réduire l'espace colorimétrique et effectuer rapidement les recherches des plus proches voisins. L'ensemble s'exécute sur le GPU et permet ainsi d'obtenir des résultats beaucoup plus probants puisque les auteurs revendiquent une segmentation d'image couleur de 6.6 millions de pixels en 0.2 secondes. Malheureusement, il n'est pas dit combien de segments comprend l'image et il n'est fait référence qu'à une seule image, dont on déduit qu'il s'agit de l'image de la figure 5.8, reproduite afin de montrer les différences avec une implémentation standard du *mean-shift*.

(a) Image originale (b) $\tau = 10$ et $\sigma = 2$ (c) $\tau = 10$ et $\sigma = 10$ (d) $\tau = 20$ et $\sigma = 10$

FIGURE 5.7 – Segmentation d'une image couleur de 512×512 pixels par l'implémentation GPU quick-shift de [40].



(a) Image originale (b) Image segmentée par mean-shift standard (c) Image segmentée par mean-shift kd-tree

FIGURE 5.8 – Comparaison des segmentations d'une image couleur de 2256×3008 pixels réalisées par *mean-shift* standard et par le *mean-shift kd tree* de [106].

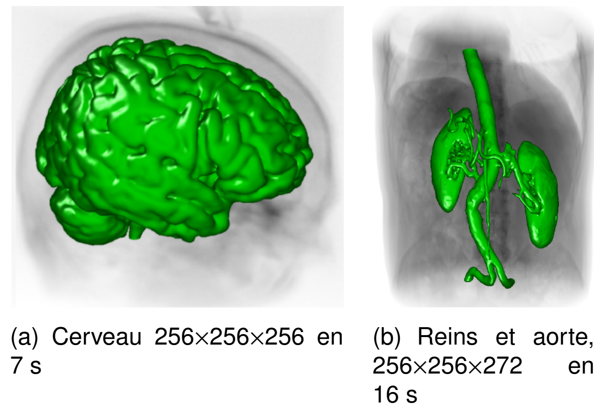
5.2.4/ Level set et snakes

Dès 2003, on recense d'importants travaux liés à l'imagerie médicale mettant en œuvre des algorithmes *level set* sur GPU. C'est le cas de [55, 56] où les auteurs décrivent une solution de visualisation des coupes d'une mesure volumique réalisée par résonance magnétique (IRM) en exploitant pour la première fois le caractère creux du système d'équations à résoudre. Cette variante est nommée *narrow-band* et diffère de la première solution 2D présentée dans [83] qui implémente la version standard des *level set*. En ne transférant au GPU, pour chaque itération, que les petits pavés de données actifs et en les rangeant de manière contiguë en texture pour optimiser les accès en lecture, les auteurs sont parvenu à effectuer, pour des données volumiques de 256×256×175, entre 3,5 et 70 itérations par seconde, qu'il faut comparer aux 50 itérations par seconde en 2D sur image de 128×128 pixels obtenues dans [83]. La limitation principale de cette solution est celle des dimensions maximales admises pour une texture qui était de 2048×2048 pour le GPU ATI Radeon 9800 pro employé (et programmé en openGL, car ni openCL ni CUDA n'étaient encore disponible à l'époque).

Les autres solutions GPU proposées depuis sont également basées sur la variante *narrow-band* (bande étroite) des *level-set* [57, 18, 49], mais seule [49] s'affranchit des transferts CPU/GPU à chaque itération pour déterminer et transférer les pavés actifs. La solution retenue est d'employer les opérations atomiques pour assurer l'accès exclusif à la liste des pavés en mémoire GPU. Cela permet de descendre à 3 ms par itération pour une image de 512×512 pixels.

L'implémentation la plus performante à ce jour est celle décrite dans [81] qui parvient à des itérations dont la durée varie, sur GTX280, de 1,8 à 6,5 ms pour des données volumiques de 256×256×256 pixels, issues d'examens IRM, pour une moyenne de 3,2 ms sur les 2200 itérations de l'exemple fourni (cerveau en 7 s, Figure 5.9(a)). Une optimisation poussée y a été effectuée pour rendre l'algorithme efficace, en particulier grâce à la refonte du code responsable de la détermination des pavés actifs, qui parvient cette fois à déterminer l'ensemble minimal de pavés actifs et à rendre cette détermination efficace sur le GPU en gérant parallèlement plusieurs tampons, chacun associé à une direction particulière en 6-connexité. Une étape de résolution des doublons est ensuite effectuée avant de les compacter de manière contiguë comme cela était déjà fait dans [55].Tout cela est réalisé sans recourir à la mémoire partagée qui s'avère complexe, voire impossible, à utiliser efficacement lorsque les éléments à accéder sont très irrégulièrement répartis en mémoire.

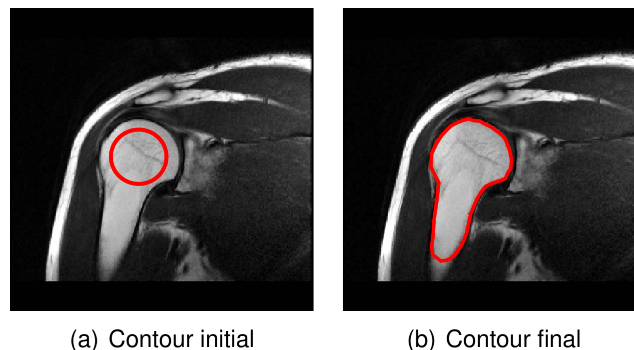
Ce faisant, les auteurs parviennent à réduire le nombre total cumulé de pavés qu'il est nécessaire de traiter lors des 2200 itérations de la segmentation de l'image d'exemple, avec seulement 294 millions de pavés, alors que l'algorithme *narrow-band* standard devait en traiter 4877 millions. Il est à noter que la durée d'exécution d'une itération dans cette variante dépend plus fortement de la proportion de pavés actifs que pour le *narrow-band* standard. Les deux courbes sont globalement affines et se croisent pour une proportion de pavés actifs proche de 10%. Si l'on considère que malgré les stratégies adoptées, tenir à jour cette liste de pavés représente encore 77% du temps de calcul, cela peut représenter une piste pour une optimisation supplémentaire qui, bien qu'apparemment superflue pour l'image du cerveau, pourrait se justifier pour d'autres, comme le suggère le temps de segmentation de 16 s nécessaire pour l'image des reins (Figure 5.9(b)) et de l'aorte, aux dimensions comparables.



(a) Cerveau 256×256×256 en 7 s

(b) Reins et aorte, 256×256×272 en 16 s

FIGURE 5.9 – Segmentation d'images issues d'examens IRM par la méthode des level set à bande étroite.



(a) Contour initial

(b) Contour final

FIGURE 5.10 – Segmentation d'une image d'épaule en 1024^2 pixels issue d'un examen IRM par l'implémentation du snake GVF de [44]. Le contour est représenté en rouge et le contour final est obtenu en 11 s. Le tracé initial du contour a été artificiellement épaissi pour le rendre visible à l'échelle de l'impression.

Les algorithmes de type *snake*, très coûteux en temps de calcul, pouvaient prétendre à bénéficier largement de la technologie des GPU pour améliorer leurs performances, mais seule la variante paramétrique GVF à été véritablement implémentée de manière spécifique et efficace [44, 8, 59, 89]. Les variantes de type géométrique restent à ce jour sans implémentation GPU, principalement en raison de l'irrégularité des motifs d'accès à la mémoire.

Parmi les premières solutions décrites, [44] propose une implémentation réalisée en openGL, où les données de gradient sont compactées en texture RVBA de manière à s'affranchir du format 16 bits de la représentation : les deux premiers canaux R et V contiennent les valeurs représentant respectivement les gradients selon dx et dy sous une forme codée par la valeur des 2 autres canaux. Par ailleurs, une approximation du système linéaire à résoudre est proposée afin de donner une structure bande symétrique à la matrice à inverser, ce qui améliore considérablement l'efficacité des accès aux données au travers du cache.

Les performances annoncées montrent tout d'abord que l'approximation adoptée n'a qu'un impact extrêmement limité sur le résultat de la segmentation avec un écart radial maximal inférieur à 1.3 pixel par rapport au calcul exact effectué sur CPU. Enfin, la seg-

mentation de l'image d'exemple en 1024×1024 pixels s'effectue en un total de 11 s après l'initialisation manuelle reproduite à la figure 5.10. Cela est annoncé comme presque 30 fois plus rapide que l'implémentation CPU de référence, mais demeure beaucoup trop lent pour un usage interactif.

Une solution directe employant la transformée de Fourier pour inverser le système à résoudre a été décrite récemment dans [110], et programmée en employant la bibliothèque openGL. Les exemples fournis montrent des objets segmentés dans des images d'environ 10000 pixels en une durée de l'ordre de la demi seconde.

En adaptant sur GPU une variante du snake GVF dite FD-snake (pour *Fourier Descriptors*) [59] permettant une convergence plus rapide et un calcul parallèle beaucoup plus adapté au GPU, Li *et al.* parviennent à suivre les déformations d'un contour en temps réel dans des images issues d'examens échographiques. Un contour de 100 points peut ainsi converger convenablement en à peine 30 ms. Une contribution supplémentaire de cette implémentation est de permettre une initialisation simplifiée et semi-automatique du contour.

La plus aboutie des implémentations actuelles du snake GVF est celle présentée par Smistad *et al.* dans [89] où les auteurs ont concentré leur effort sur l'optimisation des accès mémoire lors du calcul du GVF. Ils ont comparé 8 combinaisons possibles impliquant l'emploi des mémoires partagée et de texture ainsi que la représentation des nombres selon le format classique 32 bits ou selon un format compressé sur 16 bits. Il en ressort que l'association la plus performante est celle des textures et du format de données sur 16 bits. Les performances sont alors nettement en hausse avec des segmentations d'images médicales d'IRM de 512×512 pixels effectuées en 41 ms sur Nvidia C2070 et 28 ms sur ATI 5870 (512 itérations). L'implémentation réalisée en openCL permet d'exécuter le code sur les GPUs des deux principaux fabricants.

5.2.5/ Algorithmes hybrides

Le détecteur de contour *gPb* décrit dans [6] et que l'on considère comme la référence actuelle pour la segmentation d'objets et personnages dans des image naturelles, a été implémenté en CUDA par Cantazaro *et al.* et est décrit dans [17]. Dans cette implémentation GPU, La qualité des contours extraits est préservée et le temps de traitement est réduit d'un facteur supérieur à 100 par rapport à la version CPU de [6] : les contours des images de 0.15 MP de la base de test BSDS [63] sont ainsi traitées en 2 secondes environ sur GPU C1060. L'apport principal de ces travaux réside dans la solution conçue pour le calcul des histogrammes locaux, qui dans l'algorithme original s'étendaient sur des demi-disques centrés sur chaque pixel. La parallélisation réalisée fait l'approximation de chaque demi-disque en un rectangle de même surface dont un des grands cotés a le centre du disque pour milieu. Les rectangles sont ensuite pivotés par une rotation basée sur la discrétisation de Bresenham [12] pour en aligner les cotés avec les cotés de l'image et pouvoir employer la technique des images cumulées pour calculer rapidement l'histogramme. La figure 5.11 présente quelques résultats d'extraction de contours.

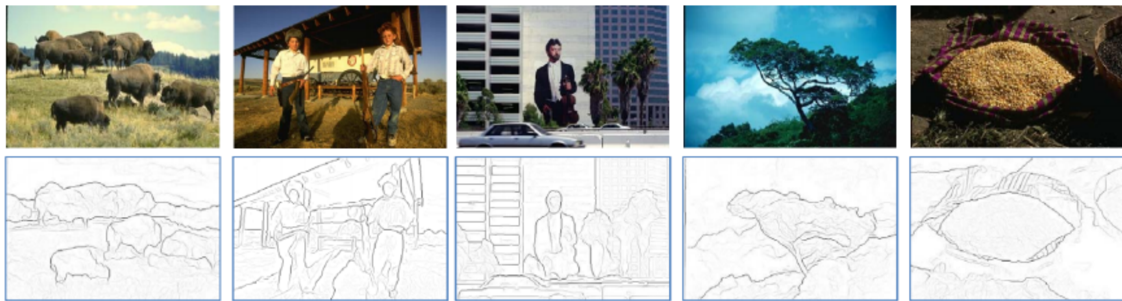


FIGURE 5.11 – Extraction de contour par la version GPU de l'algorithme gPb. Les images sont issues de la base BSDS [63]

5.3/ CONCLUSION

La présentation que nous venons de faire des principales techniques de filtrage et de segmentation ainsi que des implémentations sur GPU qui leur ont été consacrées nous ont permis de confirmer le fait que, malgré leur orientation grand public et les langages de haut niveau permettant d'accéder rapidement à la programmation GPU, la parallélisation efficace d'un algorithme séquentiel destiné à s'exécuter sur ces processeurs n'est pas triviale. Le modèle et les contraintes de programmation sont très spécifiques et obtenir un code efficace découle nécessairement d'un compromis qui peut parfois être complexe à mettre au point.

Ajoutons que les générations successives de GPU conservent certes des caractéristiques communes mais diffèrent suffisamment quant à la distribution des ressources, rendant toute généralisation vaine et faisant qu'un code optimisé pour un modèle donné peut devenir moins rapide avec un modèle plus récent. Prenons l'exemple du nombre maximal de registres utilisables par thread ; il est de 128 sur GPU C1060 contre seulement de 63 pour un C2070. Un code faisant un usage optimisé des registres sur C1060 pourra s'exécuter plus lentement sur C2070. C'est un cas de figure sur lequel nous reviendrons plus en détail dans le chapitre 8 consacré au filtre médian.

Cet état de fait rend les résultats publiés par les chercheurs souvent délicats à interpréter et plus encore à reproduire lorsque l'on souhaite comparer les performances de nos propres codes avec les références du moment, sauf à disposer d'un panel de cartes GPU représentant toutes les évolutions de l'architecture et ce pour au moins les deux grands fabricants de GPUs que sont ATI et Nvidia.

Pour aider les développeurs à allouer les ressources de manière optimale, ou tout du moins estimer le degré d'optimisation de leur code à l'aune de la vitesse d'exécution, Nvidia fournit une feuille de calcul appelée *occupancy calculator* dans laquelle on peut entrer les paramètres d'exécution d'un *kernel* parallèle : nombre de registres utilisés par chaque thread, quantité de mémoire partagée, modèle de GPU, dimensions de la grille. Le tableur retourne alors l'indice de charge (l'*occupancy*) qui traduit le rapport, à chaque instant, entre le nombre de warps actifs et le nombre maximal de warps par SM. L'occupancy se traduit donc par un indice compris entre 0 et 100% et la recherche de performance semble devoir être la recherche de l'occupancy maximale.

Toutefois, comme l'a clairement démontré Volkov dans [100], ce paradigme peut aisément être remis en cause et Volkov parvient effectivement à améliorer les performances d'un

certain nombre d'exemples génériques dans des conditions de faible valeur d'*occupancy*. Enfin, nous avons pu constater deux grands modèles d'accès aux données : les algorithmes de filtrage usent quasiment tous de la mémoire partagée comme tampon d'accès aux données de l'image en mémoire globale (ou texture) alors que les algorithmes de segmentation performants s'en affranchissent. La raison en est clairement des motifs d'accès très irréguliers et non contigus pour ces derniers, rendant la gestion efficace de la mémoire partagée délicate et potentiellement si coûteuse qu'elle en devient sans intérêt. Les chapitres présentant nos contributions reviendront sur ces aspects en proposant des solutions pour accroître les performances des algorithmes parallèles.

Algorithmes GPU rapides pour la réduction de bruit et la segmentation

6

LA SEGMENTATION PAR SNAKE POLYGONAL ORIENTÉ RÉGIONS

6.1/ INTRODUCTION

La principale difficulté soulevée par l'emploi d'algorithmes de type *snake* orientés contour est le choix de la fonction d'énergie externe et la détermination de la nature des images auxquelles elle convient. Dans l'approche orientée régions, les deux régions que sont l'extérieur et l'intérieur du contour (cas mono cible) sont prises en compte dans l'estimation de la forme du contour ; cela permet d'extraire des formes dans des images où les contours de la cible sont mal définis, en raison d'un fort niveau de bruit par exemple. Les algorithmes découlant de cette approche n'ont fait l'objet, à notre connaissance, d'aucune parallèlisation sur GPU, malgré le grand intérêt qu'elles revêtent dans l'interprétation d'images fortement bruitées (RADAR, médicales,...) et le besoin de réduire suffisamment les temps de traiement pour permettre l'interactivité.

Nous proposons dans la suite de ce chapitre de commencer par détailler l'algorithme séquentiel que nous avons pris comme référence, puis d'en présenter la version parallèle pour GPU que nous avons conçue. L'algorithme a été décrit et proposé initialement en 1999 par Chesnaud *et al.* dans [25]. L'implémentation que les auteurs ont développée a continué d'être améliorée jusqu'à aujourd'hui et est employée comme brique élémentaire dans des algorithmes plus complexes. La version qui sert de référence ici est une implémentation séquentielle optimisée qui met à profit les capacités de parallélisme des CPU actuels en employant le jeu d'instruction SSE2 des microprocesseurs. La description que nous en faisons dans les lignes qui suivent est très largement inspirée de [25] à la différence que nous n'implémentons pas les critères de régularisation du contour ni de minimisation de la longueur de description pour nous focaliser sur la déformation du contour et sa convergence.

6.2/ PRÉSENTATION DE L'ALGORITHME

6.2.1/ FORMULATION

À l'intérieur de l'image observée $\bar{v}$, soient $\bar{t}$ le vecteur composé par les niveaux de gris des N_t pixels de la région cible Ω_t et $\bar{b}$ celui des N_b pixels du fond Ω_b. Les vecteurs $\bar{t}$ et $\bar{b}$ sont supposés non corrélés et sont caractérisés par leurs densités de probabilité (PDF) respectives p^{Θ_t} et p^{Θ_b} ; Θ_t et Θ_b étant les vecteurs des paramètres de leurs PDF. Dans

le cas gaussien que nous supposerons ici parcequ'il est considéré comme prépondérant dans les images naturelles, $\Theta = (\mu, \sigma)$ où μ est la moyenne et σ^2 est la variance. On note Γ le contour de la région cible ($\Gamma \in \Omega_t$), que l'on suppose continu en connexité à 8 voisins.

Le but de la segmentation est alors de déterminer la géométrie de Γ qui maximise un critère de vraisemblance généralisée (GL). La vraisemblance sur l'ensemble de l'image, *i.e* la région Ω, est donnée par

$$P(\bar{v}|\Omega_t, \Omega_b, \Theta_t, \Theta_b) = P(\bar{v}|\Omega_t, \Theta_t)\, P(\bar{v}|\Omega_b, \Theta_b) \tag{6.1}$$

soit en développant

$$P(\bar{v}|\Omega_t, \Omega_b, \Theta_t, \Theta_b) = \prod_{x_k \in \Omega_t} p^{\Theta_t}(v_k, \Theta_t) \prod_{x_k \in \Omega_b} p^{\Theta_b}(v_k, \Theta_b) \tag{6.2}$$

Dans le cas gaussien, la PDF étant de la forme

$$p(\alpha) = \frac{1}{\sigma\sqrt{2\pi}} \mathrm{e}^{-\frac{(\alpha-\mu)^2}{2\sigma^2}} \tag{6.3}$$

La substitution de (6.3) dans (6.2), suivie du logarithme, permet d'obtenir l'expression de la *log-vraisemblance*

$$-N_t ln\left(\sqrt{2\pi}\right) - N_t ln(\sigma_t) - \frac{1}{2\sigma_t^2} \sum_{x_k \in \Omega_t} (v_k - \mu_t)^2 - N_b ln\left(\sqrt{2\pi}\right) - N_b ln(\sigma_b) - \frac{1}{2\sigma_b^2} \sum_{x_k \in \Omega_b} (v_k - \mu_b)^2 \tag{6.4}$$

dans laquelle les vecteurs Θ_t et Θ_b sont estimés suivant la méthode du maximum de vraisemblance, qui donne l'expression générique suivante pour l'estimée de Θ_t, notée $\widehat{\Theta_t}$ et transposable à l'identique pour Θ_b

$$\widehat{\Theta_t} \begin{cases} \widehat{\mu_t} = \frac{1}{N_t} \sum_{x_k \in \Omega_t} v_k \\ \widehat{\sigma_t^2} = \frac{1}{N_t} \sum_{x_k \in \Omega_t} (v_k - \widehat{\mu_t})^2 \end{cases} \tag{6.5}$$

En insérant les expressions de (6.5) dans (6.4), il reste, à une constante près, le critère de vraisemblance généralisée suivant, noté GL, que l'on cherche à optimiser en déterminant la géométrie de Γ qui en maximise la valeur et épousera alors au mieux la forme du contour de la cible.

$$GL = \frac{1}{2}\left(N_t ln\left(\widehat{\sigma_t^2}\right) + N_b ln\left(\widehat{\sigma_b^2}\right)\right) \tag{6.6}$$

6.2.2/ Optimisation des calculs

La maximisation de GL est effectuée en employant une technique itérative où sa valeur doit être calculée à chaque déformation du contour Γ. Si l'on se reporte à l'équation (6.5), on voit que l'obtention de la valeur de GL nécessite, à chaque évaluation d'une géométrie

donnée de Γ, le calcul des sommes

$$S_v(\Omega_t) = \sum_{x_k \in \Omega_t} v_k \qquad S_{v^2}(\Omega_t) = \sum_{x_k \in \Omega_t} v_k^2$$
$$S_v(\Omega_b) = \sum_{x_k \in \Omega_b} v_k \qquad S_{v^2}(\Omega_b) = \sum_{x_k \in \Omega_b} v_k^2 \tag{6.7}$$

Considérons la région cible Ω_t, les pixels de coordonées (i, j) qui la composent, et généralisons l'écriture des sommes de (6.7) en

$$S_g(\Omega_t) = \sum_{i=i_{min}}^{i=i_{max}} \sum_{j=j_{min}(i)}^{j=j_{max}(i)} g\left(v(i, j)\right) \tag{6.8}$$

où g représente la fonction de valeurs de niveaux de gris à sommer.

En posant

$$T_g(y, \tau) = \sum_{j=0}^{\tau} g\left(v(y, j)\right) \tag{6.9}$$

L'équation (6.8) devient

$$S_g(\Omega_t) = \sum_{i=i_{min}}^{i=i_{max}} \left[T_g(i, j_{max}(i)) - T_g(i, j_{min}(i) - 1)\right] \tag{6.10}$$

qui représente une sommation sur le contour Γ que l'on peut écrire

$$S_f(\Omega_t) = \sum_{(i,j) \in \Gamma} C(i, j)\gamma(i, j) \tag{6.11}$$

où $C(i, j)$ est un coefficient lié à la direction du contour au point (i, j) et $\gamma(i, j)$ prend sa valeur selon les règles suivantes

$$\gamma(i, j) = \begin{cases} T(i, j) & \text{si } C(i, j) = 1 \\ T(i, j-1) & \text{si } C(i, j) = -1 \\ 0 & \text{si } C(i, j) = 0 \end{cases} \tag{6.12}$$

La valeur de $C(i, j)$ est déterminée pour chaque pixel d'indice l du contour en considérant les pixels d'indices $l - 1$ et $l + 1$ qui définissent les deux vecteurs f_{in} et f_{out} et leur code selon le codage de Freeman, comme l'illustre la figure 6.1. La table 6.1 donne les valeurs de $C(i, j)$ selon les valeurs des codes de Freeman des vecteurs f_{in} et f_{out}. Il faut noter que les valeurs dans la table 6.1 diffèrent de celles proposées initialement dans [25]. Cette modification a été proposée pour permettre de s'adapter à la segmentation multi-cibles.

L'intérêt de cette transformation est majeur :
- La sommation en deux dimensions sur la région Ω_t est ainsi réduite à une sommation à une dimension sur le contour Γ.
- Les valeurs $T_g(i, j)$ peuvent être calculées préalablement à la phase de segmentation proprement dite. Pour le cas gaussien qui nous intéresse, cela revient à pré-calculer les trois images *cumulées* S_1, S_x et S_{x^2} définies par

$$S_1(i, j) = \sum_{x=0}^{j} x \quad , \quad S_x(i, j) = \sum_{x=0}^{j} v(i, x) \quad \text{et} \quad S_{x^2}(i, j) = \sum_{x=0}^{j} v(i, x)^2 \tag{6.13}$$

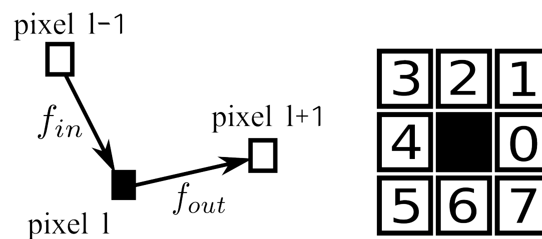


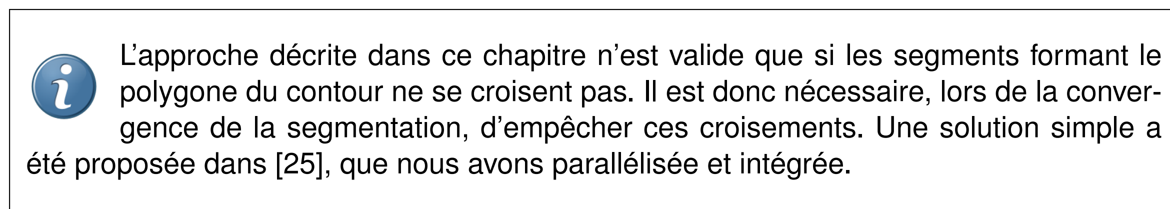
FIGURE 6.1 – À gauche : détermination des vecteurs f_{in} et f_{out}. À droite : code de Freeman d'un vecteur en fonction de sa direction, l'origine étant supposée au pixel central, en noir.

	f_{out}							
f_{in}	0	1	2	3	4	5	6	7
0	0	0	0	0	0	-1	-1	-1
1	1	1	1	1	1	0	0	0
2	1	1	1	1	1	0	0	0
3	1	1	1	1	1	0	0	0
4	0	0	0	0	0	-1	-1	-1
5	0	0	0	0	0	-1	-1	-1
6	0	0	0	0	0	-1	-1	-1
7	0	0	0	0	0	-1	-1	-1

TABLE 6.1 – Valeur du coefficient $C(i, j)$ en fonction des valeurs des codes de Freeman des vecteurs f_{in} et f_{out}.

– Les valeurs du coefficient $C(i, j)$ se calculent très facilement durant la génération du contour Γ.

Par ailleurs, le choix d'un contour polygonal permet d'améliorer l'efficacité de l'algorithme. Lors de la phase de segmentation, le déplacement d'un sommet du polygone n'influe ainsi que sur les pixels des deux segments qui s'y rapportent, réduisant ainsi la quantité de calculs à effectuer à chaque nouvelle déformation du contour.


L'approche décrite dans ce chapitre n'est valide que si les segments formant le polygone du contour ne se croisent pas. Il est donc nécessaire, lors de la convergence de la segmentation, d'empêcher ces croisements. Une solution simple a été proposée dans [25], que nous avons parallélisée et intégrée.

6.2.3/ IMPLÉMENTATION SÉQUENTIELLE

Un des inconvénients des algorithmes de type *snake* est l'influence du contour initial sur la convergence de la segmentation. Pour pallier simplement ce défaut, une technique progressive est adoptée, en initialisant le contour avec peu de sommets (par exemple 4) puis en augmentant le nombre au fur et à mesure de la convergence. L'algorithme 2 décrit macroscopiquement la solution mise œuvre tandis que l'algorithme 3 en présente

les détails.

Algorithme 2 : Principe mis en œuvre pour la convergence du snake polygonal

```
1 Calculer les images cumulées;
2 Initialiser le contour avec 4 sommets;
3 répéter                                                    /* niveau contour */
4 |  répéter                                                 /* niveau sommet */
5 |  |  Déplacer chaque sommet autour de sa position actuelle.;
6 |  |  Déplacer le sommet vers la position induisant le meilleur GL;
7 |  jusqu'à aucun sommet ne peut être déplacé;
8 |  Ajouter un sommet au milieu de chaque grand segment;
9 jusqu'à aucun sommet ne peut être ajouté;
```

Algorithme 3 : Détail de l'implémentation du snake polygonal

```
1  Lire l'image v̄;
2  Calculer les images cumulées S_1, S_x S_{x^2} ;            /* en parallèle via SSE2 */
3  n ← 0 ;                                          /* indice de boucle niveau contour */
4  N_n ← 4 ;                                                    /* nombre de nœuds */
5  Γ ← {Γ_0, Γ_1, Γ_2, Γ_3};
6  d ← d_max ;                                   /* pas de déplacement des nœuds */
7  l_min = 32 ;                           /* longueur mini des segments sécables */
8  Γ_i ← Γ_0 ;                                                  /* sommet courant */
9  GL_ref ← GL(Γ, N_n, v̄, S_y, S_{y^2}) ;   /* voir à partir de ligne 18 pour le détail */
10 répéter                                                      /* niveau contour */
11 |  N_add ← 0;
12 |  répéter                                                       /* niveau nœud */
13 |  |  N_move ← 0;
14 |  |  pour i = 0 à i = N_n − 1 faire
15 |  |  |  Calculer les positions {Γ_i^0, ..., Γ_i^7} ;      /* les 8 voisins de Γ_i */
16 |  |  |  GL_w ← GL_ref − la contribution des segments Γ_{i−1}Γ_i et Γ_iΓ_{i+1};
17 |  |  |  pour w = 0 à w = 7 faire
18 |  |  |  |  Discrétiser les segments Γ_{i−1}Γ_i^w et Γ_i^wΓ_{i+1};
19 |  |  |  |  Lire dans S_1, S_x et S_{x^2} les contributions des pixels de Γ_{i−1}Γ_i^w et Γ_i^wΓ_{i+1};
20 |  |  |  |  Calculer les directions et lire les codes de Freeman ;
21 |  |  |  |  Calculer GL_w incluant les contributions de Γ_{i−1}Γ_i^w et Γ_i^wΓ_{i+1} ;
22 |  |  |  |  si GL_w > GL_ref alors
23 |  |  |  |  |  GL_ref ← GL_w;
24 |  |  |  |  |  Γ_i ← Γ_i^w;
25 |  |  |  |  |  N_move ← N_move + 1;
26 |  |  |  |  fin
27 |  |  |  fin
28 |  |  fin
29 |  |  l ← l + 1;
30 |  jusqu'à N_move = 0;
31 |  pour chaque segment Γ_iΓ_{i+1} faire
32 |  |  si ||Γ_iΓ_{i+1}|| > l_min alors
33 |  |  |  Ajouter un nœud au milieu de Γ_i et Γ_{i+1};
34 |  |  |  N_add ← N_add + 1;
35 |  |  fin
36 |  fin
37 |  N_n ← N_n + N_add;
38 |  si d > 1 alors d ← d/2 sinon d ← 1 ;
39 |  GL_ref ← GL(Γ, N_n, v̄, S_y, S_{y^2}) ;
40 jusqu'à N_add = 0;
```

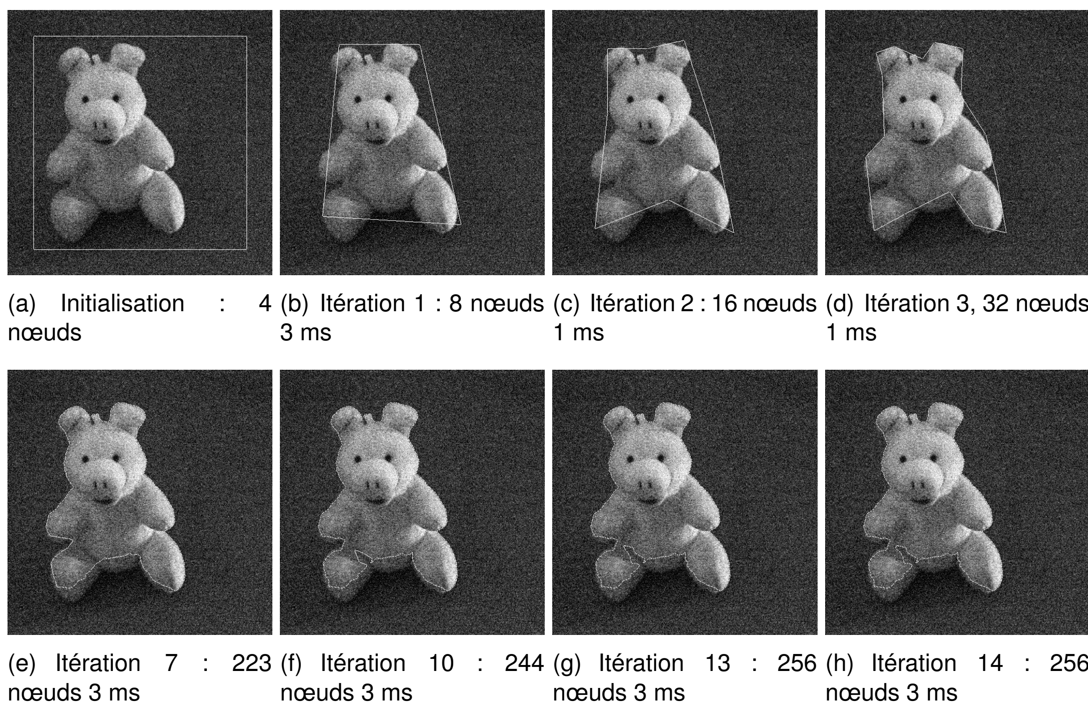
Les différentes sommations nécessaires au calcul de la valeur du critère GL sont effectuées en parallèle à l'aide du jeu d'instructions SSE2, qui permet de travailler avec des registres de grande capacité (128 bits) et d'envisager d'y ranger côte à côte les opérandes des trois sommes pour les effectuer simultanément. Si l'on cherche à traiter des images en niveaux de gris sont codés sur 16 bits, les sommes S_1, S_X et S_{X^2} vont utiliser :
– N_c bits par opérande de chaque somme pour représenter les coordonnées des pixels.
– N_p bits pour traduire le nombre d'opérandes dans chaque somme.
– 16 bits par valeur de niveau de gris dans S_X.
– 32 bits par valeur de niveau de gris au carré dans S_{X^2}.
Les trois sommes utilisent donc, par opérande, un total de $\left(3\left(N_c + N_p\right) + 16 + 32\right)$ bits devant être contenu dans un registre de 128 bits, ce qui nous donne un maximum de 26 bits pour $N_c{+}N_p$. La longueur des segments pouvant être au maximum $\sqrt{2}$ fois supérieure au coté de l'image, on peut donc considérer qu'il est nécessaire d'avoir $N_p = N_c + 1$ pour ne pas générer de restriction sur la longueur des segments. Cela nous conduit donc à $N_c = 12$ et $N_p = 13$ $(12 + 13 = 25 < 26)$. La répartition retenue pour les données dans les registres SSE2 de 128 bits est alors la suivante :
– $N_c + N_p = 25$ bits pour les opérandes des sommes de S_1.
– $N_c + N_p + 16 = 41$ bits pour les opérandes des sommes de S_X.
– $N_c + N_p + 32 = 57$ bits pour les opérandes des sommes de S_{X^2}.

