UMONS
Université de Mons

Faculté Polytechnique

Traitement Efficace d'Objets Multimédias sur Architectures Parallèles et Hétérogènes

THÈSE

présentée en vue de l'obtention du grade de Docteur en Sciences de l'Ingénieur

par

SIDI AHMED MAHMOUDI

Jury :

Pr Gaëtan LIBERT	Université de Mons	Président
Pr Pierre MANNEBACK	Université de Mons	Promoteur
Pr Olivier DEBEIR	Université Libre de Bruxelles	
Pr Bernard GOSSELIN	Université de Mons	
Dr Emmanuel JEANNOT	INRIA Bordeaux Sud-Ouest	
Pr Nouredine MELAB	Université Lille 1	
Pr Carlos VALDERRAMA	Université de Mons	

29 Janvier 2013

ACADÉMIE UNIVERSITAIRE WALLONIE-BRUXELLES

À mes très chers parents, mes frères et sœurs « Saïd, Fethi, Amine, Naziha, Nour el Houda et Abderrahim », mes belles sœurs Samira et Nadia. Et bien sur, je ne pourrai pas oublier les petits de la famille Mahmoudi : « Imane, Rani, Lamia, Hacène, Amani, Yaniss, Mehdi et Meriem ». Ce travail est également dédié à mes très chers amis « Aboubakr, Fettah, Mohammed, Djawad, Youcef, Abdelkader, Benamar et Nourelyakine ».

Résumé

Nous proposons dans ce travail d'exploiter, de manière efficace, les architectures parallèles (GPU) et hétérogènes (Multi-CPU/Multi-GPU) afin d'améliorer les performances des méthodes de traitement d'objets multimédias, telles que celles utilisées dans des algorithmes de traitement d'images et de vidéos de haute définition, des applications médicales ainsi que l'analyse et le suivi de mouvements en temps réel. Notre intérêt pour l'accélération de ces méthodes est dû principalement à l'augmentation de l'intensité de calcul de ce type d'applications, et à la forte croissance de la taille des objets multimédias (HD/Full HD) ces dernières années.

Nous proposons un modèle de traitement d'objets multimédias (image unique, images multiples, vidéos multiples, vidéo en temps réel) basé sur l'exploitation de l'intégralité de la puissance de calcul des machines hétérogènes. Ce modèle permet de choisir les ressources à utiliser (CPU ou/et GPU) ainsi que les méthodes à appliquer selon la nature des médias à traiter et la complexité des algorithmes. Le modèle proposé s'appuie sur des stratégies d'ordonnancement efficaces assurant une exploitation optimale des ressources hybrides. Il permet également de réduire les temps de transfert de données grâce à une gestion efficace des mémoires GPU ainsi qu'au recouvrement des copies de données par les fonctions d'exécution sur les GPU multiples. Ce modèle est utilisé pour la mise en œuvre de plusieurs algorithmes tels que l'extraction efficace de contours, la détection de points d'intérêt, la soustraction d'arrière-plan (background), la détection des silhouettes et le calcul des vecteurs du flot optique permettant l'estimation du mouvement. Ces mises en œuvre ont été exploitées pour accélérer différentes applications telles la segmentation des vertèbres dans

des images médicales, l'indexation de séquences vidéo et le suivi de mouvements en temps réel à partir d'une caméra mobile.

Des résultats expérimentaux ont été obtenus par l'application du modèle proposé sur différents types de médias (images et vidéos HD/Full HD, bases d'images médicales). Ces résultats montrent des accélérations globales allant d'un facteur de 5 à 100 par rapport à une implémentation séquentielle sur CPU.

Mots-clés : GPU, architectures hétérogènes, traitement d'images et de vidéos, imagerie médicale, suivi de mouvements.

Abstract

We propose in this work to exploit parallel (GPU) and heterogeneous (Multi-CPU/Multi-GPU) architectures for boosting performance of multimedia processing algorithms, such as exploited in high definition image and video processing methods, medical applications and real time video processing (analysis, tracking) algorithms. Our interest for accelerating these methods is due mainly to the high increase of calculation intensity of such applications, and the strong growth in the size of multimedia objects (HD/Full HD) during the recent years.

We propose a model for multimedia (single image, multiple images, multiple videos, video in real time) processing based on exploiting the full computing power of heterogeneous machines. This model enables to select firstly the computing units (CPU or/and GPU) for processing, and secondly the methods to be applied depending on the type of media to process and the algorithm complexity. The proposed model exploits efficient scheduling strategies which allow an optimal exploitation of hybrid machines. Moreover, it enables to reduce significantly data transfer times thanks to an efficient management of GPU memories and to the overlapping of data copies by kernels executions on multiple GPUs. This model was exploited for implementing several algorithms on GPU such as effective contours extraction, points of interest detection, background subtraction, silhouette detection and optical flow computation used for motion estimation. These implementations are exploited in different applications such as vertebra segmentation in medical images, videos indexation and real time motion tracking using mobile camera.

Experimental results have been obtained by applying the proposed model on different types of media (HD/Full HD images and videos, databases of medical images). These results showed a global speedup ranging from 5 to 100, by comparison with sequential CPU implementations.

Keywords : GPU, heterogeneous architectures, image and video processing, medical imaging, motion tracking.

Remerciements

J'aimerais tout d'abord remercier mon directeur de thèse, le professeur Pierre Manneback, pour m'avoir soutenu, guidé, encouragé, conseillé durant mes recherches. Je désire lui témoigner toute ma reconnaissance pour le temps qu'il m'a consacré et toutes les opportunités qu'il m'a données au cours de cette thèse.

Cette thèse a été rendue possible grâce au soutien de la Fédération Wallonie Bruxelles à travers le projet ARC-OLIMP (*Optimization for Live Interaction Multimedia Processing*), convention AUWB-2008-12. Elle a également bénéficié du soutien de l'action européenne COST IC805 (*Open Network for High-Performance Computing on Complex Environments*), 2009-2013.

Je tiens à remercier les membres de mon comité d'accompagnement pour leur suivi de l'évolution de ma thèse. Leurs remarques, critiques, suggestions et conseils m'ont été d'une grande utilité pour l'avancement de mes travaux. Je vous remercie profondément, Professeurs Gaëtan Libert et Carlos Valderrama de l'Université de Mons, ainsi que le Docteur Emmanuel Jeannot de l'Université de Bordeaux en France. Je remercie également tous les membres du jury d'avoir accepté d'assister à la présentation de ce travail. Mes remerciements vont également à mes collaborateurs Drs Samuel Thibault et Cédric Augonnet de l'Université de Bordeaux, ainsi que Michal Kierzynka du laboratoire PSNC (Poznan Supercomputing and Networking Center) qui m'ont chaleureusement accueilli dans leurs laboratoires. Ils m'ont beaucoup aidé à éclaircir et résoudre plusieurs problématiques grâce à leurs sens du partage.

Je voudrais remercier tous ceux qui ont apporté une contribution dans la réalisation de cette thèse. Je pense à remercier mes collègues du service d'Informatique pour leur aide, amitié, bonne humeur durant toutes ces dernières années. Avec eux, j'ai appris énormément de concepts théoriques et pratiques, en passant des journées très agréables. Un grand merci à : Fabian Lecron, Mohamed Amine Larhmam, Guttadoria Adriano, Michel Bagein, Mohammed Benjelloun, Mathieu Van Sevenant, Sébastien Noël, Sébastien Frémal, Dominique De Beul et finalement mon frère et collègue Saïd qui était toujours présent pour m'aider, m'orienter et me conseiller. Merci mon frère.

À Sarah, ma femme, merci pour tout le soutien que tu m'as apporté. Sans toi, les choses auraient sans doute été plus difficiles.

Table des matières

Introduction 1

I État de l'art 7

1 Présentation des GPU 8

1.1 Architecture des GPU 8

1.2 Programmation des GPU 12

1.2.1 OpenGL 13

1.2.2 CUDA 15

1.2.3 OpenCL 18

2 Exploitation des architectures multicœurs hétérogènes 21

3 Traitement d'objets multimédias sur GPU 29

3.1 Traitement d'images sur GPU 29

3.2 Traitement de vidéos sur GPU 32

3.3 Estimation de complexité de traitement d'images sur GPU 35

3.3.1 Fraction parallélisable 36

3.3.2 Ratio de temps entre calcul et accès mémoire 36

3.3.3 Taux de calcul par pixel 37

3.3.4 Taux d'accès aux mémoires GPU par pixel 37

3.3.5 Diversité et branchements 37

3.3.6 Dépendance de tâches 37

4 Exemples de traitements intensifs d'objets multimédias 38

4.1 Segmentation des vertèbres 38

4.1.1 Apprentissage 39

4.1.2 Modélisation 39
4.1.3 Initialisation 40
4.1.4 Segmentation 41
4.2 Indexation de bases de données multimédias 43
4.2.1 AudioCycle 43
4.2.2 ImageCycle 43
4.2.3 VideoCycle 45
4.3 Détection de mouvements à partir de caméra mobile 46
4.3.1 Estimation du mouvement de caméra 46
4.3.2 Détection du mouvement 47

II Traitement d'images sur GPU **49**
1 Schéma de développement du traitement d'images sur GPU 51
2 Optimisations proposées sur GPU 53
2.1 Traitement d'image unique 53
2.1.1 Chargement d'image en mémoire de texture 53
2.1.2 Traitement CUDA *via* la mémoire partagée 53
2.1.3 Visualisation OpenGL 54
2.2 Traitement d'images multiples 54
2.2.1 Chargement des images multiples *via* le streaming 55
2.2.2 Traitement CUDA *via* le streaming 55
2.2.3 Transfert des résultats 55
3 Mise en œuvre d'algorithmes de traitement d'images sur GPU 57
3.1 Méthodes classiques de traitement d'images sur GPU 57
3.1.1 Transformations géométriques 58
3.1.2 Elimination des bruits 59
3.2 Extraction des points d'intérêt sur GPU 60
3.2.1 Calcul des dérivées et du gradient spatial 60
3.2.2 Calcul des valeurs propres de la matrice du gradient . . . 61
3.2.3 Recherche de la valeur propre maximale 61
3.2.4 Suppression des petites valeurs propres 61
3.2.5 Sélection des meilleures valeurs 61
3.3 Détection des contours sur GPU 62
3.3.1 Lissage gaussien récursif 63
3.3.2 Calcul des gradients de Sobel 63

3.3.3 Calcul de magnitude et de direction du gradient 64
3.3.4 Suppression des non maxima 64
3.3.5 Seuillage des contours 64
4 Analyse des résultats 66
5 Conclusion 71

III Traitement hétérogène d'images multiples **73**
1 Schéma proposé du traitement hétérogène d'images multiples 75
2 Optimisation des traitements hétérogènes 78
2.1 Ordonnancement efficace des tâches hybrides 79
2.1.1 Ordonnancement statique 79
2.1.2 Ordonnancement dynamique 79
2.2 Recouvrement des transferts au sein des GPU multiples 80
2.2.1 Chargement des images d'entrée 80
2.2.2 Traitements hybrides et transfert des résultats 80
3 Analyse des résultats 81
3.1 Détection hybride des coins et contours 82
3.2 Performances du calcul hétérogène : ordonnancement statique . . . 85
3.3 Performances du calcul hétérogène : ordonnancement dynamique . . 85
3.4 Performances du calcul hétérogène : CUDA streaming 86
4 Conclusion 88

IV Traitement Multi-GPU de vidéo Full HD **89**
1 Détection d'objets en temps réel sur GPU 91
1.1 Soustraction de l'arrière-plan sur GPU 91
1.2 Extraction de silhouette sur GPU 94
1.3 Analyse des résultats 96
2 suivi de mouvements en parallèle sur GPU 97
2.1 Mesure du flot optique 97
2.2 Algorithme de suivi de mouvements 99
2.2.1 Détection des coins 99
2.2.2 Suivi des coins détectés 99
2.2.3 Elimination des régions statiques 102
2.3 Suivi de mouvements sur GPU uniques ou multiples 104
2.3.1 suivi de mouvements sur un GPU 104

2.3.2 Suivi de mouvements sur GPU multiples 106
2.4 Analyse des résultats 106
2.4.1 Comparaison des temps de calcul du flot optique sur GPU 107
2.4.2 Résultats d'exploitation des processeurs graphiques multiples 109
2.4.3 Performances en nombre d'images par secondes (fps) . . . 111
3 Conclusion 114

V Modèle de traitement hétérogène d'objets multimédias 115
1 Modèle de traitement hétérogène d'objets multimédias 117
1.1 Traitement CPU/GPU d'image individuelle 117
1.1.1 Estimation de complexité de calcul 117
1.1.2 Choix de ressource adaptée 119
1.1.3 Application du traitement 121
1.2 Traitement hétérogène d'images multiples 122
1.3 Traitement hétérogène de vidéos multiples 122
1.4 Traitement GPU/Multi-GPU de vidéo en temps réel 123
2 Premier cas : segmentation des vertèbres 127
3 Deuxième cas : indexation de vidéos 132
4 Troisième cas : détection de mouvements avec caméra mobile 134
5 Conclusion 139

Conclusion générale et perspectives 141

Références bibliographiques 147

Annexe 1 : Extraction de caractéristiques d'images 161
1 Détection des points d'intérêts 161
1.1 Détection à base de contours 162
1.2 Détection à base de la fonction d'intensité 162
1.3 Détection à base de modèles 162
2 Détection des contours 164
3 Calcul des moments de Hu 165

Annexe 2 : Méthodes de suivi de mouvements 167
4 Mesure du flot optique 167
5 Descipteurs SIFT 169

6 Mise en correspondance d'objets (block matching) 171
6.1 Principe de mise en correspondance d'objets 172
6.1.1 Critères de similtitude 172
6.1.2 Fenêtre de recherche 173
6.1.3 Méthodes de recherche 173

Annexe 3 : publications **175**
7 Revues internationales 175
8 Chapitre de livre (en soumission) : 175
9 Conférences internationales 176
10 Workshops nationaux et internationaux 177
11 Publications nationales 177
12 Communications orales 178
13 Posters 178
14 Visites scientifiques et Collaboration 179

Table des figures

1 Évolution de la puissance de pointe des CPU Intel et GPU Nvidia [100] . 3

I.1 Répartition des transistors dans les architectures CPU et GPU 8
I.2 Architecture générale d'un processeur graphique 10
I.3 Évolution de la bande passante CPU Intel et GPU Nvidia [45] 13
I.4 Hiérarchie des threads CUDA 15
I.5 Hiérarchie de mémoires sous CUDA 16
I.6 Étapes d'exécution d'un programme GPU sous CUDA 18
I.7 Hiérarchie de mémoires sous OpenCL 20
I.8 Aperçu du support StarPU 22
I.9 Aperçu du support SWARM [38] 26
I.10 SWARM au présent et au futur [38] 27
I.11 (a). Classification foie (b). Détection tête de fœtaux (c). Détection des bords [15] 31
I.12 Détection et suivi de visages multiples en temps réel sur GPU [91] 34
I.13 Marquage des points repères de vertèbres C3 à C7 40
I.14 Illustration de la méthode de segmentation des vertèbres 42
I.15 Vue d'ensemble de ImageCycle [119] 44
I.16 VideoCycle : Navigation dans des bases de séquences vidéo [126]. 45
I.17 Estimation du mouvement de la caméra 47
I.18 Estimation du mouvement de la caméra avec objets statiques [136]. . . . 48
I.19 Estimation du mouvement de la caméra avec objets mobiles [136]. 48

II.1 Modèle de traitement d'images sur GPU avec CUDA et OpenGL. 52
II.2 Fusion d'accès à la mémoire graphique 54
II.3 Otpimisations GPU proposées pour le traitement d'image unique 54
II.4 Traitement d'images multiples avec quatre CUDA streams 56
II.5 Accélération GPU *vs* nombre de CUDA streams 56
II.6 Transformations géométriques sur processeur graphique 58
II.7 Masques de lissage 59
II.8 Détection des points d'intérêt (coins) sur GPU 62
II.9 Détection GPU des contours basée sur le principe de Deriche-Canny . . 65
II.10 Résultats qualitatifs de détection des coins et contours 66
II.11 Gains, en temps, obtenus grâce à la visualisation OpenGL. GPU FX4800 68
II.12 Gains obtenus par l'utilisation de la mémoire partagée et de textures. GPU FX4800 69
II.13 Gains obtenus par à l'utilisation de la technique de streaming CUDA. GPU FX4800 71

III.1 Schéma proposé du traitement hétérogène d'images multiples 78
III.2 Optimisation du traitement hétérogène d'images multiples 82
III.3 Gains, en temps, obtenus grâce au streaming CUDA 87

IV.1 Etapes de soustraction du background sur GPU d'une vidéo de N (1. . N) trames 92
IV.2 Soustraction de l'arrière plan sur processeur graphique 93
IV.3 Etapes d'extraction de silhouettes sur GPU 94
IV.4 Extraction de silhouettes sur GPU 95
IV.5 Performances de soustraction d'arrière-plan (background) sur GPU . . . 96
IV.6 Performances d'extraction de silhouettes sur GPU 96
IV.7 Étapes d'implémentation pyramidale de la mesure du flot optique sur GPU103
IV.8 Vecteurs de flot optique extraits à partir des points d'intérêt 105
IV.9 Évolution du temps de calcul du flot optique par rapport au nombre d'itérations 107
IV.10 Évolution du temps de calcul du flot optique par rapport à la taille de la fenêtre 108
IV.11 Évolution du temps de calcul du flot optique par rapport au nombre de niveaux de pyramides 109

IV.12 Performances de détection des points d'intérêt sur processeurs graphiques multiples . . . 110
IV.13 Performances de calcul du flot optique (Lucas-Kanade) sur processeurs graphiques multiples . . . 111
IV.14 Performances (en fps) des implémentations GPU sur vidéos Full HD et 4K 112
IV.15 Répatition des temps pris par chaque étape des implémentations proposées sur une vidéo Full HD . . . 113
IV.16 Répatition des temps pris par chaque étape des implémentations proposées sur une vidéo Full 4K . . . 113

V.1 Traitement CPU/GPU d'image individuelle . . . 121
V.2 Acquisition directe de flux vidéo sur GPU avec les cartes de captures SDI [99] . . . 124
V.3 Modèle de traitement d'objets multimédias sur architectures parallèles et hétérogènes . . . 126
V.4 Calcul Multi-CPU/Multi-GPU pour la détection des vertèbres . . . 127
V.5 Réultats qualitatifs de détection des verèbres . . . 129
V.6 Calcul hétérogène pour l'indexation de vidéos . . . 132
V.7 Performances du calcul hétérogène appliqué à l'indexation de vidéos (VideoCycle) . . . 133
V.8 Processus de détection de mouvements avec caméra mobile sur GPU . . . 135
V.9 Estimation et élimination du mouvement de la caméra . . . 137
V.10 Estimation et élimination du mouvement de la caméra . . . 138

A.1 (a). Gradients de l'image. (b). Descripteurs de points clés [69] . . . 170
A.2 Correspondance d'objets avec les descripteurs SIFT [95] . . . 171
A.3 Estimation par blocs du vecteur de mouvement entre deux images successives [130] . . . 172
A.4 Algorithme de recherche en 3 pas [130] . . . 174

Liste des tableaux

I.1 Comparaion de caractéristiques entre CPU Intel I7-980x [57] et GPU GTX 580 [27] 9

I.2 Propriétés de mémoires des processeurs graphiques (G80, GT200, GF100) 11

II.1 Optimisation des traitements GPU d'images uniques et multiples 57

II.2 Détection des coins et contours d'image unique (Vis OpenCV) : GPU FX4800 67

II.3 Détection des coins et contours d'image unique (Vis OpenCV) : GPU Tesla M2070 67

II.4 Détection des coins et contours d'image unique (Vis OpenGL) : GPU GPU FX4800 68

II.5 Détection des coins et contours d'image unique (Vis OpenGL) : GPU Tesla M2070 68

II.6 Détection des coins et contours d'image unique (Mem. tex et Par) : GPU FX4800 69

II.7 Détection des coins et contours d'image unique (Mem. tex et Par) : Tesla M2070 69

II.8 Détection des coins et contours d'images multiples (Mem. tex et Par) : GPU FX4800 70

II.9 Détection de coins et contours d'images multiples (Mem. tex et Par) : GPU Tesla M2070 70

II.10 Détection de coins et contours d'images multiples (CUDA streaming) : FX4800 70

II.11 Détection des coins et contours d'images multiples (Streaming) : GPU Tesla M2070 70

III.1 Détection des coins et contours d'images multiples : ordonnancement statique 85
III.2 Détection des coins et contours d'images multiples : ordonnancement dynamique 86
III.3 Détection des coins et contours d'images multiples : CUDA streaming (4 streams) 86
III.4 Recommandations d'optimisation du traitement d'images multiples . . . 88

V.1 Évolution des accélérations en fonction des paramètres de complexité estimés 118
V.2 Évolution des accélérations en fonction des paramètres de complexité estimés 120
V.3 Étapes de traitements d'images et de vidéos à partir du modèle proposé 125
V.4 Estimation de la complexité temporelle des étapes de détection des vertèbres 128
V.5 Performances du calcul hétérogène et d'optimisations appliquées 130
V.6 Détection des vertèbres sur plates-formes Multi-CPU/Multi-GPU pour un ensemble de 200 images (1476×1680) 131
V.7 Performances GPU de la détection de mouvements à partir d'une caméra mobile 137

Listings

III.1 Chargement des images d'entrée 76
III.2 La codelet StarPU 76
III.3 Soumission des tâches StarPU à l'ensemble des images 77
III.4 Modèle de performances utilisé pour l'ordonnancement 80
III.5 fonction CPU de détection des coins et contours 83
III.6 fonction GPU de détection des coins et contours 84

Acronymes

ACP	Analyse en Composante Principale
ALU	Arithmetic Logic Unit
API	Application Programming Interface
APU	Accelerated Processing Unit
ASM	Active Shape Model
BLAS	Basic Linear Algebra Subprograms
CT	Computed Tomography Scan
CUBLAS	CUDA Basic Linear Algebra Subroutines
CUDA	Compute Unified Device Architecture
dmda	deque model data aware
DSM	Distributed Shared Memory
ETI	ET International, Inc
FBP	Filtered Back Projection
FFT	Fast Fourier transform
fps	frames per second
Full HD	Full High Definition

GHz	Gigahertz
GL	OpenGL Library
GLAUX	Auxiliary Library
GLTk	OpenGL Tk Library
GLU	OpenGL Utility Library
GLUT	OpenGL Utility Toolkit
GLX	OpenGL extension to X-Windows
GPGPU	General Purpose Graphic Processing Unit
GPU	Graphic Processing Unit
HD	High Definition
IRM	Imagerie par Résonance Magnétique
KLT	Kanade Lucas Tomasi feature tracker
OpenCL	Open Computing Language
OpenCV	Open Source Computer Vision
OpenGL	Open Graphic Language
ROI	Region Of Interest
SAD	Sum of Absolute Difference
SCALE	SWARM Codelet Association LanguagE
SDI	Serial Digital Interface
SFU	Unité Fonctionnelle Spéciale
SIFT	Scale-Invariant Feature Transform
SIMD	Single Instruction Stream-Multiple Data Stream
SM	Streaming Multi-processor
SMP	Symmetric Multi Processing
SP	Streaming Processor
SSD	Sum of Squared Difference
SWARM	Swift Adaptative Runtime Machine

TclTk Tool Command Language Toolkit
TPC Thread Processing Cluster

VCM Vector Coherence Mapping
VTK Visualization Tool Kit

Introduction

Gordon Moore est l'un des fondateurs principaux d'Intel, Ce dernier a prédit en 1965 que le nombre de transistors qu'on serait capable de placer sur un circuit électronique à prix constant doublerait tous les 2 ans. Ce postulat, connu sous le nom de « loi de Moore » a été remarquablement bien vérifié depuis lors. Et grâce à l'augmentation simultanée de la fréquence de fonctionnement des circuits électroniques, la puissance des processeurs a effectivement doublé tous les 18 mois. Mais depuis quelques années, cette fréquence s'est retrouvée plafonnée à environ 4 Gigahertz (GHz), pour des raisons thermiques, de consommation énergétique et de coût. Par conséquent, les industriels ont proposé de nouvelles solutions qui ont permis de contourner cette limitation. Ils ont notamment préconisé un changement des architectures internes en multipliant les unités de calcul, ou cœurs, intégrés dans les processeurs. L'objectif de ces nouvelles architectures multicœurs est de multiplier le nombre d'unités de traitement indépendantes partageant la mémoire centrale (RAM CPU), permettant ainsi de faire tourner plusieurs processus légers (threads) simultanément.

Dans ces nouvelles architectures, on trouve les processeurs graphiques, appelés Graphic Processing Unit (GPU), qui possèdent nativement une structure de cœurs massivement parallèle. Ces GPU qui équipent les ordinateurs personnels, étaient dévolus à la base pour des traitements spécialisés au rendu d'images 2D/3D, aux applications graphiques et jeux vidéo. Les processeurs graphiques sont dédiés pour des traitements synchrones de grosses quantités de données. À la différence des CPU multicœurs, les cœurs des GPU fonctionnent de manière synchrone en appliquant simultanément la même opération sur des données

multiples (Single Instruction Stream-Multiple Data Stream (SIMD)) dans leur propre espace de mémoire graphique (RAM GPU). La tendance actuelle se retrouve aussi dans la convergence de ces deux types d'unité de calcul, avec les tout récents processeurs accélérés, appelés Accelerated Processing Unit (APU), combinant CPU et GPU sur la même puce, partageant le même espace mémoire [2] et permettant un accès plus rapide aux données en mémoire. Les ordinateurs présentent aujourd'hui une nouvelle architecture complexe et hétérogène (ou hybride) associant des processeurs centraux multicœurs à des processeurs graphiques de plus en plus aptes à exécuter des calculs génériques en parallèle.

Lors des dernières années, l'architecture des GPU a significativement évolué et leur puissance de calcul brute a largement supplanté celle des meilleurs CPU, tel que montré à la figure 1. Avec une telle puissance de calcul, disponible dans la plupart des ordinateurs à un prix modique, il eut été dommage de ne réserver cette puissance qu'à l'unique utilisation de jeux vidéo et au rendu d'images 2D/3D.

Avec l'ouverture des interfaces de programmation, appelés Application Programming Interface (API), des GPU, de nombreux chercheurs ont entrepris d'exploiter les processeurs graphiques pour accélérer les traitements, habituellement destinés aux processeurs centraux. C'est ce qu'on appelle le General Purpose Graphic Processing Unit (GPGPU), ou la programmation parallèle générique des processeurs graphiques. Les calculs peuvent donc être portés sur GPU, mais le mode de fonctionnement SIMD de ces derniers impose la nécessité d'une grande quantité de données à traiter afin de les exploiter efficacement. Les GPU tirent leur force du recouvrement des threads inactifs (en attente de données localisées dans des grandes mémoires à forte latence) par l'exécution d'autres threads actifs (dont les données sont localisées dans des mémoires réduites mais à faible latence). S'il n'y a pas suffisamment de données, les threads seront en nombre insuffisant pour assurer un fonctionnement à plein régime du processeur graphique, et celui-ci verra ses performances se dégrader.

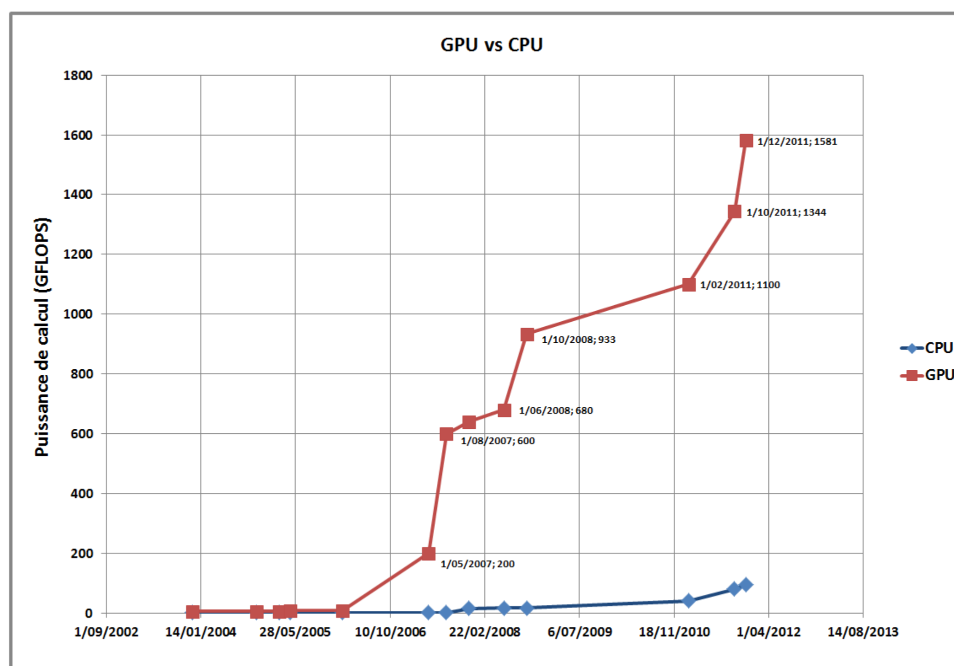


Figure 1 – Évolution de la puissance de pointe des CPU Intel et GPU Nvidia [100]

L'exploitation des architectures parallèles de processeurs graphiques n'est pas aisée, de nombreuses questions et contraintes sont liées à la gestion des différents types de mémoire d'un GPU. De manière générale, les principaux points qui doivent être traités sont : la distribution des données entre CPU et GPU, la synchronisation des threads GPU, l'optimisation et la réduction des transferts de données entre les différents types de mémoire, le respect des contraintes de capacité des différents espaces de mémoire, l'ordonnancement efficace des tâches lors du lancement des calculs sur des machines multicœurs hétérogènes, etc.

Les algorithmes de visualisation et de traitement d'objets multimédias sont des grands consommateurs de puissance de calcul et de mémoire, et en particulier lors de l'utilisation de gros volumes d'images ou de vidéos de haute définition (High Definition (HD) ou Full High Definition (Full HD)). En outre, les méthodes de traitement et d'analyse de vidéos en temps réel requièrent un calcul très rapide afin d'assurer un traitement de 25 images par seconde (frames per second (fps)). Ceci nécessite de traiter chaque image en moins de 40 millisecondes. Dans le domaine du traitement d'images, l'étude d'algorithmes parallèles date de plusieurs dizaines d'années. Il existe des librairies telles que Visualization Tool Kit (VTK) exhibant un parallélisme statique de données. Cependant, l'évolution rapide

des architectures parallèles nécessite une adaptation régulière des algorithmes pour pouvoir profiter pleinement de ces plates-formes. Les grilles ainsi que les clusters de calcul ont été également exploitées pour la parallélisation et l'accélération de ces algorithmes, mais avec un coût relativement élevé. L'arrivée de l'API Compute Unified Device Architecture (CUDA) [98] ainsi que d'autres langages tels que Open Computing Language (OpenCL) [63] ou encore Ati streams [1] ont permis une exploitation plus efficace des GPU, avec un coût modéré puisque les processeurs graphiques équipent différents types de machines : ordinateurs portables, ordinateurs de bureau, clusters, etc.

Notre contribution porte sur le développement d'approches efficaces tant sur le choix et la parallélisation d'algorithmes de traitement d'objets multimédias, que sur l'adaptation de ces algorithmes pour exploiter au mieux la puissance de calcul des nouvelles architectures parallèles (GPU) et hétérogènes (Multi-CPU/Multi-GPU). Ces algorithmes doivent être adaptés pour exploiter au mieux la puissance de ces nouvelles architectures, et aussi pour aider aux choix des ressources (CPU et/ou GPU) à utiliser, selon la nature des calculs et des données à traiter. Certains points sont cruciaux et difficiles à maitriser tels que la complexité des machines parallèles et hétérogènes, la gestion complexe des données dans différents types de mémoire, le choix efficace des ressources selon le type et la complexité estimée d'algorithmes ainsi que les stratégies d'ordonnancement employées. Notre contribution porte également sur l'accélération des méthodes de traitement d'objets multimédias, telles que les algorithmes de traitements d'images, les applications médicales comme la détection et segmentation des vertèbres, les algorithmes d'indexation et de navigation dans des bases d'objets multimédias (images et de vidéos), ainsi que les méthodes de traitement, d'analyse et de suivi de mouvements en temps réel.

Organisation du document

Ce rapport est réparti en cinq chapitres :

Chapitre 1

Le premier chapitre décrit les concepts généraux du calcul GPU et de son utilité pour les algorithmes de traitement d'objets multimédias. Dans ce contexte, nous introduisons en premier lieu l'architecture des processeurs graphiques ainsi que les outils de la programmation GPU. Ensuite, nous présentons les techniques d'exploitation des architectures multicœurs hétérogènes (Multi-CPU/Multi-GPU). La dernière partie du chapitre est dédiée à la présentation de l'état de l'art des méthodes de traitement d'images et de vidéos

sur GPU. Nous donnons aussi des exemples d'applications intensives traitant des objets multimédias, et qui peuvent bénéficier de la puissance de calcul des architectures parallèles (GPU) et hétérogènes (Multi-CPU/Multi-GPU).

Chapitre 2

Nous proposons dans le deuxième chapitre un schéma de développement pour le traitement parallèle d'images sur processeurs graphiques. Ce schéma s'appuie sur l'API CUDA pour les traitements parallèles et la bibliothèque graphique Open Graphic Language (OpenGL) [103] pour la visualisation des résultats. À partir de ce schéma, nous présentons l'implémentation efficace de différents algorithmes de traitement d'images tels que l'extraction efficace des contours basée sur la méthode Deriche-Canny [34], ainsi que la détection des points d'intérêt basée sur la technique décrite par Bouguet [17] et basée sur le principe de Harris [49]. Nous illustrons aussi les différentes techniques d'optimisation appliquées sur les traitements GPU d'images. Ces optimisations permettent de réduire significativement les temps de transfert de données entre la mémoire du CPU et celle du GPU. Elles permettent également une exploitation pleine des unités de calcul du processeur graphique.

Chapitre 3

Le troisième chapitre est dévolu au traitement efficace d'images multiples sur architectures multicœurs hétérogènes (Multi-CPU/Multi-GPU). Nous proposons un schéma de développement basé sur l'API CUDA pour les traitements parallèles, et sur le support d'exécution StarPU [8] pour l'exploitation des unités de calcul hybrides. À partir de ce schéma, nous proposons l'implémentation hétérogène des algorithmes de détection des contours (Deriche-Canny) et de points d'intérêt (Bouguet et Harris) appliquées sur des grandes bases d'images de haute définition (HD/Full HD). La dernière partie de ce chapitre décrit les différentes optimisations appliquées aux traitements hybrides tels que l'ordonnancement dynamique des tâches ainsi que le recouvrement des transferts de données par les kernels (fonctions GPU) exécutés sur GPU multiples.

Chapitre 4

Le quatrième chapitre contribue à la mise en œuvre sur un ou plusieurs GPU d'algorithmes de traitement de vidéos de haute définition (HD/Full HD) en temps réel. Nous développons sur GPU différents algorithmes d'analyse et de suivi de mouvements tels que la soustraction de l'arrière-plan (background), l'extraction de silhouettes et l'estimation du

mouvement basée sur le calcul de vecteurs de flot optique. Notons que ces implémentations sont basées sur un schéma de développement s'appuyant sur CUDA pour les traitements GPU ou Multi-GPU parallèles et OpenGL pour la visualisation des résultats. Ce chapitre décrit également les optimisations CUDA proposées offrant une exploitation efficace des processeurs graphiques multiples.

Chapitre 5

Le dernier chapitre est dédié à la description du modèle proposé pour le traitement d'objets multimédias de haute définition (image unique, images multiples, vidéos multiples, vidéo en temps réel). Ce modèle est basé sur l'exploitation de l'intégralité de la puissance des machines hétérogènes (Multi-CPU/Multi-GPU). Il permet de choisir les ressources à utiliser (CPU et/ou GPU) ainsi que les méthodes à appliquer selon la nature des médias à traiter et la complexité des algorithmes, tout en assurant un ordonnancement efficace des tâches. Nous décrivons ensuite les résultats expérimentaux obtenus en appliquant ce modèle aux méthodes implémentées dans les chapitres précédents, et utilisant différents types de médias (image, vidéo, etc.). Nous présentons aussi l'intérêt de l'exploitation du modèle proposé dans trois cas d'utilisation concrets. Le premier est une application médicale de segmentation des vertèbres [77], le deuxième est une application de navigation dans des bases de données multimédias [119]. Enfin, le troisième cas d'application du modèle est une méthode de détection de mouvements en temps réel lors de l'utilisation d'une caméra mobile [136]. Les résultats obtenus montrent une accélération globale allant d'un facteur de 10 à 50 par rapport à une implémentation séquentielle sur CPU.

Finalement, la dernière section est consacrée à la conclusion et aux perspectives.

Chapitre I

État de l'art

Sommaire

1 **Présentation des GPU** 8
1.1 Architecture des GPU 8
1.2 Programmation des GPU 12
2 **Exploitation des architectures multicœurs hétérogènes** 21
3 **Traitement d'objets multimédias sur GPU** 29
3.1 Traitement d'images sur GPU 29
3.2 Traitement de vidéos sur GPU 32
3.3 Estimation de complexité de traitement d'images sur GPU . . . 35
4 **Exemples de traitements intensifs d'objets multimédias** . . . 38
4.1 Segmentation des vertèbres 38
4.2 Indexation de bases de données multimédias 43
4.3 Détection de mouvements à partir de caméra mobile 46

Dans ce chapitre, nous décrivons les concepts généraux du calcul sur GPU et de son utilité pour les algorithmes de traitement d'objets multimédias. Nous introduisons en premier lieu l'architecture des processeurs graphiques ainsi que les langages de programmation GPU. Ensuite, nous présentons un état des lieux des outils d'exploitation des architectures multicœurs hétérogènes (Multi-CPU/Multi-GPU). En outre, nous décrivons un état de l'art des méthodes de traitement d'images et d'analyse de vidéos (détection et suivi de mouvements) sur GPU, ainsi que les techniques d'estimation de complexité de calcul de ces méthodes sur GPU. La dernière partie du chapitre est dédiée à la présentation d'exemples d'applications intensives traitant des objets multimédias, et pouvant bénéficier de la puissance de calcul des architectures parallèles (GPU) et hétérogènes (Multi-CPU/Multi-GPU).

1 Présentation des GPU

Les processeurs graphiques équipent la majorité des ordinateurs personnels actuels. Ces GPU, conçus à la base pour accélérer les applications graphiques (rendu d'images 2D/3D, jeux vidéo, etc.) présentent un grand nombre d'unités de calcul, fournissant une puissance de calcul largement supérieure à celle des CPU. Cette puissance a suscité l'intérêt de nombreux chercheurs pour exploiter les GPU afin de réaliser d'autres types de traitements effectués jusqu'alors sur CPU. Nous commençons par présenter succintement l'architecture des processeurs graphiques pour ensuite aborder les principaux langages de programmation GPU.

1.1 Architecture des GPU

Les processeurs graphiques présentent une architecture hautement parallèle, fournissant une haute puissance de calcul, qui peut dépasser largement celle des processeurs centraux. La figure I.1 illustre la répartition des transistors entre les deux architectures (CPU et GPU). D'une part, les processeurs centraux disposent de peu d'unités arithmétiques de calcul (Arithmetic Logic Unit (ALU)), d'une large mémoire cache ainsi que d'une grande unité de contrôle. Les CPU sont donc spécialisés dans le traitement de tâches différentes utilisant des gros volumes de données. L'unité de contrôle permet de gérer le flux d'instructions afin de maximiser l'occupation des unités de calcul, et d'optimiser la gestion de la mémoire cache.

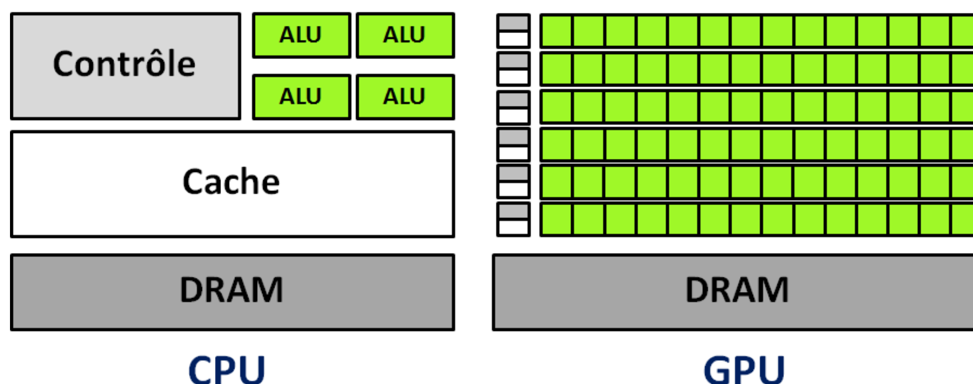


Figure I.1 – Répartition des transistors dans les architectures CPU et GPU

D'autre part, les processeurs graphiques sont composés d'un grand nombre d'unités de calcul avec un espace de mémoire cache limité et peu d'unités de contrôle. Les unités de

calcul sont destinées aux calculs massivement parallèles, permettant d'exécuter simultanément la même instruction sur une grande quantité de données (selon le paradigme SIMD). Le tableau I.1 présente une comparaison des caractéristiques entre deux processeurs (CPU Intel I7-980x [57] et GPU GTX 580 [27]) de nouvelle génération, dont nous disposons dans notre laboratoire. Notons que les GPU bénéficient d'une bande passante largement supérieure à celle des CPU. Cependant, les processeurs graphiques consomment plus d'énergie que les processeurs centraux. Un calcul accéléré sur GPU peut recouvrir cette consommation d'énergie grâce à l'exploitation des unités de calcul en parallèle.

Processeurs	**CPU Intel I7-980x**		**GPU GTX 580**
Nombre de cœurs	Cœurs	Threads	512
	6	12	
Mémoire RAM	24 Go		1.5 Go
Mémoire Cache	12 Mo		816 Ko
Energie	130 watts		244 watts
Bande passante	25.6 GB/s		192.4 GB/s

Tableau I.1 – Comparaion de caractéristiques entre CPU Intel I7-980x [57] et GPU GTX 580 [27]

La figure I.2 présente l'architecture générale des GPU. Ils sont découpés en multiprocesseurs, appelés Streaming Multi-processor (SM), composés eux-mêmes de 8, 16 ou 32 cœurs, appelés Streaming Processor (SP). Chaque cœur SP exécute une seule tâche dans un mode SIMD avec l'unité d'instructions permettant de distribuer les tâches aux différents cœurs. Chaque cœur dispose d'une unité arithmétique effectuant des calculs en simple ou double précision. De plus, chaque SM dispose d'une Unité Fonctionnelle Spéciale (SFU), permettant d'exécuter des opérations plus complexes telles que sinus, cosinus, racine carrée.