6.2.4/ PERFORMANCES

Les images de 1024×1024 pixels de la figure 6.2 montrent l'évolution du contour lors de la segmentation d'une image photographique prise en faible éclairement et bruitée artificiellement par un bruit gaussien d'écart type 25. Les paramètres de la séquence sont fixés empiriquement aux valeurs $d_{max} = 16, l_{min} = 8$. Les temps d'exécution indiqués sont mesurés sur Intel Xeon E5530-2.4GHz with 12Go RAM et sont les valeurs moyennes obtenues pour 10 exécutions.

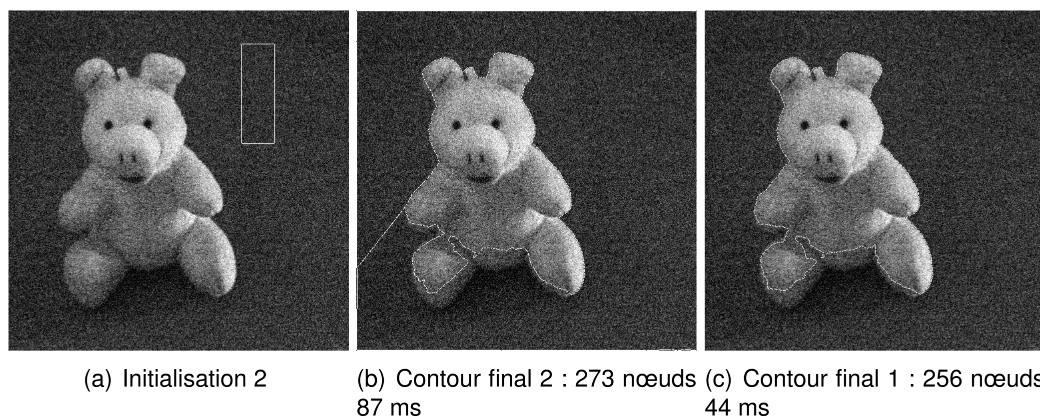
La dépendance vis à vis du contour initial, qui est un des principaux soucis liés au snake est ici fortement relativisée. La figure 6.3 montre le contour final segmentant l'image de test de la figure 6.2 à partir d'un état initial très éloigné du précédent et, *a priori*, très défavorable compte tenu du fait qu'il est loin de la cible et sans intersection avec elle. Toutefois, le contour final est très proche de celui obtenu à partir d'un état initial englobant la cible, malgré un nœud qui s'est « accroché » au bord de l'image. La convergence est également plus longue à obtenir dans ce cas avec 87 ms pour de 17 itérations et 273 nœuds.

La dimension de l'image à traiter a également un effet sur le résultat et naturellement sur le temps de calcul. Si l'on conserve les mêmes paramètres d'optimisation que pour la segmentation de l'image 512×512 pixels et un contour initial dont les cotés sont à une distance des bords équivalente à 10% des cotés de l'image, le résultat sur une image identique de $4000^2 4000$ pixels est obtenu en 1,3 s avec 1246 nœuds ; il est reproduit à la figure 6.4(a). Le nombre de pixels appartenant à la région cible est tel que l'amplitude des déplacements autorisés pour chaque nœud (d) peut se révéler trop faible vis-à-vis du seuil d'acceptation des mouvements. On observe que les zones à gradient élevé ne posent pas de problème et sont détourées de la même manière, alors que dans le bas de l'image où figure une zone de gradient faible (ombre), la cible se trouve maintenant quelque peu surévaluée en surface là ou elle était plutôt sous évaluée dans l'image en



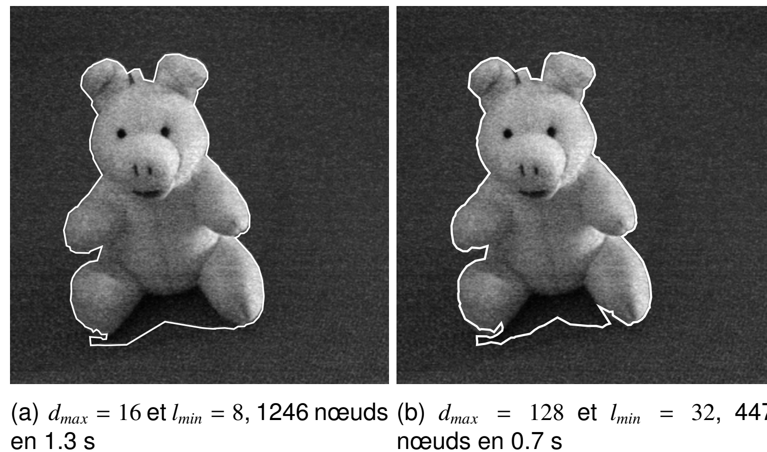
(a) Initialisation : 4 nœuds
(b) Itération 1 : 8 nœuds 3 ms
(c) Itération 2 : 16 nœuds 1 ms
(d) Itération 3, 32 nœuds 1 ms
(e) Itération 7 : 223 nœuds 3 ms
(f) Itération 10 : 244 nœuds 3 ms
(g) Itération 13 : 256 nœuds 3 ms
(h) Itération 14 : 256 nœuds 3 ms

FIGURE 6.2 – Évolution du contour lors de la segmentation d'une image de 512^2 pixels. La convergence est obtenue à l'itération 14 après 44 ms pour un total de 256 nœuds.



(a) Initialisation 2
(b) Contour final 2 : 273 nœuds 87 ms
(c) Contour final 1 : 256 nœuds 44 ms

FIGURE 6.3 – Influence du contour initial sur la segmentation. Le contour final 1 est celui de la figure 6.2.



(a) $d_{max} = 16$ et $l_{min} = 8$, 1246 nœuds en 1.3 s (b) $d_{max} = 128$ et $l_{min} = 32$, 447 nœuds en 0.7 s

FIGURE 6.4 – Segmentation de l'image de test en 4000×4000 pixels. Le tracé du contour a été artificiellement épaissi pour le rendre visible à l'échelle de l'impression.

512×512 pixels. On parvient à un résultat très proche beaucoup plus rapidement en adaptant les paramètres à la taille de l'image, comme le montre par exemple la segmentation de la figure 6.4(b), effectuée avec $d_{max} = 128$ et $l_{min} = 32$ et qui converge vers un contour de 447 nœuds en moins de 0,7 s. Au delà des 16 millions de pixels (4000×4000 pixels), l'implémentation séquentielle est toujours possible mais doit se priver des instructions SSE. Nous avons, avec l'accord des auteurs, adapté leur code en ce sens et réalisé les mesures pour des tailles allant jusqu'à 150 MP. La table 6.2 en synthétise les résultats en distinguant chaque fois le temps pris par les pré-calculs et celui nécessaire à la convergence de la segmentation. On constate que les deux étapes et donc le temps total varient linéairement avec la taille de l'image.

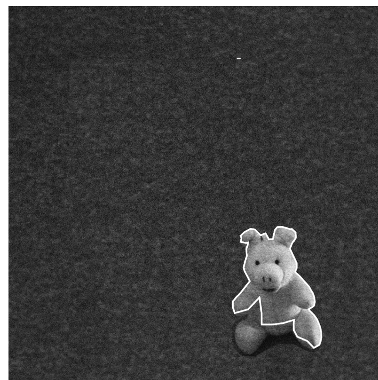


FIGURE 6.5 – Segmentation de l'image de test en 4000×4000 pixels avec une cible de petite taille. Le contour initial est la transcription de celui utilisé à la figure 6.2. Le tracé du contour a été artificiellement épaissi pour le rendre visible à l'échelle de l'impression.

Enfin, il faut aussi considérer les tailles relatives de la cible et de l'image. Ainsi, si on fait l'hypothèse d'une cible de petite taille « noyée » dans une image de grandes dimensions, les résultats de la segmentation seront impactés en raison, cette fois, d'une moindre adaptation à la cible lors des toutes premières itérations, les plus grossières, où le nombre de nœuds est réduit et le pas de déplacement potentiellement grand vis à vis de la cible. Ce cas de figure est illustré par la segmentation reproduite à la figure 6.5 et qui met en

	Taille de l'image (millions de pixels)		
	15	100	150
Pré-calculs	0,13	0,91	1,4
Segmentation	0,46	3,17	4,3
Total	0,51	4,08	5,7

TABLE 6.2 – Performances (en secondes) de la segmentation par snake polygonal sur CPU en fonction de la taille de l'image à traiter. Les temps sont obtenus avec la même image de test dilatée et bruitée et un contour initial carré dont la distance aux bords est proportionnelle à la taille de l'image. Seule l'image en 15 MP a pu être traitée par une implémentation utilisant SSE2.

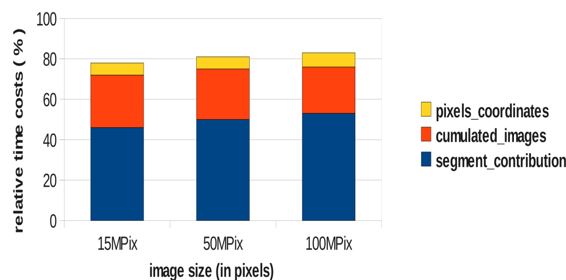


FIGURE 6.6 – Évolution du coût relatif des trois fonctions les plus consommatrices en temps de calcul en fonction de la taille de l'image à traiter.

évidence une qualité moindre par la confusion des zones les plus sombres de la cible avec le fond.

6.3/ IMPLÉMENTATION PARALLÈLE GPU DU SNAKE POLYGONAL

L'analyse de l'exécution du programme séquentiel révèle la prépondérance des blocs fonctionnels suivants, qui occupent à eux seuls 80% du temps total d'exécution :
- Le calcul de la contribution des segments, pour environ 50% (lignes 19 à 21 dans l'algorithme 3)
- La génération des trois images cumulées, avant le début des itérations, pour environ 20% (ligne 2).
- La discrétisation des segments définis par les coordonnées de leurs extrémités, pour environ 7% (ligne 18).

Cette proportion est globalement conservée lorsque la taille de l'image à traiter varie, comme le montre le graphique de la figure 6.3

Si l'effort de parallélisation porte essentiellement sur ces fonctions coûteuses, l'ensemble du traitement est réalisé sur le GPU afin de réduire autant que possible les transferts entre le GPU et le système hôte qui, selon le volume concerné, sont susceptibles de grever considérablement la performance globale. L'hôte ne conserve que l'initiative du transfert initial et le contrôle de la boucle principale, ne nécessitant l'échange que d'un seul octet à chaque itération (représentant le nombre de nouveaux nœuds N_{add}).

Les traitements étant totalement indépendants, nous traitons séparément la parallélisation des pré-calculs et celle de la segmentation.

6.3.1/ PRÉ-CALCULS DES IMAGES CUMULÉES

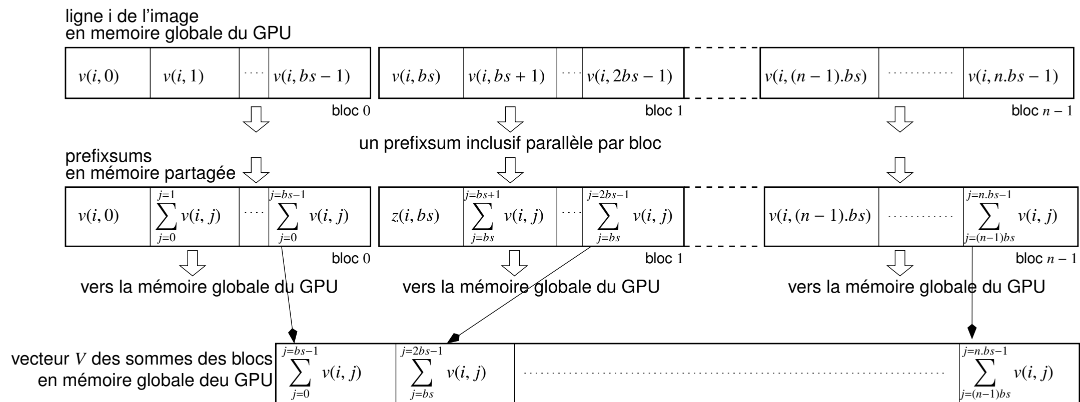
Pour réduire la quantité de mémoire requise, nous avons choisi de ne pas générer l'image S_1 mais plutôt d'en calculer les valeurs à la volée. L'expression en est simple et le temps pris par les opérations élémentaires qu'elle met en jeu est largement compensé par le gain obtenu en économisant les accès mémoire qui auraient été nécessaires, ce qui n'est pas le cas des deux autres images S_x et S_x^2 dont le calcul est quant à lui réalisé en appliquant une variante de la méthode des sommes préfixées (*prefixsums*) décrite dans [10] et qui permet d'évaluer les expressions de l'équation (6.13).

Les sommations se font au niveau de chaque ligne de l'image, que l'on décompose en n blocs de bs pixels où bs correspond aussi au nombre de threads exécutés par chaque bloc de la grille de calcul. La valeur bs étant obligatoirement une puissance de 2 supérieure à 32, le bloc de pixels d'indice $n-1$ doit éventuellement être complété par des valeurs nulles. Chaque bloc de thread réalise son traitement indépendemment des autres, mais l'ensemble des sommes de bloc étant requise pour le calcul des sommes globales, une synchronisation est nécessaire à deux endroits du calcul. Nous avons choisi d'assurer ces synchronisations en découpant le traitement en trois *kernels* distincts, rendant par la même occasion le code plus concis :

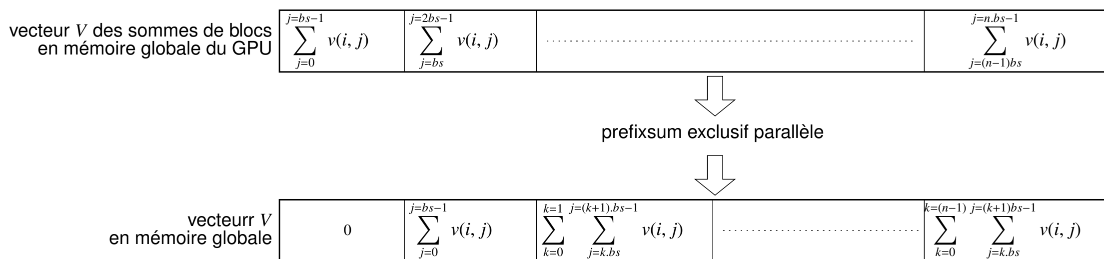
- `compute_block_prefixes()` est le *kernel* effectuant, en mémoire partagée, la *prefixsum* inclusive de chaque bloc, puis qui en mémorise la sommes, *i.e* le dernier élément, dans deux vecteurs V_x et V_x^2 en mémoire globale. L'ensemble des prefixsums est également mémorisé en mémoire globale. La largeur de l'image n'étant pas nécessairement une puissance de 2, il est nécessaire de faire du remplissage avec des valeurs nulles dans le dernier bloc (indice $n-1$).
- `scan_blocksums()` est le *kernel* effectuant les prefixsum exclusifs des vecteurs V_x et V_x^2. Les résultat demeurent respectivement dans V_x et V_x^2.
- `add_sums2prefixes()` est le *kernel* effectuant les additions de chaque élément d'indice i des vecteurs V_x (respectivement V_x^2) avec tous les éléments du prefixsum du bloc de même indice i.

Les diagrammes de la figure 6.7 donnent le détail des opérations effectuées par ces trois *kernels* pour l'image cumulée S_x. La seconde image cumulée S_x^2 est obtenues exactement de la même manière en sommant non plus les valeurs v_k mais v_k^2.

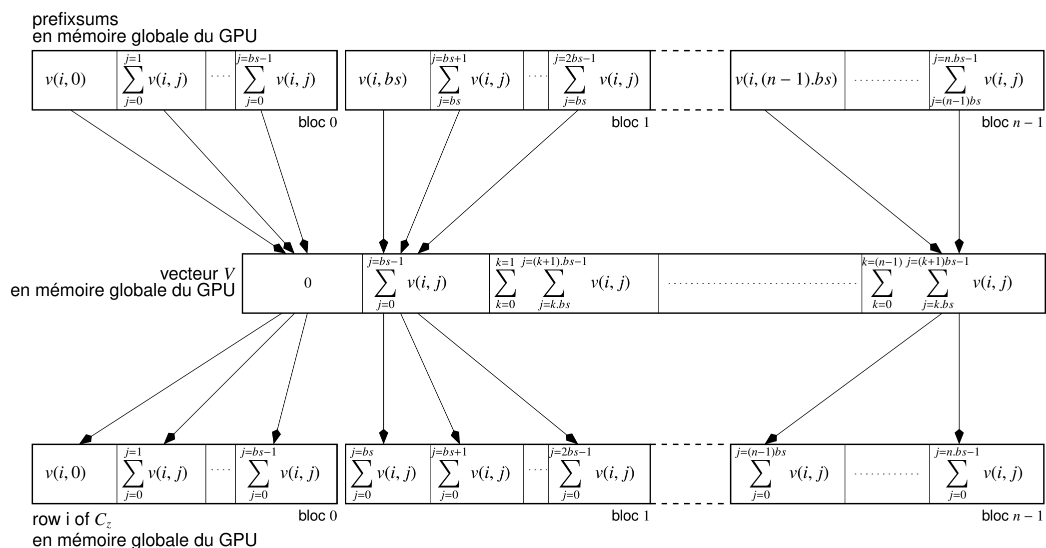
Les gains de performance de cette implémentation GPU comparée à l'implémentation CPU/SSE2 sont ceux de la table 6.3, soit un GPU environ 7 fois plus rapide pour des images de taille 15 à 150 millions de pixels. L'influence de la taille d'image sur le gain est faible, mais on peut toutefois noter que plus l'image est grande plus le gain est important. Les accélérations constatées peuvent sembler faibles en regard de ce que l'on attend d'un GPU, mais il faut rappeler que ce type d'opération (les réductions) n'est pas véritablement adapté à leur architecture en raison d'une grande inter-dépendance des données d'une étape de calcul à l'autre. Sans une implémentation optimisée, cette opération s'exécuterait même plus lentement sur GPU que sur un CPU. On obtient des accélérations supérieures en rendant le calcul moins générique et en développant des versions spécifiques des trois *kernels*, dédiées par exemple au traitement des images dont largeur est multiple de 256 pixels.



(a) Détail des opérations effectuées par le *kernel* `compute_block_prefixes()`. La valeur bs correspond au nombre de pixels de chaque bloc, qui est aussi le nombre de threads exécuté par chaque bloc de la grille de calcul.



(b) Détail des opérations effectuées par le *kernel* `scan_blocksums()`.



(c) Détail des opérations effectuées par le *kernel* `add_sums2prefixes()`.

FIGURE 6.7 – Calcul des images cumulées S_x et S_x^2 en trois étapes successives. a) cumul partiel bloc par bloc et mémorisation de la somme de chaque bloc. b) cumul sur le vecteur des sommes partielles. c) ajout des sommes partielles à chaque élément des blocs cumulés.

	Taille de l'image (millions de pixels)		
	15	100	150
temps CPU (s)	0,13	0,91	1,4
temps GPU (s)	0,02	0,13	0,2
Accélération	**6,5**	**6,9**	**7,0**

TABLE 6.3 – Accélération constatée, pour le calcul des images cumulées, de l'implémentation GPU (C2070) par rapport à l'implémentation CPU de référence.

6.3.2/ CALCUL DES CONTRIBUTIONS DES SEGMENTS

Le déplacement d'un des N_n nœuds du contour Γ vers l'une des 8 positions voisines permises, impose d'évaluer les contributions des 8 paires de segments associées, soit $16N_n$ segments pour la totalité du contour, que nous évaluons en parallèle au sein du *kernel* `GPU_compute_segments_contribs()`. Pour ce faire, chaque segment doit tout d'abord être discrétisé en une suite de pixels puis, en conservant la règle *1 pixel par thread* la contribution de chaque pixel est déterminée avant de toutes les additionner pour obtenir la contribution du segment. Les pixels représentant les nœuds font l'objet d'un traitement spécifique impliquant les codes de Freeman, pour ne pas fausser les contributions globales (voir paragraphe 6.2.3).

Pour optimiser l'exécution de ce kernel et réduire l'effet de la disparité des longueurs des segments, nous avons crée un motif régulier en mémoire, basé sur la longueur $npix_{max}$ du plus grand segment et avons complété les blocs associés aux segments de longueur inférieure à $npix_{max}$ avec des valeurs neutres pour l'opération réalisée, c'est-à-dire des valeurs nulles.

Si bs_{max} est la taille de bloc maximale admissible par le GPU, la taille bs des blocs de threads/pixels employée pour le calcul des contributions des segments est alors déterminée de la façon suivante :

$$bs = \begin{cases} 2^p & \text{avec } 2^{p-1} < npix_{max} \leq 2^p \text{ si } npix_{max} \in [33; bs_{max}] \\ 32 & \text{si } npix_{max} \leq 32 \\ bs_{max} & \text{si } npix_{max} > bs_{max} \end{cases}$$

Dans notre implémentation, les calculs sont faits en mémoire partagée et la quantité nécessaire limite la taille de bloc admissible. Nous limitons celle-ci à 256 sur C1060 et 512 sur C2050. Toutefois, les tests ont montré que sur ces deux versions de l'architecture, la taille maximale conduisant aux meilleures performances est de 256 threads par bloc.

Le *kernel* `GPU_compute_segments_contribs()` calcule alors en parallèle pour tous les segments les coordonnées de tous les pixels qui les composent. Nous mettons pour cela en œuvre l'algorithme de Bresenham, *i.e* la méthode du segment semi-ouvert, en distinguant les cas où
- la valeur absolue de la pente k du segment à discrétiser est supérieure à 1 ; on applique alors la méthode au segment *horizontal* semi-ouvert et on obtient un pixel par ligne.
- la valeur absolue de la pente k du segment à discrétiser est inférieure ou égale à 1 ; on applique alors la méthode au segment *vertical* semi-ouvert et on obtient un pixel par colonne.

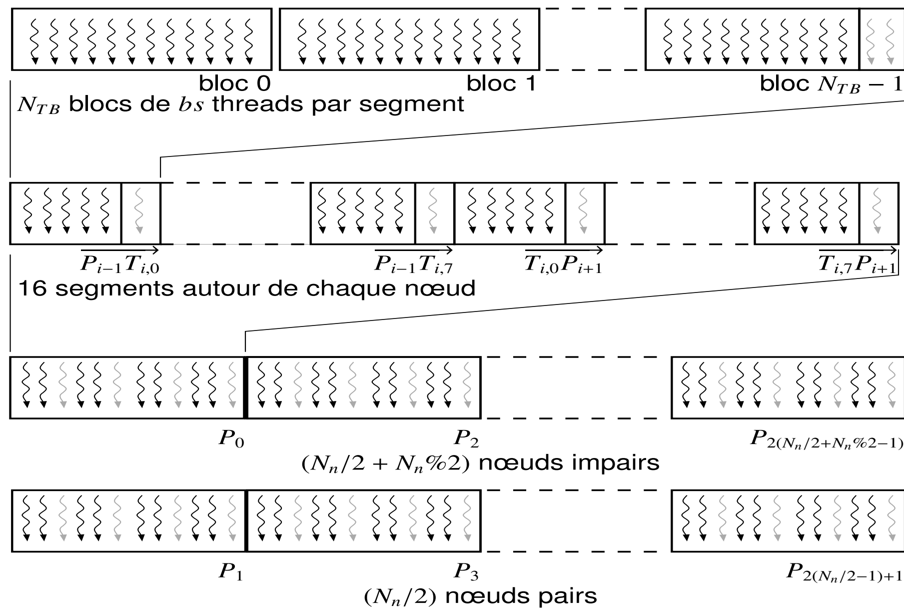


FIGURE 6.8 – Structuration des données en mémoire du GPU pour l'évaluation en parallèle de l'ensemble des évolutions possibles du contour.

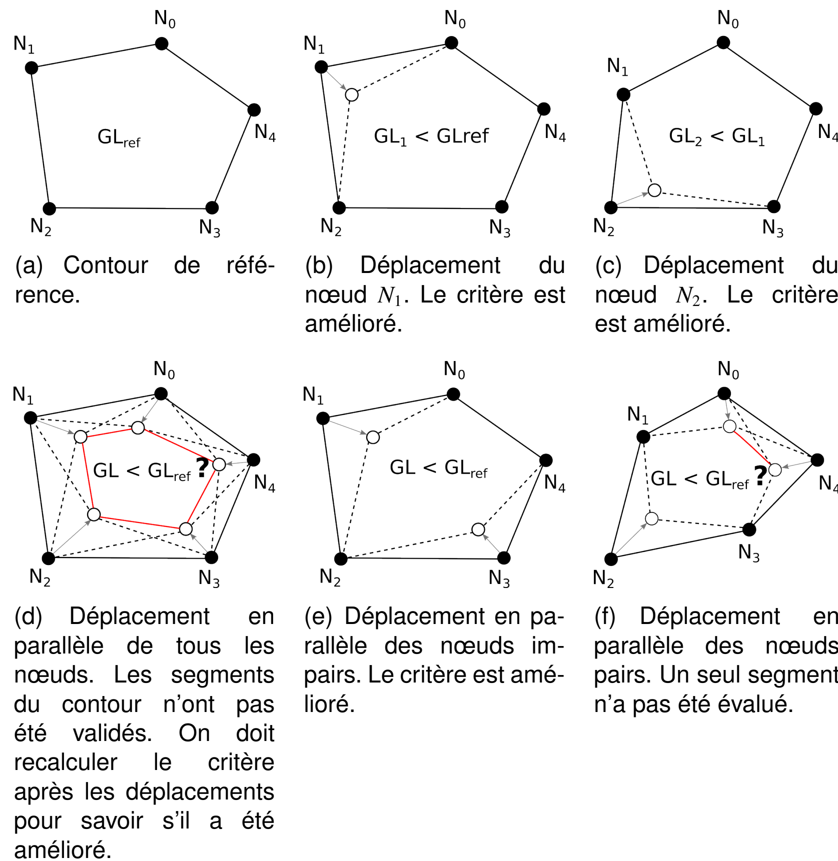
Cette distinction nous permet de conserver la règle *1 pixel par thread* importante pour la régularité des motifs d'accès en mémoire et aussi pour *charger* au maximum le GPU.

La figure 6.8 représente la structure décrite ci-dessus pour la représentation en mémoire des segments à évaluer. La première ligne montre le détail du premier segment, avec la correspondance *1 pixel par thread* et le découpage en un nombre de blocs suffisant pour permettre de décrire le plus long des segments.

La seconde ligne présente l'ordre dans lequel sont concaténés les 16 groupes de blocs-segment associés au déplacement d'un nœud particulier.

Les deux dernières lignes décrivent la concaténation des ensembles de 16 blocs-segment, avec la particularité de séparer la description des positions des nœuds d'indices pairs et ceux d'indices impairs. Cela permet de moins s'écarter de l'heuristique d'optimisation en vigueur dans la version séquentielle où les statistiques globales comme la valeur de critère GL sont recalculées après chaque déplacement (figures 6.9(a), 6.9(b) et 6.9(c)) .

En version parallèle, si les « meilleures » positions de tous les nœuds sont calculées simultanément, le contour généré est constitué de segments qui n'ont pas été validés pendant la phase de déplacement des nœuds, comme l'illustre la figure 6.9(e). La valeur du critère GL doit donc être calculée après coup sur les segments réels du nouveau contour. Dans l'absolu, nous ne sommes donc pas assurés d'améliorer réellement la valeur du critère par rapport au contour de l'itération précédente. Pour limiter ce phénomène, qui pourrait provoquer des oscillations et empêcher la convergence, nous avons effectué les déplacements en alternant ceux des nœuds d'indices pairs et ceux d'indices impairs. Cela permet de régler le problème lorsque le nombre de nœuds du contour est pair. Comme le montrent les figures 6.9(e) et 6.9(e), un segment du contour demeure non validé lorsque le nombre de nœuds est impair et nous impose toujours de recalculer, *a posteriori*, la valeur du critère GL pour s'assurer de l'amélioration apporté par les



(a) Contour de référence.

(b) Déplacement du nœud N_1. Le critère est amélioré.

(c) Déplacement du nœud N_2. Le critère est amélioré.

(d) Déplacement en parallèle de tous les nœuds. Les segments du contour n'ont pas été validés. On doit recalculer le critère après les déplacements pour savoir s'il a été amélioré.

(e) Déplacement en parallèle des nœuds impairs. Le critère est amélioré.

(f) Déplacement en parallèle des nœuds pairs. Un seul segment n'a pas été évalué.

FIGURE 6.9 – Comparaison des cycles de déplacement des nœuds. Ligne du haut : version séquentielle. Ligne du bas : version parallèle. Les segments en rouge sont des segments du contour non évalués, alors que ceux en pointillés sont les paires ayant reçu les meilleures évaluations parmi les 8 déplacements possibles des nœuds correspondant.

déplacements des nœuds.

La représentation en mémoire des segments conduit à avoir un certain nombre non prévisible de threads inactifs dans la grille, sans que cela soit préjudiciable aux performances car cela n'engendre pas de branches d'exécution divergentes, qui sont à proscrire sur GPU.