Les SM contiennent aussi d'autres ressources, soit la mémoire partagée et le « fichier de registres ». Les groupes de SM appartiennent aux clusters de traitement, appelés Thread Processing Cluster (TPC). Ceux-ci contiennent aussi des mémoires caches et des textures qui sont partagées entre les SM. L'architecture d'un processeur graphique est donc composée d'un ensemble de clusters de traitement (TPC), d'un réseau d'interconnexion et d'un système de mémoires.

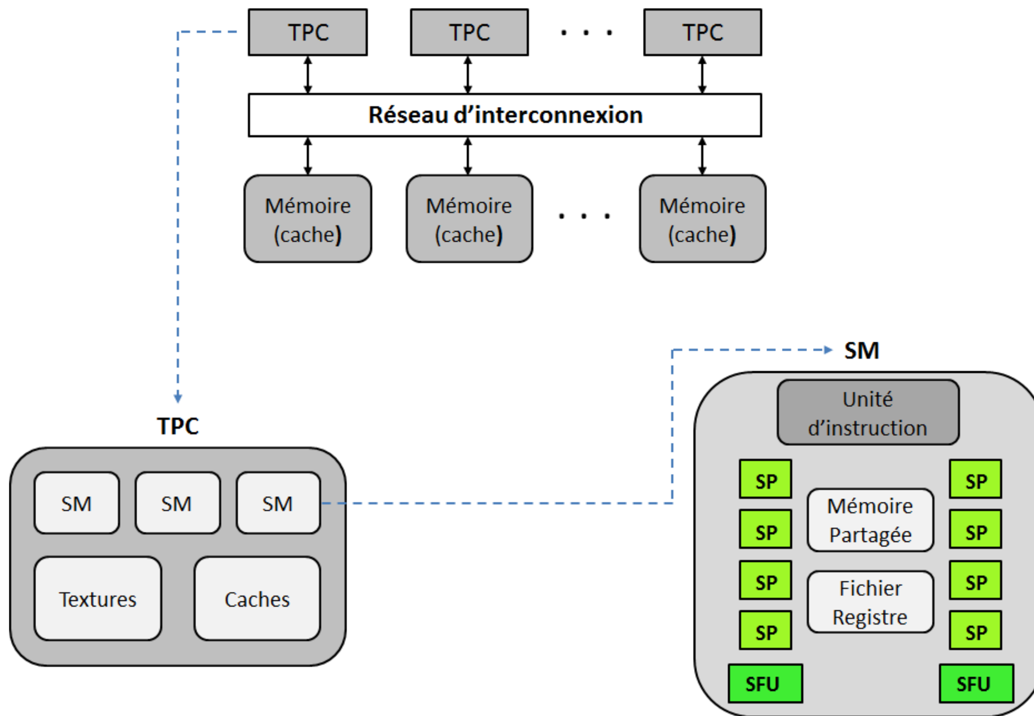


Figure I.2 – Architecture générale d'un processeur graphique

La structure de la mémoire GPU peut être décrite par six zones hiérarchiques. On distingue notamment : la mémoire globale, la mémoire partagée, la mémoire constante, la mémoire de texture, les registres et la mémoire locale (tableau I.2).

1. **La mémoire globale :** correspond à la mémoire de masse du processeur graphique, accessible par le CPU. La capacité de cette mémoire varie de quelques centaines de Mo à quelques Go (en fonction du type de la carte graphique). Cependant, sa latence d'accès aux données est élevée, soit de 400 à 600 cycles d'horloge.
2. **La mémoire partagée :** elle se retrouve à l'intérieur de chaque multi-processeur SM, permettant de partager les données entre les cœurs GPU (SP). L'accès à cette mémoire est très rapide, soit environ 4 cycles, mais sa taille est limitée à environ 48 ko seulement (16 ko pour les G80 [121] et GT200 [26], 48 Ko pour les GF100 [25]).
3. **La mémoire constante :** elle est localisée dans les multi-processeurs SM, accessible en écriture par le CPU et en lecture par le GPU. La lecture depuis cette mémoire ne coûte qu'un cycle, sa taille est limitée à 64 ko par GPU et à 8ko par SM.

Mémires GPU	Utilité	Latence	Taille/Nombre		
			G80	**GT200**	**GF100**
Registres (Nbr)	par thread	1 cycle	8192	16384	32768
Partagée	communication	4 cycles	16 Ko	16 Ko	16/48 Ko
Constante	lecture seule	1 à 4	64 Ko	64 Ko	64 Ko
Texture	traitement d'images	1 à 4 cycles	64 unités	64 unités	64 unités
Globale	mémoire principale	400 à 600	512 Mo	1 Go	1.5 Go
Cache	accès rapide	1 à 4	/ /	/ /	L1 : 16/48 Ko L2 : 768 Ko
Locale	complète les registres	400 à 600	définie lors de la compilation		

Tableau I.2 – Propriétés de mémoires des processeurs graphiques (G80, GT200, GF100)

4. **La mémoire de texture :** tout comme la mémoire constante, elle est accessible en écriture par le CPU et en lecture par le GPU. Le coût de la lecture est également très faible (4 cycles). Les textures sont d'une grande utilité pour les algorithmes de traitement d'images, permettant la mise en œuvre de filtrage bilinéaires [21] et tri-linéaires [120] et assurant un accès accéléré aux pixels.
5. **Les registres :** ils représentent des zones mémoires très rapides (y accéder coûte généralement un seul cycle d'horloge), mais leur taille est très limitée. Chaque cœur dispose d'un certain nombre de registres qui lui sont réservés.
6. **La mémoire locale :** c'est une mémoire privée propre à chaque cœur GPU. Elle vient compléter les registres automatiquement si ceux-ci sont en nombre insuffisant. Elle a une latence très élevée par rapport aux registres (400 à 600 cycles d'horloge).

Les nouvelles cartes graphiques disposent également d'une mémoire cache à un ou deux niveaux. En effet, les cartes graphiques de la firme GF100 (Fermi) [25], développés depuis 2010, disposent de deux niveaux de cache (L1 et L2) fournissant un espace plus grand. Toutefois, les GPU des générations précédentes (G80 [121] et GT200 [26]) ne disposent pas d'espace cache. Le tableau I.2 décrit les propriétés des différentes mémoires présentes dans les cartes graphiques de différentes générations (G80, GT200, GF100). Les tailles données sont des ordres de grandeur, car elles peuvent varier selon le modèle de la carte graphique.

Les processeurs graphiques bénéficient également de leur large bande passante tel que montré dans la figure I.3. Un développeur CUDA doit gérer efficacement les mémoires et threads GPU, afin de pouvoir atteindre la bande passante maximale des GPU.

1.2 Programmation des GPU

Comme évoqué dans la section précédente, les GPU possèdent une architecture hautement parallèle. Cependant, jusqu'au début des années 2000, les GPU n'étaient pas facilement programmables. Ils ne pouvaient qu'exécuter une suite fixe de traitements appliqués aux objets géométriques pour obtenir des rendus 2D ou 3D. Cette suite de traitements appelée « *pipeline graphique* » était figée, seules quelques expériences marginales utilisant certaines fonctions câblées du hardware furent menées et permirent d'accélérer quelques tâches simples avec ce qu'on appelle les Shaders [4].

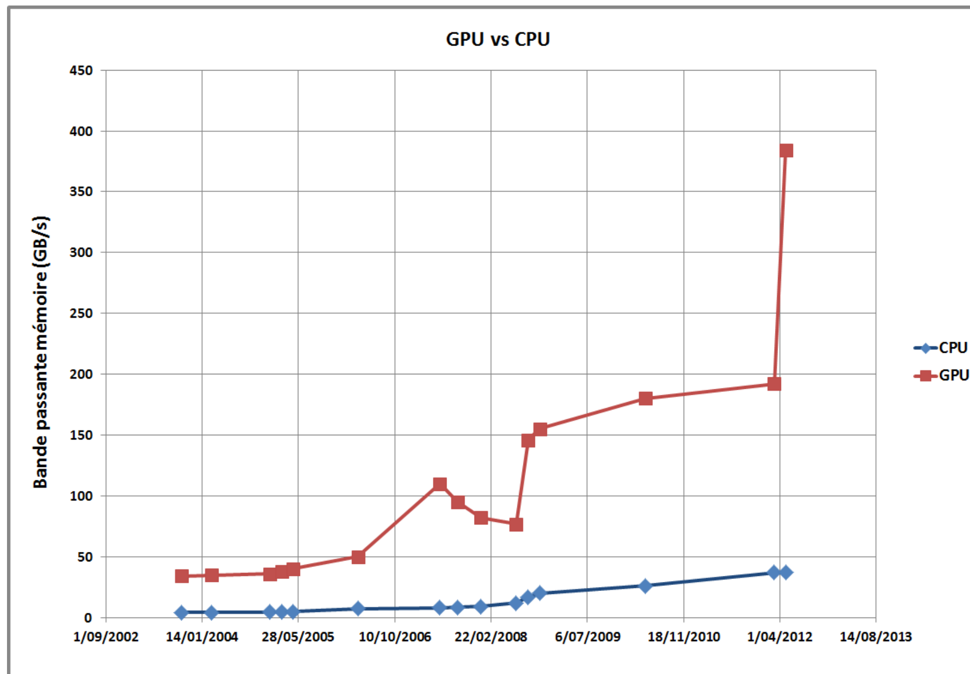


Figure I.3 – Évolution de la bande passante CPU Intel et GPU Nvidia [45]

Les Shaders sont des programmes qui permettent de paramétrer une partie du pipeline graphique. Apparus dans les années 90, ils ont commencé à être beaucoup plus utilisés à partir de 2001. Depuis, ils n'ont pas cessé d'évoluer et d'offrir de plus en plus de possibilités aux programmeurs grâce à des langages permettant leur programmation. On peut citer principalement OpenGL [103] et Direct3D [16] qui sont de plus en plus utilisés dans la création d'applications générant des images 3D. Les dernières années ont été marquées par l'arrivée des langages destinés plus particulièrement à la programmation sur GPU (GPGPU). On peut citer essentiellement CUDA [98] et OpenCL [63] qui sont les plus utilisés actuellement. Nous présentons ci-dessous une description succinte de chacun des environnements utilisés dans nos applications sur GPU : OpenGL, CUDA et OpenCL.

1.2.1 OpenGL

OpenGL (Open Graphic Library) [62, 103] est une bibliothèque graphique ouverte, développée depuis 1992 par Silicon Graphics [117], une des entreprises leaders de l'informatique graphique. Le point fort d'OpenGL est sa compatibilité avec les constructeurs de cartes graphiques grand public. L'aspect ouvert d'OpenGL lui a permis d'être portée sur des plates-formes et systèmes d'exploitation variés. OpenGL est également reconnue comme

un standard puisqu'elle est utilisée dans la plupart des logiciels de synthèse d'images, ainsi que dans le domaine des jeux vidéo 3D. L'objectif majeur d'OpenGL est de construire et de visualiser des scènes 2D/3D (images de synthèse, photos, etc.) sur écran. Elle permet aussi une interaction temps réel avec l'utilisateur, ce qui veut dire qu'un programme OpenGL permet à l'utilisateur d'agir au moyen des périphériques d'entrée (clavier, souris, etc.) sur la scène (déplacement des objets, changement des points de vue de la caméra, etc.), et d'en visualiser l'effet immédiatement. OpenGL est souvent préconisée dans les milieux scientifiques et académiques du fait de son ouverture, de sa souplesse d'utilisation et de sa disponibilité sur des plates-formes variées. Dans ce travail, nous utilisons OpenGL pour la visualisation des images ou vidéos résultantes après application des traitements parallèles sur la carte graphique.

OpenGL se base pour son fonctionnement sur plusieurs modules : OpenGL Library (GL), OpenGL Utility Library (GLU), OpenGL extension to X-Windows (GLX), Auxiliary Library (GLAUX), OpenGL Tk Library (GLTk), OpenGL Utility Toolkit (GLUT).

1. GL : il propose les fonctions d'affichage de base d'OpenGL. Son préfixe d'utilisation est : gl ;
2. GLU : il propose des commandes de bas niveau de transformations géométriques, triangulation des polygones, rendu des surfaces, etc. Son préfixe d'utilisation est : glu ;
3. GLX : il permet d'associer OpenGL avec X-windows pour l'interface utilisateur. Son préfixe d'utilisation est : glx ;
4. GLAUX : il permet de simplifier la programmation d'applications de gestion de fenêtres d'affichage, de la souris et du clavier, etc. Son préfixe d'utilisation est : aux ;
5. GLTk : équivalent a Auxiliary Library mais avec l'utilisation de Tool Command Language Toolkit (TclTk) [132] pour l'interface graphique. Son préfixe d'utilisation est : tk ;
6. GLUT : il permet d'utiliser des fonctions plus complexes : multifenêtrage, menus popup, etc. Son préfixe d'utilisation est : glut.

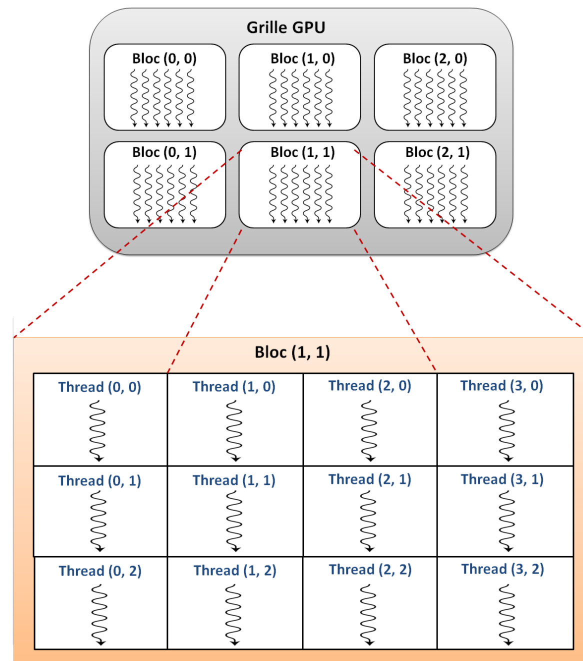


Figure I.4 – Hiérarchie des threads CUDA

1.2.2 CUDA

L'API CUDA représente l'une des technologies bas niveau les plus utilisées pour la programmation parallèle sur GPU. CUDA a été développé par la firme nVidia en 2007, permettant de programmer les processeurs graphiques issus de cette firme dans un langage proche du C standard.

Du point de vue utilisateur, chaque programme CUDA se compose de deux parties, un code CPU faisant appel au GPU, et un code CUDA, appelé kernel, exécuté en parallèle sur GPU. Les fonctions CUDA permettent au programmeur de définir le nombre de threads qui exécutent le kernel sur GPU. Toutefois, le nombre de threads peut largement dépasser le nombre d'unités de traitement. Ce paradigme permet de considérer le GPU comme un dispositif abstrait fournissant un grand nombre de ressources virtuelles. CUDA permet ainsi de masquer la latence élevée de la mémoire globale par une communication très rapide entre threads. Les threads exécutés sur GPU sont regroupés en blocs, l'ensemble des blocs représente la grille de calcul du processeur graphique. Chaque thread peut être exclusivement identifié par ses coordonnées dans un bloc, et par les coordonnées de ce bloc à l'intérieur de la grille (figure I.4). Les threads d'un même bloc peuvent coopérer entre

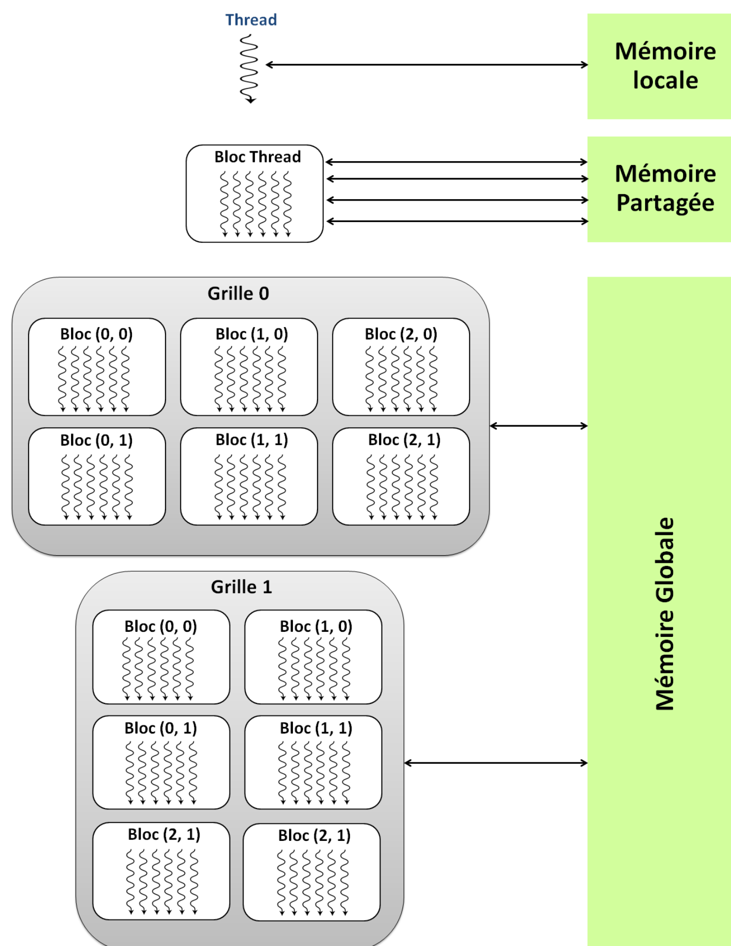


Figure I.5 – Hiérarchie de mémoires sous CUDA

eux *via* la mémoire partagée. Ils peuvent aussi synchroniser leur exécution afin de coordonner leurs accès à la mémoire. La figure I.5 montre les différentes zones hiérarchiques de mémoires GPU vues par un développeur CUDA.

L'exécution d'un programme sur GPU sous CUDA requiert cinq étapes principales (figure I.6) : sélection du processeur graphique, allocation de mémoire sur GPU, transfert de données vers la mémoire GPU, exécution des fonctions GPU (kernels), transfert des résultats vers la mémoire hôte (CPU).

1. **Sélection du processeur graphique :** les traitements GPU requis par un processeur central sont affectés par défaut à un processeur graphique (généralement le premier). Toutefois, l'utilisateur peut choisir de changer de processeur s'il en possède plusieurs. Cette sélection peut se faire *via* la fonction « cudaSetDevice » ;

2. **Allocation de mémoire sur GPU :** avant de lancer les traitements sur GPU, il faut allouer la mémoire nécessaire aux calculs sur processeur graphique (données d'entrée et de sortie). L'allocation de mémoire sur GPU peut se faire avec la fonction « cudaMalloc » ;

3. **Transfert de données vers la mémoire GPU :** afin d'effectuer les traitements sur GPU, les données doivent être transférées depuis la mémoire hôte (CPU) vers la mémoire graphique. Ce transfert peut se faire *via* la fonction « cudaMemcpyHostToDevice ». Il existe également d'autres types de copies qui peuvent être effectuées par les fonctions suivantes :
   – **cudaMemcpyHostToHost :** permet de copier les données depuis un espace de mémoire hôte vers un autre espace de mémoire hôte (cette fonction correspond à memcpy) ;
   – **cudaMemcpyDeviceToHost :** permet de copier les données depuis la mémoire graphique vers la mémoire hôte ;
   – **cudaMemcpyDeviceToDevice :** permet de copier les données depuis un espace de mémoire graphique vers un autre espace de cette mémoire graphique.

4. **Exécution des fonctions GPU (kernels) :** le processeur graphique fonctionne en mode SIMD, il est conçu pour permettre à tous les cœurs d'exécuter la même fonction simultanément sur des données différentes. La fonction GPU exécutée par chaque cœur est appelée « kernel ». Lors de l'appel de ce kernel, le développeur doit indiquer le nombre de threads qui seront créés, qui exécuteront ce même kernel. Les threads sont gérés par les contrôleurs des multiprocesseurs par paquets appelés « warp ». Un warp CUDA est donc représenté par un groupe de 32 threads, il s'agit de la taille minimale de données traitées de façon SIMD par un multiprocesseur en CUDA. L'utilisateur peut également regrouper les threads au sein de blocs, le nombre de threads par bloc dépend du type de la carte graphique et de l'application implémentée.

   La répartition de la mémoire partagée est définie par la taille du bloc, les données stockées dans cette mémoire sont accessibles par tous les threads d'un même bloc. En effet, l'utilisateur doit définir le nombre de blocs ainsi que le nombre de threads lors de l'appel d'un kernel. Lors de l'exécution, les blocs sont assignés aux SM, ils

seront décomposés en warps qui seront exécutés par les SP. Lorsqu'un warp devient inactif (accès mémoire, synchronisation, etc.), il sera interchangé avec un autre warp en attente d'exécution. Ce mécanisme permet de recouvrir les temps d'inactivité de certains warps par les temps d'activités d'autre warps ;

5. **Transfert des résultats vers la mémoire hôte :** afin de récupérer les résultats des traitements effectués sur GPU, il faut faire un transfert des résultats depuis la mémoire graphique vers la mémoire hôte. Cette copie est effectuée *via* la fonction « cudaMemcpyDeviceToHost ». Enfin, toutes les ressources allouées sur GPU doivent être libérées par la fonction « cudaFree ».

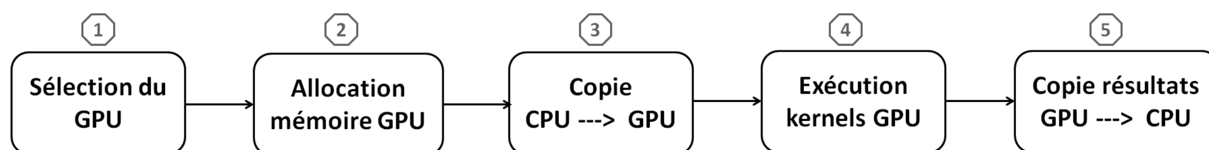


Figure I.6 – Étapes d'exécution d'un programme GPU sous CUDA

1.2.3 OpenCL

OpenCL (Open Computing Language) se veut au GPGPU ce qu'OpenGL est au rendu 3D, à savoir un standard ouvert. OpenCL se compose d'une API et d'un langage de programmation dérivé du C. Son but est de faciliter la programmation d'applications utilisant les divers processeurs disponibles dans les machines : les CPU, les GPU, mais aussi d'autres types de processeurs éventuels.

Le projet de création d'OpenCL a été initié par Apple, qui en a ensuite confié la gestion au *Khronos Group* [47]. Le projet a depuis été rejoint par les principaux acteurs industriels du secteur : AMD, nVidia et Intel. On peut donc s'attendre à ce qu'OpenCL devienne un standard très utilisé. La première version est disponible depuis Avril 2010. OpenCL bénéficie essentiellement de son indépendance vis-à-vis des API 3D, sa compatibilité avec les cartes de tous les constructeurs ainsi que sa portabilité sur différentes plates-formes.

OpenCL peut être présenté par trois couches principales : la spécification de langage, la couche plateforme API et le runtime API.

– **La spécification de langage :** décrit la syntaxe et l'interface de programmation permettant d'écrire des kernels fonctionnant sur différents GPU (ATI, nVidia) et CPU. Cette interface de programmation multi plates-formes est basée sur le langage C, en raison de sa large diffusion avec la communauté des développeurs. Une précision numérique IEEE 754 [131] a été définie afin d'obtenir des résultats cohérents lors des calculs à virgule flottante sur différentes plates-formes.
– **La couche plateforme API :** permet l'accès à des routines différentes pour effectuer des requêtes sur les unités de calcul disponibles. Le développeur peut également initialiser et choisir les ressources nécessaires à son calcul. En effet, la création des contextes, des files d'attente et des requêtes est effectuée au niveau de cette couche.
– **Le runtime API :** permet d'exécuter les kernels en assurant une gestion efficace du calcul et des ressources mémoire dans le système OpenCL.

Par ailleurs, OpenCL est conçu pour cibler non seulement les GPU mais également les CPU multicœurs. De ce fait, les kernels de calcul peuvent être affectés à des tâches parallèles, correspondants à l'architecture des GPU ou à celle des CPU. L'exécution d'un kernel est définie par un domaine à N dimensions. Chaque élément dans le domaine d'exécution est un *work-item* (équivalent à un thread avec CUDA). Ces *work-item* peuvent être regroupés en *work-group* pour une meilleure communication et synchronisation.

De la même manière que CUDA, OpenCL définit quatre espaces de mémoire différents sur GPU : privée, locale, constante et globale (figure I.7).

1. **La mémoire privée :** elle ne peut être utilisée que par une seule unité de calcul. Cette mémoire est équivalente aux registres dans le cadre de la programmation CUDA.
2. **La mémoire locale :** elle peut être utilisée (partagée) par les *work-item* d'un *work-group*. Cette mémoire est équivalente à la mémoire partagée dans le cadre de la programmation CUDA.
3. **La mémoire constante :** elle peut être utilisée pour stocker les données afin d'avoir un accès en lecture seule par toutes les unités de calcul du GPU. L'allocation et l'initialisation de cette mémoire se fait au niveau du processeur hôte, tandis que la lecture se fait depuis le processeur graphique.
4. **La mémoire globale :** elle peut être utilisée par toutes les unités de calcul disponibles dans la carte graphique.

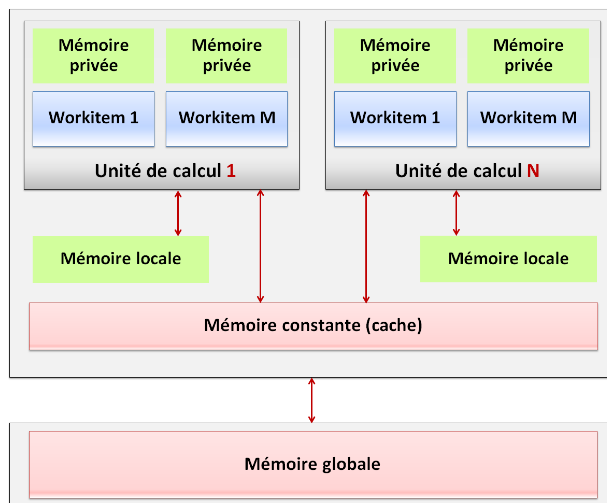


Figure I.7 – Hiérarchie de mémoires sous OpenCL

Dans ce travail, nous utilisons principalement CUDA puisqu'il représente la technologie la plus aboutie actuellement. En effet, les nouvelles versions de CUDA offrent la possibilité d'exploiter efficacement les processeurs graphiques en lançant un grand nombre de threads simultanément, et en assurant un accès plus rapide aux données sur GPU. L'API CUDA bénéficie également de sa grande utilisation par les développeurs du domaine suite à son lancement en 2007, deux ans avant la sortie d'OpenCL. Toutefois, ce dernier permet de programmer tout type de GPU programmable et non seulement les cartes nVidia. Malgré cela, OpenCL n'est pas encore capable d'atteindre les mêmes performances que CUDA. En effet, OpenCL gère également les ressources hétérogènes, ce qui l'empêche d'être aussi performant que CUDA. Par ailleurs, la compatibilité de CUDA avec OpenGL nous a poussé à proposer une visualisation OpenGL des images offrant une présentation rapide des résultats.

2 Exploitation des architectures multicœurs hétérogènes

Les processeurs graphiques permettent de fournir une solution très efficace pour l'accélération des applications à calcul intensif. Toutefois, cette solution peut être améliorée par l'exploitation simultanée des cœurs CPU et GPU multiples dont disposent la majorité des ordinateurs d'aujourd'hui. Dans ce contexte, il existe différents travaux permettant de faciliter l'exploitation des plates-formes multicœurs et hétérogènes, tels que : StarPU, StarSS, OpenACC, Swift Adaptative Runtime Machine (SWARM) et Grand Central Dispatch.

1. **StarPU**

   La librairie StarPU [8], développée à l'INRIA Bordeaux Sud-Ouest(France) offre un support exécutif unifié pour exploiter les architectures multicœurs hétérogènes, tout en s'affranchissant des difficultés liées à la gestion des transferts de données. En outre, StarPU propose plusieurs stratégies d'ordonnancement efficaces et offre la possibilité d'en concevoir aisément de nouvelles. StarPU se base pour son fonctionnement sur deux structures principales : la codelet et les tâches.

   – **La codelet :** permet de préciser sur quelles architectures le noyau de calcul peut s'effectuer, ainsi que les implémentations associées ;
   – **Les tâches :** elles consistent à appliquer la codelet sur l'ensemble des données, en utilisant la stratégie d'ordonnancement sélectionnée.

   Le cheminement d'une tâche, depuis sa soumission jusqu'à la notification de sa terminaison à l'application, peut être résumé en cinq étapes :

   (a) **Soumission des tâches :** StarPU reçoit toutes les tâches soumises à partir de l'application ;
   (b) **Choix des ressources de calcul :** l'ordonnanceur de StarPU distribue les tâches selon la politique d'ordonnancement sélectionnée. Si la tâche a été attribuée à un GPU, la description de la tâche est donc envoyée au pilote associé à ce GPU (chaque unité de calcul est associée à un pilote spécifique). Dès que la tâche est terminée, le pilote réclame une nouvelle tâche à l'ordonnanceur ;
   (c) **Chargement des données :** pour les tâches attribuées au GPU, les données doivent être transférées vers la carte graphique. StarPU permet de masquer ces

transferts puisqu'il dispose d'une bibliothèque qui met en œuvre une mémoire partagée virtuelle, appelée Distributed Shared Memory (DSM). La gestion des mouvements de données est donc transparente pour le programmeur ;

(d) **Calcul StarPU :** une fois les données chargées, les tâches peuvent être exécutées sur les ressources hétérogènes de calcul indiquées dans la codelet ;

(e) **Libération des ressources :** une fois la tâche terminée, StarPU peut le notifier à l'application afin de pouvoir lancer une autre tâche. Les éventuelles tâches qui dépendaient des résultats de celle qui vient d'être effectuée sont alors lancées par l'ordonnanceur.

Le but de StarPU est de maximiser l'utilisation des unités de calcul (CPU, GPU NVIDIA ou ATI, accélérateurs Cell) tout en minimisant les coûts de transfert de données. En effet, StarPU utilise les implémentations multiples (séquentielles sur CPU, parallèles sur GPU) d'une méthode pour exploiter aux mieux toutes les ressources de calcul disponibles (figure I.8).

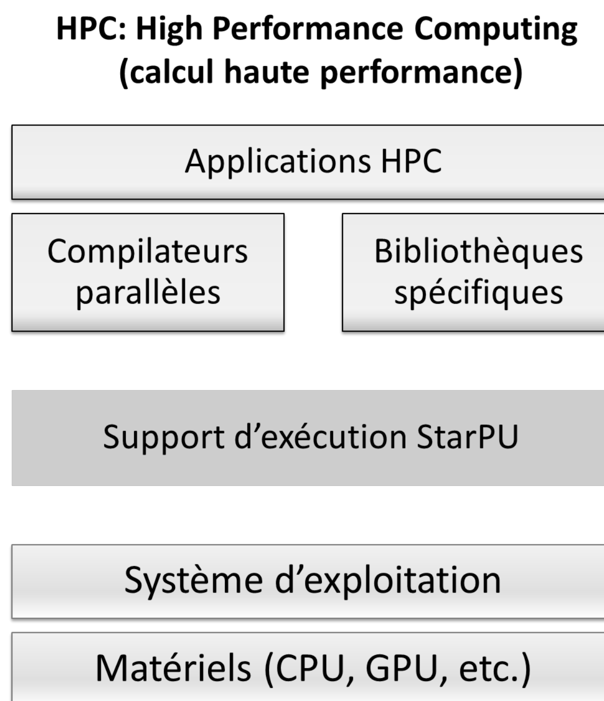


Figure I.8 – Aperçu du support StarPU

2. **StarSs**

   StarSs [11] a été développé à l'université de Catalunia (Barcelone. Espagne). Il fournit un modèle de programmation flexible pour les architectures multicœurs, en assurant une portabilité des applications ainsi qu'une amélioration des performances. Il est composé de six composantes principales : CellSs, SPMSs, GPUSs, ClearSpeedSs, ClusterSs et GridSs.

   (a) **CellSs :** permet une exploitation automatique du parallélisme fonctionnel à partir d'un programme séquentiel à travers les éléments de traitement de l'architecture Cell BE [12], en s'appuyant sur un modèle de programmation simple et flexible. En effet, à partir d'une simple annotation du code source, le compilateur de CellSs génère le code nécessaire. Ensuite, le support d'exécution de CellSs exploite les unités de calcul en parallèle par la construction d'un graphe de dépendances de tâches lors de l'exécution ;

   (b) **SPMSs :** permet d'exploiter de manière automatique et parallèle les cœurs multiples des processeurs Symmetric Multi Processing (SMP) à partir d'un code séquentiel écrit par le programmeur. SPMSs se base également sur un support d'exécution permettant de construire le graphe de dépendances de tâches ainsi que leur ordonnancement ;

   (c) **ClearSpeedSs :** permet aussi une exploitation automatique et parallèle des cœurs multiples des architectures ClearSpeed, assurant une gestion et un ordonnancement efficace des tâches ;

   (d) **GPUSs :** permet l'exploitation de processeurs graphiques (GPU) multiples. GPUSs gère les processeurs graphiques ainsi que leurs espaces de mémoire de façon efficace, tout en assurant la simplicité et la portabilité des implémentations ;

   (e) **ClusterSs :** Cluster Superscalar (ClusterSs) [128] permet l'exploitation parallèle des serveurs présents dans les clusters. Les tâches sont créées de manière asynchrone et sont assignées aux ressources disponibles à l'aide du support IBM APGAS, qui fournit une couche de communication efficace et portable ;

   (f) **GridSs :** permet une exploitation parallèle des ressources de calcul disponibles dans une grille ;

3. **OpenACC**

   OpenACC [20] présente un nouveau standard de programmation multiplateforme permettant aux développeurs C et Fortran de bénéficier de la haute puissance de calcul des nouvelles machines hétérogènes disposant de processeurs multicœurs et d'accélérateurs tels que les GPU. OpenACC, considéré comme une API de haut niveau, est supporté par les leaders de l'industrie parallèle : CAPS entreprise [19], CRAY Inc [28], The Portland Group Inc (PGI) [48] et NVIDIA.

   OpenACC présente une collection de directives de compilation pour la spécification des boucles ainsi que d'autres régions du code développé en C, C++ ou Fortran. Ces directives servent à porter les codes CPU vers les accélérateurs. Ils permettent ainsi de créer des programmes accélérés sans avoir besoin d'initialiser explicitement les accélérateurs ni de transférer les données entre les deux types de machines (CPU et accélérateurs). Le lancement et l'arrêt des accélérateurs est également pris en charge par OpenACC. Le modèle de programmation d'OpenACC permet de fournir l'information nécessaire aux compilateurs, y compris la spécification de données locales d'un accélérateur, la gestion des boucles ainsi que d'autres détails liés aux performances de calcul. Nous décrivons brièvement les modèles d'exécution et de mémoire d'OpenACC.

   (a) **Modèle d'exécution d'OpenACC**

      Le modèle d'exécution de l'API OpenACC est dirigé par l'accélérateur attaché. Une partie (séquentielle) de l'application utilisateur est exécutée sur la machine hôte (CPU), tandis que les parties ayant un calcul intensif sont exécutées sur l'accélérateur sous contrôle de l'hôte. Les unités de calcul attachées (accélérateurs) exécutent les régions parallélisables telles que les boucles ou les noyaux de calcul intensif. Pour les tâches affectées aux accélérateurs, la machine hôte doit également gérer l'exécution en assurant l'allocation de mémoire des machines attachées, l'initialisation et le transfert de données, l'envoi du code vers l'accélérateur, le passage d'arguments, la gestion de la file d'attente (des tâches à exécuter sur l'accélérateur), le transfert des résultats vers la mémoire hôte ainsi que la libération des espaces de mémoire. Généralement, la machine hôte peut mettre en attente une séquence d'opérations à exécuter sur l'accélérateur.

(b) **Modèle mémoire d'OpenACC**

Lors du lancement des calculs exploitant les accélérateurs, tous les mouvements de données doivent être effectués par la machine hôte. Ces mouvements de données peuvent occuper une partie considérable du code utilisateur telles que rencontré sous CUDA ou OpenCL. OpenACC permet à l'utilisateur de réduire significativement le code en gérant ces transferts *via* les directives fournies au compilateur. Toutefois l'utilisateur doit toujours prendre en considération l'aspect des mémoires séparées (CPU, accélérateurs) pour deux raisons majeures :

- La bande passante entre la mémoire hôte et celle de l'accélérateur permet de déterminer le niveau d'intensité de calcul exigé afin d'accélérer de façon efficace une région du code ;
- La taille limitée de l'espace mémoire des accélérateurs peut contraindre le fonctionnement d'une application en raison de l'impossibilité de copier l'intégralité des données requises.

Toutefois, les accélérateurs tels que les processeurs graphiques ne permettent pas souvent de gérer la mémoire de manière automatique. En effet, ils n'assurent pas une cohérence de mémoire (synchronisation) entre les opérations exécutées *via* des unités de calcul différentes. De même, lors de l'utilisation d'une seule unité de calcul, la synchronisation n'est garantie que lorsque les opérations de gestion de mémoire sont séparées explicitement. Certains accélérateurs actuels possèdent une mémoire cache qui peut être gérée soit au niveau matériel soit au niveau logiciel (dépend de l'accélérateur en question). Pour les outils de programmation bas niveau tels que CUDA et OpenCL, c'est à l'utilisateur de gérer ses espaces de mémoire cache. Pour l'API OpenACC, ces espaces sont gérés par le compilateur *via* des directives fournies par l'utilisateur.

OpenACC offre en effet une exploitation simplifiée et transparente des accélérateurs grâce à l'utilisation des directives. Par ailleurs, il n'offre pas les mêmes avantages si l'on désire exploiter des ressources hétérogènes. L'utilisateur devra s'occuper de la distribution et de l'ordonnancement des tâches lancées.

4. **SWARM**

SWARM [38] présente une technologie d'accélération de performances des systèmes multicœurs/multi-nœuds. Il fournit un système d'exécution dynamique et adapté permettant de minimiser l'intervention de l'utilisateur aux tâches de parallélisation et de gestion des systèmes hétérogènes. SWARM est conçu pour programmer sur des architectures multicœurs hétérogènes en utilisant un modèle d'exécution basé sur un ordonnancement dynamique.

Le modèle d'exécution de SWARM est basé sur un graphe de tâches, ou codelets. Chaque codelet représente une petite unité de travail qui ne nécessite pas de grande latence mémoire ni d'opérations bloquantes (*i.e.* accès à la mémoire distante). Les arcs liant les codelets dans le graphe représentent les dépendances entre données qui doivent être respectées avant l'exécution de la codelet, en imposant un ordre séquentiel minimal entre eux (figure I.9).

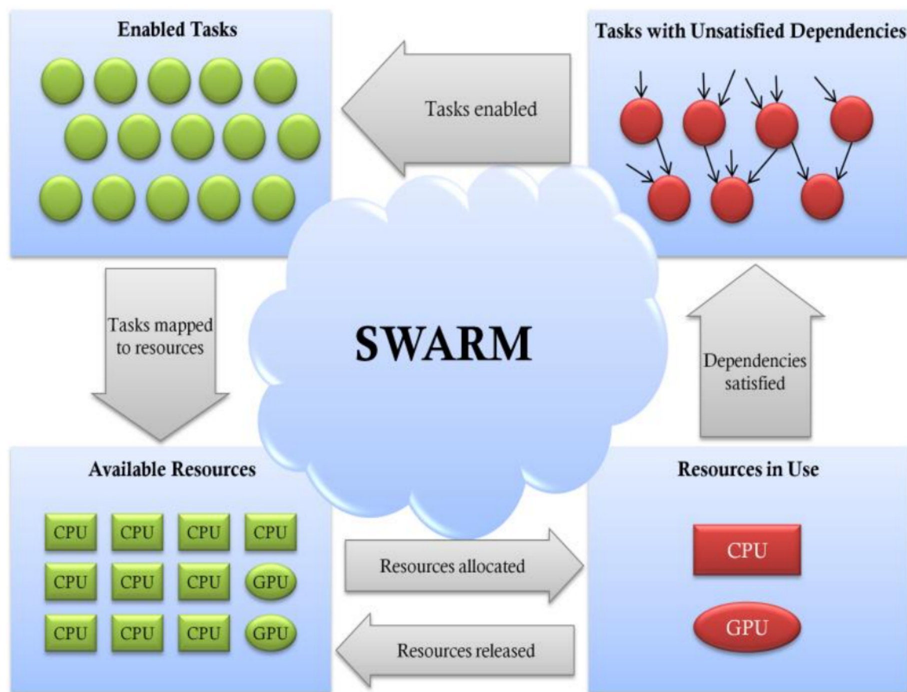


Figure I.9 – Aperçu du support SWARM [38]

Les opérations exigeant une forte latence peuvent être décomposées , de manière automatique, en phases non bloquantes et asynchrones, permettant à chaque code-

let d'être compressée en moins de temps et en utilisant peu de mémoire. SWARM présente en effet un système offrant une mise en œuvre d'applications à partir des codelets. Il propose en outre la possibilité de suivre automatiquement les dépendances liées à chaque codelet afin de les ordonnancer correctement. Aujourd'hui, les utilisateurs de SWARM peuvent exploiter les accélérateurs en codant avec une extension de C appelée SWARM Codelet Association LanguagE (SCALE), qui simplifie la création des supports d'exécution sur des plates-formes multi-coeurs.

Comme futur objectif, ET International, Inc (ETI) est en train de développer de nouvelles fonctionnalités pour SWARM comme une intégration plus étroite des réseaux, le support des architectures de GPU et d'Intel ainsi que les outils de partitionnement de mémoire. En outre, ETI envisage de créer un langage de haut niveau permettant de profiter pleinement de SWARM en simplifiant l'expression des modèles de conception parallèles et distribués. Par conséquent, cette approche offrira la possibilité de développer plus rapidement des programmes hautement parallèles et qui peuvent profiter non seulement des plates-formes multicœurs homogènes d'aujourd'hui, mais aussi des architectures multicœurs hétérogènes de demain.

Cependant, malgré ses caractéristiques intéressantes, SWARM ne représente qu'une première version d'un support d'exploitation des architectures multicœurs hétérogènes. En effet, une grande partie des fonctionnalités citées ci-dessous sont encore en développement ou en phase d'amélioration (figure I.10). SWARM fut lancée très récemment (fin 2011).