Les calculs liés à l'évaluation des contributions des pixels sont réalisés en mémoire partagée. Seule une très petite quantité de données doit être stockée en mémoire globale. Il s'agit, pour chaque **segment** :
- des coordonnées de son milieu. Cela permet l'ajout efficace de nœud quand c'est possible.
- les coordonnées des deux derniers pixels de chaque extrémité. Ils sont nécessaires pour calculer la dérivée aux extrémités et ainsi déterminer le code de Freeman des nœuds.

Pour obtenir les contributions des segments, c'est-à-dire les sommes des contributions des leurs pixels, une première phase de réduction partielle est effectuée au niveau de chaque bloc.

Une synchronisation est alors nécessaire avant d'effectuer les sommes de l'ensemble des contributions partielles qui fournissent les contributions globales des segments. Le

contour modifié est alors construit comme la suite des meilleures positions déterminées pour chaque nœud, pour peu que ces nouvelles positions ne génèrent pas de croisement de segments.

La solution retenue pour vérifier l'absence de croisement est celle de l'implémentation séquentielle, parallélisée simplement par paire de segments. Cela n'apporte pas de véritable gain de performance par rapport à la version CPU, mais, contraints de conserver les données en mémoire GPU pour limiter les transferts entre l'hôte et son périphérique, nous avons fait en sorte que cette fonctionnalité ne grève pas les performances globales.

Les valeurs obtenues après détermination du nouveau contour, calcul des statistiques globales et évaluation du critère GL, servent de référence pour les prochaines déformations du contour. Les techniques appliquées pour ces calculs sont de nouveau celles décrites au début ce paragraphe. Enfin, l'ajout des nouveaux nœuds se fait simplement, pour les segments suffisamment grands, en utilisant les coordonnées des pixels milieux mémorisées lors de la discrétisation des segments.

6.3.2.1/ Cas particulier des segments dont la pente k vérifie $|k| \leq 1$

Comme nous venons de le voir, les segments dont la pente k vérifie $|k| \leq 1$ sont discrétisés à raison de *1 pixel par colonne* et comportent donc le plus souvent plusieurs pixels sur une ligne donnée, comme le montrent les schémas de la figure 6.10.

D'après la formulation générale du snake faite au paragraphe 6.2.2, le coefficient $C(i, j)$ est à appliquer en chaque point du contour. La technique de discrétisation employée conduit à des coefficients $C(i, j)$ constants sur l'ensemble des pixels des segments dont la pente k vérifie $|k| > 1$, mais ce n'est pas le cas pour ceux dont la pente k est inférieure ou égale à 1. Les quatre cas, un par quadrant, qui peuvent se présenter sont représentés à la figure 6.10.

D'un point de vue opérationnel, on constate en se reportant à la table 6.1, que tout pixel dont les voisins immédiats sont sur la même ligne, à un coefficient $C(i, j) = 0$ ($F_{in} = f_{out} = 0$). Les deux pixels des extrémités, n'ayant qu'un voisin sur la même ligne, ont un coefficient qui dépend du quadrant :

- dans les quandrant 1 et 2
  - le premier pixel d'une ligne a un coefficient $C(i, j) = 1$.
  - le dernier pixel d'une ligne a un coefficient $C(i, j) = 0$
- dans les quandrant 3 et 4
  - le dernier pixel d'une ligne à un coefficient $C(i, j) = -1$.
  - le premier pixel d'une ligne a un coefficient $C(i, j) = 0$.

Les accès en mémoire aux contributions des pixels de coefficient $C(i, j) = 0$, dans les images cumulées, sont évités et une contribution nulle leur est automatiquement attribuée dès l'étape de discrétisation au sein du kernel `GPU_compute_segments_contribs()`.

6.3.3/ Performances

Dans l'hypothèse la plus contraignante d'images en niveaux de gris codés sur 16 bits, l'implémentation parallèle que nous venons de décrire utilise de manière permanente 20 octets par pixel de l'image d'entrée, qui se détaillent en

- l'image d'entrée pour 4 octets par pixel (1 entier).

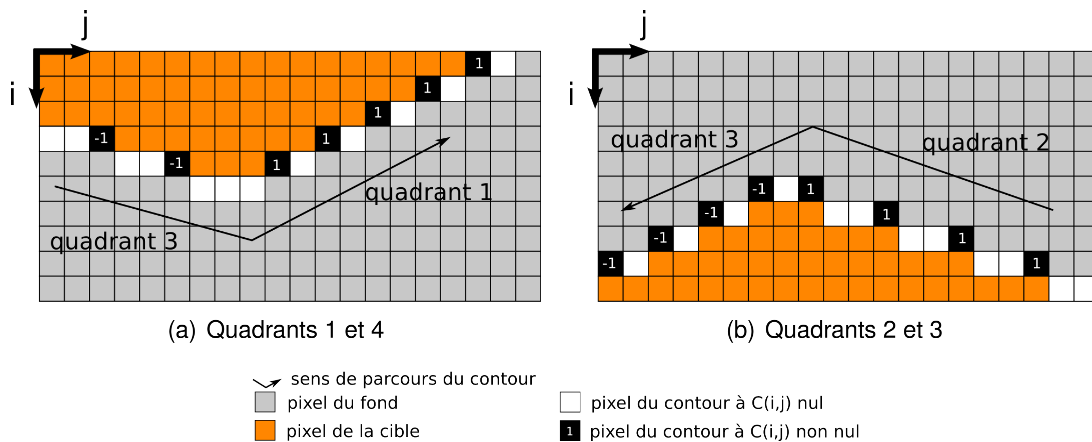


FIGURE 6.10 – Détermination des coefficients $C(i, j)$ des pixels du contour.

– l'image cumulée S_x pour 8 octets par pixel (1 entier long)
– l'image cumulée S_x^2 pour 8 octets par pixel (1 entier long)
auxquels il faut ajouter un maximum d'environ 50 Mo d'espace nécessaire à la mémorisation des variables temporaires des calculs et données diverses comme le contour lui-même (nœuds, milieux, Freemans, etc.).

Sur un GPU de type C1060 disposant de 3 Go de mémoire, cela permet de traiter des images jusqu'à presque 150 millions de pixels. Il est possible de réduire cette empreinte jusqu'à 13 octets par pixel, mais cela soulève la question de l'alignement des données en mémoire, sans objet si on emploie les types entier et entier long (32 et 64 bits) pour la représentation des données et qui permet de préserver les performances maximales des opérations et des accès aux données du GPU. On pourrait tout de même porter ainsi la limite de taille de l'image d'entrée à 230 millions de pixels.

La convergence de notre implémentation intervient en un nombre généralement plus réduit d'itérations vers un contour final qui diffère par essence de celui obtenu avec la solution de référence. Ces effets sont la conséquence déjà abordée de l'heuristique d'optimisation appliquée à l'implémentation parallèle qui conduit à la création de certains segments non validés au préalable (voir fig. 6.9).

Les comparaisons visuelles et de valeur du critère GL qui peuvent être faites pour les images de taille inférieure à 4096×4096 pixels nous renseignent toutefois sur la qualité de la segmentation obtenue. Pour les tailles au delà et jusqu'au maximum de 12000×12000 pixels, le comportement est globalement conservé, mais on note qu'il n'est pas pertinent de permettre des tailles de segments trop petites vis-à-vis de la taille d'image. Les déplacements des nœuds ne génèrent alors plus de variations significatives des contributions correspondantes.

La figure 6.11 présente une segmentation effectuée sur une image de 100 millions de pixels. La table 6.4 résume les performances obtenues pour différentes tailles de la même image.

		Performances		
		CPU	GPU	CPU/GPU
Image 15 MP	**total**	**0,51 s**	**0,06 s**	**x8,5**
	images cumulées	0,13 s	0,02 s	x6,5
	segmentation	0,46 s	0,04 s	x11,5
Image 100 MP	**total**	**4,08 s**	**0,59 s**	**x6,9**
	images cumulées	0,91 s	0,13 s	x6,9
	segmentation	3,17 s	0,46 s	x6,9
Image 150 MP	**total**	**5,70 s**	**0,79 s**	**x7,2**
	images cumulées	1,40 s	0,20 s	x7,0
	segmentation	4,30 s	0,59 s	x7,3

TABLE 6.4 – Comparaison des temps d'exécution de l'implémentation GPU (C2070) par rapport à l'implémentation CPU de référence, appliqués à une même image dilatée (fig. 6.2) pour en adapter la taille.

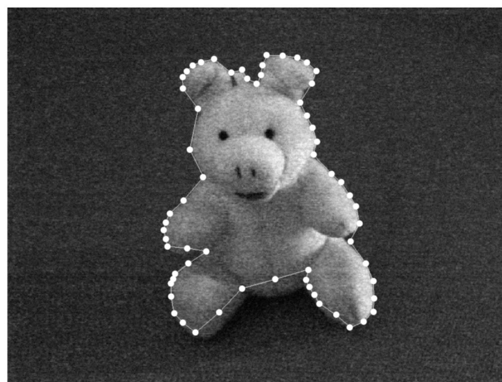
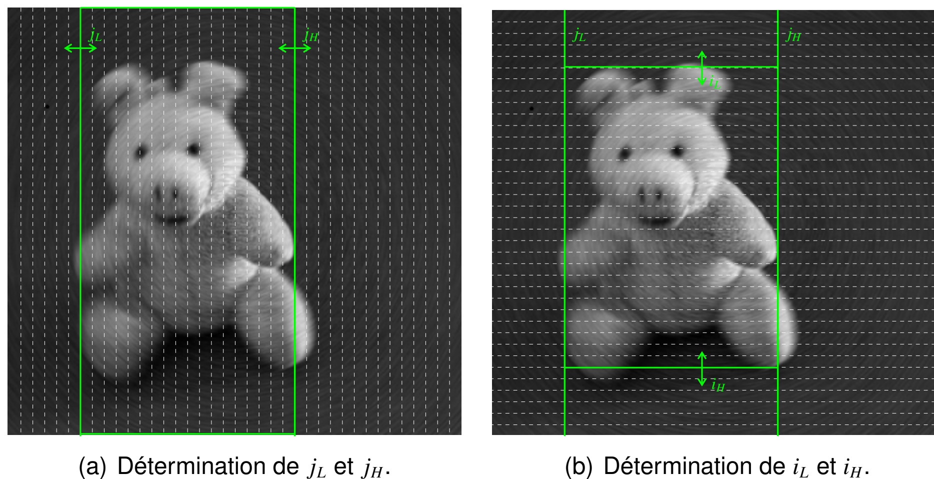


FIGURE 6.11 – Segmentations d'une image de 100 MP en 0,59 s pour 5 itérations. Le contour initial conserve les proportions de celui de la figure 6.2.

6.3.4/ DÉTERMINATION INTELLIGENTE DU CONTOUR INITIAL

Nous avons déjà discuté de l'influence du contour initial sur le résultat de la segmentation, mais il faut ajouter que la durée d'exécution est aussi impactée par ce choix, dans des proportions qui peuvent être importantes selon la distance, la taille et dans une moindre mesure la forme de la cible.

Ces effets se mesurent lors de la première itération, celle qui va cerner grossièrement la cible avec un polygone à quatre cotés. Si le contour initial se trouve très éloigné, comme dans la situation de la figure 6.5, notre choix maintenant habituel d'un rectangle près des bords de l'image s'avère peu adapté et conduit à une première itération très longue. Dans un tel cas, pour une image de 10000×10000 pixels, si la cible est un carré de 1000×1000 pixels dont le sommet du bas à droite se confond avec celui du contour et que l'on approche par pas de 64 pixels, on devra dans le meilleur des cas déplacer les 4 nœuds du contour 110 fois de suite avant de pouvoir passer à la deuxième itération. Un pas de 128 permet de réduire ces valeurs, mais l'expérience montre qu'au delà, l'approche initiale de la cible est trop grossière et les itérations suivantes en pâtissent pour un résultat souvent



(a) Détermination de j_L et j_H. (b) Détermination de i_L et i_H.

FIGURE 6.12 – Détermination intelligente du contour initial en deux phases successives. a) La première étape repose sur un échantillonnage horizontal. b) La seconde étape repose sur un échantillonnage vertical.

dégradé. En revanche, si les proportions sont celles de la figure 6.2, seules 31 passes de déplacement des 4 nœuds initiaux sont nécessaires.

Pour optimiser l'initialisation, nous avons donc proposé de tirer parti du GPU pour évaluer une grande quantité de contours initiaux rectangulaires et réduire ainsi le coût de la première itération. Pour pouvoir employer la mémoire partagée comme tampon de données, il faut limiter le nombre de contours à évaluer. Nous avons donc effectué un échantillonnage spatial des images et déterminé le contour initial en deux temps, en mettant à profit la propriété qu'ont les segments horizontaux d'avoir une contribution nulle, comme on peut le vérifier en se reportant à la figure 6.1 et à la table 6.1. Le principe mis en œuvre, illustré par la figure 6.3.4 est le suivant :

1. on réalise un échantillonnage horizontal pour ne considérer que les colonnes d'indice $j = 8k$.
2. on évalue alors tous les contours rectangulaires de diagonale $(0, j_L) - (J_H, H)$
3. on identifie le contour présentant le meilleur critère GL, ce qui détermine j_L et j_H.
4. on fait de même en échantillonnant verticalement : les lignes d'indice $i = 8t$ permettent de décrire tous les contours de diagonale $(i_L, j_L) - (i_H, j_H)$. Le meilleur contour est celui retenu pour l'initialisation de la segmentatation.

Le gain de performance apporté par cette initialisation « intelligente » est naturellement très variable selon la cible, mais dans des situations favorables comme celle de l'image de la figure 6.5, on parvient à une accélération proche de 15 alors qu'elle n'est que d'environ 7 avec l'initialisation basique. Cette proportion est conservée pour les tailles supérieures et signifie que la phase de segmentation est tout de même effectuée 30 fois plus rapidement qu'avec l'implémentation CPU, grâce à une première itération optimisée.

6.3.5/ CONCLUSION

Nous avons conçu une implémentation parallèle de *snake* polygonal orienté régions, ce qui n'avait pas encore été réalisé, n'ayant recensé à ce jour aucune publication y faisant

référence. Elle a fait l'objet d'une publication et d'une communication à la conférence *Computer and Information Technology* (voir [73]). Les objectifs étaient d'étendre les capacités de traitement de l'implémentation CPU de référence en terme de taille d'image en conservant des temps d'exécution acceptables ce qui, de l'avis des auteurs de la version CPU, impose de se situer *a minima* sous la seconde pour pouvoir envisager l'intégration dans une application interactive.

Sur ce point, les performances de notre version sont satisfaisantes, puisque nous avons repoussé la limite de taille de 16 à 150 millions de pixels et parvenons à segmenter ces grandes images en moins d'une seconde. Le temps de calcul dépend très fortement du contenu de l'image et la segmentation est le plus souvent obtenue en un temps plus court.

L'emploi du GPU dans notre implémentation ne parvient pas à être optimal car, par essence, la répartition des pixels d'intérêt est mouvante et ne permet pas de construire des accès mémoire coalescents. Les opérations de type réduction sont également nombreuses et ne sont pas les plus efficaces sur GPU. Dans notre situation, elles peuvent même représenter une perte de performances, car effectuées sur des vecteurs de tailles insuffisantes.

S'il s'agit de parler d'accélération, notre implémentation divise les temps de traitement précédents par un facteur allant de 6 à 15 selon l'image et le contour initial adopté. Rappelons encore que l'implémentation CPU de référence n'est pas une implémentation naïve, mais une solution optimisée employant déjà les capacités de parallélisme des microprocesseurs modernes et affichant les performances les plus élevées dans ce domaine ; il n'était pas trivial d'en surpasser les performances, même avec un GPU.

Par nécessité, notre solution s'écarte cependant quelque peu de l'algorithme original pour permettre les déplacements simultanés de l'ensemble des sommets du polygone. Ce faisant, la décroissance du critère n'est pas certaine à toutes les étapes de la segmentation et l'on observe cette conséquence, en particulier lors des dernière itérations lorsque le pas de déplacement ainsi que les variations du critère sont faibles. Ce comportement provoque parfois la convergence prématurée de la segmentation, mais n'influe toutefois que sur quelques nœuds et le contour ainsi obtenu ne s'éloigne que très peu du contour obtenu par l'algorithme de référence.

La technique que nous avons proposée pour la détermination intelligente du contour initial permet d'augmenter encore les performances, surtout dans les grandes images lorsque la cible est petite vis-à-vis des dimensions de l'image. Il reste toutefois à concevoir une technique permettant de prévoir si cette recherche intelligente serait génératrice de gain de performance.

L'analyse fine des séquences de segmentation montre enfin que les première étapes, qui mettent en œuvre les segments les plus longs, génèrent des grilles de calcul suffisamment chargées et homogènes pour présenter de bonnes performances. Les dernières étapes, en revanche, traitent un grand nombre de petits segments, générant beaucoup de trous dans la grille de calcul et induisant des performances moindres.

Pour résumer, l'accélération globale obtenue est principalement déterminée par le calcul des images cumulées et des toutes premières étapes de déplacements. Une possibilité à explorer serait de construire une version hybride réalisant le début de la segmentation sur GPU, puis la terminant sur le CPU hôte. Ceci est envisageable en raison du très petit volume de données à transférer que constituent les paramètres du contour (2 ko pour

100 nœuds).

7

Réduction de bruit par recherche des lignes de niveaux

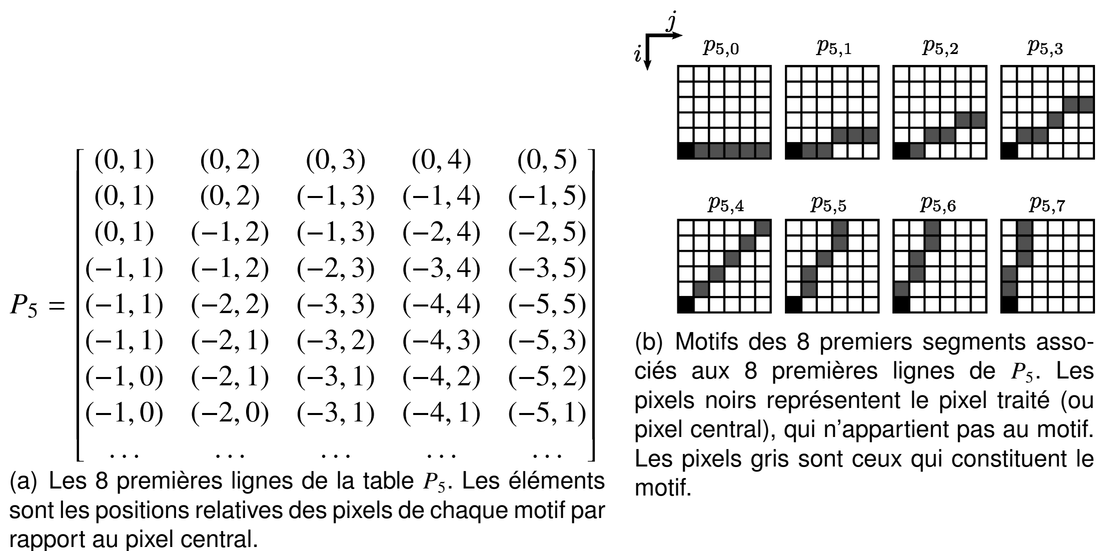
7.1/ Introduction

Le concept de ligne de niveau dans les images a été introduit dès 1975 par Matheron [64], puis Caselles *et al.* [16] l'ont exploité et proposé le cadre définissant les *images naturelles* comme les scènes photographiées, en intérieur ou en extérieur, à l'aide d'un appareil standard. Ces images vérifient alors l'hypothèse de gradient à valeurs bornées et peuvent être décomposées en un ensemble de lignes de niveaux. Bertaux *et al.* ont plus récemment proposé un algorithme de réduction du *speckle* dans les images éclairées en lumière cohérente en introduisant, pour les pixels de l'image observée, une contrainte d'appartenance aux lignes de niveaux du modèle d'image non bruitée [9]. L'image observée étant perturbée, les lignes de niveaux ne sont pas accessibles et il s'agit donc d'en estimer, localement par morceaux, la valeur et la forme, en se basant sur un modèle pré établi. Pour un pixel dont on cherche à estimer la valeur du niveau de gris, la contrainte d'appartenance à une ligne de niveau demeure locale, avec cependant un voisinage de forme et de taille (en nombre de pixels) variables en fonction des propriétés de l'image bruitée dans la zone concernée. Ce voisinage, dont la forme, l'étendue et le niveau de gris sont déterminés par maximum de vraisemblance, appelé une *isoline*, est une estimation locale de la ligne de niveau à laquelle appartient le pixel concerné. Cette technique a montré qu'elle permettait de réduire très significativement le niveau de bruit tout en préservant les contours des objets. Elle s'est en revanche averée gourmande en ressources, ce qui a initialement conduit les auteurs à réduire la résolution de calcul des *isolines* par application d'un maillage sur l'image bruitée. Malgré cela, les temps de calcul demeuraient prohibitifs, avec une image de 2 millions de pixels traitée en 1 minute par un PIII-1GHz. Comme nous l'avons déjà évoqué, l'amélioration des performances des microprocesseurs permet aujourd'hui de réduire assez considérablement ce temps de calcul. Cependant, la résolution des images à traiter à crû dans des proportions comparables, laissant les termes du compromis qualité/performance inchangés.

7.2/ Présentation de l'algorithme

7.2.1/ Formulation

Les *isolines* sont des lignes brisées composées d'un ou plusieurs segments et construites par allongements successifs. Le niveau de gris affecté en sortie au pixel considéré est la valeur moyenne des niveaux de gris des pixels appartenant à l'*isoline*. Les segments sont de longueur n fixe mais paramétrable et leur « forme » est sélectionnée parmi 32 motifs prédéterminés et mémorisés dans une table de référence notée P_{n-1} dont un extrait est reproduit à la figure 7.1 avec les motifs des segments correspondants. Tous les motifs sont composés du même nombre $a = n - 1$ de pixels. Pour chaque pixel de l'image d'entrée, le premier segment est choisi comme celui présentant la meilleure vraisemblance parmi les 32 possibles. Le choix d'intégrer ou non d'autres segments à l'*isoline* et la sélection des segments à intégrer sont effectués par évaluation d'un critère de vraisemblance généralisée dont l'obtention est détaillée dans la suite.



(a) Les 8 premières lignes de la table P_5. Les éléments sont les positions relatives des pixels de chaque motif par rapport au pixel central.

(b) Motifs des 8 premiers segments associés aux 8 premières lignes de P_5. Les pixels noirs représentent le pixel traité (ou pixel central), qui n'appartient pas au motif. Les pixels gris sont ceux qui constituent le motif.

Figure 7.1 – Détail des motifs et de leur représentation interne, pour la taille $a = 5$.

7.2.1.1/ Isolines à un seul segment

Pour chacun des pixels (i, j) de l'image corrompue, on calcule la vraisemblance associée à chaque segment candidat de la table P_{n-1} dans la région carrée ω centrée en (i, j) et de côté $2n - 1$. La région ω est l'union des deux sous régions S^n et $\overline{S^n}$ telles que S^n décrit le segment candidat à évaluer comme un ensemble de n pixels de coordonnées (i_q, j_q) où $q \in [0..n[$. La figure 7.2 montre cette répartition pour $a = 5$ et le motifs $p_{5,3}$.

La densité de probabilité des valeurs des niveaux de gris des pixels de S^n est supposée gaussienne de paramètres μ_{S^n} (moyenne) et σ (écart type) inconnus. Les moyennes des niveaux de gris de chaque pixel sur S^n sont inconnues et supposées indépendantes.

Soit Z l'ensemble des niveaux de gris des pixels de ω et $\{\mu_{ij}\}_{\overline{S^n}}$ l'ensemble des valeurs

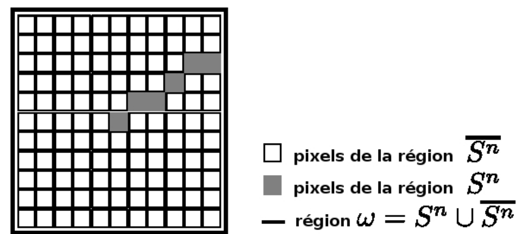


FIGURE 7.2 – Exemple de la répartition des pixels dans la région ω pour le calcul de la vraisemblance, pour $n = 6$ ($a = 5$).

moyennes des pixels de $\overline{S^n}$. On peut écrire la probabilité

$$P[Z|S^n, \mu_{S^n}, \{\mu_{ij}\}_{S^n}, \sigma]$$

qui se développe comme suit, en distinguant les contributions de S^n et $\overline{S^n}$

$$\prod_{(i,j)\in S^n} P\left[v(i,j)|\mu_{S^n}, \sigma\right] \prod_{(i,j)\in \overline{S^n}} P\left[v(i,j)|\left\{\mu_{ij}\right\}_{\overline{S^n}}, \sigma\right] \tag{7.1}$$

Nous cherchons alors à déterminer l'ensemble S^n qui maximise la valeur de l'expression (7.1) ci dessus. Or, sur S^n, les niveaux de gris $z(i,j)$ peuvent aussi être pris comme les estimations $\widehat{\mu_{ij}}$ des moyennes μ_{ij}. Le second terme de l'expression (7.1) devient donc

$$\prod_{(i,j)\in \overline{S^n}} P\left[v(i,j)|\left\{\widehat{\mu_{ij}}\right\}_{\overline{S^n}}, \sigma\right] = 1. \tag{7.2}$$

Il reste alors le premier terme de (7.1), soit l'expression de la vraisemblance généralisée donnée par :

$$\prod_{(i,j)\in S^n} P\left[v(i,j)|\mu_{S^n}, \sigma\right] \tag{7.3}$$

On peut développer cette expression en remplaçant la probabilité, gaussienne, par son expression. Il vient :

$$\prod_{(i,j)\in S^n} \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{\left(z(i,j)-\mu_{S^n}\right)^2}{2\sigma^2}} \tag{7.4}$$

En prenant le logarithme et en développant, on obtient alors l'expression suivante de la *log-vraisemblance*

$$-\frac{n}{2} log\,(2\pi) - \frac{n}{2} log\left(\sigma^2\right) - \frac{n}{2} \tag{7.5}$$

où le vecteur des paramètres (μ_{S^n}, σ) est lui même obtenu par estimation au sens du maximum de vraisemblance.

$$\begin{cases} \widehat{\mu_{S^n}} = \frac{1}{n} \sum_{(i,j)\in S^n} v(i,j) \\ \widehat{\sigma^2} = \frac{1}{n} \sum_{(i,j)\in S^n} \left(v(i,j) - \widehat{\mu_{S^n}}\right)^2 \end{cases}$$

Le motif retenu pour le segment est celui qui maximise l'expression de (7.5).

7.2.1.2/ Isolines composées de plusieurs segments - critère d'allongement

L'objectif poursuivi en cherchant à étendre la portée des isolines est d'améliorer la force du filtrage en intégrant plus de valeurs de niveaux de gris dans le calcul de la moyenne qui deviendra la valeur de sortie filtrée. Pour cela nous permettons à chaque isoline, comportant initialement un seul segment, d'être prolongée par d'autres segments, chaque allongement faisant l'objet d'une validation selon un critère de vraisemblance généralisée. L'évaluation de l'ensemble des isolines pouvant être construite sur ce modèle présente un coût prohibitif en temps de calcul et l'idée en a donc été abandonnée. À la place, nous effectuons une sélection à chaque étape d'allongement : on évalue l'ensemble des 32 allongements possibles et si au moins un des motifs est accepté, on retient l'*isoline* ayant la meilleure vraisemblance. Ce processus est répété tant qu'au moins un motif représente un allongement valide.

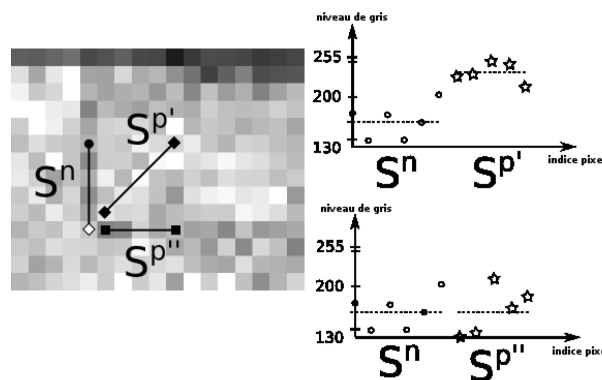


Figure 7.3 – Allongement du segment S^n. Deux candidats $S^{p'}$ et $S^{p''}$ sont évalués au travers du critère GLRT de l'équation (7.8) que seul $S^{p''}$ s'avère satisfaire. a) Représentation dans le plan de l'image. b) Évolution des niveaux de gris en fonction de la position des pixels dans les lignes brisées ainsi formées.

Soit S^n une isoline précédemment validée et S^p un segment connecté à S^n de telle sorte qu'il représente un allongement potentiel de S^n. Une situation de cette nature est représentée à la figure 7.3 avec un premier segment valide et deux candidats $S^{p'}$ et $S^{p''}$. À gauche de la figure est reproduite une petite zone d'image réelle, suffisamment grossie pour permettre de bien individualiser les pixels et à laquelle ont été superposés les trois segments en question. Les deux relevés de la partie droite montrent quant à eux l'évolution des valeurs des niveaux de gris des pixels des deux isolines possibles que sont $S^nS^{p'}$ et $S^nS^{p''}$. On a également identifié les différents sous-ensembles de pixels S^n, $S^{p'}$ et $S^{p''}$ ainsi que les valeurs moyennes de chacun. À la lecture de ces deux représentations, on peut aisément imaginer que $S^{p'}$ ne soit pas retenu comme extension valide de S^n, au contraire de $S^{p''}$.

Pour formaliser ce que notre intuition semble nous dicter dans l'exemple précédent, nous comparons les log-vraisemblances des deux situations suivantes :

1. Le segment S^p **est** une extension valide pour S^n. Ils forment donc tous deux une isoline S^nS^p.
   Dans ce cas et par hypothèse, la valeur moyenne des niveaux de gris est définie sur S^nS^p et vaut $\mu_{S^nS^p}$. D'après (7.1), la log-vraisemblance est alors donnée par

$$-\frac{(n+p)}{2}(log(2\pi)+1)-\frac{(n+p)}{2}log\left(\widehat{\sigma_1}^2\right) \tag{7.6}$$

où $$\widehat{\sigma_1^2} = \frac{1}{n+p} \sum_{(i,j)\in S^n S^p} (v(i,j) - \widehat{\mu_{S^n S^p}})^2.$$

2. Le segment S^p **n'est pas** une extension valide pour S^n. Les deux parties ont des valeurs moyennes distinctes (μ_{S^n}, μ_{S^p}) et la log-vraisemblance est alors la somme des log-vraisemblances des deux portions.

$$-\frac{(n+p)}{2}\left(log\,(2\pi)+1\right) - \frac{n}{2}log\left(\widehat{\sigma_2}^2\right) - \frac{p}{2}log\left(\widehat{\sigma_2}^2\right) \quad (7.7)$$

où $$\widehat{\sigma_2^2} = \frac{1}{n+p}\left(\sum_{(i,j)\in S^n} (v(i,j) - \widehat{\mu_{S^n}})^2 + \sum_{(i,j)\in S^p} (v(i,j) - \widehat{\mu_{S^p}})^2\right).$$

La différence entre (7.6) et (7.7) nous donne l'expression du critère GLRT (*Generalized Likelihood Ratio Test*)

$$T(S^n, S^p, T_{max}) = T_{max} - (n+p)\left[log\left(\widehat{\sigma_1}^2\right) - log\left(\widehat{\sigma_2}^2\right)\right] \quad (7.8)$$

où T_{max} est un seuil arbitrairement fixé de sorte à produire des résultats visuels et chiffrés satisfaisant. Un allongement de S^n par S^p est validé si $T(S^n, S^p, T_{max}) > 0$.