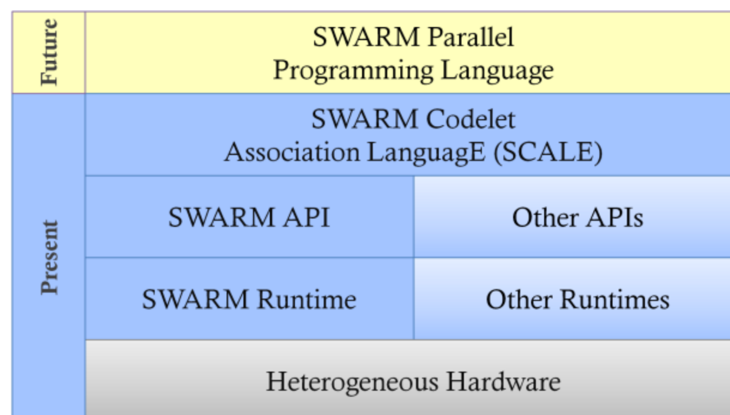


Figure I.10 – SWARM au présent et au futur [38]

5. **Grand Central Dispatch**

   Grand Central Dispatch [5] est une technologie développée par Apple pour mieux exploiter les processeurs multicœurs et distribués (différentes machines) dans les plates-formes Mac (systèmes iOS et OS X). Cette nouvelle architecture permet aux développeurs d'utiliser pleinement la puissance de calcul des processeurs multicœurs. Elle travaille en distribuant de manière efficace les traitements (processus) aux différents cœurs. En effet, Grand Central Dispatch permet de mieux répondre aux besoins des applications en cours d'exécution, et de les adapter aux ressources systèmes disponibles, de façon équilibrée. Le code de Grand Central Dispatch est ouvert depuis septembre 2009.

Une comparaison entre ces technologies d'exploitation de plates-formes mulicœurs (StarPU, StarSs, OpenACC, SWARM et Grand Central Dispatch) nous permet de distinguer que StarPU est la plus prometteuse grâce à son exploitation de l'intégralité des ressources hétérogènes de calcul (Multi-CPU/Multi-GPU), à l'inverse de StarSs et Grand Central Dispatch qui ne permettent qu'une exploitation distincte des cœurs CPU ou GPU multiples. OpenACC et SWARM présentent des nouvelles techniques proposant des solutions efficaces mais limitées à certains cas précis (boucles, itérations, etc.). En outre, StarPU offre la possibilité de concevoir de nouvelles stratégies d'ordonnancement. Ceci permet d'intégrer des paramètres supplémentaires (nombre d'opérations, dépendance de tâches, etc.) afin d'avoir une meilleure affectation des ressources (CPU ou GPU) aux tâches.

3 Traitement d'objets multimédias sur GPU

La majorité des méthodes de traitement d'objets multimédias, et en particulier les algorithmes de traitement d'images, disposent de sections de traitements identiques entre les pixels. Ce type de calcul se prête bien à une parallélisation GPU permettant d'exploiter les unités de traitement (cœurs GPU). Ceci est particulièrement important dans les applications de traitement de grandes bases d'images, et également dans les applications de traitement des vidéos de haute définition (HD ou Full HD) en temps réel. Cependant, une exploitation efficace des processeurs graphiques nécessite une bonne sélection des algorithmes à paralléliser en appliquant des techniques d'estimation de leur complexité temporelle. Nous présentons dans cette section un état de l'art des méthodes permettant d'accélérer ces deux types d'applications (traitement d'images et de vidéos) sur des plates-formes parallèles telles que les GPU. Nous présentons ensuite un état des lieux des techniques d'estimation de complexité de calcul des méthodes de traitement d'images et de vidéos sur GPU.

3.1 Traitement d'images sur GPU

Les algorithmes de traitement d'images représentent des outils nécessaires à de nombreux domaines de vision par ordinateur tels que la vidéosurveillance, l'imagerie médicale, la reconnaissance de formes, etc. On trouve ainsi de nombreuse méthodes de traitement d'images classiques telles que les transformées de Fourier, de Hough, des ondelettes [58, 108], les détecteurs de coins [49] et de contours [18], l'Analyse en Composante Principale (ACP) [59], etc. Cependant, ces méthodes sont entravées par leur consommation de calcul et de mémoire, qui augmente significativement lors du traitement de gros volumes d'images. Depuis l'arrivée des technologies de programmation GPU telles que CUDA et OpenCL, on trouve de plus en plus de travaux permettant l'accélération des calculs dans ce domaine, exploitant la puissance des processeurs graphiques.

Dans ce contexte, Castaneda *et al.* [32] ont présenté une évaluation détaillée des implémentations de différents algorithmes sur GPU telles que la transformée de Fourier, les filtres linéaires et non linéaires, ou encore l'ACP. Ces implémentations basées sur CUDA et Cg [89] ont permis d'accélérer les versions CPU d'un facteur de 10 en moyenne (2,4 secondes sur GPU au lieu de 25 secondes sur CPU). En outre, Yang *et al.* [134] ont mis en œuvre plusieurs algorithmes classiques de traitement d'images sur GPU avec CUDA,

tels que l'égalisation d'histogramme, la suppression des bruits et la transformée de Fourier. Ces implémentations ont permis d'avoir des facteurs d'accélération de l'ordre de 70 (0,4 secondes sur GPU au lieu de 31 secondes sur CPU pour la suppression des bruits d'images). Cependant, les temps de transfert de données entre les mémoires CPU et GPU n'ont pas été pris en compte. Luo *et al.* ont proposé une implémentation GPU de la méthode d'extraction de contours [72] basée sur le détecteur de Canny [18]. Cette mise en œuvre était également intégrée sous MATLAB.

On trouve aussi dans le projet OpenVIDIA [42] une collection d'implémentations d'algorithmes de vision par ordinateur qui peuvent être exécutés sur un ou plusieurs processeurs graphiques, utilisant OpenGL [103], Cg [89] et CUDA [98]. Le projet OpenVIDIA exploite la puissance totale des GPU pour avoir des performances largement supérieures à celles obtenues sur CPU. Il permet aussi de libérer le processeur central pour effectuer d'autres tâches au-delà de la vision par ordinateur. La bibliothèque gpuCV [40] propose un ensemble de méthodes de traitement d'images sur GPU. Elle se distingue par sa compatibilité avec Open Source Computer Vision (OpenCV) [102] (bibliothèque de traitement d'images sur CPU). Le but de gpuCV est de permettre un portage aisé d'applications écrites avec OpenCV. GpuCV peut être utilisé de la même façon qu'OpenCV, profitant d'une accélération GPU sans devoir s'occuper des différentes configurations nécessaires. Cette bibliothèque est entravée par le nombre limité de fonctions gpuCV, et par l'impossibilité d'accélérer les performances du code gpuCV *via* de nouvelles techniques d'optimisation sur GPU.

Dans la littérature, il existe également de nombreux travaux de traitement d'images sous OpenCL. Wenyu Zhang [135] a proposé une étude évaluant l'implémentation OpenCL (par rapport à CUDA) de la méthode de reconstruction d'images tomographiques Computed Tomography Scan (CT) [39], *via* la technique de rétroprojection filtrée, appelée Filtered Back Projection (FBP). Cette étude propose également la mise en œuvre (sous OpenCL) de l'approche de détection de textures de Haralick [22]. Antao *et al.* [113] ont présenté de leur côté la mise en œuvre d'algorithmes de traitements linéaires d'images sur GPU sous OpenCL, montrant une exploitation efficace des architectures SIMD. Kim *et al.* [64] ont développé un modèle d'exploitation de GPU multiples (sous OpenCL) à partir d'un programme écrit initialement pour l'utilisation d'un seul processeur graphique. Ce modèle permet de considérer les mémoires de GPU multiples comme une seule unité de stockage,

en assurant la cohérence des différentes zones de mémoire disponibles.

D'autre part, les applications médicales sont connues par leur forte intensité de calcul due au traitement de grands nombres d'images de haute définition HD, et aussi à la nature des images bruitées qui nécesitent plus de traitements. Ainsi, Il existe de nombreuses implémentations GPU dédiées au monde médical, permettant notamment le calcul du rendu d'images volumineuses [50, 123], ainsi que des reconstructions d'images IRM (Imagerie par Résonance Magnétique (IRM)) [115]. Brickbek *et al.* [15] ont développé une méthode GPU de détection des poses 3D dans des images médicales (figure I.11). Cette méthode permet d'accélérer d'un facteur de 80 le processus de classification et de segmentation de foie dans des images IRM par rapport à une implémentation CPU. Une accélération de 30 a été également obtenue pour la localisation des têtes de fœtaux dans les images IRM.

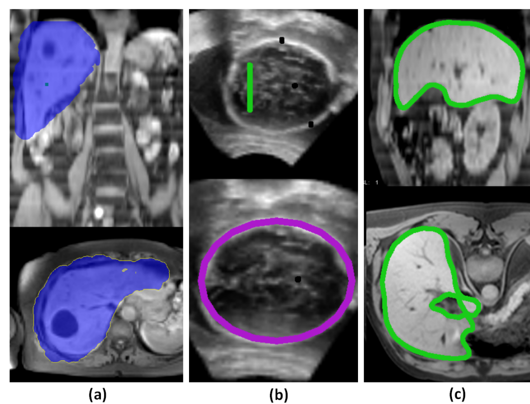


Figure I.11 – (a). Classification foie (b). Détection tête de fœtaux (c). Détection des bords [15]

Cependant, ces accélérations peuvent être insuffisantes lors du traitement de grandes bases d'images ayant des résolutions différentes. En effet, le GPU ne peut pas être plus efficace dans tous les cas (les petites images ainsi que les simples algorithmes ne requièrent pas beaucoup de calcul), ce qui implique une nécessité de bien analyser les complexités spatiale et temporelle de calcul des méthodes avant de les paralléliser sur GPU.

Notre contribution porte sur l'analyse et l'estimation de complexité des différents algorithmes appliqués à l'aide de plusieurs paramètres afin d'affecter les GPU aux algorithmes appropriés. Nous contribuons également par l'exploitation efficace des architectures multicœurs hétérogènes, en proposant des optimisations offrant une exploitation optimale des ressources disponibles. Cette contribution est décrite en détail dans les troisième et cinquième chapitres.

3.2 Traitement de vidéos sur GPU

Les méthodes de vision par ordinateur, et en particulier les techniques de traitement de vidéos, sont basées sur des algorithmes d'analyse d'images, puisqu'une vidéo est toujours représentée par une succession de trames. Cependant, les méthodes de traitement de vidéos exigent généralement de respecter les contraintes de temps réel. Ceci veut dire que chaque trame (image) de la séquence vidéo doit être traitée très rapidement (en moins de 40 millisecondes) afin d'assurer une vitesse de défilement suffisante (typiquement 25 trames visualisées par seconde) pour la perception humaine. Dans ce domaine, on trouve de nombreuses méthodes nécessitant un traitement en temps réel telles que les applications d'étude de comportements, de détection, reconnaissance et suivi d'objets. Ces méthodes proposent souvent des solutions basées sur des approches de suivi et d'analyse de mouvements utilisant les vecteurs du flot optique [33], les descripteurs Scale-Invariant Feature Transform (SIFT) [68], etc.

Les algorithmes de suivi de mouvements présentent aujourd'hui une thématique de recherche très active dans les domaines de vision par ordinateur et de vidéosurveillance. Les méthodes de suivi de mouvements peuvent être décrites par l'estimation des déplacements et de vitesses des caractéristiques (coins, formes, etc.) d'une image (trame) de la vidéo par rapport à la trame précédente. Ils présentent ainsi des outils nécessaires à de nombreuses applications telles que l'analyse des mouvements d'humains, la détection d'événements, l'indexation de séquences vidéo. Ces méthodes s'appuient sur différents algorithmes tels que l'estimation du flot optique [33], les descripteurs SIFT [68], la mise en correspondance de blocs (block matching) [14]. Dans ce travail, nous nous concentrons principalement sur les méthodes d'estimation de mouvements basées sur le calcul du flot optique. En effet, elles offrent une solution pertinente pour le suivi d'objets en mouvement (personnes, voitures, etc.) même lors de l'utilisation de vidéos de foules bruitées. Les vecteurs du flot optique présentent une distribution de vitesses des mouvements apparents dans une image, fournissant une information importante des déplacements spatiaux des objets dans la scène. Plusieurs approches s'appuient sur la mesure du flot optique telles que l'estimation de vitesses de véhicules [56] à partir de vidéos capturées *via* une caméra fixe. Dans ce cas, le mouvement des véhicules est détecté puis suivi le long des trames en utilisant les mesures du flot optique à travers la technique Horn & Shunk [52]. Andrade *el al.* [3] ont développé une méthode de modélisation de mouvements normaux afin de détecter tout évènement anormal. Cette solution applique premièrement un calcul des vecteurs du flot optique, qui seront regroupés

en segments *via* l'ACP. Une similarité entre ces segments est calculée à partir des observations des chaînes de Markov cachées [133]. Ces observations serviront à définir un seuil de détection des évènements anormaux avant de le comparer aux différentes observations des exemples de scénarios. Par ailleurs, Kalal *et al.* [61] ont proposé une méthode de détection et de suivi de visages basée sur l'approche TLD (Tracking-Learning-Detection *i.e.* Suivi-Apprentissage-Detection), tandis que [29] ont développé quelques techniques de suivi de mouvements à partir de caméras multiples. On trouve également dans [55] des travaux de détection d'évènements anormaux dans la foule à partir de l'analyse des mouvements au lieu de les suivre un par un.

Cependant, ces méthodes deviennent très consommatrice (en temps et en mémoire) lors du traitement de vidéos de haute résolution (HD ou Full HD), et par conséquent, il devient très difficile de respecter la contrainte du temps réel. Les processeurs graphiques représentent une solution prometteuse pour contourner cette limitation, en exploitant leur grande puissance de calcul. On trouve ainsi deux catégories de travaux d'analyse et de suivi de mouvements à partir des mesures du flot optique. La première concerne le calcul du flot optique dense qui permet de suivre tous les pixels de trames sans appliquer de sélection préalable de caractéristiques. Dans ce contexte, Marzat *et al.* [90] ont proposé une implémentation GPU de la méthode Lucas-Kanade d'estimation du flot optique. Une mise en œuvre programmée sous CUDA calculant des vecteurs de vitesses (de tous les pixels) à environ 15 fps pour une vidéo de 640×480 au lieu d'une seule fps sur CPU. Les auteurs dans [92] ont présenté une implémentation CUDA de la méthode Horn-Shunck de calcul du flot optique, offrant un traitement en temps réel de vidéos de faible résolution (316×252).

La deuxième catégorie est dévolue aux méthodes de suivi de mouvements à partir des caractéristiques extraites préalablement, telles que décrit dans [54], ou les auteurs proposent une mise en œuvre sous CUDA de la technique de calcul de cartographie de cohérence de vecteurs de mouvements, appelée Vector Coherence Mapping (VCM). Ceci a permis d'avoir un suivi de mouvements efficace avec une accélération de 40 par rapport aux versions CPU. Les auteurs dans [122] ont présenté une implémentation GPU du descripteur SIFT d'extraction de caractéristiques, ainsi que la méthode Kanade Lucas Tomasi feature tracker (KLT) utilisée pour le suivi de mouvements. Ces techniques ont été développées avec les bibliothèques OpenGL et Cg, permettant un traitement GPU à 10 fps au lieu de 1 fps sur CPU. On trouve également dans [109] une autre implémentation GPU, basée

sur OpenGL, de la méthode de calcul de correspondance (block matching) entre blocs de pixels des trames successives. Ceci permet d'atteindre un traitement en temps réel (25 fps) d'une vidéo (640×480) au lieu de 8 fps sur CPU. Sundaram *et al.* [125] ont proposé une méthode de calcul de trajectoires de points basée sur une mise en œuvre parallèle sur GPU d'un algorithme de calcul du flot optique. Ceci permet d'appliquer un traitement GPU à 22 fps au lieu de 0.3 fps sur CPU pour une vidéo de résolution 640×480. Cependant, ces accélérations restent insuffisantes pour assurer un traitement temps réel des vidéos de haute définition.

On trouve aussi des travaux d'exploitation des GPU pour l'accélération de détection des visages en temps réel [91]. Les traitements GPU permettent dans ce cas de détecter cinq différents visages sur la séquence vidéo en temps réel (figure I.12), avec un taux de rafraichissement de 36 fps, largement supérieur aux résultats obtenus sur CPU (20 fps).

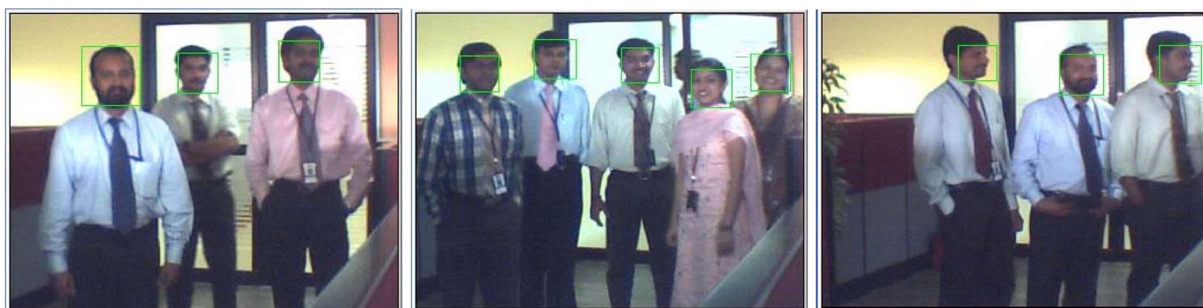


Figure I.12 – Détection et suivi de visages multiples en temps réel sur GPU [91]

Toutefois, et malgré ces accélérations, de nombreux algorithmes ne peuvent pas fonctionner en temps réel lors du traitement de vidéos de haute définition (HD/Full HD). Dans ce cadre, une gestion efficace des espaces mémoires ainsi qu'une exploitation maximale des threads GPU deviennent nécessaires pour assurer un traitement suffisamment rapide.

Notre contribution porte principalement sur l'exploitation, plein régime, du processeur graphique en proposant différentes techniques d'optimisation telles que le recouvrement des transferts de données par les calculs (streaming), l'exploitation efficace de mémoires à accès rapide, la réduction des copies de données, la visualisation rapide des résultats, l'exploitation simultanée des GPU multiples. Ces contributions sont décrites en détail dans le quatrième chapitre.

3.3 Estimation de complexité de traitement d'images sur GPU

Les algorithmes de traitement de gros volumes d'images et de vidéos nécessitent d'appliquer un grand nombre d'opérations mathématiques et logiques. Ceci les rend bien adaptés aux architectures massivement parallèles des processeurs graphiques. Cependant, plusieurs méthodes n'arrivent pas à atteindre des performances satisfaisantes à cause de l'incompatibilité de leurs algorithmes et implémentations par rapport à l'architecture des GPU. En effet, les nouvelles architectures de ces processeurs doivent prendre en compte différents paramètres lors de la mise en œuvre des algorithmes sur GPU. Pour cela, il est nécessaire d'établir des recommandations permettant un portage efficace des calculs sur processeurs graphiques. Il est donc nécessaire d'évaluer la compatibilité des algorithmes avec les plates-formes (GPU), par une estimation de leur complexité algorithmique.

Dans ce contexte, Ryoo *et al.* [112] ont présenté différentes optimisations permettant d'exploiter au mieux les GPU GeForce 8800. Leur objectif est de maximiser le nombre de threads actifs sur GPU, et de bien gérer les dispositifs associés (nombre de registres, nombre de threads par bloc, bande passante de mémoire globale, etc.). Un exemple de multiplication de matrices a été utilisé pour illustrer l'intérêt de ces optimisations. Ces dernières ont permis d'appliquer un traitement GPU à environ 90 GFLOPS au lieu de 10 GFLOPS avec la version non optimisée du code CUDA de la multiplication de matrices.

Kothapalli *et al.* [60] ont proposé un modèle de prédiction de performances pour les applications développées sous CUDA. Ce modèle englobe les différentes facettes de l'architecture des processeurs graphiques telles que l'ordonnancement des tâches, la hiérarchie de mémoires GPU ainsi que le pipeline graphique. À partir de ces facteurs, le modèle applique une analyse des codes CUDA pour obtenir une estimation des performances. Une illustration des résultats obtenus *via* ce modèle est présentée avec les applications de multiplication de matrices, classement de listes (list ranking) et génération d'histogrammes. En effet, les temps de calcul estimés par ce modèle sont très proches des temps réel du traitement. Cependant, ce modèle ne prend pas en compte les échanges ainsi que la synchronisation entre blocs de threads effectués *via* la fonction syncthreads(). De plus, le modèle ne gère pas les opérations atomiques.

Les auteurs dans [114] proposent une méthode d'exploitation efficace des processeurs graphiques multiples assurant la scalabilté des résultats. La méthode proposée permet aux

développeurs de prédire le temps d'exécution par rapport au nombre de GPU et de la taille des données utilisées. Cette méthode a été utilisée pour prédire les performances de différentes applications telles que la convolution, le tracé de rayons [111] ainsi que la transformée de Fourier, appelée Fast Fourier transform (FFT), offrant une amélioration des performances d'un facteur allant de 11 % à 40 %.

Par ailleurs, les auteurs dans [105] ont proposé un ensemble de paramètres permettant l'évaluation de la complexité des algorithmes de traitement d'images sur GPU sous CUDA. Ces paramètres se présentent comme suit :

1. Fraction parallélisable ;
2. Ratio de temps entre calcul et accès mémoire ;
3. Taux de calcul par pixel ;
4. Taux d'accès aux mémoires GPU par pixel ;
5. Diversité et branchements ;
6. Dépendance de tâches.

3.3.1 Fraction parallélisable

Dans le domaine du calcul parallèle, La loi d'Amdahl [46] est souvent utilisée pour estimer l'accélération théorique maximale qu'on peut atteindre sur plusieurs processeurs. Cette loi estime que si f est la fraction du programme pouvant être parallélisée, l'accélération maximale S qui peut être atteinte utilisant N processeurs est :

$$S \leq \frac{1}{1 - f + \frac{f}{N}} \tag{I.1}$$

Les algorithmes de traitement d'images sont souvent composés de plusieurs étapes. La fraction de parallélisation de ces algorithmes est directement liée à chacune de ces étapes.

3.3.2 Ratio de temps entre calcul et accès mémoire

Sous CUDA, les instructions d'accès à la mémoire incluent chaque opération de lecture/écriture aux mémoires, globale, locale et partagée du processeur graphique. Les temps de latence élevés dus à l'accès aux mémoires, globale et locale, peuvent être négligés s'il y

a suffisamment d'instructions indépendantes lancées sur GPU (gains importants en temps de calcul). Le ratio entre le nombre d'opérations en virgule flottante et l'accès à la mémoire globale (ou locale) permet de quantifier la propriété citée ci-dessus. Plus le ratio est élevé, meilleures seront les performances.

3.3.3 Taux de calcul par pixel

Les processeurs graphiques offrent généralement des accélérations significatives, grâce au grand nombre d'unités de traitement parallèle. Ces accélérations deviennent plus importantes lors de l'application des traitements intensifs. Pour cela, le nombre d'opérations (en virgule flottante) par pixel permet de définir une estimation de la complexité temporelle de l'algorithme, et de son efficacité pour une mise en œuvre parallèle.

3.3.4 Taux d'accès aux mémoires GPU par pixel

Les GPU actuels disposent d'une bande passante d'accès à la mémoire largement supérieure à celle des CPU. Un algorithme de traitement d'image ayant un accès fréquent à la mémoire par pixel peut exploiter cette large bande passante afin d'accélérer les calculs. De plus, les processeurs graphiques permettent un accès beaucoup plus rapide aux données *via* la mémoire partagée. Le nombre d'accès aux mémoires GPU (globale, locale, partagée et texture) par pixel peut donc définir une bonne estimation de la complexité temporelle des algorithmes de traitement d'images.

3.3.5 Diversité et branchements

Les instructions de contrôle et de branchement peuvent dégrader considérablement les performances des traitements parallèles. Ces instructions peuvent causer des blocages dans le déroulement des threads qui les contraignent à être exécutés séquentiellement. L'augmentation du nombre d'instructions de contrôle a un effet négatif sur les performances GPU.

3.3.6 Dépendance de tâches

Une implémentation CUDA peut être performante si elle est composée de plusieurs kernels lancés indépendamment. Chaque kernel pourra traiter un sous ensemble de données en utilisant un certain nombre de threads. La dépendance des tâches peut donc dégrader les performances des traitements parallèles sur GPU.

4 Exemples de traitements intensifs d'objets multimédias

Comme évoqué dans la section précédente, les processeurs graphiques ne sont pas toujours plus performants que les processeurs centraux. En effet, les applications doivent avoir suffisamment de calculs pour pouvoir profiter de la puissance des GPU. Dans ce contexte, plusieurs méthodes peuvent présenter de bons candidats pour une exploitation efficace de ces processeurs graphiques. On peut citer dans ce rapport trois applications issues de domaines variés. La première est une méthode de segmentation des vertèbres pour des images X-ray [66]. La deuxième présente un algorithme de navigation et d'indexation dans des bases de données multimédias [119], tandis que la troisième application est utilisée pour la détection de mouvements en temps réel à partir d'une caméra mobile [136]. Nous présentons ici le principe de ces méthodes tout en montrant les étapes pouvant être portées et implémentées sur GPU. Notre choix d'accélérer ces applications est dû premièrement à leur temps de calcul élevé (calcul intensif), et deuxièmement à l'intérêt de nos collaborateurs scientifiques (Numédiart [1], Laboratoire PSNC, Pologne [2], Université d'Ankara [3]) pour leur accélération.

4.1 Segmentation des vertèbres

À l'heure actuelle, la radiographie de la colonne vertébrale est l'une des solutions les plus rapides et économiques utilisées pour détecter les anomalies vertébrales pour un spécialiste. Du point de vue patient, cette procédure présente l'avantage d'être sûre et non invasive. Pour ces raisons, cet examen est largement utilisé et demeure incontestable pour des traitements et/ou des diagnostics urgents. Malgré ces avantages, un examen lourd et méticuleux est demandé au spécialiste, la nature même des images radiographiques en est la raison. Celles-ci révèlent la densité des tissus osseux sur un film radiographique. Un os est défini par du blanc, un tissu mou par un niveau de gris et un espace vide par du noir. Dès lors, les images présentent de faibles contrastes et certaines zones peuvent être partiellement cachées par d'autres organes du corps humain. Il en découle donc que le contour des vertèbres reste délicat à détecter. Cette application s'inscrit dans le cadre d'une aide au diagnostic des maladies liées aux anomalies vertébrales (scoliose, ostéoporose, etc.) par une

1. Institut Numédiart, UMONS, Université de Mons, Belgique
2. PSNC, Poznan Supercomputing and Networking Center, Polgne
3. Université d'Ankara. Turquie

détection et segmentation automatique du contour des vertèbres. Ce processus fournit des informations quantitatives à l'examen qualitatif d'un spécialiste. La méthode utilisée est basée sur l'Active Shape Model (ASM), ou modèle actif de formes, développée par Cootes *et al.* dans [24].

L'ASM représente un modèle statistique déformable caractérisant la forme d'un objet. Il est déterminé par le traitement statistique d'un échantillon d'images représentatives de l'objet à modéliser. Pour cela, des points de référence sont judicieusement identifiés sur la frontière des objets constituant l'échantillon. L'un des grands intérêts de cette méthode est donc de pouvoir tirer parti des connaissances d'un spécialiste. Toutefois, un inconvénient concerne la phase d'initialisation de recherche de cette méthode. En effet, pour réaliser la segmentation, une forme moyenne issue du modèle doit être positionnée à un endroit pertinent de l'image. Cette forme sera amenée à évoluer afin que chaque point de référence la caractérisant épouse au mieux le contour de l'objet à segmenter. L'application présentée dans cette section vise à automatiser cette phase d'initialisation. La méthode de segmentation basée sur les modèles actifs de formes est donc composée de quatre étapes : apprentissage, modélisation, initialisation et segmentation.

4.1.1 Apprentissage

Cette étape commence par la création d'un échantillon d'images X-ray centrées sur les vertèbres cervicales C3 à C7. Chaque image de cet échantillon est décrite par une information, c.a.d. les coordonnées de quelques points de repère situés sur les contours de vertèbres (figure I.13). Ces points seront alignés entre les différentes images de l'échantillon afin d'extraire un modèle moyen de vertèbres. Pour ce faire, une approche basée sur l'alignement de Procuste est utilisée [44].

4.1.2 Modélisation

Une fois les formes de vertèbres alignées, elles peuvent être employée pour construire le modèle moyen. Ensuite d'autres formes admissibles sont extraites en déplaçant leurs points de repère dans des directions précises, obtenues par L'ACP des données. Cette étape de modélisation est décrite en détail dans [23].

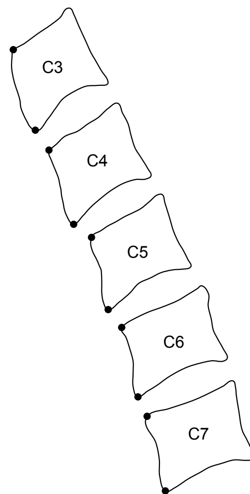


Figure I.13 – Marquage des points repères de vertèbres C3 à C7

4.1.3 Initialisation

Afin d'initialiser le processus de segmentation, la forme moyenne doit être placée à proximité des coins de vertèbres. Pour ce faire, l'utilisateur est invité à marquer deux points de l'image seulement afin de déterminer une région d'intérêt, appelée Region Of Interest (ROI), à partir du coin supérieur de la vertèbre C3 ainsi que le coin inférieur de la vertèbre C7. Ensuite, tous les coins des autres vertèbres seront extraits *via* un processus composé de quatre étapes : égalisation d'histogramme, détection des contours, détection des coins et localisation des vertèbres.

1. **Egalisation d'histogramme :** étant donné que les images X-ray présentent des faibles contrastes, une amélioration de la qualité de ces images est réalisée afin d'avoir une meilleure détection des contours par la suite. Ceci peut se faire *via* la technique d'égalisation d'histogramme permettant d'uniformiser les contrastes de l'image en le faisant correspondre à une distribution donnée (idéalement uniforme) tel que décrit dans [106].

2. **Détection des contours :** après l'amélioration du contraste des images, une détection des contours est appliquée *via* l'algorithme de Canny ou Deriche-Canny qui s'appuient sur un filtre particulièrement performant mais coûteux en temps de calcul (environ trois minutes pour la détection des contours de 200 images de 1476×1680 sur un CPU dual-core). Ce filtre a été conçu pour optimiser trois critères :

- La bonne détection : faible taux d'erreur dans la signalisation des contours ;
- La bonne localisation : distance minimale entre contours détectés et contours réels ;
- La clarté de la réponse : une seule réponse par contour et pas de faux positifs.

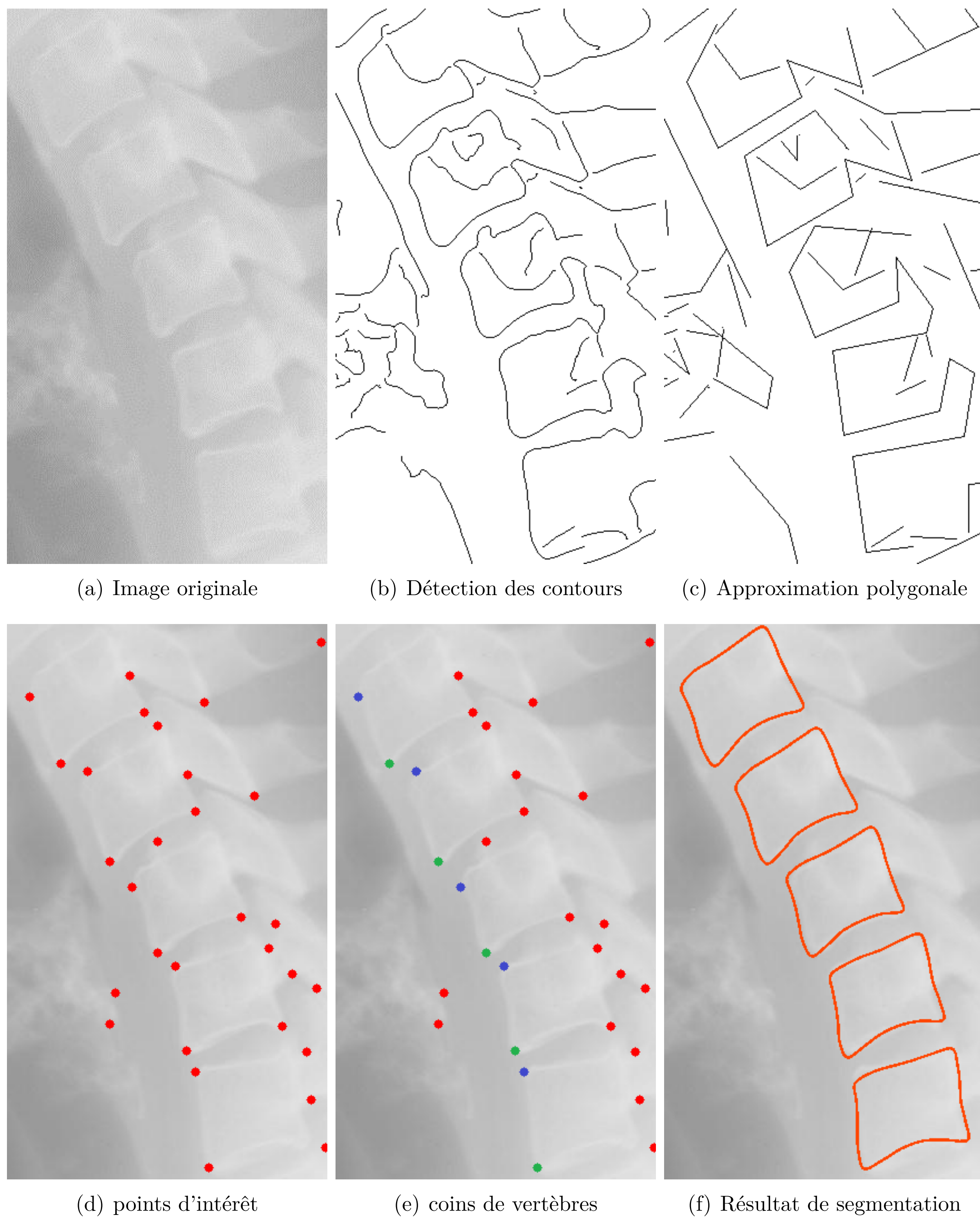
3. **Détection des coins :** la détection des coins imaginée s'effectue en plusieurs étapes. En premier lieu, un suivi des contours est réalisé dont le résultat servira d'entrée à un algorithme d'extraction de polygones. L'intérêt de ce dernier est de diminuer de façon significative le nombre de points qui composent un contour. Dès lors, les courbes qui représentaient les contours deviennent des droites dont les extrémités constituent les points d'intérêt. L'algorithme utilisé dans le cadre de cette application est l'algorithme de Douglas-Peucker [36].

4. **Localisation des vertèbres :** Parmi tous les coins détectés lors de l'étape précédente, nous devons extraire uniquement les coins de vertèbres. Pour ce faire, nous appliquons un alignement entre les deux points de repère marqués par l'utilisateur (phase d'initialisation) et le modèle de forme moyenne extrait précédemment. Ensuite, pour chaque point de repère, nous cherchons le coin le plus proche, détecté à partir de l'approximation polygonale.

4.1.4 Segmentation

Finalement, chaque point de repère de la forme moyenne évolue de telle sorte que son contour correspond au mieux à celui de la vertèbre. La figure I.14 illustre les étapes de la méthode de segmentation des vertèbres. Pour une question de simplicité, l'image d'origine (1476×1680) est réduite pour représenter la région d'intérêt.

Cependant, la méthode de segmentation des vertèbres décrite ci-dessus souffre d'un temps de calcul très élevé (environ cinq (5) minutes lors de l'utilisation d'un ensemble de 200 images médicales avec une résolution de 1476×1680 sur un CPU dual-core). Nous présentons dans le cinquième chapitre une solution permettant de contourner cette contrainte par l'exploitation des architectures parallèles et hétérogènes (Multi-CPU/Multi-GPU).



(a) Image originale (b) Détection des contours (c) Approximation polygonale

(d) points d'intérêt (e) coins de vertèbres (f) Résultat de segmentation

Figure I.14 – Illustration de la méthode de segmentation des vertèbres

4.2 Indexation de bases de données multimédias

Nous présentons dans cette section le projet MediaCycle [119]. Ce dernier fournit un nouvel environnement de navigation et d'indexation dans des bases de données multimédias (images, vidéos, sons), il offre une alternative à la recherche classique par requête. Les bases de données sont indexées de telle sorte que les utilisateurs peuvent facilement récupérer les éléments dont ils ont besoin. MediaCycle dispose de trois composantes principales : AudioCycle, ImageCycle et VideoCycle.

4.2.1 AudioCycle

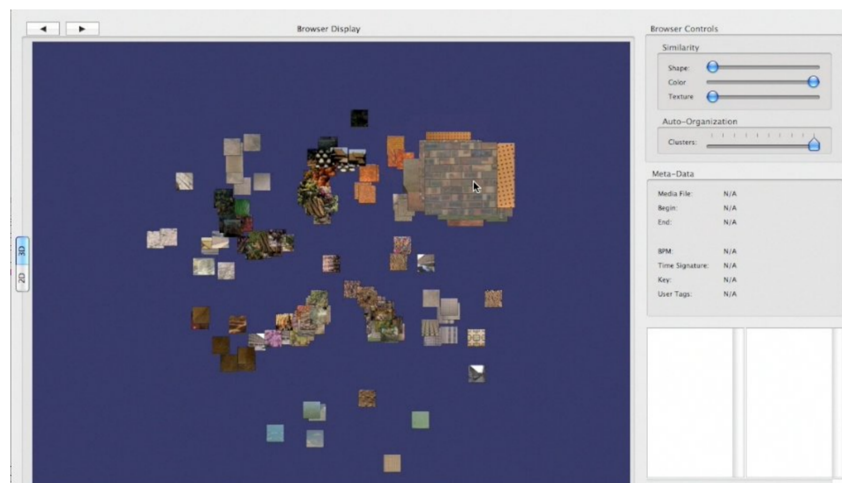
AudioCycle [37] propose un prototype de navigation dans des bibliothèques de boucles musicales. Il fournit à l'utilisateur une vue graphique des extraits audio qui sont visualisés et organisés en fonction de leur similitude en termes de paramètres musicaux tels que le timbre, l'harmonie et le rythme. L'utilisateur est en mesure de naviguer dans cette représentation visuelle, d'écouter des extraits audio et de rechercher ceux qui l'intéressent. Ce projet s'appuie sur différentes technologies telles que l'analyse des fichiers audio à partir des informations musicales, la visualisation 3D, le rendu sonore spatial. L'application proposée peut être exploitée par les remixeurs, musiciens, compositeurs de bandes sonores ainsi que les artistes.

4.2.2 ImageCycle

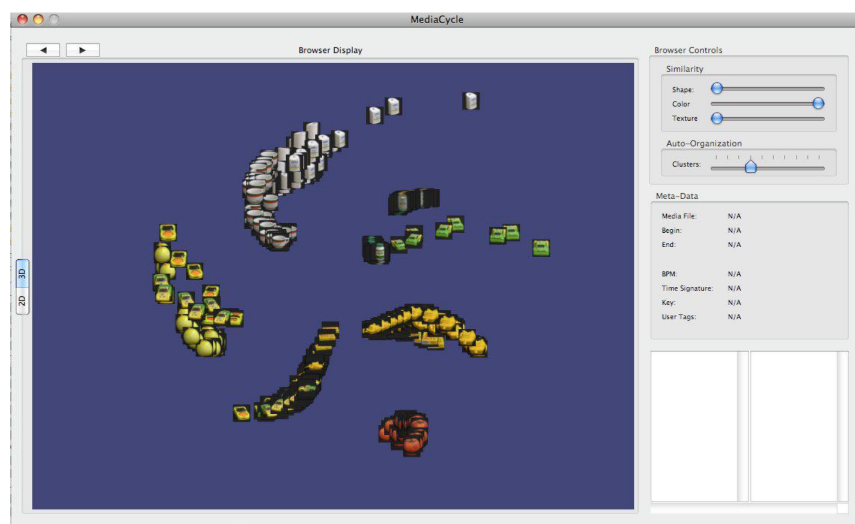
ImageCycle [119] permet la navigation et l'indexation d'images à partir de leur classification prenant en compte les caractéristiques de couleur, de forme, de texture, etc. Ce projet propose également des fonctions de zoom et de rotation des images. L'interface de navigation présente cinq fonctionnalités principales (figure I.15(a)) :

1. **interaction :** trois curseurs (coin haut à droite de la figure I.15(a)) permettant à l'utilisateur de définir les poids de la forme, la couleur et la texture. Ces poids offrent une alternative pour classifier les images selon les préférences de l'utilisateur ;
2. **visualisation :** la fenêtre principale permet d'afficher les images de telle sorte que la distance entre chaque couple d'images reflète leur degré de similarité calculé à partir des poids définis par l'utilisateur ;
3. **classification :** les images sont groupées en fonction de la distance indiquée ci-dessus. L'utilisateur peut naviguer à l'intérieur ou à l'extérieur des classes ;

4. **affichage individuel :** lorsque la souris passe au-dessus d’une image, celle-ci devient instantanément plus grande de telle sorte que l’utilisateur peut naviguer rapidement à travers la base de données (figure I.15(a)) ;
5. **description :** un bouton placé en bas à droite de la fenêtre principale (figure I.15(a)) permet d’afficher des informations supplémentaires sur les images.



(a) Navigation dans une base d’images



(b) Inedxation d’images avec ImageCycle

Figure I.15 – Vue d’ensemble de ImageCycle [119]

La figure I.15(b) montre un exemple illustrant l’utilisation du projet MediaCycle pour la navigation dans une base d’images. Cette figure montre les classes regroupant les images ayant des caractéristiques similaires.

4.2.3 VideoCycle

VideoCycle [126] offre, à son tour, une application de navigation et d'indexation de séquences vidéo. La navigation dans ce cas est effectuée à partir des caractéristiques du mouvement : silhouette, zone du mouvement, moments de Hu [53], contours, etc. L'interface de navigation est similaire à celui de ImageCycle.

La figure I.16 montre une vue de l'application VideoCycle permettant la navigation dans une base de séquences vidéo à partir du calcul de leurs similarités.

Figure I.16 – VideoCycle : Navigation dans des bases de séquences vidéo [126].

Cependant, MediaCycle est entravé par la forte intensité ainsi que les temps de calcul qui augmentent en fonction du nombre et de la taille des objets multimédias utilisés (grandes bases d'images, images de haute résolution, etc.). L'exploitation des plates-formes parallèles (GPU) et hétérogènes (Multi-CPU/Multi-GPU) représente une bonne solution pour remédier à cette contrainte. Nous présentons les résultats que nous avons obtenus à base de cette solution dans le cinquième chapitre.