7.3/ Modélisation des isolines pour l'implémentation parallèle sur GPU

Les isolines sont construites segment après segment. Cela permet de suivre des formes courbes. La validité d'un segment et son éventuelle sélection sont soumises au critère décrit dans le paragraphe précédent. Il nous est également apparu pertinent de limiter le nombre de segments candidats, ce qui permet d'apporter une réponse aux points suivants :

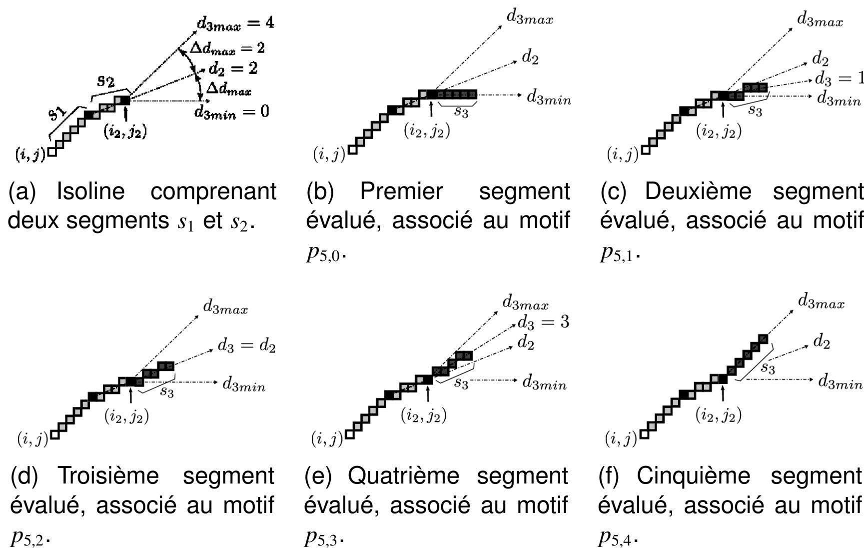
1. la sélection du premier segment est cruciale mais il n'est pas prouvé que la meilleure isoline soit celle qui a pour premier segment celui qui a été effectivement sélectionné en premier. Une telle erreur sur la direction primaire peut s'avérer très pénalisante pour la qualité du traitement. C'est pourquoi nous conduisons en parallèle les allongements des 32 isolines, chacune ayant l'un des motifs permis comme premier segment (voir figure 7.1).
2. évaluer systématiquement les 32 motifs pour chaque extension peut alors rendre l'algorithme très coûteux. En effet, si q est le nombre de segments maximum autorisés pour une isoline, le nombre d'évaluations à effectuer se monte à 32^q par pixel. Cela représente par exemple un total de $\mathbf{3,5.10^{13}}$ évaluations pour des isolines de $q = 5$ segments dans une image de 1024×1024 pixels.
3. permettre à tout allongement de se faire dans chacune des 32 directions risque de générer des isolines oscillant entre les deux extrémités de l'un de ses segments, ou bien s'enroulant sur elles-même au delà du simple rebouclage.
4. une ligne de niveau ne peut pas se couper, donc une isoline ne peut pas être composée de segments qui se croisent.

Les contraintes des points 3 et 4 ci-dessus nous ont conduit à limiter la déviation angulaire pouvant résulter de toute procédure d'allongement. Nous notons Δd_{max} l'écart maximal toléré entre les indices des motifs de deux segments successifs. Le choix d'une

valeur de Δd_{max} adaptée dépend de la taille des segments ainsi que du nombre maximal de segments que peut comporter une isoline. L'autre conséquence de cette limitation est la diminution du nombre total d'évaluations nécessaires. Si $\Delta d_{max} = 2$, le nombre d'évaluations effectuées dans l'exemple du point 2 passe ainsi à $1024^2 \times 32 \times 5^{q-1} = \mathbf{2,0.10^{10}}$ soit 1500 fois moins (avec $q = 5$).

7.3.1/ ISOLINES ÉVALUÉES SEMI-GLOBALEMENT

La première implémentation proposée et notée PI-LD (*Poly Isolines with Limited Deviation*), consiste donc à conduire l'allongement des 32 isolines candidates à leur terme, puis de sélectionner la plus vraisemblable parmi celles qui partagent la plus grande longueur. L'exemple de la figure 7.4 illustre ce processus pour la sélection du deuxième segment d'une isoline avec $a = 5$ et $\Delta d_{max} = 2$.



(a) Isoline comprenant deux segments s_1 et s_2.

(b) Premier segment évalué, associé au motif $p_{5,0}$.

(c) Deuxième segment évalué, associé au motif $p_{5,1}$.

(d) Troisième segment évalué, associé au motif $p_{5,2}$.

(e) Quatrième segment évalué, associé au motif $p_{5,3}$.

(f) Cinquième segment évalué, associé au motif $p_{5,4}$.

FIGURE 7.4 – Processus de sélection lors de l'allongement d'une isoline comportant initialement deux segments s_1 et s_2. Dans cet exemple $a = 5$ et $\Delta d_{max} = 2$. Chaque segment évalué est soumis au critère GLRT. Si au moins un des segments présente un test GLRT positif, alors l'allongement est réalisé avec le segment qui forme l'isoline la plus vraisemblable.

La rapidité de cette implémentation est très supérieure à celle des algorithmes *état de l'art* comme BM3D ([30]), la qualité du débruitage étant tout de même moindre. Le tableau 7.2 rassemble les performances comparées de nos implémentations et de celle du BM3D en y ajoutant comme référence de vitesse d'exécution un simple filtre moyenneur. Les mesures ont été réalisées sur l'ensemble des images de la base de test de S. Lansel (université de Berkeley), devenue entre temps indisponible au téléchargement, mais qui représente toujours une base de référence pour comparer des implémentations d'algorithmes de débruitage. Les images en sont reproduites à la figure 7.3.1.

L'adaptation de ce modèle au fonctionnement du GPU n'est pas non plus optimale du fait de la nécessité de réaliser, à chaque étape d'allongement, deux différents types de validation : un test GLRT et une minimisation de log-vraisemblance. Cela induit de nombreuses branches d'exécution divergentes dans le kernel principal, qui sont sérialisées

(a) airplane (b) boat (c) barbara (d) couple (e) elaine

(f) fingerprint (g) goldhill (h) lena (i) man (j) mandrill

(k) peppers (l) stream (m) zelda

FIGURE 7.5 – Images non bruitées de la base d'images en niveaux de gris de S. Lansel.

par le GPU et causent une perte de performance considérable.

Une analyse plus poussée des isolines construites nous montre qu'il y a une proportion relativement faible d'isolines optimales dont le premier segment s'écarte notablement de celui sélectionné l'absence d'allongement, c'est-à-dire par PI-LD avec $q = 1$. L'exemple représentatif de la figure 7.6 montre l'histogramme des différences constatée pour l'image du singe. Les autres images de l'ensemble de test fournissent des histogrammes très semblables qui sont reproduits en petit format à la figure 7.7. On y observe que pour environ 60% des pixels de l'image, il y a correspondance des directions et que pour 80% des pixels, l'écart angulaire reste inférieur à 2 (en indices des motifs).

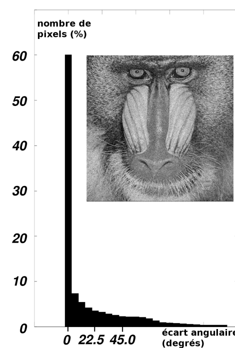
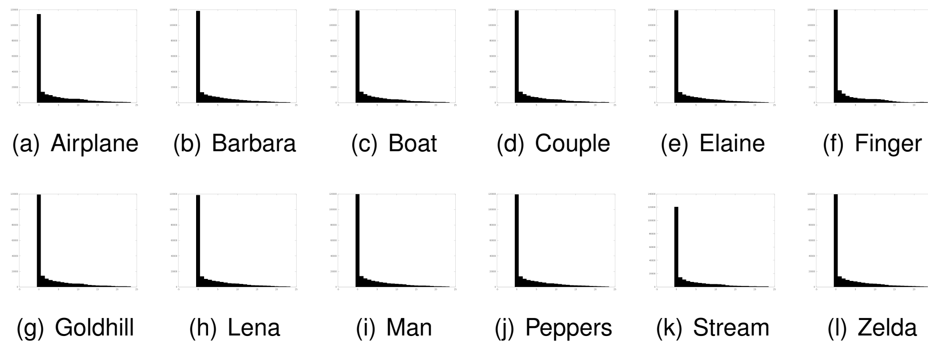


FIGURE 7.6 – Histogramme des écarts angulaires entre la direction primaire de l'isoline optimale et celle du segment sélectionné par PI-LD avec $q = 1$ (sans allongement), pour l'image du singe (Mandrill). Pour la très grande majorité des pixels, l'écart est nul.



(a) Airplane (b) Barbara (c) Boat (d) Couple (e) Elaine (f) Finger

(g) Goldhill (h) Lena (i) Man (j) Peppers (k) Stream (l) Zelda

FIGURE 7.7 – Histogrammes des écarts angulaires entre la direction primaire de l'isoline optimale et celle de l'isoline sélectionnée, pour les images de l'ensemble de test de S. Lansel. La répartition des erreurs est semblable dans toutes ces images, mais également dans toute image naturelle.

On observe également que les pixels pour lesquels la sélection du premier segment n'est pas robuste sont situés dans les zones de l'image ne contenant pas de forts gradients de niveaux de gris, ce qui est cohérent avec l'impossibilité d'identifier une direction privilégiée dans ces régions.

7.3.2/ ISOLINES À SEGMENTS PRÉ-ÉVALUÉS - MODÈLE PI-PD

Les observations précédentes nous indiquent que, dans les zones où la sélection du premier segment est robuste, il n'est pas nécessaire de conduire l'étape de sélection consécutive à chaque allongement. Sous cette hypothèse, étendre une isoline se terminant au point final (i, j) revient à sélectionner le premier segment de l'isoline débutant en (i, j).

Cette technique réduit considérablement la quantité d'évaluations à effectuer, la faisant passer de 32^q à seulement **160** évaluations par pixel (pour $q = 5$), soit un total de $\mathbf{1,7.10^8}$ pour une image de 124×1024 avec $a = 5$.

Ce nouveau modèle, nommé PI-PD (*Poly Isolines with Precomputed Directions*), permet donc de séparer complètement la phase de sélection du segment initial par maximum de vraisemblance des phases d'allongements successifs soumis au seul test GLRT. Pour implémenter efficacement cet algorithme sur GPU, il faut alors répartir les calculs en deux kernels principaux :

1. `kernel_precomp()` réalise la sélection du premier segment en chaque pixel (i, j). La direction $d_1(i, j)$ ainsi déterminée est mémorisée dans une matrice I_Θ. Pour effectuer les calculs relatifs au GLRT, il faut aussi connaître, pour chaque segment s_1, la valeur des sommes partielles

$$C_x(Z(S_1)) = \sum_{(i,j)\in s_1} v(i, j) \tag{7.9}$$

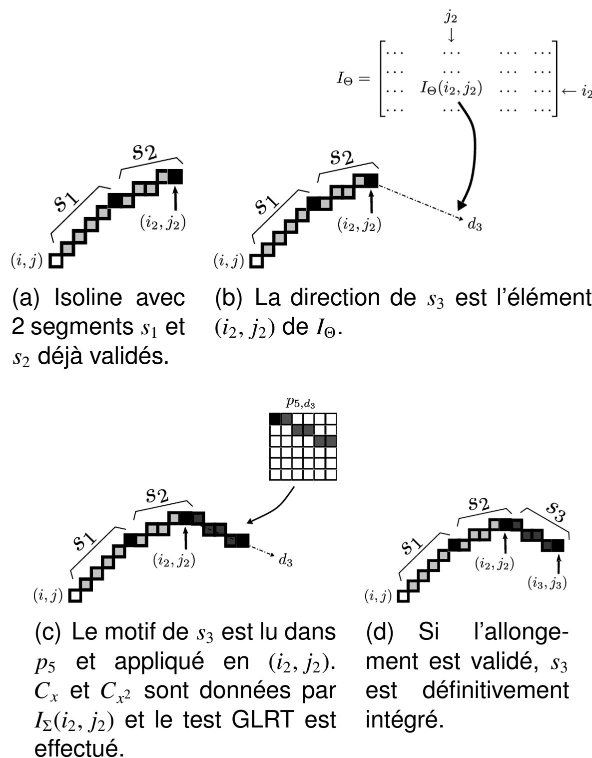
et

$$C_{x^2}(Z(S_1)) = \sum_{(i,j)\in s_1} v(i, j)^2 \tag{7.10}$$

Elles sont calculées et mémorisées dans une seconde matrice notée I_Σ. Remarquons que les traitements réalisés par ce kernel correspondent exactement au modèle d'isoline à un seul segment présenté au début. Les détails de son implémentation sont donnés dans l'algorithme 5, les initialisations étant données dans l'algorithme 4.

2. `kernel_PIPD()` évalue les allongements successifs, qui ne nécessitent plus de sélection par maximum de vraisemblance, mais uniquement la validation par GLRT. Les données nécessaires à l'évaluation du critère GLRT sont regroupées, outre dans l'image d'entrée, dans les matrices P_d, I_Θ et I_Σ et ne sont donc plus à calculer à ce stade. Cela permet d'envisager des performances en hausse par rapport à la solution PI-LD. L'algorithme 6 fournit les détails de l'implémentation de ce kernel.

Les schémas de la figure 7.8 illustrent les étapes décrites ci-dessus de l'allongement d'une isoline par la méthode PI-PD.



(a) Isoline avec 2 segments s_1 et s_2 déjà validés.

(b) La direction de s_3 est l'élément (i_2, j_2) de I_Θ.

(c) Le motif de s_3 est lu dans p_5 et appliqué en (i_2, j_2). C_x et C_{x^2} sont données par $I_\Sigma(i_2, j_2)$ et le test GLRT est effectué.

(d) Si l'allongement est validé, s_3 est définitivement intégré.

FIGURE 7.8 – Exemple d'application du procédé d'allongement à une isoline comprenant initialement 2 segments. la longueur des segments est $a = 5$. Le procédé se répète jusqu'à ce que le test GLRT échoue.

Le processus d'allongement du modèle PI-PD est également soumis aux restrictions sur les oscillations et retours en arrière des segments, déjà énoncées pour le modèle PI-LD. Par ailleurs, nous lui avons ajouté la possibilité de gérer des segments plus épais, composés de 2 ou 3 segments parallèles aux motifs décrits par la matrice P_d. Pour l'épaisseur 2, on utilise chaque segment motif et le segment parallèle situé immédiatement avant (au sens trigonométrique), pour l'épaisseur 3, on ajoute le segment parallèle situé immédiatement après le motif. Cela a pour effet d'intégrer plus de pixels dans les calculs statistiques et d'augmenter en conséquence les gains sur le PSNR, en particulier pour traiter des images de grandes dimensions qui ne contiendraient pas de *trop petits* dé-

Algorithme 4 : Initialisations du modèle PI-PD, en mémoire du GPU.

1 $l \leftarrow$ taille segments;
2 $D \leftarrow$ nombre de motifs/directions;
3 $I_n \leftarrow$ image d'entrée bruitée;
4 $I_{ntex} \leftarrow I_n$; /* copie en texture */
5 $P_l \leftarrow$ kernel_genPaths ; /* génération de la matrice P_l */
6 $P_{ltex} \leftarrow P_l$; /* copie en texture */
7 $T_{max} \leftarrow$ seuil GLRT pour les allongements;
8 $T2_{max} \leftarrow$ seuil GLRT pour la détection de bords;

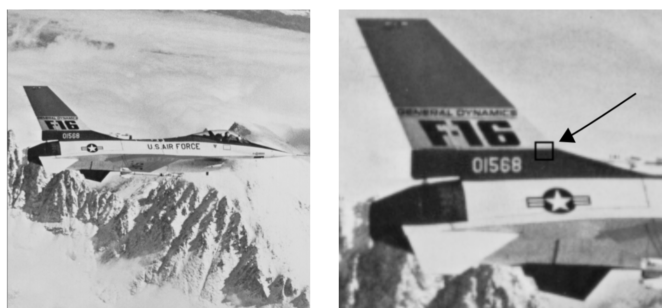
tails que l'épaisseur des isolines risquerait de flouter. Cette possibilité rend notre solution encore plus versatile que la référence BM3D dont les temps de calcul s'avèrent prohibitifs sur des images de grandes dimensions, avec par exemple plus de 5 minutes pour 4096×4096 pixels (Xeon quad core E312453.3GHz, 8Go RAM).

Toutefois, il demeure que l'isoline construite n'est pas nécessairement la plus vraisemblable pour tous les pixels de l'image, les optimisations étant faites sous l'hypothèse de robustesse énoncée au paragraphe 7.3.2.

7.3.3/ Modèle PI-PD hybride

Le manque de robustesse de la sélection des segments dans certaines zones provient du petit nombre de pixels impliqués dans les calculs statistiques. Ces régions sont celles où la pente de la surface définie par les niveaux de gris des pixels, pris comme élévations, est faible vis à vis du bruit qui perturbe l'image. Par souci de concision, nous nommons ces régions LSR (*Low Slope Regions*, régions à faible pente).

Pour illustrer ce comportement du modèle PI-PD, on peut sélectionner une région de petite taille (11×11 pixels) au sein d'une image de test. La zone d'étude, repérée à la figure 7.9, est choisie pour ses propriétés particulières : deux *plateaux* séparés par une transition nette, que l'on observe sur la représentation en trois dimensions de la figure 7.10(a).



(a) Image de référence non bruitée.

(b) La région de 11×11 pixels étudiée.

FIGURE 7.9 – Situation de la région servant à illustrer le comportant du modèle PI-PD dans les zones à faible pente (LSR).

Les figures 7.10(b) et 7.10(c) montrent la même zone de l'image après qu'elle ait été

Algorithme 5 : `kernel_precomp()` : génération des matrices I_Θ et I_Σ.

```
1  foreach pixel (i, j) do                                          /* en parallèle */
2      C_{x-best} ← Σ_{(y,x)∈p_{l,0}(i,j)} I_{ntex}(i + y, j + x) ;
3      C_{x2-best} ← Σ_{(y,x)∈p_{l,0}(i,j)} I^2_{ntex}(i + y, j + x) ;
4      σ_{best} ← écart type sur p_{l,0}(i, j) ;
       /* pour chaque motif de segment                                            */
5      foreach d ∈ [1..D − 1] do
6          C_x ← Σ_{(y,x)∈p_{l,d}(i,j)} I_{ntex}(i + y, j + x);
7          C_{x2} ← Σ_{(y,x)∈p_{l,d}(i,j)} I^2_{ntex}(i + y, j + x);
8          σ ← écart type sur p_{l,d}(i, j);
9          if σ_d < σ_{best} then                                   /* sélection par MV */
10             C_{x-best} ← C_x ;
11             C_{x2-best} ← C_{x2} ;
12             Θ_{best} ← d ;
13         end
14     end
15     I_Σ(i, j) ← [C_{x-best}, C_{x2-best}] ;                      /* mémorisation */
16     I_Θ(i, j) ← Θ_{best} ;                                        /* dans I_Θ et I_Σ */
17 end
```

corrompue par deux tirages d'un bruit gaussien de mêmes paramètres μ et σ. Les deux diagrammes 7.10(d) et 7.10(e) représentent quant à eux les directions primaires des isolines, modulo π, débutant en chaque pixel de la fenêtre. On observe que la détermination de la direction est robuste dans la bande de transition entre les LSR, alors que pour les LSR elles-mêmes, on constate une grande variabilité.

Ainsi, dans les LSR, l'application du modèle PI-PD n'a que peu de sens, mais la quête de performance nous interdit d'y appliquer par exemple le modèle PI-LD décrit précédemment. Le meilleur estimateur dans une zone LSR étant la valeur moyenne, nous proposons donc, à la place :

1. d'identifier les zones à faible pente en concevant un kernel détecteur (`kernel_LSR_detector()`).
2. d'appliquer un simple filtre moyenneur dans les zones désignées LSR par le détecteur et le PI-PD partout ailleurs.
3. de n'appliquer le moyenneur que sur les pixels appartenant à la zone LSR lorsque la fenêtre du détecteur se trouve à cheval sur deux zones de types différents.

7.3.3.1/ Le détecteur de zone à faible pente

Le principe retenu pour réaliser le détecteur de LSR est proche de celui mis en oeuvre pour valider les allongements des isolines : il s'agit de séparer la fenêtre d'observation autour du pixel considéré en deux régions, puis d'effectuer un test GLRT pour déterminer

Algorithme 6 : `kernel_PIPD()` : gestion du processus d'allongement.

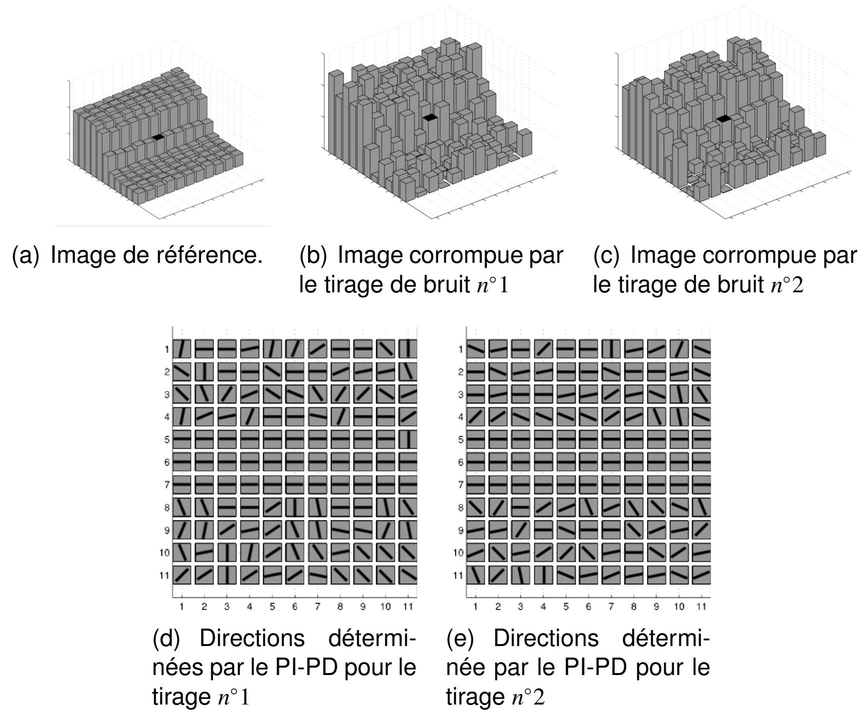
```
1  foreach pixel (i, j) do                                   /* en parallèle */
2      (C_x^1, C_x2^1) ← z(i, j) ;                           /* pixel de départ */
3      (i_1, j_1) ← (i, j) ;                                  /* premier segment */
4      (C_x^1, C_x2^1) ← I_Σ(i_1, j_1) ;                      /* lecture depuis I_Σ */
5      d_1 ← I_Θ(i, j) ;                                      /* lecture depuis I_Θ */
6      l ← n ;                                                /* longueur de l'isoline */
7      σ_1 ← (C_x2^1/l − C_x^1)/l;
8      (i_2, j_2) ← fin du premier segment;
9      (C_x^2, C_x2^2) ← I_Σ(i_2, j_2) ;                      /* 2^nd segment */
10     d_2 ← I_Θ(i_2, j_2);
11     σ_2 ← (C_x2^2/n − C_x^2)/n ;
12     while GLRT(σ_1, σ_2, l, n) < T_max do
13         l ← l + n ;                                        /* allongement */
14         (C_x^1, C_x2^1) ← (C_x^1, C_x2^1) + (C_x^2, C_x2^2);
15         σ_1 ← (C_x2^1/l − C_x^1)/l ;                       /* mise à jour */
16         (i_1, j_1) ← (i_2, j_2) ;                          /* décalage */
17         d_1 ← d_2;
18         (i_2, j_2) ← fin du segment suivant;
               /* segment suivant */ (C_x^2, C_x2^2) ← I_Σ(i_2, j_2);
19         d_2 ← I_Θ(i_2, j_2);
20         σ_2 ← (C_s2^2/n − C_s^2)/n ;
21     end
22 end
23 Î(i, j) ← C_x^1/l ;                                        /* niveau de gris en sortie */
```

s'il est vraisemblable ou non que ces deux régions forment un seul et même plan. Pour garantir la prise en compte d'éventuelles transitions dans toutes les directions, il faut effectuer le test avec des séparations de fenêtre dont les directions couvrent toute la plage angulaire, de 0 à π.

L'utilisation d'un test GLRT semblable à celui de l'équation (7.8) sous-entend que les ensembles considérés n'ont aucun pixel en commun. Afin d'éviter de devoir déterminer de nouveaux ensembles de pixels pertinents, nous avons utilisé les motifs de la matrice P_d, n'ayant pas d'intersection entre eux et de directions $\Theta_{4i} = 4i\frac{\pi}{4}$. Ces motifs remplissent les critères pour établir l'expression d'un critère GLRT. La ligne de séparation entre les deux régions de la fenêtre est donc composée par les motifs de directions Θ_{4i} et $\Theta_{4(i+4)}$. Ces deux régions sont respectivement nommées arbitrairement T et B, T étant représentée comme la région *haute* et B comme la région *basse* sur le schéma explicatif de la figure 7.11 où $\Theta_{4i} = \frac{\pi}{4}$ et où les pixels affectés d'une élévation nulle sont les pixels non impliqués dans le calcul du critère GLRT. En outre, les pixels de la limite sont supposés appartenir à la région T, ce qui implique qu'elle comprend au total les pixels correspondant à cinq motifs plus le pixel central, tandis que B n'en comprend que l'équivalent de 3 motifs.

Les équations (7.6), (7.7) et (7.8) nous permettent d'obtenir l'expression suivante pour le critère GLRT $T2$

$$T2 = T2_{max} - (8a+1)\left[log\left(\widehat{\sigma_3}^2\right) - log\left(\widehat{\sigma_4}^2\right)\right] \tag{7.11}$$



(a) Image de référence. (b) Image corrompue par le tirage de bruit $n°1$ (c) Image corrompue par le tirage de bruit $n°2$

(d) Directions déterminées par le PI-PD pour le tirage $n°1$ (e) Directions déterminée par le PI-PD pour le tirage $n°2$

FIGURE 7.10 – Comportement du modèle PI-PD dans les zones de faible et à forte pente. On constate un manque de robustesse dans les zones à faible pente : les directions ne sont pas reproduites d'un tirage à l'autre, contrairement à celles de la zone de transition.

où $\widehat{\sigma_3}$ est l'estimation de l'écart type dans le cas où les deux demi régions en formeraient une seule et $\widehat{\sigma_4}$, l'estimation de l'écart type pour le cas où une transition serait détectée entre les deux. Leurs expressions sont donc :

$$\widehat{\sigma_3}^2 = \frac{1}{8a+1} \sum_{(i,j)\in T\cup B} (v(i,j) - \widehat{\mu_{T\cup B}})^2$$

et

$$\widehat{\sigma_4}^2 = \frac{1}{8a+1} \left(\sum_{(i,j)\in T} (v(i,j) - \widehat{\mu_T})^2 + \sum_{(i,j)\in B} (v(i,j) - \widehat{\mu_B})^2 \right)$$

Le seuil de décision est noté $T2_{max}$ et d'après l'expression du critère (7.11), une valeur négative du critère signifie la détection d'une transition. Ainsi, lorsque les valeurs du critère $T2$ sont connues pour toutes les 8 directions $\Theta_{4i}(i \in [0..7])$, la valeur du niveau de gris de sortie pour le pixel central est déterminée selon la stratégie suivante :

- si plus d'une valeur du critère est négative, alors on applique la valeur issue du modèle PI-PD.
- si une seule valeur du critère est négative, le pixel central est vraisemblablement situé sur une transition nette et on applique la valeur moyenne des motifs de la région T à laquelle il appartient. Cela permet de garantir des transitions visuellement plus douces entre les zones où le PI-PD est appliqué et les zones moyennées.
- si aucune valeur du critère n'est négative, alors la région autour du pixel central est vraisemblablement une LSR. En conséquence, on applique la valeur moyenne de la zone.

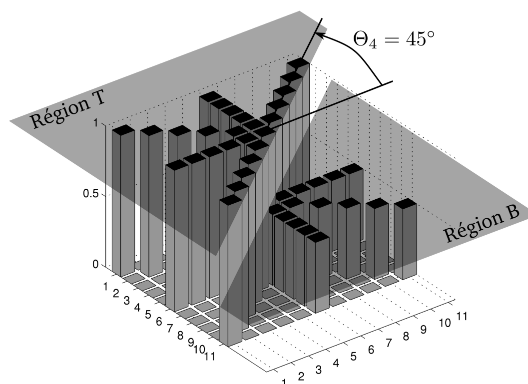
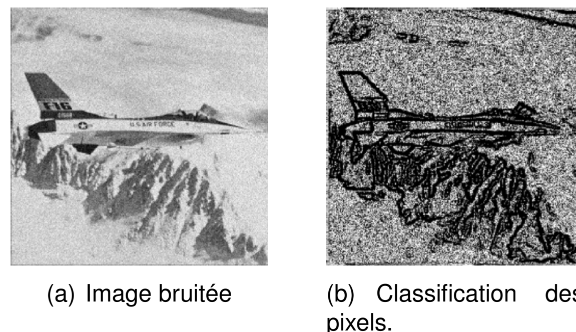


FIGURE 7.11 – Motif de détection des zones à faible pente, pour le cas $\Theta = \Theta_4 = 45°$. L'élévation des pixels permet juste de les distinguer selon 3 classes : l'élévation 1 est associée aux pixels de la région T, l'élévation 0.5 est associée à ceux de la région B et l'élévation 0 désigne les pixels n'intervenant pas dans la détection.

La figure 7.12 présente le résultat de la classification des pixels d'une image bruitée, pour $T2_{max} = 2$. On y remarque en particulier que les pixels noirs, pour lesquels s'appliquera le PI-PD, sont situés sur des transitions bien définies.



(a) Image bruitée (b) Classification des pixels.

FIGURE 7.12 – Classification des pixels d'une image bruitée, pour une valeur de seuil $T2 = 2$ du détecteur. b) Les pixels en noir sont ceux à qui le PI-PD sera appliqué. Les pixels en blancs se verront appliquer une moyenne sur tout ou partie du voisinage.

Les détails d'implémentation du détecteur sont donnés par l'algorithme 7. Pour en optimiser les performances, les sommes individuelles sum_Θ sont pré-calculées aux lignes 7 à 10 pour les 8 motifs concernés. L'évaluation des 8 configurations angulaires est effectuée ensuite de la ligne 11 à la ligne 25.

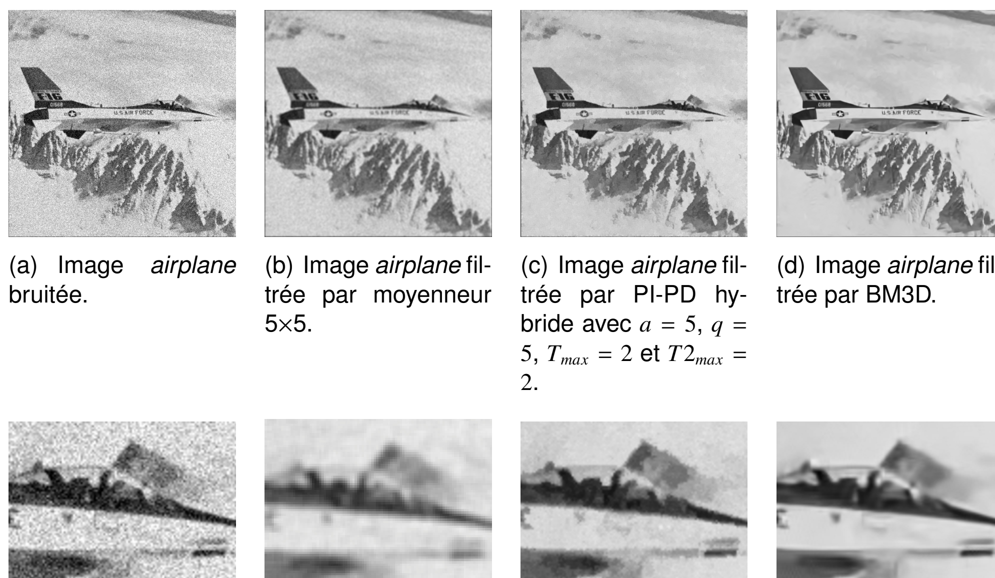
7.4/ RÉSULTATS

L'implémentation du PI-PD hybride a été appliquée aux 13 images de la base de test, dans leurs versions les plus bruitées, perturbées par un bruit gaussien de moyenne nulle et d'écart type 25. Pour ce type d'images (taille, détails), les paramètres qui se sont avérés optimaux sont $a = 5$ pour la longueur des segments avec un maximum de $q = 5$ segments. En ce qui concerne les seuils GLRT, nous avons testé l'ensemble des com-

binaisons de valeurs T_{max} et $T2_{max}$ variant de 1 à 10 par pas de 0,5. La combinaison $T_{max} = 1$ et $T2_{max} = 2$ s'est révélée la plus appropriée, en ce sens qu'elle représente l'optimum pour 11 des 13 images, sauf *peppers* et *zelda*, pour lesquelles une combinaison $T_{max} = 2$ et $T2_{max} = 2$ permet d'améliorer l'indice de similarité MSSIM respectivement de 0,03 et 0,02.