4.3 Détection de mouvements à partir de caméra mobile

Nous décrivons dans cette section une application de détection d'objets en mouvements à partir d'une caméra mobile [136]. Cette application a été développée à l'université d'Ankara en Turquie avec laquelle nous collaborons *via* le projet européen COST IC 805. Cette application se compose de deux étapes principales : estimation du mouvement de caméra et soustraction du background.

4.3.1 Estimation du mouvement de caméra

Le processus d'estimation du mouvement de la caméra se base sur le calcul du flot optique à partir des points d'intérêt. Une détection des coins est appliquée sur chaque image de la vidéo afin de pouvoir les suivre (coins) *via* le flot optique. Le résultat de ce calcul est présenté par un ensemble de n vecteurs tel que montré dans l'équation (I.2).

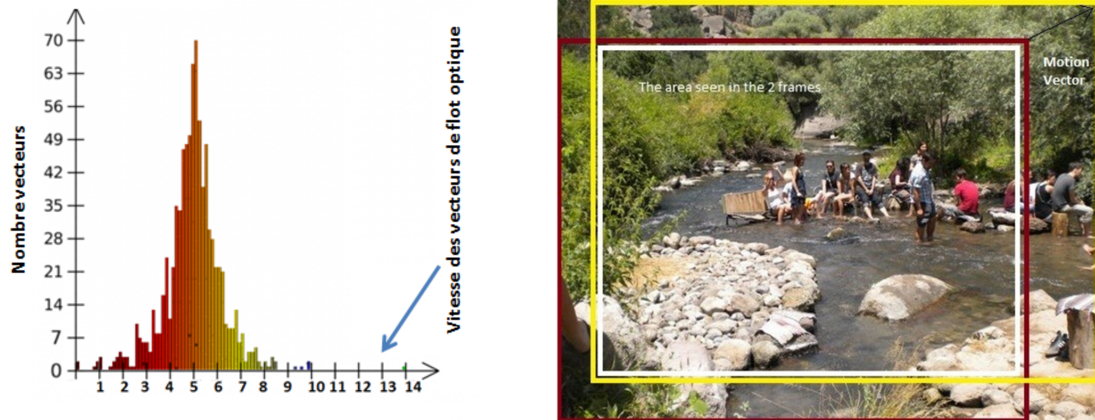
$$\Omega = \{\omega_1 \ldots \omega_n \mid \omega_i = (x_i, y_i, vx_i, vy_i)\} \tag{I.2}$$

où :

- x_i : coordonnée x du point i ;
- y_i : coordonnée y du point i ;
- vx_i : déplacement du point i selon l'axe des x ;
- vy_i : déplacement du point i selon l'axe des y.

En général, le mouvement de la caméra implique le déplacement des coins des images de la vidéo. De ce fait, une sélection des vecteurs du flot optique est appliquée en premier lieu afin d'estimer le mouvement de la caméra. Dans ce cas les déplacements des coins sont dus au mouvement de la caméra, et par conséquent les longueurs (vitesses) des vecteurs du flot optique associés auront des valeurs très proches. Ces dernières représentent ainsi le mouvement estimé de la caméra. Un histogramme des vecteurs du flot optique est calculé afin de pouvoir visionner et sélectionner les vecteurs décrivant ce mouvement.La figure I.17(a) présente un histogramme, extrait à partir d'une séquence vidéo d'une taille de 640×480, montrant les différentes valeurs de vitesses des vecteurs du flot optique (axe horizontal), ainsi que le nombre de vecteurs associés à chacune des vitesses (axe vertical). La figure I.17(a) montre également que la majorité des vecteurs ont une longueur (vitesse) de 5 pixels par image, et par conséquent la vitesse de la caméra est estimé à 5. La direction du mouvement de la caméra est estimée de la même manière.

Une fois le mouvement (vitesse et direction) de la caméra estimé, la région commune entre deux frames (images) successive sera extraite. Ceci s'effectue par la soustraction des parties disparues de l'image (frame) de départ, ainsi que les parties apparues dans l'image suivante (frame + 1) suite au mouvement de la caméra. La soustraction s'effectue à partir des valeurs de vitesse et de direction obtenues par les mesures du flot optique. La figure I.17(b) montre un exemple d'extraction de la partie commune entre deux images successives. Les cadres, rouge et jaune, montrent les images de départ (frame i) et d'arrivée (frame i+1), tandis que le cadre blanc présente le résultat de l'extraction de partie commune tel que décrit ci-dessus.



(a) Histogramme des vecteurs de flot optique [136]

(b) Partie commune entre images successives

Figure I.17 – Estimation du mouvement de la caméra

4.3.2 Détection du mouvement

Une fois le mouvement de la caméra estimé et éliminé, la détection des objets en mouvement dans la scène devient possible. En effet, la soustraction d'images (frames) successives permet de localiser les mouvements apparus. Si le résultat de cette soustraction donne une image noire, on peut déduire qu'il n'y pas eu de mouvement. Sinon, on peut déduire qu'un mouvement a eu lieu. Notons qu'un seuil est utilisé afin d'éliminer les bruits.

La figure I.18 montre le résultat de soustraction entre couples d'images successives en utilisant une caméra mobile. Le résultat présenté par une image noire est interprété par l'absence de mouvement et que le déplacement de la caméra était bien estimé et éliminé. La figure I.19 présente le même cas mais avec des personnes en mouvement en même temps

avec le déplacement de la caméra. Le résultat montre que la silhouette des personnes en mouvement a bien été détectée grâce à l'estimation du mouvement de la caméra et la soustraction entre images successives.

Figure I.18 – Estimation du mouvement de la caméra avec objets statiques [136].

Figure I.19 – Estimation du mouvement de la caméra avec objets mobiles [136].

Toutefois, les performances de calcul de cette implémentation séquentielle sur CPU ne sont pas suffisantes pour assurer un traitement en temps réel. En effet, cette application fonctionne à environ 5 fps pour une vidéo de 640×480. Ceci est dû principalement à l'intensité de calcul des étapes de détection des coins et d'estimation du flot optique. Suite à la forte croissance des résolutions de vidéo fournies par les caméras d'aujourd'hui ainsi que l'intérêt de vidéos HD ou Full HD pour l'application d'un traitement plus précis de pixels, nous proposons d'accélérer cette application en exploitant les processeurs graphiques (unique ou multiples) de manière efficace tous en assurant une bonne gestion et cohérence des mémoires CPU et GPU. Cette solution est présentée en détail dans le cinquième chapitre.

Chapitre II

Traitement d'images sur GPU

Sommaire

1 **Schéma de développement du traitement d'images sur GPU** **51**

2 **Optimisations proposées sur GPU** **53**

2.1 Traitement d'image unique 53

2.2 Traitement d'images multiples 54

3 **Mise en œuvre d'algorithmes de traitement d'images sur GPU** **57**

3.1 Méthodes classiques de traitement d'images sur GPU 57

3.2 Extraction des points d'intérêt sur GPU 60

3.3 Détection des contours sur GPU 62

4 **Analyse des résultats** **66**

5 **Conclusion** **71**

Dans ce chapitre, nous montrons l'intérêt de l'exploitation efficace des processeurs graphiques lors du traitement d'images uniques et multiples. Ces méthodes sont connues pour leurs grands besoins de calcul, dus principalement aux volumes importants de données à traiter (images HD/Full HD). Ce chapitre est présenté en quatre parties : la première décrit le schéma de développement proposé pour le traitement d'images uniques et multiples sur GPU. La deuxième partie décrit les différentes optimisations employées permettant de réduire significativement les temps de transferts de données entre la mémoire du CPU et celle du GPU. Ces optimisations offrent également une meilleure exploitation des unités de calcul intégrées dans les processeurs graphiques. Nous présentons ensuite dans la troisième partie nos mises en œuvres de différents algorithmes (élimination des bruits, détection des points d'intérêt, détection des contours, etc.) effectuées à partir du schéma proposé. La quatrième partie est dévolue à la présentation et l'interprétation des résultats obtenus.

Publications principales liées au chapitre :

1. **Revue internationale :**

   P. D. Possa, **S. A. Mahmoudi**, N. Harb, C. Valderrama "A Challenging/Adaptable FPGA-Based Architecture for Feature Detection", *IEEE Transactions on Computers*, Soumis le 30/10/2012.

2. **Conférences internationales :**

   [73] : **S. A. Mahmoudi**, S. Frémal, M. Bagein, P. Manneback, "Calcul intensif sur GPU : exemples en traitement d'images, en bioinformatique et en télécommunication", *CIAE 2011 : Colloque d'Informatique, Automatique et Electronique*, Casablanca, Maroc. Mars 2011.

   [30] : P. D. Possa, **S. A. Mahmoudi**, N. Harb, C. Valderrama "A New Self-Adapting Architecture for Feature Detection", *FPL 2012 : 22 nd International Conference on Field Programmable Logic and Applications*, Oslo, Novège. Août 2012.

3. **Workshops nationaux et internationaux**

   [81] : **S. A. Mahmoudi**, P. Manneback, "Parallel Image Processing with CUDA and OpenGL", *1st Workshop of COST Action IC 0805. Network for High-Performance Computing on Complex Environments*. Lisbon, Portugal. Octobre 2009.

   [79] : **S. A. Mahmoudi**, P. Manneback, "Traitements d'images sur GPU sous CUDA et OpenGL : application à l'imagerie médicale", *Journées CIGIL : Calcul Intensif et Grilles Informatiques à Lille*. Lille, France. Décembre 2009.

   [80] : **S. A. Mahmoudi**, Pierre Manneback, "Le traitement d'objets multimédias sur GPU", *Seconde journée scientifique du pôle hainuyer*. Mons, Belgique, Mai 2010.

1 Schéma de développement du traitement d'images sur GPU

Les algorithmes de traitement d'images présentent un excellent champ d'applications pour l'accélération sur GPU. En effet, la majorité de ces algorithmes disposent de sections de traitements identiques entre pixels, ce qui les rend bien adaptés à un traitement parallèle exploitant la grande puissance de calcul que fournissent les processeurs graphiques. Toutefois, ces processeurs requièrent une gestion efficace de leur espace mémoire et unités de calcul afin de profiter pleinement de leur puissance. De ce fait, nous proposons un modèle de traitement d'images sur GPU, permettant le chargement, le traitement et l'affichage des images sur processeurs graphiques. Notre modèle s'appuie sur CUDA pour les traitements parallèles et OpenGL pour la visualisation des résultats. Ceci permet de réduire les coûts de transfert de données entre la mémoire CPU et la mémoire GPU. Ce modèle repose sur trois étapes principales : chargement des images d'entrée, traitement CUDA en parallèle, présentation des résultats (figure V.3).

1. **Chargement des images d'entrée :**
   Le transfert des images d'entrée depuis la mémoire hôte (CPU) vers la mémoire graphique (GPU) permet de les traiter sur GPU par la suite. Cette étape inclue également l'initialisation du processeur graphique ainsi que l'allocation des espaces de mémoire sur GPU ;
2. **Traitement CUDA en parallèle :**
   Le traitement parallèle des pixels de l'image est effectué sous CUDA en deux phases principales :
   (a) **Allocation des threads** : avant de lancer les traitements parallèles, le nombre de threads de la grille de calcul GPU doit être déterminé de telle sorte que chaque thread puisse effectuer des traitements sur un ou plusieurs pixels groupés. Ceci permet d'appliquer des traitements en mode SIMD, bien adapté à la programmation GPU. Notons que la sélection du nombre de threads dépend du nombre de pixels de l'image ;
   (b) **Traitements CUDA** : les fonctions CUDA (kernels) sont exécutées M fois en utilisant les M threads créés lors de l'étape précédente.

3. **Présentation des résultats :**

   À l'issue des traitements, les résultats peuvent être présentés en utilisant deux scénarios différents :

   (a) **Visualisation OpenGL :** l'affichage des images de sortie avec la bibliothèque OpenGL permet une visualisation rapide grâce à la réutilisation de zones mémoires allouées par CUDA. Ceci permet de réduire significativement les coûts de transfert de données. Ce scénario est utile lors du traitement GPU appliqué sur une seule image ;

   (b) **Transfert des résultats :** la visualisation sous OpenGL n'est plus requise lorsque l'on désire sauvegarder plusieurs images traitées. Dans ce cas, le transfert des images de sortie depuis la mémoire GPU vers la mémoire CPU est indispensable. Le temps de transfert de ces images représente un coût supplémentaire pour l'application.

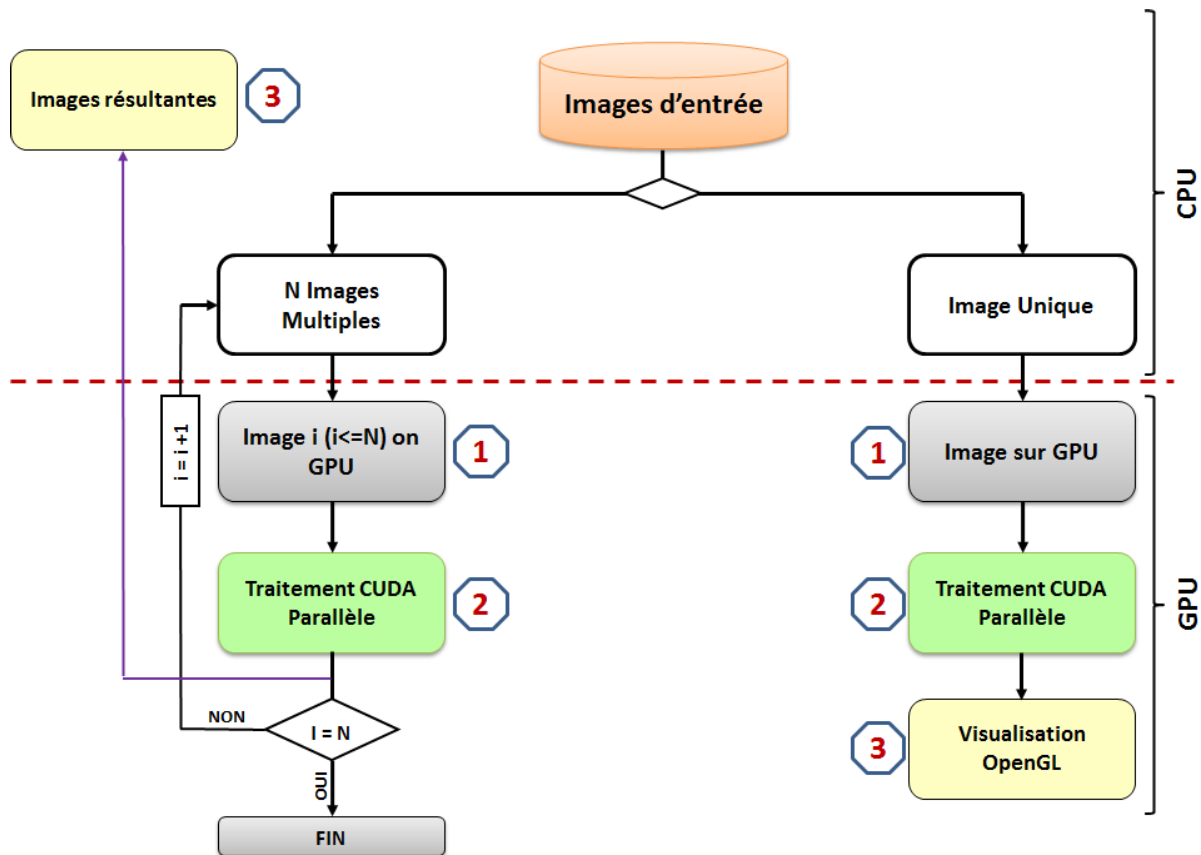


Figure II.1 – Modèle de traitement d'images sur GPU avec CUDA et OpenGL.

2 Optimisations proposées sur GPU

Une exploitation des unités de calcul du processeur graphique en parallèle permet d'accélérer les calculs de différentes méthodes de traitement d'images. Cette accélération devient plus importante lors du traitement de grands volumes de données permettant d'utiliser la majorité des unités de calcul disponibles dans la carte graphique. Cependant, les calculs parallèles sur processeur graphique peuvent être même plus lents que leurs équivalents séquentiels sur CPU, lors de l'application de traitements légers sur des petits volumes de données. Ceci est dû principalement aux temps de transfert de données entre mémoires, centrale et graphique, ainsi que les délais de synchronisation entre les calculs des différents threads GPU. La latence élevée des espaces de mémoire de la carte graphique présente également un grand handicap pour l'accélération de ces calculs. Pour faire face à ces contraintes, nous proposons d'employer différentes techniques d'optimisation GPU afin de mieux exploiter le processeur graphique. Ces optimisations dépendent principalement du type d'images (unique ou simple) à traiter :

2.1 Traitement d'image unique

Les optimisations proposées pour le traitement d'image unique sont appliquées sur les différentes étapes du traitement proposées dans le schéma de développement (section II.1). Pour chaque étape, nous appliquons une optimisation adaptée à son fonctionnement (figure II.3) :

2.1.1 Chargement d'image en mémoire de texture

Nous proposons de charger l'image d'entrée en mémoire de texture du processeur graphique afin d'avoir un accès plus rapide aux pixels par la suite. Cette mémoire, optimisée pour l'accès en 2D, est accessible en écriture par le CPU et en lecture par le GPU avec une latence de 4 cycles seulement. Même si les temps de chargement en mémoire de texture par rapport à la mémoire globale sont très proches, nous pouvons bénéficier des textures lors de l'étape suivante (section 2.1.2) grâce à l'accès rapide aux pixels de l'image.

2.1.2 Traitement CUDA *via* la mémoire partagée

Avant de lancer les traitements parallèles *via* CUDA, nous proposons de charger pour chaque pixel les valeurs de ses voisins en mémoire partagée du GPU. Ceci permet un traitement plus rapide des pixels utilisant les valeurs du voisinage. Nous proposons également de

fusionner les accès à la mémoire afin de réduire la latence d'accès aux pixels de l'image (figure II.2). Si des threads adjacents accèdent à des espaces mémoire non-adjacents, les accès sont effectués séquentiellement, si les threads accèdent à des espaces mémoires adjacents, les accès sont fusionnés et les données sont chargées en une fois.

2.1.3 Visualisation OpenGL

Lors du traitement d'image unique, nous pouvons bénéficier de la bibliothèque OpenGL offrant une visualisation rapide des résultats (image de sortie) grâce à son utilisation (OpenGL) des espaces mémoires déjà alloués sur GPU. Par conséquent aucun transfert n'est requis entre la mémoire du CPU et celle du GPU pour présenter les résultats.

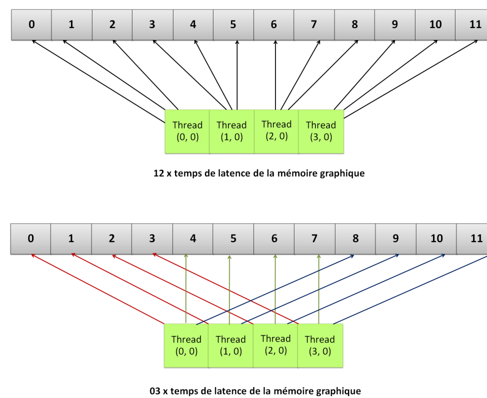


Figure II.2 – Fusion d'accès à la mémoire graphique

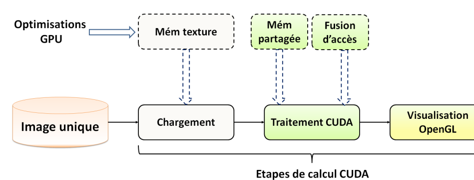


Figure II.3 – Otpimisations GPU proposées pour le traitement d'image unique

2.2 Traitement d'images multiples

Lors du traitement d'images multiples, les optimisations proposées sont également appliquées aux différentes étapes du traitement GPU dédié à ce type de données (images multiples) tel que montré dans la section II.1. Chaque étape est donc optimisée de manière adaptée à son fonctionnement :

2.2.1 Chargement des images multiples *via* le streaming

Le chargement de plusieurs images sur GPU nécessite un temps considérable qui peut entraver les performances de l'application. Nous proposons ainsi de créer quatre streams CUDA avant de charger les images d'entrée en mémoire globale. En effet, les streams permettent de recouvrir les kernels d'exécution par les transferts de données entre mémoires CPU et GPU. Chaque stream pourra traiter un sous ensemble d'images au lieu de les traiter séquentiellement *via* un seul stream (configuration par défaut de CUDA). La mémoire de texture n'est pas utilisée dans ce cas puisqu'elle n'est pas adaptée à l'utilisation de streams.

2.2.2 Traitement CUDA *via* le streaming

Nous proposons d'appliquer les traitements CUDA de chaque image à l'intérieur d'un stream parmi les quatre créés lors de l'étape précédente, afin de recouvrir les copies vers GPU par les kernels d'exécutions. Le but est de pouvoir traiter chaque image avec un stream en même temps qu'une autre image se fait copier sur GPU *via* un autre stream. Nous proposons également de charger pour chaque image les valeurs de ses voisins en mémoire partagée du GPU ce qui offre un traitement plus rapide des pixels utilisant les valeurs du voisinage.

2.2.3 Transfert des résultats

La présentation des résultats du traitement d'images multiples sur GPU nécessite un transfert de ces images vers la mémoire hôte (CPU). Ce transfert prend également un temps considérable en raison du volume de données à copier. De ce fait, nous proposons d'inclure cette étape à l'intérieur des streams CUDA créés lors de l'étape de chargement, afin de recouvrir cette copie par les étapes du chargement et du traitement CUDA.

Pour résumer, chaque stream CUDA se compose de trois étapes (figure II.4) :

1. Copie du sous ensemble d'images depuis la mémoire centrale vers la mémoire graphique ;
2. Calculs appliqués par les kernels CUDA ;
3. Copie des images résultantes vers la mémoire CPU.