Les images filtrées ont été caractérisées en termes de PSNR et de MSSIM et les résultats, regroupés dans la table 7.2, sont comparés à ceux de la référence BM3D, ainsi qu'à ceux d'un simple filtre moyenneur GPU 5×5, choisi comme référence en terme de rapidité et dont la taille de fenêtre permet des gains théoriques en PSNR du même ordre de grandeur que le PI-PD.

Les mesures de qualité montrent que le PI-PD hybride améliore en moyenne le PSNR de 1,5 dB et le MSSIM de 7,3% par rapport au moyenneur, au prix d'un temps de calcul multiplié par 100, soit environ 7,3 ms, là où l'algorithme PI-LD prenait 35 ms. Le BM3D fait encore progresser la qualité de 2,4 dB et 4,6% en moyenne par rapport au PI-PD hybride, mais en mettant 590 fois plus de temps que ce dernier, soit environ 4,3 s. Le principal défaut du filtre proposé est la génération d'artefacts de type marches d'escalier (staircase effect), inhérente à tous les filtres de voisinage. Cependant, nous avons implémenté sur GPU la solution proposée par Buades dans [14] et ainsi attenué nettement cet effet indésirable pour un coût de 0,2 ms. La valeur du PSNR de chaque image débruitée a ainsi été encore améliorée de 1 dB. La figure 7.13 permet de constater le rendu visuel des traitements comparés, sur l'image entière ainsi que sur une zone grossie de l'image *airplane*.



(a) Image *airplane* bruitée.

(b) Image *airplane* filtrée par moyenneur 5×5.

(c) Image *airplane* filtrée par PI-PD hybride avec $a = 5$, $q = 5$, $T_{max} = 2$ et $T2_{max} = 2$.

(d) Image *airplane* filtrée par BM3D.

FIGURE 7.13 – Comparaison des rendus des traitements comparés. Rangée du haut : les images complètes. Rangée du bas : Zooms sur une zone de l'ímage au dessus.

Les temps de calcul des différentes implémentations testées dépendent très peu du contenu de l'image, voire pas du tout pour le moyenneur. Ils sont présentés à la table 7.1. Pour les implémentations GPU, il faut ajouter, dans le cas de traitements uniques (hors séquences d'images), les temps de transfert des images vers la mémoire texture du GPU puis vers une zone de mémoire non paginée de l'hôte CPU, qui représentent un total de 0,15 ms pour les images de test, soit moins de 2% du temps total du PI-

PD hybride. Notons que l'emploi de mémoire pré-allouée (ne générant pas de défaut de page) pour la mémorisation des données côté CPU permet d'économiser 0,09 ms par image 8 bits, soit environ 1% du temps total du PI-PD. Notons enfin que le traitement de séquences en haute définition (1920×1080 pixels) au taux 20 images par seconde est rendu possible.

7.5/ EXTENSION AUX IMAGES COULEURS

7.5.1/ EXPRESSION DU CRITÈRE

Considérons une image couleur à 3 canaux RVB (Rouge, Vert et Bleu). La valeur v_k observée au pixel k est alors un vecteur à trois éléments. Nous faisons ici l'hypothèse de canaux décorrelés, conduisant à une matrice de covariance diagonale de la forme $R = \sigma^2 \mathbb{1}_3$ où σ^2 est la puissance du bruit gaussien perturbant les trois canaux, chaque canal pouvant être corrompu par un tirage de bruit particulier. La probabilité de v_k est alors

$$P(v_k|R) = \left(\frac{1}{2\pi^{3/2}\sqrt{|R|}} e^{-\frac{1}{2}(v_k-\mu)^T R^{-1}(v_k-\mu)}\right)$$

Pour exprimer le critère GLRT de validation des allongements, nous procédons comme précédemment, c'est-à-dire en distinguant les deux hypothèses :

1. le segment candidat S^p prolonge effectivement l'isoline S^n : ils partagent donc la même valeur moyenne μ et la log-vraisemblance s'écrit

$$\sum_{(i,j)\in S^p\cup S^n} -\frac{3}{2}log(2\pi) - \frac{1}{2}log(|R|) - \frac{1}{2}\left(v_{(i,j)} - \mu\right)^T R^{-1}\left(v_{(i,j)} - \mu\right)$$

Soit

$$-\frac{3}{2}(n+p)log\left(2\pi\right) - \frac{3}{2}(n+p)log\left(\widehat{\sigma_0^2}\right) - \frac{3}{2}(n+p) \tag{7.12}$$

où

$$\widehat{\sigma_0^2} = \frac{1}{3(n+p)} \sum_{(i,j)\in S^p\cup S^n} \left(v_{(i,j)} - \widehat{\mu}\right)^T \left(v_{(i,j)} - \widehat{\mu}\right) \tag{7.13}$$

et

$$\widehat{\mu} = \frac{1}{(n+p)} \sum_{(i,j)\in S^p\cup S^n} v_{(i,j)} \tag{7.14}$$

2. le segment candidat ne prolonge pas l'isoline S^n : on distingue alors leur deux valeurs moyennes μ_p et μ_n et la log-vraisemblance s'écrit

$$\begin{aligned}&\sum_{(i,j)\in S^n} -\frac{3}{2}log(2\pi) - \frac{1}{2}log(|R|) - \frac{1}{2}\left(v_{(i,j)} - \mu_n\right)^T R^{-1}\left(v_{(i,j)} - \mu_n\right)\\ +&\sum_{(i,j)\in S^p} -\frac{3}{2}log(2\pi) - \frac{1}{2}log(|R|) - \frac{1}{2}\left(v_{(i,j)} - \mu_p\right)^T R^{-1}\left(v_{(i,j)} - \mu_p\right)\end{aligned} \tag{7.15}$$

Soit

$$-\frac{3}{2}(n+p)log\,(2\pi) - \frac{3}{2}(n+p)log\left(\widehat{\sigma_1{}^2}\right) - \frac{3}{2}(n+p) \tag{7.16}$$

où

$$\widehat{\sigma_1{}^2} = \frac{1}{3(n+p)}\left(\sum_{(i,j)\in S^n}\left(v_{(i,j)} - \widehat{\mu_n}\right)^T\left(v_{(i,j)} - \widehat{\mu_n}\right) + \sum_{(i,j)\in S^p}\left(v_{(i,j)} - \widehat{\mu_p}\right)^T\left(v_{(i,j)} - \widehat{\mu_p}\right)\right) \tag{7.17}$$

et

$$\widehat{\mu_n} = \frac{1}{n}\sum_{(i,j)\in S^n} v_{(i,j)} \tag{7.18}$$

$$\widehat{\mu_p} = \frac{1}{p}\sum_{(i,j)\in S^p} v_{(i,j)} \tag{7.19}$$

Le critère GLRT s'obtient par la soustraction des deux expressions de (7.16) et (7.12) :

$$T_{rvb} = 3(n+p)\left(-log\left(\widehat{\sigma_1{}^2}\right) + log\left(\widehat{\sigma_0{}^2}\right)\right)$$

On notera $T_{rvb-max}$ la valeur de seuil au delà de laquelle on ne validera pas l'allongement de l'isoline.

7.5.2/ Résultats

Nous avons retenu la base d'images de test tid2008 [78] pour évaluer la qualité du traitement PI-PD sur les images couleurs. Cet ensemble d'images a été utilisé avec nombre d'algorithmes de débruitage et les résultats de mesure sont disponibles. Chacune des 25 images de référence (non bruitées) a subi 4 niveaux de distorsion, pour 17 types de bruit différents. Pour nos expérimentations, nous avons selectionné les 25 images corrompues par un bruit gaussien RVB (type 2 dans tid2008) d'écart type $\sigma = 25$ (niveau 4 dans tid2008), où chaque canal RVB est perturbé par un tirage de bruit gaussien scalaire. La figure 7.5.2 présente les vignettes des 25 images de référence, soit 24 images *naturelles* et une image de synthèse.

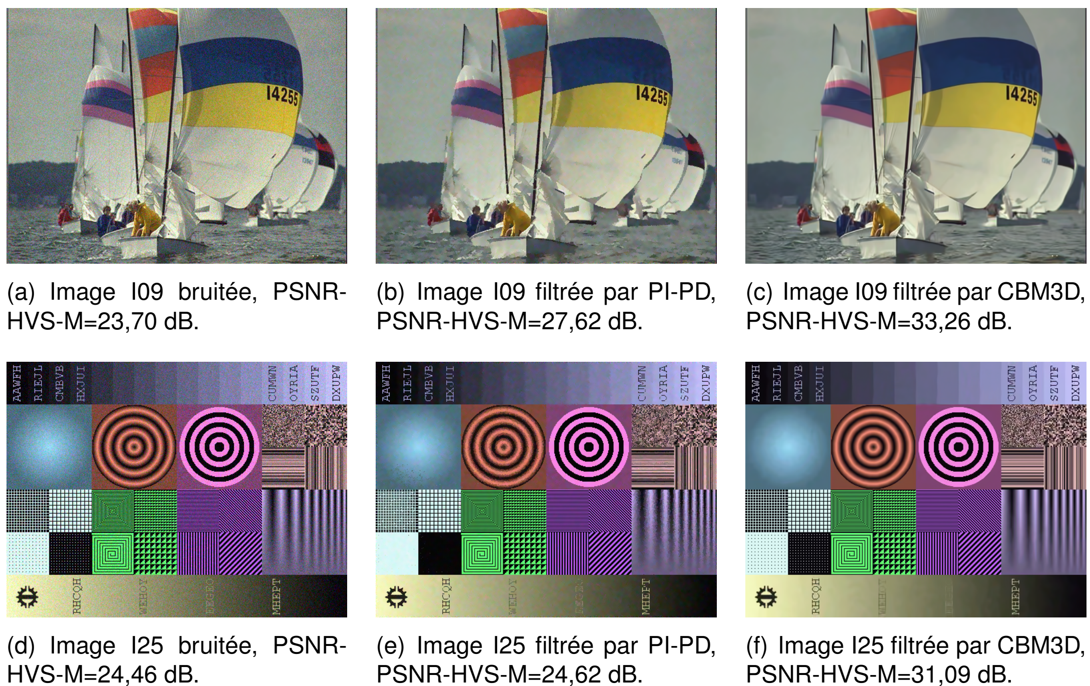
Notre référence est ici encore l'implémentation BM3D dans sa variante couleurs (CBM3D) et nous avons choisi d'exprimer la qualité de débruitage au travers la valeur du PSNR-HVS-M (voir [79]) qui est une extension du simple PSNR prenant en compte des caractéristiques structurelles de l'image. Les expérimentations décrites dans [78] montrent en outre, que pour les perturbations de la catégorie *noise* à laquelle appartient le type 2 qui nous intéresse, le PSNR-HVS-M présente les meilleures corrélations avec la perception humaine de la qualité, que ce soit au sens de Spearman ou de Kendall. Comme pour les images en niveaux de gris, notre implémentation RVB intègre la réduction de l'effet *marches d'escalier*, que nous avons adapté à la couleur en choisissant la norme 2 comme mesure de distance dans l'espace RVB. Nous avons aussi expérimenté une variante employant la norme 1, avec des résultats moins satisfaisants. Cette étape améliore le rendu visuel mais représente cette fois une proportion plus importante du temps de calcul, en raison du calcul de la norme, plus coûteux. Sur les images de 512×512, cela représente environ 1 ms, soit environ 25% du temps de calcul.

(a) I01 (b) I02 (c) I03 (d) I04 (e) I05
(f) I06 (g) I07 (h) I08 (i) I09 (j) I10
(k) I11 (l) I12 (m) I13 (n) I14 (o) I15
(p) I16 (q) I17 (r) I18 (s) I19 (t) I20
(u) I21 (v) I22 (w) I23 (x) I24 (y) I25

FIGURE 7.14 – Images non bruitées de la base tid2008.

Le PI-PD en couleur s'exécute quant à lui à la même vitesse qu'en niveaux de gris, soit environ 4,0 ms ; c'est aussi le cas de CBM3D avec une moyenne de 4,3 secondes. Sur les 25 images de test, le gain moyen apporté par PI-PD s'élève à 2,84 dB (PSNR-HVS-M) contre 7,09 dB pour CBM3D, ce qui constitue indéniablement un échelon supérieur en terme de qualité, au prix d'un temps de calcul multiplié par 1000.

L'ensemble des résultats de mesure est consigné dans le tableau 7.3 et deux exemples de résultats sont reproduits en figure 7.5.2 pour une des images naturelles ainsi que pour l'image de synthèse. Les valeurs des paramètres sont identiques pour toutes les images et ont été déterminées empiriquement par analyse systématique des résultats produits par les combinaisons permises dans les intervalles de 3 à 7 pour la taille n des segments, de 25 à 70 pour la longueur maximale l des isolines et de 1 à 10 pour le seuil GLRT $T_{rvb-max}$. Cette analyse extensive a mis en évidence la combinaison $n = 4$, $l = 48$ et $T_{rvb-max} = 5$ comme permettant au PI-PD d'apporter les meilleurs résultats d'ensemble. Certaines des images, comme l'image de synthèse n°25, bénéficieraient d'un ajustement des paramètres, mais conscients de la contrainte que cela représente, nous avons choisi de faire prévaloir un réglage unique.



(a) Image I09 bruitée, PSNR-HVS-M=23,70 dB.

(b) Image I09 filtrée par PI-PD, PSNR-HVS-M=27,62 dB.

(c) Image I09 filtrée par CBM3D, PSNR-HVS-M=33,26 dB.

(d) Image I25 bruitée, PSNR-HVS-M=24,46 dB.

(e) Image I25 filtrée par PI-PD, PSNR-HVS-M=24,62 dB.

(f) Image I25 filtrée par CBM3D, PSNR-HVS-M=31,09 dB.

FIGURE 7.15 – Exemples de résultat de traitement par PI-PD RVB et par CBM3D pour deux images de la base tid2008 (une image naturelle et l'image de synthèse). Il peut être nécessaire de zoomer sur le document numérique pour visualiser les détails.

7.6/ CONCLUSION

L'algorithme PI-PD hybride permet de débruiter 19 images en haute définition à la seconde tout en réduisant de manière importante le niveau de bruit gaussien. La démarche adoptée pour sa conception a été de se baser sur des opérations élémentaires dont nous connaissions ou avions démontré l'efficacité sur GPU. Nous jugeons ce principe essentiel pour la conception d'algorithmes GPU performants et robustes tant le débogage peut s'avérer délicat sur ces plateformes. Par ailleurs, il nous semble peu pertinent systématiquement comparer les implémentations CPU et GPU pour en déduire un facteur d'accélération comme on le rencontre trop souvent. La plupart des algorithmes qui s'avèrent rapides sur GPU ne le sont vraisemblablement pas sur CPU et il est donc tout à fait illusoire de penser qu'il en existe une implémentation optimisée. Comparer alors une implémentation GPU performante avec son pendant CPU naïf ne présente aucun intérêt. La réciproque étant généralement vraie, nous avons choisi, en particulier en ce qui concerne le filtrage dont il est question ici, de chercher à assembler des blocs fonctionnels simples mais robustes et performants avec l'objectif opérationnel de réduire la puissance de bruit.

L'algorithme et les résultats que nous avons détaillés dans ce chapitre ont été publiés dans le *Journal of real-time image processing* dans un article intitulé *Fast GPU-based denoising filter using isoline levels* [76].

Algorithme 7 : Détecteur de zones à faible pente (LSR) `kernel_LSR_detector()`

```
1  foreach pixel (i, j) do                                          /* en parallèle */
2      Θ ← 0;                                               /* Indice de la direction */
3      edgeCount ← 0;
4      sumEdge ← 0;
5      nT ← 5l + 1;
6      nB ← 3l;
7      while (Θ < 32) do
8          sum_Θ ← ( Σ_{(y,x)∈P_{l,α}(i,j)} I_ntex(i+y, j+x), Σ_{(y,x)∈P_{l,α}(i,j)} I²_ntex(i+y, j+x) );
9          Θ ← Θ + 4;
10     end
11     while (Θ < 32) do
12         sumT ← (I_ntex(i, j), I²_ntex(i, j));
13         sumB ← (0, 0) ;
14         for (α = Θ to α = Θ + 16 par pas de 4) do
15             sumT ← sumT + sum_Θ;
16         end
17         for (α = Θ + 20 to α = Θ + 28 par pas de 4) do
18             sumB ← sumB + sum_Θ ;
19         end
20         if (GLRT(sumT, nT, sumB, nB) > T2_max) then
21             edgeCount ← edgeCount + 1;
22             sumEdge ← sumT.x;
23         end
24         Θ ← Θ + 4;
25     end
       /* niveau de gris de l'isoline                                            */
26     if (edgeCount == 0) then
27         Î(i, j) ← (sumT.x + sumB.x) / (nT + nB) ;                          /* LSR */
28     end
29     if (edgeCount == 1) then
30         Î(i, j) ← (sumEdge) / nT
31     end
32     if (edgeCount > 1) then
33         Î(i, j) ← Î_PIPD(i, j);                                           /* PI-PD */
34     end
35 end
```

Ligne 8 : $sum_\Theta \leftarrow \left(\sum_{(y,x)\in P_{l,\alpha}(i,j)} I_{ntex}(i+y, j+x), \sum_{(y,x)\in P_{l,\alpha}(i,j)} I^2_{ntex}(i+y, j+x) \right);$

Ligne 27 : $\widehat{I}(i,j) \leftarrow \frac{(sumT.x + sumB.x)}{nT + nB};$

Ligne 30 : $\widehat{I}(i,j) \leftarrow \frac{(sumEdge)}{nT}$

Ligne 33 : $\widehat{I}(i,j) \leftarrow \widehat{I_{PIPD}}(i,j);$

	Temps de calcul (ms)	Temps de transfert (ms)
Moyenneur	0.07	0.15
PI-PD hybride	7.30	0.15
BM3D	4300	...

TABLE 7.1 – Temps de calcul et de transfert des implémentations comparées.

Image	**Bruitée**	**Moyenneur** 5×5	**PI-LD**	**PI-PD hybride**	**BM3D**
	PSNR (dB) MSSIM	gain (dB)/noisy MSSIM	gain (dB)/noisy MSSIM	gain (dB)/noisy MSSIM	gain (dB)/noisy MSSIM
airplane	19.49 0.58	6.90 0.84	8.94 0.78	8.97 0.88	11.39 0.93
barbara	20.04 0.70	2.72 0.76	4.84 0.79	4.22 0.83	10.56 0.94
boat	20.33 0.66	5.25 0.81	6.86 0.81	7.21 0.87	9.69 0.91
couple	20.28 0.69	4.97 0.79	6.77 0.82	7.05 0.87	9.49 0.91
elaine	19.85 0.59	8.86 0.86	8.16 0.79	9.09 0.87	10.75 0.91
fingerprint	20.34 0.93	2.99 0.87	6.00 0.95	5.73 0.95	7.59 0.96
goldhill	19.59 0.67	6.88 0.82	8.02 0.81	7.84 0.87	9.63 0.88
lena	19.92 0.60	8.07 0.84	8.37 0.78	9.22 0.88	11.88 0.93
man	20.38 0.71	4.36 0.80	6.49 0.83	6.36 0.86	7.76 0.87
mandrill	19.34 0.77	1.00 0.69	4.20 0.83	3.04 0.83	5.41 0.88
peppers	19.53 0.61	7.77 0.86	8.66 0.79	9.15 0.87	11.34 0.92
stream	20.35 0.80	2.88 0.78	4.97 0.87	5.00 0.87	5.99 0.88
zelda	17.71 0.58	10.42 0.87	11.13 0.79	10.00 0.88	12.78 0.93

TABLE 7.2 – Comparaison image par image de la qualité de débruitage des filtres PI-LD et PI-PD hybride proposé par rapport à BM3D pris comme référence de qualité et à un moyenneur GPU 5×5 pris comme référence de rapidité. Les paramètres du PI-LD/PI-PD sont $n = 5$, $l = 25$, $T_{max} = 1$ et $T2_{max} = 2$. La colonne 'Bruitée' donne les mesures relatives à l'image d'entrée corrompue par un bruit gaussien de moyenne nulle et d'écart type $\sigma = 25$. PI-LD s'exécute en 35 ms, PI-PD en 7,3 ms et BM3D en 4,3 s.

Image	Noisy PSNR-HVS-M (dB)	PI-PD gain (dB HVS-M)/noisy	BM3D gain (dB HVS-M)/noisy
1	23.91	26.03	31.02
2	23.39	26.31	28.96
3	23.31	28.17	33.04
4	23.21	27.59	31.30
5	24.52	26.26	30.63
6	23.91	25.61	29.45
7	23.68	27.21	32.66
8	24.51	25.81	30.75
9	23.70	27.62	33.26
10	23.49	27.16	32.34
11	23.95	26.21	30.63
12	23.33	27.26	31.67
13	24.17	24.91	29.45
14	24.03	25.83	30.15
15	23.55	27.20	30.63
16	23.27	27.11	31.73
17	23.74	27.47	32.18
18	24.04	25.67	28.94
19	24.20	27.06	31.45
20	23.27	26.58	26.38
21	23.75	26.70	31.41
22	23.55	26.22	29.24
23	23.48	27.98	32.11
24	23.58	26.40	30.88
25	24.46	24.62	31.09

TABLE 7.3 – Comparaison image par image de la qualité de débruitage du filtre PI-PD RVB proposé par rapport à BM3D pris comme référence de qualité. Les paramètres du PI-PD sont $n = 4$, $l = 48$, $T_{rvb-max} = 5$. La colonne 'noisy' donne les mesures relatives à l'image d'entrée corrompue par tirage de bruit gaussien sur chaque canal (moyenne nulle, écart type $\sigma = 25$).

8

LE FILTRE MÉDIAN SUR GPU

8.1/ INTRODUCTION

Au cours de nos expérimentations, en particulier concernant le débruitage par lignes de niveaux décrit dans le chapitre précédent, nous avons cherché à comparer les performances d'un certain nombre d'algorithmes de filtrage portés sur GPU. Comme nous l'avons dit dans le chapitre 4, il s'est avéré que le filtre médian n'avait pas fait l'objet de beaucoup de publications. On a tout de même recensé quelques implémentations intéressantes des algorithmes BVM et PCMF, ainsi que l'existence d'une solution commerciale libJacket/Arrayfire (se reporter au paragraphe 4.2.2).

Les performances annoncées pour des fenêtres de petite taille comme 3×3 pouvent atteindre jusqu'à 180 millions de pixels traités à la seconde dans le cas d'Arrayfire. En regard du petit nombre d'opérations à effectuer pour sélectionner la valeur médiane dans une fenêtre 3×3, il nous a semblé que ces débits étaient très en deçà des possibilités des GPUs employés.

Un rapide prototypage a conforté cette idée et nous a conduit à chercher plus avant une technique d'implémentation du filtre médian qui exploite pleinement les capacités de nos GPU.

8.2/ LES TRANSFERTS DE DONNÉES

Le chapitre 2, présentant l'architecture et les caractéristiques principales des GPUs, donne également la liste et les spécificités des types de mémoire accessibles par un kernel. Lorsqu'il s'agit de stocker des volumes importants de données, comme les images d'entrée et de sortie, les alternatives sont assez limitées. En effet, le seul espace mémoire suffisamment important est celui la mémoire dite globale, malheureusement la plus lente. On dispose cependant de plusieurs modes pour y accéder, comme la déclaration de textures, qui offre un mécanisme de cache 2D permettant d'augmenter assez nettement les débits en lecture dans le cas d'accès au voisinage d'une donnée. Dans le cadre de nos travaux, cette mémorisation sous forme de texture s'est montrée la plus performante pour les images d'entrée.

Les images de sortie filtrées sont produites en mémoire globale standard, hors texture, puis copiées vers une zone de mémoire de l'hôte (CPU) dont les pages sont réservées à l'avances et verrouillées, ce qui évite les pertes de performances liées aux défauts de page. L'algorithme 8 synthétise ces pratiques en introduisant aussi les notations pour la

suite. Cet emploi de mémoire que l'on qualifiera dorénavant de « non paginée », apporte un gain de temps important dans les transferts même s'il peut aussi s'avérer limitant lorsqu'il s'agit de traiter de très grands volumes de données. Les quantités de mémoire vive dont disposent les ordinateurs modernes permettent cependant de traiter sans restriction des images de plusieurs centaines de millions de pixels. Nos essais ont été conduits avec des images d'au maximum 100 MP.

Algorithme 8 : Gestion des transferts mémoire vers et depuis le GPU.

1. allocation et affectation en mémoire CPU **h_img_in**;
2. allocation de mémoire CPU non-paginée **h_img_out**;
3. allocation de mémoire globale GPU **d_img_out**;
4. allocation de mémoire texture GPU **tex_img_in**;
5. copie image de **h_img_in** vers **tex_img_in**;
6. kernel<<< gridDim,blockDim>>> `/* sortie dans d_img_out */;`
7. copie image de sortie de **d_img_out** vers **h_img_out**;

Ces choix concernant les types de mémoire employés sont un facteur déterminant de la performance globale de l'implémentation. Cela sera confirmé par les mesures présentées à la fin de ce chapitre, mais une première expérience permet de s'en convaincre : le kernel médian 3×3 d'Arrayfire, aimablement mis à disposition par l'un des développeurs, voit son débit global pratiquement doublé lorsqu'on remplace ses accès mémoire par la combinaison texture/non-paginée que l'on vient de présenter.

Le tableau 8.1 donne le détail des temps de transfert pour quelques tailles usuelles d'images en niveaux de gris, codés en 8 ou 16 bits, et compare les temps globaux avec ceux mesurés lorsque l'on utilise uniquement la mémoire globale. L'impact du choix de la configuration mémoire est évident, avec des gains constatés de 15% à 75%.

Dimension (pixels)	Profondeur (bits)	CPU → GPU (ms)	GPU → CPU (ms)	**Total** (ms)	**Mém. globale** (ms)
512×512	8	0.08	0.06	**0.14**	0.23
	16	0.14	0.10	**0.24**	0.42
1024×1024	8	0.24	0.19	**0.43**	0.81
	16	0.45	0.35	**0.80**	1.23
2048×2048	8	0.85	0.68	**1.53**	2.15
	16	1.59	1.32	**2.91**	3.83
4096×4096	8	3.27	2.61	**5.88**	7.10
	16	6.21	5.21	**11.42**	13.16

TABLE 8.1 – Temps de transfert vers et depuis le GPU, en fonction de la dimension de l'image et de la profondeur des niveaux de gris. La colonne "Mémoire globale" donne les temps mesurés lorsque cette seule mémoire est employée.

8.3/ UTILISATION DES REGISTRES

En traitement d'image, les filtres médians sont beaucoup employés avec des tailles de fenêtres modestes comme pré-traitement, éventuellement itératif, ou bien avec de grandes

tailles de fenêtres pour de l'estimation d'intensité d'arrière plan. Les taille intermédiaires, de l'ordre de quelques dizaines de pixels, ne sont à notre connaissance pas employées.

Un filtre médian de petite taille ne réalise que peu d'opérations, sans complexité de surcroît, et doit donc atteindre des niveaux de performances élevés. Le cadre général des traitements sur GPU présenté au paragraphe 4.2.3 n'est alors plus pertinent, pour deux raisons :

1. La phase de pré-chargement des données nécessaires en mémoire partagée prend du temps et la lecture se faisant depuis la mémoire texture, elle est soumise aux latences qui lui sont attachées.
2. l'utilisation en lecture/écriture des données en mémoire partagée, outre le fait qu'elle puisse être contraignante en terme de motifs d'accès, n'atteint pas les débits permis par les registres individuels à la disposition des threads.

Il est ainsi clair que la chaîne de traitement la plus performante consiste à ne faire qu'une lecture en texture par pixel puis d'effectuer les calculs en registres. Les limites de ce schéma général sont le nombre de registres disponibles, par thread et par bloc. Si on dépasse ces limites, le compilateur déporte les variables en mémoire locale très peu performante. Sans aller au delà de la limite, l'utilisation de trop nombreux registres va mécaniquement limiter le nombre de threads effectivement exécutés en parallèle par le GPU et bien souvent grever la performance du kernel. Un compromis est donc à définir entre la recherche de vitesse par l'emploi des registres et le ralentissement que provoque l'usage d'un trop grand nombre de ces derniers.

Prenons l'exemple d'un kernel qui ferait usage d'un total de 20 registres par thread et que l'on exécuterait par blocs de 128 threads. La limite des 63 registres par thread n'est évidemment pas atteinte, ni celle des 32K par bloc avec seulement $128 \times 20 = 2560$ registres par bloc de 128 threads. Dans ce cas, le GPU pourra exécuter en parallèle $32K/2560 = 12$ blocs, soit 1536 threads, ce qui représente le maximum possible et permet d'envisager un bon niveau de performance.

En revanche, si le même kernel utilise maintenant 24 registres, le GPU ne pourra plus exécuter en parallèle que 1280 threads sur les 1536 techniquement possibles. Une perte de performance est alors à craindre.

De ce point de vue, l'architecture Fermi, et en particulier le modèle C2070, ne proposant que 63 registres par thread, représente une régression par rapport à la génération précédente avec par exemple le GPU C1060 et ses 128 registres par thread.

8.3.1/ La sélection de la valeur médiane

Dans le cas des filtres médians à petite fenêtre, on peut envisager d'attribuer un registre par valeur à trier. Dans ce cas, un médian 3×3 emploiera 9 registres par thread, et cette méthode pourra théoriquement s'appliquer jusqu'au médian 7×7 sur C2070 et 11×11 sur C1060. Comme la recherche de performance impose de rationaliser l'utilisation des registres, nous nous sommes orientés vers l'algorithme dit *forgetful selection* (sélection par oubli) qui évite d'avoir recours à cette cardinalité de « un registre pour un pixel » de la fenêtre ([69]).

Cette méthode de « sélection par oubli » est illustrée en figure 8.1 par l'exemple de la sélection de la médiane parmi 9 valeurs. Plus généralement, il s'agit de

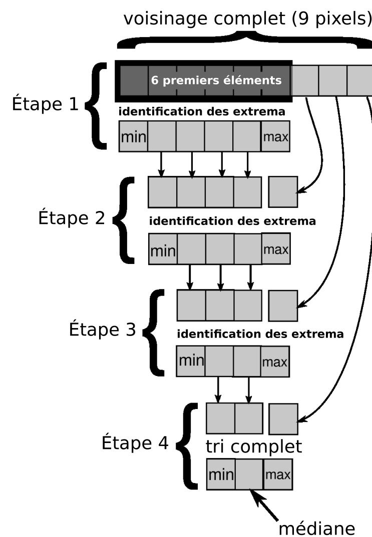


FIGURE 8.1 – Application de la sélection de médiane par oubli à une fenêtre de 3×3 pixels.

1. former une liste initiale de R_n valeurs prises parmi les $n = k \times k$ valeurs de la fenêtre du filtre,
2. identifier, puis éliminer de la liste la plus petite et la plus grande valeur,
3. insérer dans la liste une nouvelle valeur parmi celles non encore intégrées,
4. reprendre au point 2, et ce jusqu'à ce qu'il ne reste plus de valeur non utilisée. La médiane est alors la valeur restant dans la liste.

Cet algorithme nécessite un nombre constant d'étapes, égal à $n - \lceil \frac{n}{2} \rceil$, ce qui devrait assurer une charge équivalente pour tous les threads. Cependant, il existe un léger déséquilibre dû au nombre d'opérations requis par l'identification des *extrema* qui dépend des valeurs dans chaque liste. Cette variabilité, n'implique pas de branches d'exécution divergentes, et n'induit pas de perte de performances.

Nous avons par ailleurs choisi de fixer le nombre R_n de valeurs figurant initialement dans la liste, comme le plus petit nombre permettant de réaliser la sélection de la médiane. On obtient cette valeur limite en considérant qu'à chaque phase d'élimination des extrema, il faut garantir que la médiane globale n'est pas éliminée. Or, la définition de la médiane indique que dans la liste triée complète, on trouve autant de valeurs dont l'indice est supérieur à celui de la médiane que de valeurs dont l'indice lui est inférieur. Sachant que les fenêtres des filtres comportent toujours un nombre impair de valeurs, la condition suffisante pour garantir la sélection est donc que le nombre de valeurs non-intégrées dans la liste initiale soit inférieur au nombre de valeurs d'indice supérieur (ou inférieur) à la médiane dans la liste complète triée, soit

$$R_n = \lceil \frac{n}{2} \rceil + 1$$

Cette valeur de R_n représente donc aussi le nombre minimum de registres nécessaires à la sélection par oubli, ce qui permet de reculer la limite de taille admissible pour le filtre médian avec 9×9 pour le GPU C2070.

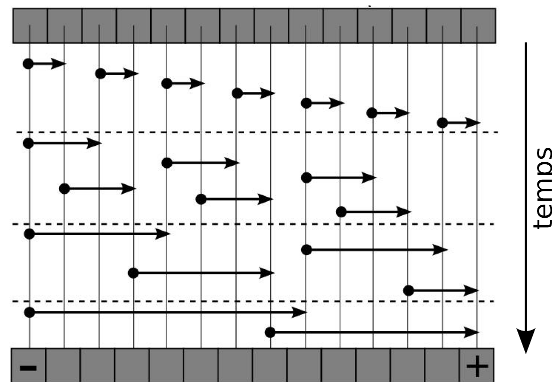


FIGURE 8.2 – Première étape d'identification des extrema pour un filtre 5×5, avec maximisation de l'ILP (Instruction Level Parallelism) pour l'identification des extrema.

8.3.2/ MASQUAGE DES LATENCES

Les lectures en texture ainsi que les écritures en mémoire globale sont soumises à des latences que nous avons déjà détaillées au chapitre 2. La mémoire texture bénéficie d'un cache permettant d'optimiser les lectures dans un voisinage à deux dimensions. Cela permet de réduire nettement les latences apparentes lors de l'accès aux éléments de la fenêtre du filtre. L'algorithme que nous proposons ne requiert qu'une lecture par élément de la fenêtre, dont la taille est assez petite pour que tous les éléments soient mis en cache. Aucune latence superflue n'est donc générée à la lecture.

Un autre moyen de réduire la latence moyenne constatée d'une séquence d'instructions est d'augmenter le niveau d'ILP (Instruction Level Parallelism ou parallélisme d'instructions). On cherche pour cela à réduire autant que possible la dépendance entre instructions successives au sein d'un kernel, de sorte à ne pas forcer les pipelines d'instructions des SMs à se vider. Nous avons appliqué ce principe à la phase d'identification des extrema de la liste en arrangeant les instructions élémentaires de permutation de sorte à éloigner au maximum les instructions inter-dépendantes. L'exemple de la figure 8.2 montre la séquence des permutations conditionnelles permettant l'identification des extrema lors de la première étape de sélection d'un filtre médian 5×5. On retrouve les $R_n = 14$ éléments de la liste initiale en haut de la figure, et la même liste au bas avec la valeur minimale à gauche et la valeur maximale à droite. Les séquences d'instructions indépendantes étant séparées par les lignes pointillées horizontales.

Enfin, il est également possible de réduire la latence moyenne d'accès à la mémoire globale en faisant en sorte que chaque thread produise, non pas la valeur de sortie d'un seul pixel, mais de plusieurs, et ce par autant d'écritures immédiatement consécutives, seule la première de la série générant une latence. Pour que l'application de ce principe produise l'effet attendu, il faut tout de même garantir la contiguïté des accès par demi warp, ce qui est le cas ici si les valeurs multiples issues par chaque thread se trouvent également à des adresses consécutives en mémoire globale.

Nous faisons l'hypothèse que chaque thread traite deux pixels voisins et cela impose de gérer la superposition partielle des fenêtres du filtre. La méthode de sélection que nous avons choisie nous interdit en effet d'employer les techniques habituelles, comme la mise à jour incrémentale de l'histogramme des niveaux de gris. Cependant, une partie des traitements est commune aux 2 processus de sélection. En effet, les fenêtres associées

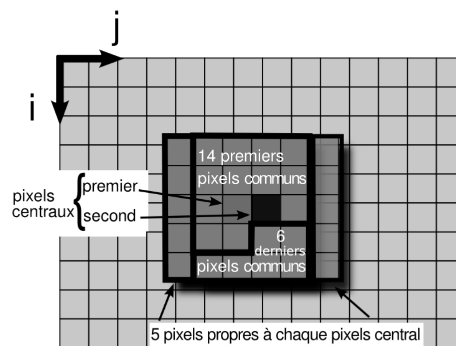


FIGURE 8.3 – Gestion des éléments communs aux fenêtres de deux pixels centraux voisins dans un filtre médian 5×5. La liste initiale comprend les 14 premiers éléments communs, puis les 7 premières étapes de sélection sont conduites en commun avant que les 5 dernières le soient en parallèle, mais de manière disjointe.

aux deux pixels partagent un certain nombre de données, égal à $S_n = n - k = n - \sqrt{n}$. Or, pour $n \geq 9$:

$$n - \sqrt{n} \geq \lceil \frac{n}{2} \rceil + 1$$

Nous pouvons donc initialiser la liste de sélection avec R_n valeurs choisies parmi les S_n valeurs communes en étant sur que le processus de sélection n'écartera pas les deux médianes. Ensuite, on mène les $(S_n - R_n + 1)$ premières étapes de sélection permettant d'intégrer progressivement l'ensemble des valeurs communes à la liste. Il ne reste alors que les k éléments propres à chaque fenêtre à intégrer dans deux séquences de sélection rendues distinctes mais réalisées en alternance par le même thread. Cette répartition des éléments pour un filtre médian 5×5 est représentée à la figure 8.3. Comme on le voit sur la figure, les R_n premières valeurs sont simplement prises au début de la liste des valeurs communes, sans que cela ne génère des défauts de cache pour les fenêtres de taille supérieure (le cache de texture 2D contient environ 5Ko).

Chaque thread utilise plus de registres $(R_n + 2k)$ pour traiter deux pixels que pour un seul (R_n), mais cela ne modifie pas les capacités théoriques de traitement, avec toujours une taille maximale de 9×9 sur C2070. Il suffit alors de réduire le nombre de threads par bloc pour retrouver le niveau de parallélisme souhaité. Mais globalement, traiter deux pixels par threads permet d'utiliser $k + 1$ registres de moins par paire de pixels par rapport au traitement par threads distincts, ce qui représente un gain de parallélisme au niveau de chaque bloc.

Nous avons fait jusqu'ici l'hypothèse d'un traitement de deux pixels par thread. Le passage à quatre pixels par thread implique $n - 3k$ éléments communs aux quatre fenêtres, ce qui est inférieur à R_n pour les tailles 3×3 et 5×5, et ne permet que deux étapes de sélections communes pour le 7×7. cela ne compense pas le coût des copies nécessaires au dédoublement. Le cas du 9×9 pourrait sembler plus pertinent, mais là encore, les sélections communes aux quatre pixels ne sont pas suffisamment nombreuses pour compenser les coûts inhérents aux sélections disjointes. La solution deux pixels par thread s'avère la plus performante.

L'ensemble des choix que nous venons de décrire et qui ont présidé à l'élaboration de notre filtre médian GPU conduisent à adopter un style de codage assez inhabituel, du fait de l'usage intensif des registres dont une des caractéristiques est de ne pas être

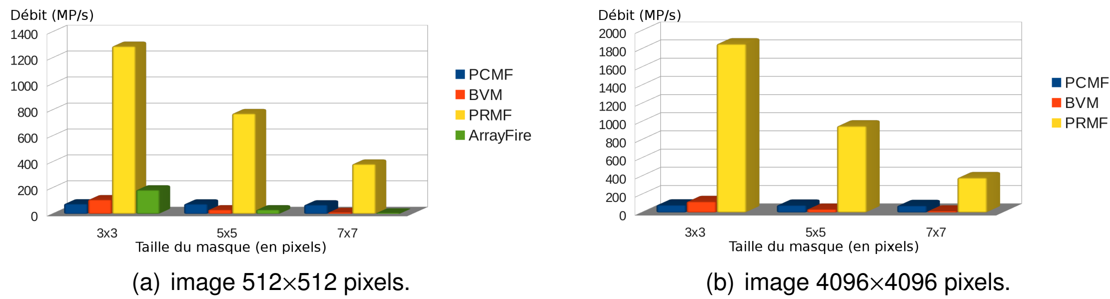
indexables. Le code du filtre médian 3×3 est reproduit au listing 8.1.

Listing 8.1 – Kernel réalisant un filtre médian 3×3 en registres.

```
__device__ inline void s(unsigned char * a, unsigned char * b)
{
  unsigned char tmp ;
  if (*a > *b)
    {
      tmp = *b ;
      *b = *a ;
      *a = tmp ;
      }
}

#define minmax3(a, b, c)          s(a, b); s(a, c); s(b,c)
#define minmax4(a, b, c, d)       s(a, b); s(c, d); s(a, c); s(b, d);
#define minmax5(a, b, c, d, e)     s(a, b); s(c, d); s(a, c); s(b, d); s(b,d); s(a,e); s(
    d,e)
#define minmax6(a, b, c, d, e, f) s(a, b); s(c, d); s(e, f); s(a, c); s(d, f); s(a, d); s
    (c ,f)

__global__ void kernel_median3_2pix( unsigned short*output, int i_dim, int j_dim)
{
  // coordonnées absolues du point
  int j = __mul24(__mul24(blockIdx.x,blockDim.x) + threadIdx.x,2) ;
  int i = __mul24(blockIdx.y,blockDim.y) + threadIdx.y ;

  int a0, a1, a2, a3, a4, a5 ;    // les 2 listes de sélection
  int b0, b1, b2, b3, b4, b5 ;    //

  a0 = tex2D(tex_img_ins, j  , i-1) ;    //
  a1 = tex2D(tex_img_ins, j+1, i-1) ;    //
  a2 = tex2D(tex_img_ins, j  , i  ) ;    // les 6 premiers éléments
  a3 = tex2D(tex_img_ins, j+1, i  ) ;    //
  a4 = tex2D(tex_img_ins, j  , i+1) ;    //
  a5 = tex2D(tex_img_ins, j+1, i+1) ;    //

  minmax6(&a0, &a1, &a2, &a3, &a4, &a5) ; // identification des extrema

  b0=a0; b1=a1; b2=a2; b3=a3; b4=a4; b5=a5; // séparation des listes

  a5 = tex2D(tex_img_ins, j-1, i  ) ;   // ajout éléments suivants
  b5 = tex2D(tex_img_ins, j+2, i  ) ;   //

  minmax5(&a1, &a2, &a3, &a4, &a5) ;    // identification des extrema
  minmax5(&b1, &b2, &b3, &b4, &b5) ;    //

  a5 = tex2D(tex_img_ins, j-1, i-1) ;
  b5 = tex2D(tex_img_ins, j+2, i-1) ;
  minmax4(&a2, &a3, &a4, &a5) ;
  minmax4(&b2, &b3, &b4, &b5) ;
  a5 = tex2D(tex_img_ins, j-1, i+1) ;
  b5 = tex2D(tex_img_ins, j+2, i+1) ;
  minmax3(&a3, &a4, &a5) ;
  minmax3(&b3, &b4, &b5) ;

  output[ __mul24(i, j_dim) +j  ] = a4 ;   // écriture des 2 valeurs
  output[ __mul24(i, j_dim) +j+1 ] = b4 ; // en mémoire globale

}
```

8.4/ RÉSULTATS

Les valeurs présentées dans les tableaux 8.2, 8.3 et la figure 8.4 sont obtenues par moyennage du chronométrage de 1000 exécutions du même kernel, développé en va-



(a) image 512×512 pixels.

(b) image 4096×4096 pixels.

FIGURE 8.4 – Comparaison des débits (MP/s) atteints par notre implémentation notée PRMF, avec les principales solutions de référence. De gauche à droite : PCMF, BVM, PRMF, ArrayFire (impossible en 4096×4096)

riantes 8 et 16 bits de profondeurs de niveau de gris.

L'analyse que nous pouvons tirer du tableau 8.2 est la pertinence des choix relatifs aux transferts de données, qui représentent entre 13% et 82% du temps total d'exécution des configurations testées.

Taille d'image	**Profondeur**	**3×3**	**5×5**	**7×7**
512×512	8 bits	73%	44%	20%
	16 bits	82%	57%	29%
1024×1024	8 bits	68%	37%	15%
	16 bits	80%	53%	25%
2048×2048	8 bits	66%	34%	14%
	16 bits	79%	59%	23%
4096×4096	8 bits	65%	33%	13%
	16 bits	78%	50%	23%

TABLE 8.2 – Pourcentage du temps d'exécution pris par les transferts de données en fonction de la taille de fenêtre du filtre, pour les profondeurs 8 et 16 bits sur GPU C2070.

Les valeurs du tableau 8.3 détaillent les débits de pixels réalisés par les différents kernels. Ils prennent en compte le temps d'exécution ainsi que les temps de transfert. Par ailleurs, afin d'évaluer le niveau de performance absolue de notre méthode, nous avons également mesuré le débit maximum effectif permis par le couple GPU/CPU, ce qui nous permet d'évaluer la pertinence d'éventuelles recherches postérieures visant à améliorer encore les débits.

La valeur de ce débit maximum est obtenue en exécutant un kernel « identité » qui n'effectue aucune opération mais se contente de faire les lectures et écritures en mémoire. Les débits ainsi mesurés sont regroupés dans le tableau 8.4 où l'on constate en particulier que plus l'image est de grande dimension, plus on peut espérer un débit élevé. On vérifie aussi notre intuition initiale avec des valeurs d'environ 2000 MP/s, à comparer aux

moins de 200 MP/s permis par les implémentations de référence.

Taille d'image	t : temps kernel T_x débit en prof. x	3×3	5×5	7×7
512×512	t (ms)	0.05	0.19	0.60
	T_8 (Mpix/s)	1291	773	348
	T_{16} (Mpix/s)	865	607	307
1024×1024	t (ms)	0.20	0.74	2.39
	T_8 (Mpix/s)	1644	889	371
	T_{16} (Mpix/s)	1045	692	329
2048×2048	t (ms)	0.79	2.95	9.53
	T_8 (Mpix/s)	1805	936	379
	T_{16} (Mpix/s)	1130	729	338
4096×4096	t (ms)	3.17	11.77	38.06
	T_8 (Mpix/s)	1854	951	382
	T_{16} (Mpix/s)	1151	738	340

TABLE 8.3 – Performances des filtres médians rapides en fonction des tailles d'image et de fenêtre du filtre, en variantes 8 et 16 bits de profondeursur GPU C2070.

Taille d'image	T_8	T_{16}
512×512	1598	975
1024×1024	2101	1200
2048×2048	2359	1308
4096×4096	2444	1335

TABLE 8.4 – Débits maximum effectifs T_8 and T_{16} (en MP/s), respectivement pour les variantes 8 et 16 bits sur C2070.

Les performances des kernels en variante 16 bits ne différent pas de celles en variante 8 bits, seuls les temps de transfert des données sont à l'origine des variations du débit global.

Enfin, considérant que l'algorithme implémenté dans les kernels d'Arrayfire est proche du notre, nous avons mesuré les débits d'une version modifiée de leur kernel appliquant nos techniques de gestion mémoire, avec pour résultat un filtre médian 3×3 capable de traiter 670 MP/s, soit 3,7 fois plus que la version commerciale. L'écart restant (×2,7) étant à mettre au crédit de notre implémentation du kernel.

8.5/ CONCLUSION

L'implémentation GPU du filtre médian que nous avons décrite permet de traiter jusqu'à 1854 millions de pixels à la seconde, soit aussi 900 images haute définition (1080p), et surclasse les solutions jusqu'alors proposées dans la littérature, dont la plus performante ne débitait que 180 millions de pixels par seconde (voir [84]). Elle a fait l'objet d'un article dans la revue *Journal of Signal Processing Systems* (voir [75]). L'important gain de

vitesse est la conséquence de l'attention toute particulière que nous avons apportée à la gestion de la mémoire, tant du côté GPU que CPU, et qui nous a conduit à concevoir des kernels utilisant exclusivement des registres pour effectuer les opérations de sélection.

Le débit de pixels constaté approche cette fois le débit maximal effectif de la plateforme (2444 MP/s), ce qui limite le gain que pourraient apporter des implémentations futures. Toutefois, notre algorithme appartient à la classe des solutions de filtrage médian par tri (incomplet) des valeurs et à ce titre, ses temps d'exécution sont fortement dépendants de la taille de la fenêtre de filtrage, comme le montrent les diagrammes de la figure 8.4. Il n'est donc pertinent que pour les petites tailles de fenêtre, qui sont aussi les plus communément employées en traitement d'image.

Dans les grandes tailles de fenêtre, la plupart des solutions adoptent des méthodes approchées de détermination de la médiane. Les principes que nos avons appliqués peuvent alors apporter des gains de performance comme nous l'avons montré dans [74], ainsi qu'à beaucoup d'autres méthodes de calcul comme les filtres de convolution abordés dans le chapitre suivant.

Enfin nous renvoyons le lecteur à la conclusion du prochain chapitre qui présentera l'outil en ligne que nous proposons et qui permet de générer les codes des kernels médians et de convolution.

9

LES FILTRES DE CONVOLUTION SUR GPU

9.1/ INTRODUCTION

Après avoir conçu des filtres médians aux performances élevées, nous avons cherché à en appliquer les principes à d'autres types d'algorithmes de filtrage. Les filtres de convolution, par la diversité des traitements qu'ils permettent de réaliser et leur universalité, nous ont semblé être un objectif particulièrement intéressant.

Le principe et la formulation de la convolution sont présentés au chapitre 4.1.1 aussi nous attacherons nous uniquement dans les paragraphes qui suivent à détailler les solutions et expérimentations permettant de concevoir des filtres rapides sur GPU.

Nous faisons l'hypothèse que les fonctions de convolution sont à support carré, de taille de côtés impaire, permettant ainsi de considérer un *pixel central*. Cette hypothèse ne constitue pas une restriction en termes de traitement car tout support non carré peut être étendu à un support carré même si, dans ce cas de figure, l'exécution impliquera plus d'opérations que nécessaire et ne sera ainsi plus optimale.

L'étude la plus complète et qui montre les performances les plus élevées émane du constructeur Nvidia lui-même dans [90]. Nous l'avons présentée au paragraphe 4.2 et nous rappellerons simplement ici qu'elle a utilisé des modèles à architecture GT200 (GTX280) et choisi comme traitement de référence une convolution non-séparable de masque 5×5 sur une image en profondeur 8 bits de 2048×2048 pixels. Leur implémentation la plus rapide effectue cette opération en 1,4 ms et permet un débit global (incluant les temps de transfert des données) de 945 millions de pixels à la seconde (MP/s). Elle servira de référence pour nos expérimentations.

9.2/ IMPLÉMENTATION GÉNÉRIQUE DE LA CONVOLUTION NON SÉPARABLE SUR GPU

L'implémentation GPU de la convolution non-séparable d'une fonction image I par une fonction masque h définie sur un support Ω peut-être décrite comme dans l'algorithme 9. Pour le cas où la somme S_h des valeurs du masque est différente de 1, l'image résultante I' est obtenue après une normalisation nécessaire pour ne pas modifier l'intensité moyenne de l'image. Par exemple, pour une profondeur de 8 bits : Si $S_h > 0$ alors $I' = I_\Omega / S_h$

Selon la valeur de la somme S_h, il peut être nécessaire de « recaler » globalement les niveaux de gris de l'image. Ainsi :

1. Si $S_h = 0$ alors $I' = I_\Omega + 128$
2. Si $S_h < 0$ alors $I' = I_\Omega + 255$

Algorithme 9 : Convolution générique sur GPU

1 **foreach** *pixel* (x, y) **do** /* en parallèle */
2   Lire les niveaux de gris $I(x, y)$ des voisins sur Ω ;
3   Calculer la somme $I_\Omega(x, y) = \sum_{(j,i)\in\Omega} I(x - j, y - j).h(j, i)$;
4   Normaliser $I_\Omega(x, y)$ pour obtenir $I'(x, y)$;
5   Mémoriser $I'(x, y)$;
6 **end**

Il est tout à fait possible d'envisager ici l'application brute des principes mis en œuvre pour les filtres médians. Cela conduit au code du listing 9.1 où chaque coefficient du masque (moyenneur 3×3) est fixé et mémorisé dans un registre, le calcul de la somme s'effectuant également dans un registre. Pour éviter des opérations coûteuses comme la division, on remarque que la normalisation est évitée et pré-effectuée au niveau des coefficients du masque dont la somme est ainsi toujours égale à 1. Par ailleurs, pour des raisons de lisibilité de ce premier code, chaque thread ne traite ici qu'un seul pixel.

Listing 9.1 – Kernel réalisant la convolution par un masque moyenneur 3×3 dont les coefficients normalisés sont codés *en dur*, dans les registres du GPU.

```
__global__ void kernel_convoGene3Reg8( unsigned char *output, int j_dim)
{
  float outval0=0.0 ;
  float n0,n1,n2,n3,n4,n5,n6,n7,n8 ;
  // coefficients du masque
  n0 = (1.0/9) ;
  n1 = (1.0/9) ;
  n2 = (1.0/9) ;
  n3 = (1.0/9) ;
  n4 = (1.0/9) ;
  n5 = (1.0/9) ;
  n6 = (1.0/9) ;
  n7 = (1.0/9) ;
  n8 = (1.0/9) ;

  // coordonnées absolues du pixel central
  int j = __mul24(blockIdx.x, blockDim.x) + threadIdx.x ;
  int i = __mul24(blockIdx.y, blockDim.y) + threadIdx.y ;
  // somme
  outval0 = n8*tex2D(tex_img_inc, j-1, i-1 )
          + n7*tex2D(tex_img_inc, j  , i-1 )
          + n6*tex2D(tex_img_inc, j+1, i-1 )
          + n5*tex2D(tex_img_inc, j-1, i   )
          + n4*tex2D(tex_img_inc, j  , i   )
          + n3*tex2D(tex_img_inc, j+1, i   )
          + n2*tex2D(tex_img_inc, j-1, i+1 )
          + n1*tex2D(tex_img_inc, j  , i+1 )
          + n0*tex2D(tex_img_inc, j+1, i+1 ) ;

  output[ __mul24(i, j_dim) + j ] = (unsigned char) outval0 ;
}
```

Les performances de cette implémentation directe ont été regroupées dans les tableaux 9.1 et 9.2 où l'on peut immédiatement constater que la solution optimale Nvidia demeure

Masque		Taille d'image 512 × 512	1024 × 1024	2048 × 2048	4096 × 4096
3×3	temps exéc. (ms)	0.077	0.297	1.178	4.700
	débit global (MP/s)	1165	1432	1549	1585
5×5	temps exéc. (ms)	0.209	0.820	**3.265**	13.050
	débit global (MP/s)	559	836	**875**	533
7×7	temps exéc. (ms)	0.407	1.603	6.398	25.560
	débit global (MP/s)	472	515	529	533

TABLE 9.1 – Performances des kernels effectuant la convolution non-séparable sur le modèle du listing 9.1, sur GPU C2070. Le temps d'exécution correspond à la seule exécution du kernel. Le débit global intègre les temps de transfert. Les valeurs en gras correspondent au traitement de référence.

Masque		Taille d'image 512 × 512	1024 × 1024	2048 × 2048	4096 × 4096
3×3	temps exéc. (ms)	0.060	0.209	0.801	3.171
	débit global (MP/s)	1186	1407	1092	1075
5×5	temps exéc. (ms)	0.148	0.556	**2.189**	8.7200
	débit global (MP/s)	848	960	**802**	793
7×7	temps exéc. (ms)	0.280	1.080	4.278	17.076
	débit global (MP/s)	594	649	573	569

TABLE 9.2 – Performances des kernels effectuant la convolution non-séparable sur le modèle du listing 9.1, sur GPU GTX280. Le temps d'exécution correspond à la seule exécution du kernel. Le débit global intègre les temps de transfert. Les valeurs en gras correspondent au traitement de référence.

plus rapide. L'analyse plus détaillée nous apprend aussi que le modèle GTX280 exécute le kernel plus vite que le récent C2070, en raison d'un plus grand nombre de registres disponibles. Malgré tout, lorsqu'on prend en compte les temps de transfert des données, l'avantage va au C2070 qui réalise ce traitement à 875 MP/s.

9.3/ IMPLÉMENTATION OPTIMISÉE DE LA CONVOLUTION NON SÉPARABLE SUR GPU

Les coefficients du masque de convolution sont indépendants et il est donc impossible, sauf cas particulier, de réduire le nombre de registres nécessaires. Pour cette même raison, multiplier le nombre de pixels traités par chaque thread, ne permet pas d'économiser des registres au niveau bloc comme il a été possible de la faire pour les médians.

De surcroît, autant il était envisageable de concevoir un kernel par taille de masque lorsqu'il s'agissait de filtres médians car ils ne comportent qu'un seul paramètre, autant cela devient inconcevable pour les filtres de convolution et l'immense variété de paramétrage qu'ils recouvrent. La contrainte de définir les valeurs des coefficients du masque de ma-

nière littérale à l'intérieur du kernel doit donc être levée pour permettre de rendre toute leur souplesse aux opérations de convolution.

Parmi les types de mémoires disponibles, nous avons opté pour le stockage des coefficients du masque en mémoire constante (*symbol memory*) en raison de ses performances et du petit volume des données. L'abandon des registres permet aussi d'adopter un style de codage beaucoup plus conventionnel utilisant des structures de contrôle classiques (itérations et tableaux).

L'augmentation du nombre de pixels traités par chaque thread est alors de nouveau envisageable, puisque l'utilisation de la mémoire constante pour les coefficients libère autant de registres. Il faut cependant organiser les calculs de manière à réduire autant que possible les accès aux valeurs des pixels de l'image ; en d'autres termes, on cherche à exploiter le recouvrement entre positions voisines du masque de convolution et n'effectuer qu'une seule lecture par pixel de l'image pour en distribuer la valeur sur l'ensemble des calculs de convolutions en cours dans le thread. Cela complique quelque peu les expressions des sommes partielles mais réalise l'objectif opérationnel de la mémoire partagée sans en subir le coût ni les contraintes d'accès, et permet ainsi d'envisager de meilleures performances.

Multiplier les pixels traités par un même thread impose également de faire un choix sur la forme de ce que l'on appellera un *paquet* de pixels (centraux, par opposition aux pixels des voisinages, même si un pixel a successivement l'un et l'autre des statuts). La contrainte de contiguïté des accès en mémoire globale pour la mémorisation des valeurs de sortie fait que seule l'organisation « en ligne » des paquets de pixels est bénéfique, bien que n'étant pas celle qui présente systématiquement les recouvrements les plus importants. Par exemple, un paquet carré ou rectangulaire présente plus de recouvrements qu'un paquet « en ligne ».

1	2	3	3	3	3	3	3	2	1
1	2	3	3	3	3	3	3	2	1
1	2	3	3	3	3	3	3	2	1

(x,y) (x+7,y)

(a) Cas d'un masque de taille 3×3 ($k = 1$) où l'on dénombre 6 colonnes centrales, soit 18 pixels de multiplicité maximale 3.

1	2	3	4	5	5	5	5	4	3	2	1
1	2	3	4	5	5	5	5	4	3	2	1
1	2	3	4	5	5	5	5	4	3	2	1
1	2	3	4	5	5	5	5	4	3	2	1
1	2	3	4	5	5	5	5	4	3	2	1

(x,y) (x+7,y)

(b) Cas d'un masque de taille 5×5 ($k = 2$) où l'on dénombre 4 colonnes centrales, soit 20 pixels de multiplicité maximale 5.

FIGURE 9.1 – Multiplicité des implications des pixels de la zone d'intérêt d'un thread dans les calculs de convolution. Le nombre de calculs dans lequel est impliqué un pixel est inscrit en son centre. Le premier pixel du paquet, ou pixel de base, est repéré par ses coordonnées (x, y) ; le dernier a pour coordonnées $(x + 7, y)$

La valeur de 8 pixels comme taille de paquet, déterminée expérimentalement, s'est avérée optimale sur les deux types d'architecture GPU et pour toutes les tailles de masques soumis aux mesures. Cela signifie que chaque thread conduit simultanément les calculs de convolution attachés à chacun des 8 pixels du paquet qu'il traite. La somme partielle de chaque convolution est mémorisée dans un registre. Sur cette base, on a schématisé

Masque		Taille d'image 512×512	1024×1024	2048×2048	4096×4096
3×3	temps exéc. (ms)	0.036	0.128	0.495	1.964
	débit global (MP/s)	1425	1862	2071	2138
5×5	temps exéc. (ms)	0.069	0.253	**0.987**	3.926
	débit global (MP/s)	1208	1524	**1666**	1711
7×7	temps exéc. (ms)	0.110	0.413	1.615	6.416
	débit global (MP/s)	1016	1237	1334	1364

TABLE 9.3 – Performances des kernels effectuant la convolution non-séparable sur le modèle du listing 9.2, sur GPU C2070. Le temps d'exécution correspond à la seule exécution du kernel. Le débit global intègre les temps de transfert. Les valeurs en gras correspondent au traitement de référence.

à la figure 9.1, l'implication de chaque pixel de la zone d'intérêt d'un thread, découlant du recouvrement des 8 positions du masque. Pour chaque pixel, cette implication est figurée par une valeur de *multiplicité* représentant le nombre de convolutions différentes dans lesquelles il est impliqué au sein d'un même thread. Tous les pixels d'une colonne partagent la même multiplicité et chaque pixel étant au moins impliqué dans l'un des 8 calculs, les valeurs de cette multiplicité varient de 1 à k, si k est le *rayon* du masque tel que $n = 2k + 1$.

On peut dénombrer globalement les multiplicités comme suit :
- les $(8 - 2k)$ colonnes centrales de la zone d'intéret, soient $(8 - 2k)(2k + 1)$ pixels sont impliqués dans k calculs.
- les paires de colonnes symétriques par rapport au bloc des colonnes centrales précédentes ont leurs $2(2k + 1)$ pixels impliqués dans $(k - 1 - e)$ calculs, si e représente l'éloignement avec le bloc de colonnes centrales.
- Les deux colonnes extérieures ont ainsi leurs pixels impliqués chacun dans un seul calcul de convolution.

Le listing 9.2 présente, pour exemple, le code implémentant ces solutions pour les masques de taille 3×3. On remarque qu'il n'y a que 30 accès à la texture, au lieu des $9 \times 8 = 72$ sans optimisation, et que la sortie opère sur 8 pixels consécutifs en mémoire globale. On obtient ainsi une utilisation optimale de la mémoire. L'ensemble des mesures de performance associées, sur C2070, est regroupé dans le tableau 9.3. On observe que, grâce à une bande passante mémoire supérieure, les débits mesurés peuvent dépasser les 2100 MP/s, pour une convolution 3×3 sur une image de 4096×4096 pixels. Le traitement de référence quant à lui est effectué en 0.987 ms pour un débit de 1666 MP/s.

Sur GTX280, cette implémentation atteint également des débits supérieurs aux précédents, mais surtout, surpasse la solution Nvidia avec une exécution du traitement de référence en 1,21 ms, soit une accélération de plus de 14%. Le gain au niveau du débit reste modeste car les transferts représentent à eux seuls plus de 72% du temps total. Le modèle GTX280 traite ainsi 962 MP à la seconde, soit un gain de seulement 1.7% par rapport à la solution de référence.

Listing 9.2 – Kernel réalisant la convolution par un masque 3×3 dont les coefficients normalisés sont en mémoire constante.

```
__global__ void kernel_convoGene8x8pL3( unsigned char  *output , int j_dim )
```

```
{
  int ic, jc ;
  const int L=3 ;
  unsigned char pix ;
  // registres pour les 8 calculs
  float outval0=0.0, outval1=0.0, outval2=0.0, outval3=0.0 ;
  float outval4=0.0, outval5=0.0, outval6=0.0, outval7=0.0 ;

  // coordonnees absolues du point de base, le premier du paquet
  int j = ( __umul24( blockIdx.x, blockDim.x) + threadIdx.x)<< 3 ;
  int i = ( __umul24( blockIdx.y, blockDim.y) + threadIdx.y) ;

  for (ic=0 ; ic<L ; ic++) // pour chaque ligne de la zone d'intérêt
   {
     pix = tex2D(tex_img_inc, j+1, i-1+ic) ;      // les colonnes centrales. Multiplicité 3
     outval0 += mask[ __umul24(ic,L) +2 ]*pix ;
     outval1 += mask[ __umul24(ic,L) +1 ]*pix ;
     outval2 += mask[ __umul24(ic,L)    ]*pix ;
     pix = tex2D(tex_img_inc, j+2, i-1+ic) ;
     outval1 += mask[ __umul24(ic,L) +2 ]*pix ;
     outval2 += mask[ __umul24(ic,L) +1 ]*pix ;
     outval3 += mask[ __umul24(ic,L)    ]*pix ;
     pix = tex2D(tex_img_inc, j+3, i-1+ic) ;
     outval2 += mask[ __umul24(ic,L) +2 ]*pix ;
     outval3 += mask[ __umul24(ic,L) +1 ]*pix ;
     outval4 += mask[ __umul24(ic,L)    ]*pix ;
     pix = tex2D(tex_img_inc, j+4, i-1+ic) ;
     outval3 += mask[ __umul24(ic,L) +2 ]*pix ;
     outval4 += mask[ __umul24(ic,L) +1 ]*pix ;
     outval5 += mask[ __umul24(ic,L)    ]*pix ;
     pix = tex2D(tex_img_inc, j+5, i-1+ic) ;
     outval4 += mask[ __umul24(ic,L) +2 ]*pix ;
     outval5 += mask[ __umul24(ic,L) +1 ]*pix ;
     outval6 += mask[ __umul24(ic,L)    ]*pix ;
     pix = tex2D(tex_img_inc, j+6, i-1+ic) ;
     outval5 += mask[ __umul24(ic,L) +2 ]*pix ;
     outval6 += mask[ __umul24(ic,L) +1 ]*pix ;
     outval7 += mask[ __umul24(ic,L)    ]*pix ;

     pix = tex2D(tex_img_inc, j, i-1+ic) ;       // les colonnes extérieures
     outval0 += mask[ __umul24(ic,L) +1 ]*pix ; // multiplicité 2
     outval1 += mask[ __umul24(ic,L)    ]*pix ;
     pix = tex2D(tex_img_inc, j-1, i-1+ic) ;
     outval0 += mask[ __umul24(ic,L) ]*pix ;    // multiplicité 1

     pix = tex2D(tex_img_inc, j+7, i-1+ic) ;
     outval6 += mask[ __umul24(ic,L) +2 ]*pix ; // multiplicité 2
     outval7 += mask[ __umul24(ic,L) +1 ]*pix ;
     pix = tex2D(tex_img_inc, j+8, i-1+ic) ;
     outval7 += mask[ __umul24(ic,L) +2 ]*pix ; // multiplicité 1
   }

  output[ __umul24(i, j_dim) + j   ] = outval0 ; // les 8 sorties
  output[ __umul24(i, j_dim) + j+1 ] = outval1 ;
  output[ __umul24(i, j_dim) + j+2 ] = outval2 ;
  output[ __umul24(i, j_dim) + j+3 ] = outval3 ;
  output[ __umul24(i, j_dim) + j+4 ] = outval4 ;
  output[ __umul24(i, j_dim) + j+5 ] = outval5 ;
  output[ __umul24(i, j_dim) + j+6 ] = outval6 ;
  output[ __umul24(i, j_dim) + j+7 ] = outval7 ;
}
```

9.4/ CAS DE LA CONVOLUTION SÉPARABLE

Dans la pratique, les traitements appliqués aux images par des opérations de convolution à deux dimensions reposent souvent sur des masques présentant une ou plusieurs symétries. On rappelle que lorsqu'un tel masque h peut s'écrire comme le produit de 2 vecteurs h_v et h_h, comme dans l'exemple ci-dessous, alors on dit que la convolution 2D est séparable et peut donc être effectuée en deux opérations de convolution 1D de masques respectifs h_v et h_h.

$$h = h_v \times h_h = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} -1 & 2 & -1 \end{bmatrix} = \begin{bmatrix} -1 & 2 & -1 \\ -2 & 4 & -2 \\ -1 & 2 & -1 \end{bmatrix}$$

Une convolution séparable $n \times n$ est donc moins coûteuse en nombre d'opérations arithmétiques, avec seulement $2n$ paires addition/multiplication par pixel contre n^2 pour une convolution non séparable. Cela représente un gain de 60% du nombre d'opérations pour un masque 5×5 et nous laisse entrevoir des performances supérieures à celles de la convolution non séparable.

Il faut cependant considérer qu'effectuer un traitement en 2 kernels consécutifs implique de multiplier aussi les écritures en mémoire globale, ce qui a un coût. La plupart des implémentations séquentielles de la convolution séparable utilisent la même fonction pour réaliser les 2 passes horizontale et verticale, la première mémorisant la transposée de l'image de sortie pour qu'elle soit traitée directement par la seconde passe. Sur GPU, cette solution se heurte aux contraintes de contiguïté dans les accès à la mémoire globale, il faut donc préférer deux kernels distincts : un pour la convolution verticale, l'autre pour l'horizontale.

Dans une convolution 1D verticale de masque h_v, il n'y a pas de recouvrement entre les différentes positions du masque associées aux pixels d'un paquet tel que nous le définissons. Aucune optimisation de la distribution des données n'est donc possible de ce côté et il s'avère même que l'utilisation de la mémoire partagée est ici la solution la plus performante. La zone d'intérêt d'un bloc de threads ne s'étendant que vers le haut et le bas, on peut appliquer une version simplifiée du cadre général d'emploi de la mémoire partagée présenté au paragraphe 4.2.3. Remarquons à ce sujet que tous les threads ne « chargent » pas systématiquement le même nombre de pixels en mémoire partagée, ce qui implique que certains threads doivent attendre avant de pouvoir entamer le calcul principal. Ceci impose l'emploi d'une barrière de synchronisation entre ces deux phases. Le listing 9.3 détaille la mise en œuvre complète de ce kernel, pour des paquets de 8 pixels, qui demeure la taille optimale dans le cas séparable. Notons que ce type de kernel permet de travailler avec une taille de masque quelconque, passée en paramètre.

Listing 9.3 – Kernel réalisant la convolution verticale k×1 avec utilisation de la mémoire partagée.

```
__global__ void kernel_convoSepShx8pV(unsigned char *output, int j_dim, int r)
{
  int ic, jc, p;
  int k = 2*r+1 ;
  float outval0=0.0, outval1=0.0, outval2=0.0, outval3=0.0 ;
  float outval4=0.0, outval5=0.0, outval6=0.0, outval7=0.0 ;
  int bdimX = blockDim.x<<3 ; // nombre de pixels traités par une ligne d'un bloc
  int tidX = threadIdx.x<<3 ; // décalage paquet
```

```
   // coordonnées absolues du point de base
   int j = (__umul24(blockIdx.x,blockDim.x) + threadIdx.x)<<3 ;
   int i = __umul24( blockIdx.y, blockDim.y) + threadIdx.y ;
   // indice absolu dans l'image
   int idx = __umul24(i,j_dim) + j ;
   // adresse d'un ligne de la ROI en mémoire partagée
   int idrow = threadIdx.y*bdimX ;

   extern __shared__ unsigned char roi8p[];

   // haut de la ROI
   for (p=0; p<8; p++)
    roi8p[ idrow  + tidX +p ] = tex2D(tex_img_inc, j+p  , i-r) ;

   // bas de la ROI
   if ( threadIdx.y < k-1 )
    {
      idrow = (threadIdx.y+blockDim.y)*bdimX ;
      for (int p=0; p<8; p++)
       roi8p[ idrow + tidX +p  ] = tex2D( tex_img_inc, j+p  , i+blockDim.y-r ) ;
    }
   __syncthreads();

   // convolution verticale
   for (ic=0 ; ic<k ; ic++)
      {
       int baseRoi = __umul24(ic+threadIdx.y,bdimX) + tidX ;
       float valMask = maskv[ ic ] ;
       outval0 += valMask*roi8p[ baseRoi    ] ;
       outval1 += valMask*roi8p[ baseRoi +1 ] ;
       outval2 += valMask*roi8p[ baseRoi +2 ] ;
       outval3 += valMask*roi8p[ baseRoi +3 ] ;
       outval4 += valMask*roi8p[ baseRoi +4 ] ;
       outval5 += valMask*roi8p[ baseRoi +5 ] ;
       outval6 += valMask*roi8p[ baseRoi +6 ] ;
       outval7 += valMask*roi8p[ baseRoi +7 ] ;
      }

   // sortie des 8 pixels
   output[ idx++ ] = outval0 ;
   output[ idx++ ] = outval1 ;
   output[ idx++ ] = outval2 ;
   output[ idx++ ] = outval3 ;
   output[ idx++ ] = outval4 ;
   output[ idx++ ] = outval5 ;
   output[ idx++ ] = outval6 ;
   output[ idx   ] = outval7 ;
}
```

Listing 9.4 – Kernel réalisant la convolution horizontale 1×k avec utilisation de la mémoire partagée.

```
__global__ void kernel_convoSepShx8pH(unsigned char *output, int j_dim, int r)
{
   int ic, jc, p;
   int k = 2*r+1 ;
   float outval0=0.0, outval1=0.0, outval2=0.0, outval3=0.0 ;
   float outval4=0.0, outval5=0.0, outval6=0.0, outval7=0.0 ;
   int bdimX = blockDim.x<<3 ; // nombre de pixels traités par une ligne d'un bloc
   int tidX = threadIdx.x<<3 ; // décalage paquet

   // coordonnées absolues du point de base
   int j = (__umul24(blockIdx.x,blockDim.x) + threadIdx.x)<<3 ;
   int i = __umul24( blockIdx.y, blockDim.y) + threadIdx.y ;
   int j0= __umul24(blockIdx.x,blockDim.x)<<3 ;
   // indice absolu dans l'image
   int idx = __umul24(i,j_dim) + j ;

   // adresse d'une ligne de la ROI en mémoire partagée
```

```
  int idrow = threadIdx.y*(bdimX+k-1) ;

20  extern __shared__ unsigned char roi8p[];

  // gauche de la ROI
  for (p=0; p<8; p++)
   roi8p[  idrow  + tidX +p ] = tex2D(tex_img_inc, j-r+p  , i) ;
25  // droite de la ROI
  if ( threadIdx.x < r )
   {
     roi8p[  idrow + bdimX + threadIdx.x    ] = tex2D( tex_img_inc, j0-r
                          +bdimX+threadIdx.x  , i ) ;
30   roi8p[  idrow + bdimX + threadIdx.x +r ] = tex2D( tex_img_inc, j0
                               +bdimX+threadIdx.x  , i ) ;
   }

  __syncthreads();
35
  // convolution horizontale
  for (jc=0 ; jc<k ; jc++)
     {
      int baseRoi = idrow + tidX +jc ;
40    float valMask = maskh[ jc ] ;
      outval0 += valMask*roi8p[  baseRoi    ] ;
      outval1 += valMask*roi8p[  baseRoi +1 ] ;
      outval2 += valMask*roi8p[  baseRoi +2 ] ;
      outval3 += valMask*roi8p[  baseRoi +3 ] ;
45    outval4 += valMask*roi8p[  baseRoi +4 ] ;
      outval5 += valMask*roi8p[  baseRoi +5 ] ;
      outval6 += valMask*roi8p[  baseRoi +6 ] ;
      outval7 += valMask*roi8p[  baseRoi +7 ] ;
     }
50
  // sortie des 8 pixels
  output[ idx++ ] = outval0 ;
  output[ idx++ ] = outval1 ;
  output[ idx++ ] = outval2 ;
55 output[ idx++ ] = outval3 ;
  output[ idx++ ] = outval4 ;
  output[ idx++ ] = outval5 ;
  output[ idx++ ] = outval6 ;
  output[ idx   ] = outval7 ;
60 }
```

Masque		Taille d'image 512 × 512	1024 × 1024	2048 × 2048	4096 × 4096
3×3	temps exéc. (ms)	0.056	0.192	0.719	2.796
	débit calcul (MP/s)	4681	5461	5834	6000
5×5	temps exéc. (ms)	0.060	0.213	0.794	3.073
	débit calcul (MP/s)	4369	4923	5282	5460
7×7	temps exéc. (ms)	0.064	0.225	0.886	3.490
	débit calcul (MP/s)	4096	4660	4734	4807

TABLE 9.4 – Performances des kernels effectuant la convolution séparable sur le modèle des listings 9.3 et 9.4, sur GPU C2070. Le temps d'exécution correspond à l'exécution des 2 kernels. Cette variante présente des performances voisines de la solution Nvidia.

Le cas de la convolution 1D horizontale est différent : il existe toujours des recouvrements entre les différentes positions du masque au sein d'un paquet de pixels. Il serait alors naturel de penser à appliquer la technique que nous avons proposée pour la convolution non-séparable et qui consiste à lire chaque donnée d'entrée une seule fois, puis de la distribuer à tous les calculs auxquels elle participe. Il faut cependant considérer que lire l'image intermédiaire depuis la mémoire texture impose de l'y copier préalablement, entre

les deux opérations de convolution 1D, ce qui représente un coût en fonction de la taille d'image, dont le détail est donné par la table 9.6.

La solution retenue pour maximiser les performances de cette passe horizontale est alors de lire les données d'entrée directement depuis la mémoire globale, bénéficiant ainsi de son cache 1D (sur C2070) tout en économisant l'opération de copie en texture. L'exploitation des recouvrements intra-paquet peut être faite de la même manière que pour la convolution non-séparable et cela conduit, pour la convolution horizontale 1×3 au kernel du listing 9.5.

Listing 9.5 – Kernel réalisant la convolution horizontale optimisée 1×3 sans utilisation de la mémoire partagée.

```
__global__ void kernel_convoSep8HL3x8pG( unsigned char *input, unsigned char  *output,
                                                                int i_dim, int j_dim)
{
  int jc, baseIdx, id ;
  int L=3 ;
  float outval0=0.0, outval1=0.0, outval2=0.0, outval3=0.0, outval4=0.0, outval5=0.0,
      outval6=0.0, outval7=0.0 ;
  float val ;
  // coordonnees absolues du point de base
  int j = (__mul24( blockIdx.x, blockDim.x ) + threadIdx.x)<<3 ;
  int i = __mul24( blockIdx.y, blockDim.y ) + threadIdx.y ;

  baseIdx = __mul24(i  , j_dim) + j ;
  if (baseIdx >0 ) id = baseIdx-1 ; else id = baseIdx ;
   val = input[id] ;          // pixel 1 : 1 calcul
   outval0 += masque[0]*val ;
  val = input[baseIdx++] ;       // pixel 2 : 2 calculs
   outval0 += masque[1]*val ;
   outval1 += masque[0]*val ;
  val = input[baseIdx++] ;            // pixels 3 à 8 : 3 calculs
   outval0 += masque[2]*val ;
   outval1 += masque[1]*val ;
        outval2 += masque[0]*val ;
  val = input[baseIdx++] ;
   outval1 += masque[2]*val ;
   outval2 += masque[1]*val ;
        outval3 += masque[0]*val ;
  val = input[baseIdx++] ;
   outval2 += masque[2]*val ;
   outval3 += masque[1]*val ;
        outval4 += masque[0]*val ;
  val = input[baseIdx++] ;
   outval3 += masque[2]*val ;
   outval4 += masque[1]*val ;
        outval5 += masque[0]*val ;
  val = input[baseIdx++] ;
   outval4 += masque[2]*val ;
   outval5 += masque[1]*val ;
        outval6 += masque[0]*val ;
  val = input[baseIdx++] ;
   outval5 += masque[2]*val ;
   outval6 += masque[1]*val ;
        outval7 += masque[0]*val ;
  val = input[baseIdx++] ;            // pixel 9 : 2 calculs
   outval6 += masque[2]*val ;
   outval7 += masque[1]*val ;
  val = input[baseIdx++] ;            // pixel 10 : 1 calcul
   outval7 += masque[2]*val ;

  baseIdx = __mul24(i  , j_dim) + j ;
  output[ baseIdx++ ] = outval0 ;       // les 8 sorties
  output[ baseIdx++ ] = outval1 ;
  output[ baseIdx++ ] = outval2 ;
  output[ baseIdx++ ] = outval3 ;
  output[ baseIdx++ ] = outval4 ;
```

```
55    output[ baseIdx++ ] = outval5 ;
      output[ baseIdx++ ] = outval6 ;
      output[ baseIdx++ ] = outval7 ;
}
```

Les performances globales de cette solution sont particulièrement élevées et surpassent assez nettement celles de la solution proposée par le constructeur qui met en œuvre deux kernels complémentaires semblables et faisant usage de la mémoire partagée. Le premier ressemble à celui du listing 9.3, pour la convolution 1D verticale et le second à celui du listing 9.4 pour la convolution 1D horizontale. Cette paire de kernels fournit une solution souple où la taille du masque est un paramètre d'entrée, mais ses performances sont voisines de celles des kernels Nvidia dont on trouve le détail dans la table 9.4, qui présente les temps d'exécution ainsi que les débits correspondants (hors transferts). La plus grande efficacité de la convolution séparable par rapport à la non-séparable y est globalement confirmée par des temps d'exécution inférieurs , à l'exception de la taille de masque 3×3 où les coûts de l'écriture intermédiaire en mémoire globale ne parviennent pas à être compensés par le plus petit nombre d'opérations arithmétiques. Les débits globaux de la table 9.8 sont obtenus après intégration des temps de transfert des données, détaillés dans la table 8.1 et rappelés dans la table 9.5 pour des images 8 bits.

Notre solution, dont les résultats détaillés sont donnés en table 9.7, présente un débit de calcul pouvant dépasser les 7000 MP/s alors que ceux de Nvidia ne dépassent jamais les 6000 MP/s, soit des accélerations de 17% à 33%. À cause de la prépondérance des transferts de données, les débits globaux ne varient que très peu, avec des maxima de 2026 MP/s pour nos kernels et 1933 MP/s pour ceux de Nvidia.

Image	**Total** (ms)
512×512	0.14
1024×1024	0.43
2048×2048	1.53
4096×4096	5.88

TABLE 9.5 – Temps de transfert total depuis et vers le GPU, en fonction de la dimension de l'image. Extrait de la table 8.1.

Image	Temps (ms)
512×512	0.029
1024×1024	0.101
1024×1024	0.387
1024×1024	1.533

TABLE 9.6 – Durée de la copie depuis la mémoire globale vers la mémoire texture, en fonction de la taille de l'image.

Masque		Taille d'image **512 × 512**	**1024 × 1024**	**2048 × 2048**	**4096 × 4096**
3×3	temps exéc. (ms) débit calcul (MP/s)	0.042 6242	0.142 7384	0.550 7626	2.390 7020
5×5	temps exéc. (ms) débit calcul (MP/s)	0.046 5699	0.160 6554	0.604 6944	2.578 6508
7×7	temps exéc. (ms) débit calcul (MP/s)	0.054 4855	0.192 5461	0.731 5738	2.987 5617

TABLE 9.7 – Performances des kernels effectuant la convolution séparable optimisée sur le modèle des listings 9.3 et 9.5, sur GPU C2070. Le temps d'exécution correspond à l'exécution des 2 kernels.

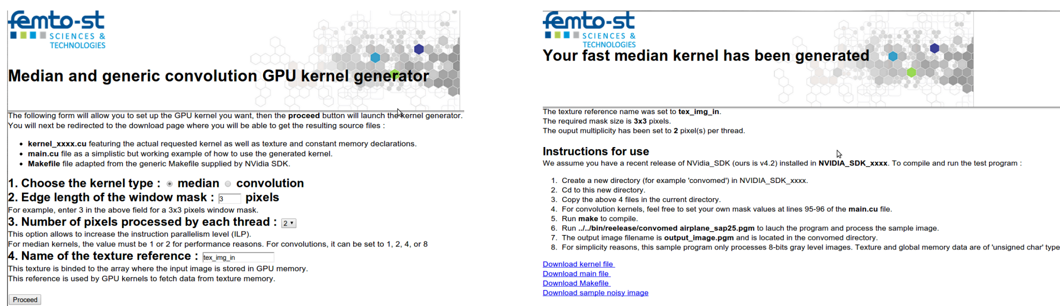
Masque	Taille d'image **512 × 512**	**1024 × 1024**	**2048 × 2048**	**4096 × 4096**
3×3	1380	1817	2016	2028
5×5	1351	1762	1965	1983
7×7	1298	1672	1855	1892

TABLE 9.8 – Débit global en ms (incluant les transferts) des kernels effectuant la convolution séparable sur le modèle des listings 9.3 et 9.5, sur GPU C2070.

9.5/ CONCLUSION

L'architecture des GPUs et le modèle de programmation CUDA permettent d'implémenter efficacement les opérations de convolution, séparable ou non. Nous avons transposé les principes appliqués aux filtres médians et montré qu'ils n'étaient pas tous pertinents dans le cas de la convolution. Nous avons malgré cela proposé des solutions adaptées qui ont permis d'atteindre des performances encore inégalées sur GPU Nvidia avec jusqu'à 2138 millions de pixels traités à la seconde, transferts inclus. Les expérimentations conduites sur les kernels de convolution tendent également à confirmer, dans un cadre plus large, ce que les travaux sur les filtres médians avaient fait apparaître : l'usage de la mémoire partagée ne représente pas forcément la solution apportant les meilleures performances. Cela peut cependant être le cas, en particulier lorsque les voisinages des pixels d'un même paquet ne se recouvrent pas, rendant sans objet toute optimisation liée à ces recouvrements.

Conscients du manque de souplesse découlant de l'optimisation de ces kernels et pour que cela ne soit pas un frein à l'utilisation de ces solutions, nous avons enfin proposé une application en ligne qui génère, à la demande, les codes des kernels médians et de convolution d'après les critères indiqués par l'utilisateur. Ce dernier peut alors télécharger un ensemble suffisant et immédiatement fonctionnel comprenant un fichier kernel GPU, un fichier main.c, un Makefile et une image de test. Il est accessible à l'adresse http ://info.iut-bm.univ-fcomte.fr/staff/perrot/convomed et ses pages d'accueil et de téléchargement sont reproduites à la figure 9.2.



(a) Sélection des paramètres.

(b) Téléchargement des fichiers.

FIGURE 9.2 – Générateur de codes sources pour les filtres GPU rapides.

10

CONCLUSION GÉNÉRALE

Les travaux présentés dans ce manuscrit partagent le même cadre et les mêmes objectifs, à savoir effectuer des opérations de filtrage ou de segmentation sur des images bruitées en exploitant au mieux les capacités de traitement des GPUs. Comme le laissait entrevoir notre intuition première, ces traitements présentent des propriétés de parallélisme très diverses, conduisant à des implémentations plus ou moins efficaces sur GPU.

Certains algorithmes, comme le filtrage contraint par les lignes de niveaux, ont pu voir leur objectif opérationnel conservé en adaptant les modèles aux caractéristiques particulières des GPUs, ce qui a permis d'atteindre des niveaux de performance élevés. Pour d'autres opérations plus classiques, comme les filtrages médian ou de convolution, la problématique est autre, puisqu'il s'agit d'effectuer des traitements de référence ayant une définition mathématique clairement établie. Toute implémentation efficace requiert la conception de structures garantissant l'exactitude du calcul tout en optimisant l'utilisation du GPU. Nous avons, dans ce domaine, contribué significativement à l'amélioration des performances, en approchant du maximum permis par nos architectures de test.

Dans tous les cas, les implémentations présentées ici sont le fruit d'une recherche approfondie et quasi-exhaustive de la meilleure utilisation des différents types de mémoire disponibles sur GPU. Nos travaux mettent en évidence des résultats qui ébranlent le paradigme de la mémoire partagée comme unique chemin vers la performance. En effet, dans de nombreux cas, l'utilisation de la mémoire partagée s'avère pénalisante et le recours aux registres internes des cœurs de calcul se révèle bien plus efficace. L'optimisation des calculs n'est de surcroît pas le seul aspect dont l'implémentation doive être soignée. Nous avons ainsi optimisé les temps de transfert entre CPU et GPU pris en compte dans nos calculs de performances, contrairement à la plupart des articles de référence qui « omettent » de les intégrer à leurs mesures.

L'algorithme de type *snake* s'est avéré particulièrement délicat à implémenter efficacement sur GPU, la version parallèle ne surpassant la version séquentielle que pour des images de très grande taille. Les motifs d'accès à la mémoire demeurent beaucoup trop irréguliers pour être performants et conduisent à exécuter des grilles de calcul creuses ne permettant pas de masquer efficacement les latences. Toutefois, les modèles de GPU les plus récents, basés sur l'architecture Kepler, permettent d'entrevoir la possibilité de minimiser ce problème en recourant à deux de leurs nouvelles fonctionnalités : la faculté d'exécuter simultanément plusieurs kernels et le parallélisme dynamique.

Nos travaux remettent également en cause l'idée trop souvent reçue que le simple portage d'applications séquentielles sur GPU permet systématiquement d'en décupler les performances. Certains algorithmes, tels le snake, font par exemple appel à des opéra-

tions de réduction sur des grilles de calcul creuses, et sont clairement peu adaptés aux GPUs. De plus, les algorithmes que l'on parvient à implémenter efficacement ne le sont qu'au prix d'optimisations particulièrement ardues.

Beaucoup d'autres traitements sont susceptibles de bénéficier des performances toujours en hausse des cartes graphiques modernes. Certains n'ont pas encore été portés sur ces plateformes, d'autres l'ont été, mais n'atteignent pas toujours les performances attendues. Sur la base des techniques et savoir-faire que nous avons pu développer durant ces années de thèse, il est permis de penser que nous pourrons à l'avenir contribuer à améliorer significativement cet état de fait, avec des solutions adaptées tant aux évolutions des matériels qu'aux diverses problématiques scientifiques.

À court terme, nous envisageons d'appliquer les techniques exposées ici aux algorithmes de traitement d'image qui font référence en termes de qualité, comme *BM3D* ou *level-sets*, qui nous paraissent susceptibles d'en tirer parti. Outre les performances obtenues dans le traitement des images 2-D, les résultats obtenus concernant la convolution séparable montrent que nos méthodes apportent aussi un gain de performances important sur les opérations 1-D, ce qui nous permettra d'étendre leur champ d'application à l'ensemble des signaux mono-dimensionnels comme les signaux audio. Enfin, le procédé d'exploitation des recouvrements de voisinages, d'autant plus efficace que ces recouvrements sont importants, ouvre des perspectives très prometteuses dans le traitement des images 3-D. Nous envisageons d'étendre nos recherches à ce domaine et à ses nombreuses applications, notamment l'interprétation des données issues de l'imagerie médicale.

Bibliographie

[1] Kdd cup data, October 1999.

[2] Ccd image sensor noise sources. Technical report, Eastman Kodak company, Rochester, August 2001.

[3] David Adalsteinsson and James Sethian. *A fast level set method for propagating interfaces*. PhD thesis, University of California, 1994.

[4] Pankaj K Agarwal and Cecilia Magdalena Procopiuc. Exact and approximation algorithms for clustering. *Algorithmica*, 33(2) :201–226, 2002.

[5] M. Aldinucci, C.S.M. Drocco, M. Torquati, and S. Palazzo. A parallel edge preserving algorithm for salt and pepper image denoising. 2012.

[6] Pablo Arbelaez, Michael Maire, Charless Fowlkes, and Jitendra Malik. Contour detection and hierarchical image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(5) :898–916, 2011.

[7] Sanjeev Arora, Prabhakar Raghavan, and Satish Rao. Approximation schemes for euclidean k-medians and related problems. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 106–113. ACM, 1998.

[8] Christian Bauer, Horst Bischof, and Reinhard Beichel. Segmentation of airways based on gradient vector flow. In *International workshop on pulmonary image analysis, Medical image computing and computer assisted intervention*, pages 191–201, 2009.

[9] Nicolas Bertaux, Yann Frauel, Philippe Réfrégier, and Bahram Javidi. Speckle removal using a maximum-likelihood technique with isoline gray-level regularization. *JOSA A*, 21(12) :2283–2291, 2004.

[10] Guy E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[11] Yuri Boykov and Vladimir Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(9) :1124–1137, 2004.

[12] Jack E Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems journal*, 4(1) :25–30, 1965.

[13] A. Buades, B. Coll, and J. M Morel. A non-local algorithm for image denoising. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 2, pages 60–65 vol. 2, 2005.

[14] Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. The staircasing effect in neighborhood filters and its solution. *IEEE Transactions on Image Processing*, 15(6) :1499–1505, 2006.

[15] Vicent Caselles and Jean michel Morel. Topographic maps and local contrast changes in natural images. *Int. J. Comp. Vision*, 33 :5–27, 1999.

[16] Vincent Caselles, Bartomeu Coll, and Jean-Michel Morel. Scale space versus topographic map for natural images. pages 29–49. Springer, 07 1997.

[17] Bryan Catanzaro, Bor-Yiing Su, N. Sundaram, Yunsup Lee, Mark Murphy, and K. Keutzer. Efficient, high-quality image contour detection. In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 2381–2388, 2009.

[18] Joshua E Cates, Aaron E Lefohn, and Ross T Whitaker. Gist : an interactive, gpu-based level set segmentation tool for 3d medical images. *Medical Image Analysis*, 8(3) :217–231, 2004.

[19] Bala G Chandran and Dorit S Hochbaum. A computational study of the pseudoflow and push-relabel algorithms for the maximum flow problem. *Operations research*, 57(2) :358–376, 2009.

[20] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using cuda. *Journal of parallel and distributed computing*, 68(10) :1370–1380, 2008.

[21] Wei Chen, M. Beister, Y. Kyriakou, and M. Kachelries. High performance median filtering using commodity graphics hardware. In *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*, pages 4142–4147, 24 2009-nov. 1 2009.

[22] Wen-Hsiung Chen, C. Smith, and S. Fralick. A fast computational algorithm for the discrete cosine transform. *Communications, IEEE Transactions on*, 25(9) :1004–1009, 1977.

[23] Yizong Cheng. Mean shift, mode seeking, and clustering. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 17(8) :790–799, 1995.

[24] Boris V Cherkassky and Andrew V Goldberg. On implementing the push—relabel method for the maximum flow problem. *Algorithmica*, 19(4) :390–410, 1997.

[25] Christophe Chesnaud, Philippe Réfrégier, and Vlady Boulet. Statistical region snake-based segmentation adapted to different physical noise models. *IEEE Trans. Pattern Anal. Mach. Intell.*, 21(11) :1145–1157, 1999.

[26] Laurent D Cohen, Eric Bardinet, Nicholas Ayache, et al. Surface reconstruction using active contour models. 1993.

[27] Dorin Comaniciu and Peter Meer. Mean shift analysis and applications. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 2, pages 1197–1203. IEEE, 1999.

[28] Dorin Comaniciu and Peter Meer. Mean shift : A robust approach toward feature space analysis. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(5) :603–619, 2002.

[29] LJ Cutrona. Synthetic aperture radar. *Radar Handbook, second edition, ed. M. Skolnik, McGraw-Hill, New York*, 1990.

[30] Kostadin Dabov, Alessandro Foi, Vladimir Katkovnik, and Karen Egiazarian. Image denoising with block-matching and 3d filtering. In *IN ELECTRONIC IMAGING'06, PROC. SPIE 6064, NO. 6064A-30*, 2006.

[31] Kostadin Dabov, Ro Foi, Vladimir Katkovnik, and Karen Egiazarian. Bm3d image denoising with shape-adaptive principal component analysis. In *Proc. Workshop on Signal Processing with Adaptive Sparse Structured Representations (SPARS'09*, 2009.

[32] Ingrid Daubechies. *Ten lectures on wavelets*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1992.

[33] Nandan Dixit, Renaud Keriven, and Nikos Paragios. Gpu-cuts : Combinatorial optimisation, graphic processing units and adaptive object extraction. 2005.

[34] Michael Elad and Michal Aharon. Image denoising via sparse and redundant representations over learned dictionaries. *Image Processing, IEEE Transactions on*, 15(12) :3736–3745, 2006.

[35] Pedro F Felzenszwalb and Daniel P Huttenlocher. Efficient graph-based image segmentation. *International Journal of Computer Vision*, 59(2) :167–181, 2004.

[36] Oliver Fluck, Shmuel Aharon, Daniel Cremers, and Mikael Rousson. Gpu histogram computation. In *ACM SIGGRAPH 2006 Research posters*, page 53. ACM, 2006.

[37] James D Foley, Andries Van Dam, Steven K Feiner, John F Hughes, and Richard L Phillips. *Introduction to computer graphics*, volume 55. Addison-Wesley Reading, 1994.

[38] Lester Randolph Ford and Delbert R Fulkerson. *A simple algorithm for finding maximal network flows and an application to the Hitchcock problem*. Rand Corporation, 1955.

[39] Keinosuke Fukunaga and Larry Hostetler. The estimation of the gradient of a density function, with applications in pattern recognition. *Information Theory, IEEE Transactions on*, 21(1) :32–40, 1975.

[40] Brian Fulkerson and Stefano Soatto. Really quick shift : Image segmentation on a gpu. In *Trends and Topics in Computer Vision*, pages 350–358. Springer, 2012.

[41] Frédéric Galland, Nicolas Bertaux, and Philippe Réfrégier. Minimum description length synthetic aperture radar image segmentation. *IEEE Transactions on Image Processing*, 12(9) :995–1006, 2003.

[42] Olivier Germain and Philippe Réfrégier. Statistical active grid for segmentation refinement. *Pattern Recognition Letters*, 22(10) :1125–1132, 2001.

[43] Bart Goossens, Hiêp Luong, Jan Aelterman, Aleksandra Pižurica, and Wilfried Philips. A gpu-accelerated real-time nlmeans algorithm for denoising color video sequences. In Jacques Blanc-Talon, Don Bone, Wilfried Philips, Dan Popescu, and Paul Scheunders, editors, *Advanced Concepts for Intelligent Vision Systems*, volume 6475 of *Lecture Notes in Computer Science*, pages 46–57. Springer Berlin Heidelberg, 2010.

[44] Zhiyu He and Falko Kuester. Gpu-based active contour segmentation using gradient vector flow. In George Bebis, Richard Boyle, Bahram Parvin, Darko Koracin, Paolo Remagnino, Ara Nefian, Gopi Meenakshisundaram, Valerio Pascucci, Jiri Zara, Jose Molineros, Holger Theisel, and Tom Malzbender, editors, *Advances in Visual Computing*, volume 4291 of *Lecture Notes in Computer Science*, pages 191–201. Springer Berlin Heidelberg, 2006.

[45] Glenn E Healey and Raghava Kondepudy. Radiometric ccd camera calibration and noise estimation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 16(3) :267–276, 1994.

[46] Dorit S Hochbaum and James B Orlin. Simplifications and speedups of the pseudoflow algorithm. *Networks*, 61(1) :40–57, 2013.

[47] Bai Hong-tao, He Li-li, Ouyang Dan-tong, Li Zhan-shan, and Li He. K-means on commodity gpus with cuda. In *Computer Science and Information Engineering, 2009 WRI World Congress on*, volume 3, pages 651–655, 2009.

[48] GEORGE Humphrey. The psychology of the gestalt. *Journal of Educational Psychology*, 15(7) :401, 1924.

[49] Won-Ki Jeong, Johanna Beyer, Markus Hadwiger, Amelio Vazquez, Hanspeter Pfister, and Ross T Whitaker. Scalable and interactive segmentation and visualization of neural processes in em datasets. *Visualization and Computer Graphics, IEEE Transactions on*, 15(6) :1505–1514, 2009.

[50] M. Kachelriess. Branchless vectorized median filtering. In *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*, pages 4099 –4105, 24 2009-nov. 1 2009.

[51] Michael Kass, Andrew P. Witkin, and Demetri Terzopoulos. Snakes : Active contour models. *International Journal of Computer Vision*, 1(4) :321–331, 1988.

[52] Yakov Keselman and EVANGELIA Micheli-Tzanakou. Extraction and characterization of regions of interest in biomedical images. In *Information Technology Applications in Biomedicine, 1998. ITAB 98. Proceedings. 1998 IEEE International Conference on*, pages 87–90. IEEE, 1998.

[53] P. Kestener, Y. Moudden, and A. Pedron. Calcul scientifique sur gpu et application en traitement d'images. Seminaire HPC-GPU, CMLA, ENS Cachan, March 2009.

[54] Pushmeet Kohli and Philip HS Torr. Dynamic graph cuts for efficient inference in markov random fields. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 29(12) :2079–2088, 2007.

[55] Aaron E Lefohn, Joshua E Cates, and Ross T Whitaker. Interactive, gpu-based level sets for 3d segmentation. In *Medical Image Computing and Computer-Assisted Intervention-MICCAI 2003*, pages 564–572. Springer, 2003.

[56] Aaron E Lefohn, Joe M Kniss, Charles D Hansen, and Ross T Whitaker. Interactive deformation and visualization of level set surfaces using graphics hardware. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 11. IEEE Computer Society, 2003.

[57] Aaron E Lefohn, Joe M Kniss, Charles D Hansen, and Ross T Whitaker. A streaming narrow-band algorithm : interactive computation and visualization of level sets. In *ACM SIGGRAPH 2005 Courses*, page 243. ACM, 2005.

[58] Peihua Li and Lijuan Xiao. Mean shift parallel tracking on gpu. In *Pattern Recognition and Image Analysis*, pages 120–127. Springer, 2009.

[59] Tao Li, Alexandre Krupa, and Christophe Collewet. A robust parametric active contour based on fourier descriptors. In *Image Processing (ICIP), 2011 18th IEEE International Conference on*, pages 1037–1040. IEEE, 2011.

[60] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, page 14. California, USA, 1967.

[61] Stphane Mallat. *A Wavelet Tour of Signal Processing, Third Edition : The Sparse Way*. Academic Press, 3rd edition, 2008.

[62] Massimo Mancuso and Sebastiano Battiato. An introduction to the digital still camera technology. *ST Journal of System Research*, 2(2), 2001.

[63] David Martin, Charless Fowlkes, Doron Tal, and Jitendra Malik. A database of human segmented natural images and its application to evaluating segmentation algorithms and measuring ecological statistics. In *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, volume 2, pages 416–423. IEEE, 2001.

[64] Georges Matheron. *Random sets and integral geometry*. Wiley, 1975.

[65] S.A. Nene, S.K. Nayar, and Murase H. Columbia object image library (coil-100). Technical Report CUCS-006-96, Computer Vision Laboratory, Columbia University, February 1996.

[66] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide v4.2*, 7 2012.

[67] Stanley Osher and James A Sethian. Fronts propagating with curvature-dependent speed : algorithms based on hamilton-jacobi formulations. *Journal of computational physics*, 79(1) :12–49, 1988.

[68] N. Otsu. A threshold selection method from gray-level histograms. *Systems, Man and Cybernetics, IEEE Transactions on*, 9(1) :62–66, 1979.

[69] Alan W. Paeth. Median finding on a 3-by-3 grid. In *Graphics Gems V*, pages 171–175. Academic Press, 1995.

[70] Fernanda Palhano Xavier De Fontes, Guillermo Andrade Barroso, Pierrick Coupé, and Pierre Hellier. Real time ultrasound image denoising. *Journal of Real-Time Image Processing*, May 2010.

[71] Dan Pelleg, Andrew W Moore, et al. X-means : Extending k-means with efficient estimation of the number of clusters. In *ICML*, pages 727–734, 2000.

[72] S. Perreault and P. Hebert. Median filtering in constant time. *Image Processing, IEEE Transactions on*, 16(9) :2389 –2394, sept. 2007.

[73] G. Perrot, S. Domas, R. Couturier, and N. Bertaux. Gpu implementation of a region based algorithm for large images segmentation. In *Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on*, pages 291 –298, 31 2011-sept. 2 2011.

[74] Gilles Perrot. Image processing. In *Designing Scientific Applications on GPUs*, pages 28,70. CRC Press, 2013.

[75] Gilles Perrot, Stéphane Domas, and Raphaël Couturier. Fine-tuned high-speed implementation of a gpu-based median filter. *Journal of Signal Processing Systems*, pages 1–6, 2013.

[76] Gilles Perrot, Stéphane Domas, Raphaël Couturier, and Nicolas Bertaux. Fast gpu-based denoising filter using isoline levels. *Journal of Real-Time Image Processing*, pages 1–12, 2013.

[77] T.Q. Pham and L.J. van Vliet. Separable bilateral filtering for fast video preprocessing. In *Multimedia and Expo, 2005. ICME 2005. IEEE International Conference on*, pages 4 pp.–, 2005.

[78] Nikolay Ponomarenko, Vladimir Lukin, Alexander Zelensky, Karen Egiazarian, M Carli, and F Battisti. Tid2008-a database for evaluation of full-reference visual quality assessment metrics. *Advances of Modern Radioelectronics*, 10(4) :30–45, 2009.

[79] Nikolay Ponomarenko, Flavia Silvestri, Karen Egiazarian, Marco Carli, Jaakko Astola, and Vladimir Lukin. On between-coefficient contrast masking of dct basis functions. In *Proceedings of the Third International Workshop on Video Processing and Quality Metrics*, volume 4, 2007.

[80] F. Porikli. Constant time o(1) bilateral filtering. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8, 2008.

[81] Mike Roberts, Jeff Packer, Mario Costa Sousa, and Joseph Ross Mitchell. A work-efficient gpu algorithm for level set segmentation. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 123–132, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[82] Rémi Ronfard. Region-based strategies for active contour models. *International Journal of Computer Vision*, 13(2) :229–251, 1994.

[83] Martin Rumpf and Robert Strzodka. Level set segmentation in graphics hardware. In *Image Processing, 2001. Proceedings. 2001 International Conference on*, volume 3, pages 1103–1106. IEEE, 2001.

[84] Ricardo M. Sanchez and Paul A. Rodriguez. Bidimensional median filter for parallel computing architectures. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 1549–1552, march 2012.

[85] R.M. Sanchez and P.A. Rodriguez. Bidimensional median filter for parallel computing architectures. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 1549 –1552, march 2012.

[86] James A Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4) :1591–1595, 1996.

[87] S.A.Arul Shalom, Manoranjan Dash, and Minh Tue. Efficient k-means clustering using accelerated graphics processors. In Il-Yeol Song, Johann Eder, and Tho-Manh Nguyen, editors, *Data Warehousing and Knowledge Discovery*, volume 5182 of *Lecture Notes in Computer Science*, pages 166–175. Springer Berlin Heidelberg, 2008.

[88] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(8) :888–905, 2000.

[89] Erik Smistad, AnneC. Elster, and Frank Lindseth. Real-time gradient vector flow on gpus using opencl. *Journal of Real-Time Image Processing*, pages 1–8, 2012.

[90] J. Stam. Convolution soup. In *GPU Technology Conference*, Aug. 2010.

[91] T. Stitch. Graph cuts with cuda. In *GPU Technology Conference*, Oct. 2009.

[92] Gilbert Strang. The discrete cosine transform. *SIAM review*, 41(1) :135–147, 1999.

[93] RicardoM. Sánchez and PaulA. Rodríguez. Highly parallelable bidimensional median filter for modern parallel programming models. *Journal of Signal Processing Systems*, 71(3) :221–235, 2013.

[94] Albert JP Theuwissen. Ccd or cmos image sensors for consumer digital still photography ? In *VLSI Technology, Systems, and Applications, 2001. Proceedings of Technical Papers. 2001 International Symposium on*, pages 168–171. IEEE, 2001.

[95] C. Tomasi and R. Manduchi. Bilateral filtering for gray and color images. In *Computer Vision, 1998. Sixth International Conference on*, pages 839–846, 1998.

[96] John Wilder Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

[97] Andrea Vedaldi and Stefano Soatto. Quick shift and kernel methods for mode seeking. In *Computer Vision–ECCV 2008*, pages 705–718. Springer, 2008.

[98] V. Vineet and P. J. Narayanan. Cuda cuts : Fast graph cuts on the gpu. In *Computer Vision and Pattern Recognition Workshops, 2008. CVPRW '08. IEEE Computer Society Conference on*, pages 1–8, 2008.

[99] Vibhav Vineet, Pawan Harish, Suryakant Patidar, and P. J. Narayanan. Fast minimum spanning tree for large graphs on the gpu. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 167–171, New York, NY, USA, 2009. ACM.

[100] Vasily Volkov. Better performance at lower occupancy. *Proceedings of the GPU Technology Conference, GTC*, 10, 2010.

[101] Song Wang and Jeffrey Mark Siskind. Image segmentation with minimum mean cut. In *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, volume 1, pages 517–524. IEEE, 2001.

[102] Song Wang and Jeffrey Mark Siskind. Image segmentation with ratio cut. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 25(6) :675–690, 2003.

[103] Zhou Wang, Alan Conrad Bovik, Hamid Rahim Sheikh, Student Member, Eero P. Simoncelli, and Senior Member. Image quality assessment : From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13 :600–612, 2004.

[104] Henry Wong, M-M Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 235–246. IEEE, 2010.

[105] Zhenyu Wu and Richard Leahy. An optimal graph theoretic approach to data clustering : Theory and its application to image segmentation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 15(11) :1101–1113, 1993.

[106] Chunxia Xiao and Meng Liu. Efficient mean-shift clustering using gaussian kd-tree. In *Computer Graphics Forum*, volume 29, pages 2065–2073. Wiley Online Library, 2010.

[107] Qingxiong Yang, Kar-Han Tan, and N. Ahuja. Real-time o(1) bilateral filtering. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 557–564, 2009.

[108] C. T. Zahn. Graph-theoretical methods for detecting and describing gestalt clusters. *IEEE Trans. Comput.*, 20(1) :68–86, January 1971.

[109] Z. Zheng, W. Xu, and K. Mueller. Performance tuning for cuda-accelerated neighborhood denoising filters. *Workshop on High Performance Image Reconstruction (HPIR)*, pages 52–55, 2011.

[110] Zuoyong Zheng and Ruixia Zhang. A fast gvf snake algorithm on the gpu. *image*, 2 :4, 2012.

TABLE DES FIGURES

2.1 Comparaison des structures d'un cœur de GPU et d'un cœur de CPU (d'après [66]). ALU = Arithmetical & Logical Unit. 13
2.2 Comparaison des performances des GPUs Nvidia et des CPU Intel (d'après [66]). 14
2.3 Organisation des GPUs d'architecture Fermi, comme le C2070 (d'après www.hpcresearch.nl). 15
2.4 Représentation d'une grille de calcul en 2D et des blocs de threads, à 2 dimensions, qui la composent. 16

4.1 Images 256×256 en niveau de gris 8 bits utilisées pour l'illustration des propriétés des filtres. a) l'image de référence non bruitée. b) l'image corrompue par un bruit gaussien d'écart type $\sigma = 25$. c) l'image corrompue par un bruit impulsionnel à 25%. 28
4.2 Filtrage par convolution. 29
4.3 Réduction du bruit impulsionnel par filtre médian. 29
4.4 Réduction de bruit gaussien par filtrage bilatéral de voisinage 5×5. σ_S et σ_I sont les écarts type des fonctions gaussiennes de pondération spatiale et d'intensité. 31
4.5 Filtrage par décomposition en ondelettes et seuillage dur des coefficients inférieurs au seuil T. 32
4.6 Filtrage par NL-means pour différentes combinaisons des paramètres de similarité f et de non localité t. 33
4.7 Filtrage par BM3D, PSNR=29.3 dB MSSIM=0.41 33
4.8 Performances relatives des filtres médians implémentés sur GPU dans libJacket/ArrayFire, PCMF et BVM et exécutés sur deux modèles de générations différentes. 34
4.9 Illustration du pré-chargement en mémoire partagée mis en œuvre dans [109] pour l'implémentation, entre autres, du filtre bilatéral. a) en vert le bloc de threads associé aux pixels centraux. b-e) les blocs de pixels successivement pré-chargés en mémoire partagée. f) la configuration finale de la ROI en mémoire partagée. 36

5.1 Segmentation d'une image en niveaux de gris de 128×128 pixels par analyse simple d'histogramme. Colonne de gauche : image d'entrée. Colonne centrale : histogramme des niveaux de gris. Colonne de droite : résultat de la segmentation. 41

5.2 Segmentation d'une image en niveaux de gris de 128 × 128 pixels par simplification de graphe de type *Normalized cut* pour un nombre s de segments variant de 2 à 5. 43

5.3 Segmentation d'une image en niveaux de gris de 128 × 128 pixels par algorithme *k-means* pour un nombre s de segments variant de 2 à 5. Chaque couleur est associée à un segment. Les couleurs sont choisies pour une meilleure visualisation des différents segments. 44

5.4 Segmentation d'une image en niveaux de gris de 128 × 128 pixels par algorithme *mean-shift* pour un rayon de voisinage r de 100, 50, 35 et 25 pixels permettant d'obtenir un nombre s de segments variant respectivement de 2 à 5. Le volume minimal admis pour un segment est fixé à 100 pixels. Chaque couleur est associée à un segment. Les couleurs sont choisies pour une meilleure visualisation des différents segments. 44

5.5 Segmentation d'une image en niveaux de gris de 128 × 128 pixels par algorithme dit du *snake*, dans sa version originale. Les paramètres d'élasticité, de raideur et d'attraction ont été fixés respectivement aux valeurs 5, 0.1 et 5. 46

5.6 Évolution du nombre de pixels actifs pour les itérations successives de l'implémentation de l'algorithme push-relabel de [91]. Les petites images montrent la localisation des pixels actifs après chaque itération, en blanc. . 49

5.7 Segmentation d'une image couleur de 512×512 pixels par l'implémentation GPU quick-shift de [40]. 51

5.8 Comparaison des segmentations d'une image couleur de 2256×3008 pixels réalisées par *mean-shift* standard et par le *mean-shift kd tree* de [106]. 51

5.9 Segmentation d'images issues d'examens IRM par la méthode des level set à bande étroite. 53

5.10 Segmentation d'une image d'épaule en 1024^2 pixels issue d'un examen IRM par l'implémentation du snake GVF de [44]. Le contour est représenté en rouge et le contour final est obtenu en 11 s. Le tracé initial du contour a été artificiellement épaissi pour le rendre visible à l'échelle de l'impression. 53

5.11 Extraction de contour par la version GPU de l'algorithme gPb. Les images sont issues de la base BSDS [63] 55

6.1 À gauche : détermination des vecteurs f_{in} et f_{out}. À droite : code de Freeman d'un vecteur en fonction de sa direction, l'origine étant supposée au pixel central, en noir. 62

6.2 Évolution du contour lors de la segmentation d'une image de 512^2 pixels. La convergence est obtenue à l'itération 14 après 44 ms pour un total de 256 nœuds. 66

6.3 Influence du contour initial sur la segmentation. Le contour final 1 est celui de la figure 6.2. 66

6.4 Segmentation de l'image de test en 4000×4000 pixels. Le tracé du contour a été artificiellement épaissi pour le rendre visible à l'échelle de l'impression. 67

6.5 Segmentation de l'image de test en 4000×4000 pixels avec une cible de petite taille. Le contour initial est la transcription de celui utilisé à la figure 6.2. Le tracé du contour a été artificiellement épaissi pour le rendre visible à l'échelle de l'impression. 67

6.6 Évolution du coût relatif des trois fonctions les plus consommatrices en temps de calcul en fonction de la taille de l'image à traiter. 68

6.7 Calcul des images cumulées S_x et S_x^2 en trois étapes successives. a) cumul partiel bloc par bloc et mémorisation de la somme de chaque bloc. b) cumul sur le vecteur des sommes partielles. c) ajout des sommes partielles à chaque élément des blocs cumulés. 70

6.8 Structuration des données en mémoire du GPU pour l'évaluation en parallèle de l'ensemble des évolutions possibles du contour. 72

6.9 Comparaison des cycles de déplacement des nœuds. Ligne du haut : version séquentielle. Ligne du bas : version parallèle. Les segments en rouge sont des segments du contour non évalués, alors que ceux en pointillés sont les paires ayant reçu les meilleures évaluations parmi les 8 déplacements possibles des nœuds correspondant. 73

6.10 Détermination des coefficients $C(i, j)$ des pixels du contour. 75

6.11 Segmentations d'une image de 100 MP en 0,59 s pour 5 itérations. Le contour initial conserve les proportions de celui de la figure 6.2. 76

6.12 Détermination intelligente du contour initial en deux phases successives. a) La première étape repose sur un échantillonnage horizontal. b) La seconde étape repose sur un échantillonnage vertical. 77

7.1 Détail des motifs et de leur représentation interne, pour la taille $a = 5$. . . . 82

7.2 Exemple de la répartition des pixels dans la région ω pour le calcul de la vraisemblance, pour $n = 6$ ($a = 5$). 83

7.3 Allongement du segment S^n. Deux candidats $S^{p'}$ et $S^{p''}$ sont évalués au travers du critère GLRT de l'équation (7.8) que seul $S^{p''}$ s'avère satisfaire. a) Représentation dans le plan de l'image. b) Évolution des niveaux de gris en fonction de la position des pixels dans les lignes brisées ainsi formées. . 84

7.4 Processus de sélection lors de l'allongement d'une isoline comportant initialement deux segments s_1 et s_2. Dans cet exemple $a = 5$ et $\Delta d_{max} = 2$. Chaque segment évalué est soumis au critère GLRT. Si au moins un des segments présente un test GLRT positif, alors l'allongement est réalisé avec le segment qui forme l'isoline la plus vraisemblable. 86

7.5 Images non bruitées de la base d'images en niveaux de gris de S. Lansel. . 87

7.6 Histogramme des écarts angulaires entre la direction primaire de l'isoline optimale et celle du segment sélectionné par PI-LD avec $q = 1$ (sans allongement), pour l'image du singe (Mandrill). Pour la très grande majorité des pixels, l'écart est nul. 87

7.7 Histogrammes des écarts angulaires entre la direction primaire de l'isoline optimale et celle de l'isoline sélectionnée, pour les images de l'ensemble de test de S. Lansel. La répartition des erreurs est semblable dans toutes ces images, mais également dans toute image naturelle. 88

7.8 Exemple d'application du procédé d'allongement à une isoline comprenant initialement 2 segments. la longueur des segments est $a = 5$. Le procédé se répète jusqu'à ce que le test GLRT échoue. 89

7.9 Situation de la région servant à illustrer le comportant du modèle PI-PD dans les zones à faible pente (LSR). 90

7.10 Comportement du modèle PI-PD dans les zones de faible et à forte pente. On constate un manque de robustesse dans les zones à faible pente : les directions ne sont pas reproduites d'un tirage à l'autre, contrairement à celles de la zone de transition. 93

7.11 Motif de détection des zones à faible pente, pour le cas $\Theta = \Theta_4 = 45°$. L'élévation des pixels permet juste de les distinguer selon 3 classes : l'élévation 1 est associée aux pixels de la région T, l'élévation 0.5 est associée à ceux de la région B et l'élévation 0 désigne les pixels n'intervenant pas dans la détection. 94

7.12 Classification des pixels d'une image bruitée, pour une valeur de seuil $T2 = 2$ du détecteur. b) Les pixels en noir sont ceux à qui le PI-PD sera appliqué. Les pixels en blancs se verront appliquer une moyenne sur tout ou partie du voisinage. 94

7.13 Comparaison des rendus des traitements comparés. Rangée du haut : les images complètes. Rangée du bas : Zooms sur une zone de l'ímage au dessus. 95

7.14 Images non bruitées de la base tid2008. 98

7.15 Exemples de résultat de traitement par PI-PD RVB et par CBM3D pour deux images de la base tid2008 (une image naturelle et l'image de synthèse). Il peut être nécessaire de zoomer sur le document numérique pour visualiser les détails. 99

8.1 Application de la sélection de médiane par oubli à une fenêtre de 3×3 pixels. 106

8.2 Première étape d'identification des extrema pour un filtre 5×5, avec maximisation de l'ILP (Instruction Level Parallelism) pour l'identification des extrema. 107

8.3 Gestion des éléments communs aux fenêtres de deux pixels centraux voisins dans un filtre médian 5×5. La liste initiale comprend les 14 premiers éléments communs, puis les 7 premières étapes de sélection sont conduites en commun avant que les 5 dernières le soient en parallèle, mais de manière disjointe. 108

8.4 Comparaison des débits (MP/s) atteints par notre implémentation notée PRMF, avec les principales solutions de référence. De gauche à droite : PCMF, BVM, PRMF, ArrayFire (impossible en 4096×4096) 110

9.1 Multiplicité des implications des pixels de la zone d'intérêt d'un thread dans les calculs de convolution. Le nombre de calculs dans lequel est impliqué un pixel est inscrit en son centre. Le premier pixel du paquet, ou pixel de base, est repéré par ses coordonnées (x, y) ; le dernier a pour coordonnées $(x + 7, y)$ 116

9.2 Générateur de codes sources pour les filtres GPU rapides. 125

Liste des tables

2.1 Caractéristiques des différents types de mémoire disponibles sur le GPU. Pour les mémoires cachées, les latences sont données selon l'accès *sans-cache/L1/L2*. Les mesures ont été obtenues à l'aide des microprogrammes de test de [104]. 16

6.1 Valeur du coefficient $C(i, j)$ en fonction des valeurs des codes de Freeman des vecteurs f_{in} et f_{out}. 62

6.2 Performances (en secondes) de la segmentation par snake polygonal sur CPU en fonction de la taille de l'image à traiter. Les temps sont obtenus avec la même image de test dilatée et bruitée et un contour initial carré dont la distance aux bords est proportionnelle à la taille de l'image. Seule l'image en 15 MP a pu être traitée par une implémentation utilisant SSE2. . 68

6.3 Accélération constatée, pour le calcul des images cumulées, de l'implémentation GPU (C2070) par rapport à l'implémentation CPU de référence. 71

6.4 Comparaison des temps d'exécution de l'implémentation GPU (C2070) par rapport à l'implémentation CPU de référence, appliqués à une même image dilatée (fig. 6.2) pour en adapter la taille. 76

7.1 Temps de calcul et de transfert des implémentations comparées. 100

7.2 Comparaison image par image de la qualité de débruitage des filtres PI-LD et PI-PD hybride proposé par rapport à BM3D pris comme référence de qualité et à un moyenneur GPU 5×5 pris comme référence de rapidité. Les paramètres du PI-LD/PI-PD sont $n = 5$, $l = 25$, $T_{max} = 1$ et $T2_{max} = 2$. La colonne 'Bruitée' donne les mesures relatives à l'image d'entrée corrompue par un bruit gaussien de moyenne nulle et d'écart type $\sigma = 25$. PI-LD s'exécute en 35 ms, PI-PD en 7,3 ms et BM3D en 4,3 s. 101

7.3 Comparaison image par image de la qualité de débruitage du filtre PI-PD RVB proposé par rapport à BM3D pris comme référence de qualité. Les paramètres du PI-PD sont $n = 4$, $l = 48$, $T_{rvb-max} = 5$. La colonne 'noisy' donne les mesures relatives à l'image d'entrée corrompue par tirage de bruit gaussien sur chaque canal (moyenne nulle, écart type $\sigma = 25$). 102

8.1 Temps de transfert vers et depuis le GPU, en fonction de la dimension de l'image et de la profondeur des niveaux de gris. La colonne "Mémoire globale" donne les temps mesurés lorsque cette seule mémoire est employée. 104

8.2 Pourcentage du temps d'exécution pris par les transferts de données en fonction de la taille de fenêtre du filtre, pour les profondeurs 8 et 16 bits sur GPU C2070. 110

8.3 Performances des filtres médians rapides en fonction des tailles d'image et de fenêtre du filtre, en variantes 8 et 16 bits de profondeursur GPU C2070.111

8.4 Débits maximum effectifs T_8 and T_{16} (en MP/s), respectivement pour les variantes 8 et 16 bits sur C2070. 111

9.1 Performances des kernels effectuant la convolution non-séparable sur le modèle du listing 9.1, sur GPU C2070. Le temps d'exécution correspond à la seule exécution du kernel. Le débit global intègre les temps de transfert. Les valeurs en gras correspondent au traitement de référence. 115

9.2 Performances des kernels effectuant la convolution non-séparable sur le modèle du listing 9.1, sur GPU GTX280. Le temps d'exécution correspond à la seule exécution du kernel. Le débit global intègre les temps de transfert. Les valeurs en gras correspondent au traitement de référence. 115

9.3 Performances des kernels effectuant la convolution non-séparable sur le modèle du listing 9.2, sur GPU C2070. Le temps d'exécution correspond à la seule exécution du kernel. Le débit global intègre les temps de transfert. Les valeurs en gras correspondent au traitement de référence. 117

9.4 Performances des kernels effectuant la convolution séparable sur le modèle des listings 9.3 et 9.4, sur GPU C2070. Le temps d'exécution correspond à l'exécution des 2 kernels. Cette variante présente des performances voisines de la solution Nvidia. 121

9.5 Temps de transfert total depuis et vers le GPU, en fonction de la dimension de l'image. Extrait de la table 8.1. 123

9.6 Durée de la copie depuis la mémoire globale vers la mémoire texture, en fonction de la taille de l'image. 123

9.7 Performances des kernels effectuant la convolution séparable optimisée sur le modèle des listings 9.3 et 9.5, sur GPU C2070. Le temps d'exécution correspond à l'exécution des 2 kernels. 124

9.8 Débit global en ms (incluant les transferts) des kernels effectuant la convolution séparable sur le modèle des listings 9.3 et 9.5, sur GPU C2070. . . . 124

Résumé :

Les cartes graphiques modernes (GPU) mettent, en théorie, la programmation parallèle à la portée de tous. Ces facilités ont éveillé l'intérêt des chercheurs et développeurs de toutes disciplines, qui ont tenté de tirer parti des performances élevées de ces matériels. Cependant, d'importants efforts de conception sont souvent nécessaires à l'obtention des vitesses de traitement espérées. Dans cette thèse, nous proposons des méthodes conduisant à des implémentations rapides de plusieurs algorithmes destinés au traitement des images fortement bruitées. La première est une transposition sur GPU d'un algorithme de segmentation dit du *snake* dont la capacité de traitement a été étendue et les performances améliorées. La seconde décrit un algorithme original, basé sur la recherche des lignes de niveaux et conçu spécifiquement pour les GPUs, qui réduit le bruit gaussien dans les images en niveaux de gris ou en couleur et dont le rapport qualité/vitesse est particulièrement intéressant. En concevant une gestion fine des mémoires du GPU, nous avons également conféré un débit de traitement inégalé au filtre médian, pouvant dépasser les 5 milliards de pixels à la seconde. Enfin nous avons étendu l'application de ces techniques à un opérateur beaucoup plus générique, le filtre de convolution, et montré qu'elles permettaient de surpasser les implémentations les plus rapides connues jusqu'alors, avec un maximum au delà des 7 milliards de pixels à la seconde. Nous mettons aussi à disposition une application en ligne permettant à tout développeur de générer les codes sources opérationnels des filtres que nous avons décrits.

Mots-clés : GPU, Filtrage, Image, Segmentation

Abstract:

In theory, modern graphical processing units (GPUs) make parallel programming accessible to all, and have triggered widespread interest among researchers or developers of all disciplines, with the hope of dramatically increasing processing speeds. Nevertheless, obtaining such performances cannot be done without considerable designing efforts : as an answer, we propose two GPU-based methods leading to fast implementations of several algorithms targeted to processing noisy images. One of them consists in porting the segmentation algorithm named *snake*, with the effect of extending its processing capacity and performance. A second involves a innovative GPU-specific algorithm, based on searching for level lines within gray-level or color images to reduce gaussian noise, whose quality-to-speed ratio is particularly interesting. Through extremely fine-tuned management of the different memory types available on GPUs, we have also conferred unprecedent flow rates to the median filter, making it able to process over 5 billion pixels per second. Eventually, we extended the above methods to the more generic convolution filter, and showed they out-perform the fastest implementations known to date, with over 7 billion pixels per second. In addition, we provide an on-line application that enables any developer to automatically generate operational source code of our filters.

Keywords: GPU, Filtering, Image, Segmentation

■ École doctorale SPIM 16 route de Gray F - 25030 Besançon cedex

■ tél. +33 (0)3 81 66 66 02 ■ **ed-spim@univ-fcomte.fr** ■ www.ed-spim.univ-fcomte.fr