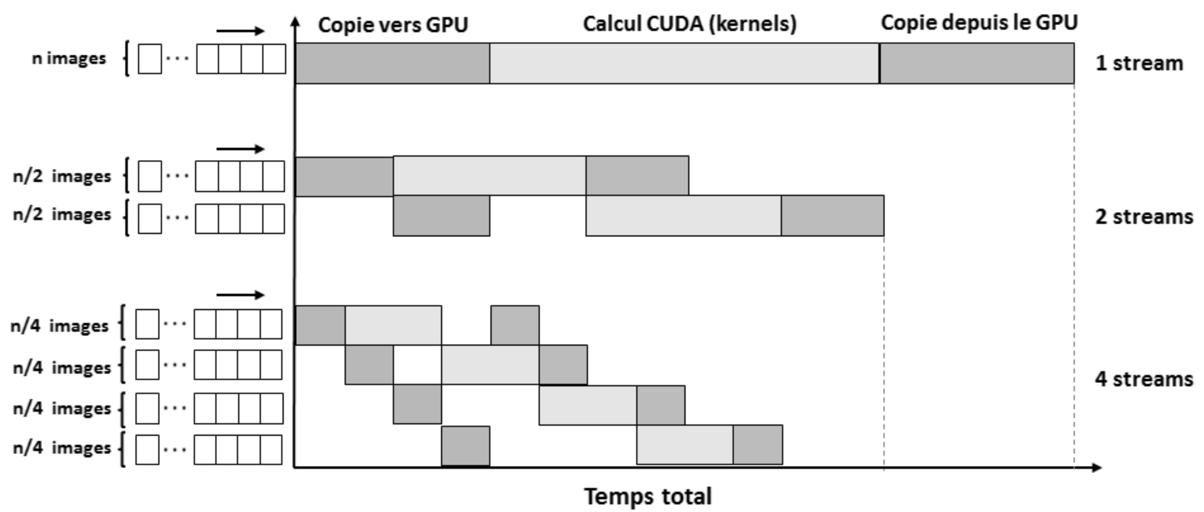


Figure II.4 – Traitement d'images multiples avec quatre CUDA streams

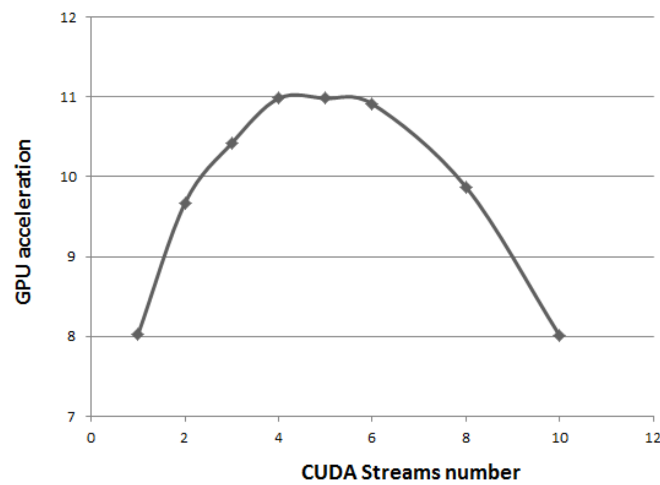


Figure II.5 – Accélération GPU *vs* nombre de CUDA streams

Notons que le choix de quatre streams CUDA est basé sur nos résultats expérimentaux. En effet, la figure V.5(b) présente l'évolution du temps de détection des coins et contours sur GPU (implémentations décrites dans le paragraphe suivant) pour un ensemble de 200 images avec une résolution de 3936×3936, par rapport au nombre de streams. Le choix de quatre streams a permis d'avoir les meilleures performances.

Le tableau II.1 résume les techniques d'optimisations appliquées à chacune des étapes du traitement GPU d'images uniques et multiples.

Etapes du traitement GPU	image unique	images multiples
Chargement	Mémoire Texture	CUDA streaming
Traitement CUDA	Mémoire Partagée	Mémoire Partagée CUDA streaming
Présentation	Visualisation OpenGL	CUDA streaming

Tableau II.1 – Optimisation des traitements GPU d'images uniques et multiples

3 Mise en œuvre d'algorithmes de traitement d'images sur GPU

Nous proposons dans cette section les implémentations GPU de différents algorithmes de traitement d'images (élimination des bruits, détection des points d'intérêt, détection des contours). Ces mises en œuvre s'appuient sur le schéma de développement ainsi que les optimisations proposées dans les sections II.1 et II.2 respectivement.

3.1 Méthodes classiques de traitement d'images sur GPU

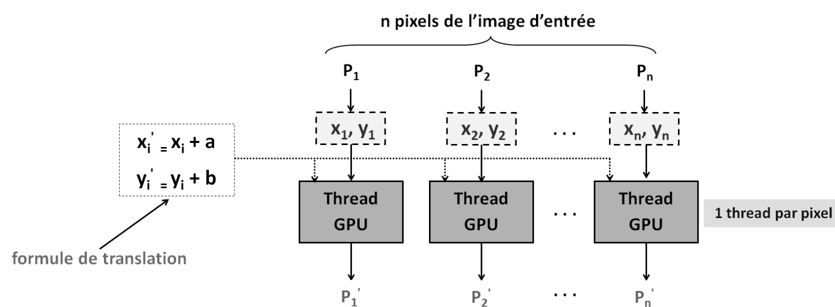
Nous avons commencé dans un premier temps par la mise en œuvre parallèle sur GPU de méthodes classiques afin de tester et de valider le schéma proposé. En effet, nous avons implémenté les méthodes suivantes sur processeur graphique : transformations géométriques (translation, rotation) et élimination des bruits.

3.1.1 Transformations géométriques

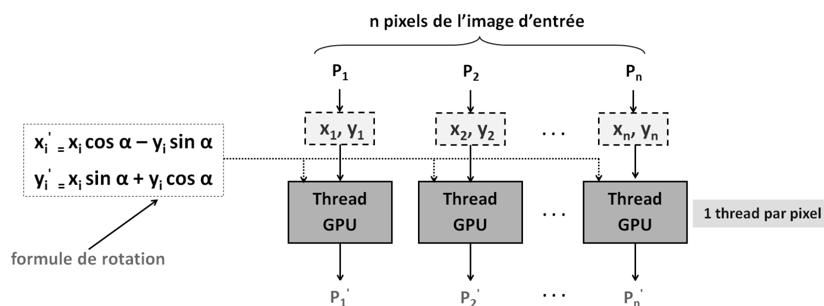
Nous avons mis en œuvre les méthodes de transformations géométriques (translation, rotation) sur GPU en utilisant un principe très simple consistant à choisir un nombre de threads (de la grille GPU) égal au nombre de pixels de l'image. Cela permet à chaque thread d'appliquer la formule de transformation géométrique *i.e.* translation (équation (II.1)) ou rotation (équation (II.2)) à un pixel de l'image. Ce traitement sera appliqué en parallèle pour tous les pixels de l'image. La visualisation des résultats est effectuée avec OpenGL. La figure II.6 décrit les étapes de mise en œuvre de ces algorithmes sur processeur graphique.

$$x^{'} = x + a \qquad y^{'} = y + b \tag{II.1}$$

$$x^{'} = x \cos\varphi - y \sin\varphi \qquad y^{'} = x \sin\varphi + y \cos\varphi \tag{II.2}$$



(a) Processus de translation d'image sur GPU



(b) Processus de rotation d'image sur GPU

Figure II.6 – Transformations géométriques sur processeur graphique

où :
- $x^{'}, y^{'}$: nouvelles coordonnée ;
- φ : angle de rotation ;
- a, b : valeurs de translations.

3.1.2 Elimination des bruits

Le lissage représente une opération très importante en traitement d'images, utilisée pour atténuer un bruit qui corrompt l'information. Les méthodes de lissage sont basées sur la convolution de l'image par des noyaux (figure II.7). En calcul numérique, l'application d'un noyau (filtre) correspond à l'application d'une matrice de transformation sur les données de l'image. Le produit de convolution (ou moyenne glissante) permet de traiter l'image de manière très simple. En effet, le centre du noyau est placé sur le pixel à traiter, les autres coefficients du filtre sont multipliés par les valeurs des pixels correspondants (voisins du pixel à traiter). La somme de ces multiplications présente le résultat du lissage appliqué (exp. 9 éléments dans le noyau présenté dans la figure II.7).

Nous illustrons dans la figure II.7 deux filtres de lissage à support 3×3. Le premier a ses coefficients constants (donc il applique aux niveaux de gris une moyenne pondérée : filtre moyen), tandis que le second voit ses coefficients diminuer du centre à la périphérie.

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

(a)

1/16	1/8	1/16
1/8	1/4	1/8
1/16	1/8	1/16

(b)

Figure II.7 – Masques de lissage

L'implémentation GPU de ces méthodes s'effectue en utilisant un nombre de threads GPU égal au nombre de pixels, de telle sorte que chaque thread puisse calculer un produit de convolution sur le pixel correspondant avec le filtre utilisé. Les lissages qui utilisent des masques contenant des valeurs identiques (figure II.7.(a)) peuvent effectuer un lissage horizontal puis vertical afin d'accélérer le calcul. Pour mettre en œuvre le lissage horizontal sur GPU, nous avons choisi un nombre de threads GPU égal au nombre de lignes de l'image. Nous appliquons ensuite la convolution de l'image par le biais d'un filtre composé d'une

seule ligne qui représente la partie horizontale du filtre moyen (lignes ayant les mêmes coefficients). Ceci permet de convoluer l'image en ne prenant en compte que les voisins gauches et droits (lissage horizontal). Cette convolution est lancée en parallèle pour toutes les lignes de l'image. De même, le lissage vertical est appliqué sur les colonnes de l'image.

Les implémentations GPU des méthodes de transformations géométriques et d'éliminations de bruits (lissage) ont permis d'avoir des facteurs d'accélérations allant de 5 à 100 par rapport à une implémentation séquentielle sur CPU. Notons que ces traitements GPU deviennent moins rapides que leurs traitements équivalents sur CPU lors de l'utilisation de petites images. Ceci est interprété par une exploitation insuffisante des unités de calcul disponibles dans le GPU.

3.2 Extraction des points d'intérêt sur GPU

Les méthodes d'extraction des points d'intérêt représentent des étapes préliminaires à de nombreux processus de vision par ordinateur. Ce paragraphe présente notre implémentation GPU du détecteur de coins utilisant la technique décrite par Bouguet [17], basée sur le principe de Harris. Cette méthode est connue pour son efficacité, due à sa forte invariance à la rotation, à la mise à l'échelle, à la luminosité et au bruit de l'image. À partir du schéma décrit dans la section II.1, nous avons parallélisé cette méthode en implémentant chacune de ses cinq étapes sur GPU (figure II.8) :

3.2.1 Calcul des dérivées et du gradient spatial

La première étape consiste au calcul de la matrice de dérivées spatiales G pour chaque pixel de l'image I, à partir de l'équation (II.5). Cette matrice (2×2) est calculée à partir des dérivées spatiales I_x, I_y calculées suivant les équations (II.3) et (II.4).

$$I_x(x,y) = \frac{I(x+1,y) - I(x-1,y)}{2} \tag{II.3}$$

$$I_y(x,y) = \frac{I(x,y+1) - I(x,y-1)}{2} \tag{II.4}$$

$$G = \begin{pmatrix} I_x^2 & I_xI_y \\ I_xI_y & I_y^2 \end{pmatrix} \tag{II.5}$$

L'implémentation GPU est effectuée par un traitement parallèle des pixels, en utilisant une grille de calcul GPU contenant un nombre de threads égal au nombre de pixels de l'image. Chaque thread calcule les dérivées spatiales d'un pixel à partir des équations (II.3) et (II.4). Ensuite, le thread peut calculer la matrice G de chaque point de l'image en appliquant l'équation (II.5). Les valeurs des pixels voisins (gauche, droit, haut et bas) de chaque point sont chargées dans la mémoire partagée du GPU, puisque ces valeurs sont utilisées pour le calcul des dérivées spatiales.

3.2.2 Calcul des valeurs propres de la matrice du gradient

À partir de la matrice G, nous calculons les deux valeurs propres de chaque pixel. Ensuite, nous gardons pour chacun la valeur propre ayant la plus grande norme. L'implémentation GPU de cette étape est effectuée par le calcul de ces valeurs en parallèle sur les pixels de l'image. La grille de calcul GPU utilisée contient un nombre de threads égal au nombre de points de l'image.

3.2.3 Recherche de la valeur propre maximale

Une fois les valeurs propres calculées, nous extrayons la valeur propre maximale. Cette valeur est obtenue en faisant appel à la librairie CUDA Basic Linear Algebra Subroutines (CUBLAS) [97]. Cette bibliothèque présente une version GPU, accélérée, de la librairie Basic Linear Algebra Subprograms (BLAS) qui fournit un ensemble de fonctions standardisées réalisant des opérations de base de l'algèbre linéaire.

3.2.4 Suppression des petites valeurs propres

La recherche des petites valeurs propres est réalisée de telle sorte que chaque thread compare la valeur propre de son pixel correspondant à la valeur propre maximale. Si cette valeur est inférieure à 5 % de la valeur maximale, ce pixel est exclu.

3.2.5 Sélection des meilleures valeurs

la dernière étape permet d'extraire pour chaque zone de l'image le pixel ayant la plus grande valeur propre. Pour l'implémentation sur GPU, nous avons affecté à chaque thread GPU un groupe de pixels représentant une zone (10×10 pixels). Chaque thread permet d'extraire la valeur propre maximale dans une zone en utilisant la libraire CUBLAS. Les pixels ayant ces valeurs extraites représentent ainsi les points d'intérêt.

La figure II.8 montre le processus d'implémentation GPU des différentes étapes de détection des points d'intérêt citées ci-dessus.

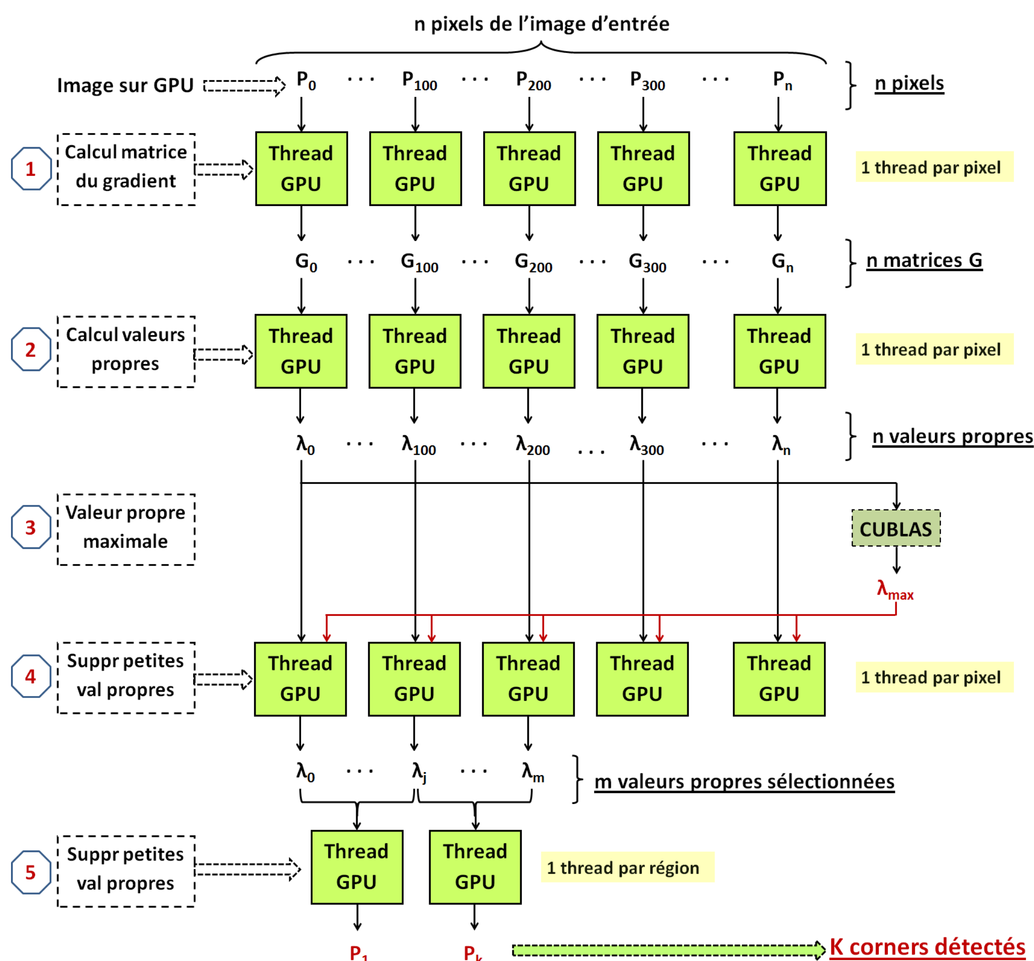


Figure II.8 – Détection des points d'intérêt (coins) sur GPU

3.3 Détection des contours sur GPU

Nous décrivons dans ce paragraphe l'implémentation GPU de la méthode de détection des contours basée sur la technique récursive de Deriche [34] tout en rappelant son principe. La robustesse au bruit de troncature et le nombre réduit d'opérations de cette approche la rendent très efficace au niveau de la qualité des contours extraits. Cependant, cette méthode est entravée par les coûts de calcul qui augmentent considérablement en fonction du nombre et de la taille des images utilisées. L'implémentation séquentielle de cette technique est composée de quatre étapes principales :

1. Calcul des gradients (Gx, Gy) ;
2. Calcul de magnitude et de direction du gradient ;
3. Suppression des non maxima ;
4. Seuillage des contours.

L'étape de calcul des gradients applique un lissage gaussien récursif avant de filtrer l'image à partir des deux noyaux de Sobel (équation (II.7)). Les étapes de calcul de magnitude et de direction du gradient, la suppression des non maxima ainsi que le seuillage sont les mêmes que celles utilisées pour le filtre de Canny [18]. L'implémentation GPU proposée pour cette méthode est basée sur la parallèlisation de ces étapes (figure II.9).

3.3.1 Lissage gaussien récursif

La première étape permet de réduire les bruits de l'image originale, en appliquant à chaque pixel une convolution utilisant le filtre gaussien (équation (II.6)).

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{II.6}$$

L'implémentation GPU du lissage gaussien récursif est fournie par le SDK CUDA [96]. Cette implémentation parallèle utilise le principe la méthode récursive de Deriche. L'image est convoluée dans la direction horizontale avec un opérateur de dérivation de manière récursive. Une convolution verticale sera ensuite appliquée sur le résultat obtenu. L'avantage de cette approche est l'indépendance du temps de calcul par a rapport à la taille du filtre utilisé. Cette implémentation permet ainsi une meilleure immunité au bruit de troncature et par conséquent une meilleure qualité du résultat.

3.3.2 Calcul des gradients de Sobel

L'opérateur de Sobel est appliqué sur l'image pour le calcul de la dérivée verticale et horizontale. Cet opérateur est composé de deux masques de convolution G$_x$ et G$_y$, tel que le montre l'équation (II.7).

$$G_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} * I \qquad G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} * I \tag{II.7}$$

L'implémentation GPU de cette étape est développée en utilisant le SDK CUDA. Elle exploite à la fois la mémoire de texture et la mémoire partagée du GPU, ce qui offre un

accès plus rapide aux données. Elle est basée sur une convolution horizontale appliquée en parallèle sur les colonnes de l'image pour calculer G_x, suivie par une convolution verticale appliquée en parallèle sur les lignes de l'image pour calculer G_y.

3.3.3 Calcul de magnitude et de direction du gradient

Une fois les gradients horizontaux et verticaux (G_x et G_y) ont été calculés, il est possible de calculer la magnitude du gradient G et la direction Θ (équation (II.8)). L'implémentation CUDA est appliquée en parallèle sur les pixels. Ceci est effectué par une grille de calcul GPU contenant un nombre de thread égal au nombre de pixels de l'image. Par conséquent, chaque thread calcule la magnitude et la direction du gradient d'un pixel.

$$G = \sqrt{G_x^2 + G_y^2} \qquad \Theta = \arctan\left(\frac{G_x}{G_y}\right) \tag{II.8}$$

3.3.4 Suppression des non maxima

Après avoir calculé les magnitudes (intensités) et directions du gradient, nous appliquons une fonction CUDA qui extrait les maxima locaux (pixels avec une forte intensité du gradient). Ces points sont considérés comme des parties du bord. Nous avons proposé de charger les valeurs des pixels voisins (gauche, droit, haut et bas) dans la mémoire partagée puisqu'elles sont utilisées pour la recherche des maxima locaux. Le nombre de threads sélectionnés pour la parallélisation de cette étape est égal au nombre de pixels de l'image.

3.3.5 Seuillage des contours

Le seuillage représente la dernière étape de l'extraction des contours. Il est basé sur l'utilisation de deux seuils T_1 et T_2. Chaque pixel ayant une magnitude de gradient supérieure à T_1 est considéré comme pixel de bord, il sera marqué immédiatement. Ensuite, tous les points connectés à ce pixel et qui ont une magnitude de gradient supérieure à T_2 seront également considérés comme pixels de bord et marqués. L'implémentation GPU de cette étape est développée en utilisant la méthode décrite dans [72]. Nous avons étendu cette implémentation par l'utilisation de la mémoire partagée pour avoir un accès plus rapide aux valeurs des pixels connectés. La figure II.9 montre le processus d'implémentation GPU des différentes étapes de détection des contours citées ci-avant.

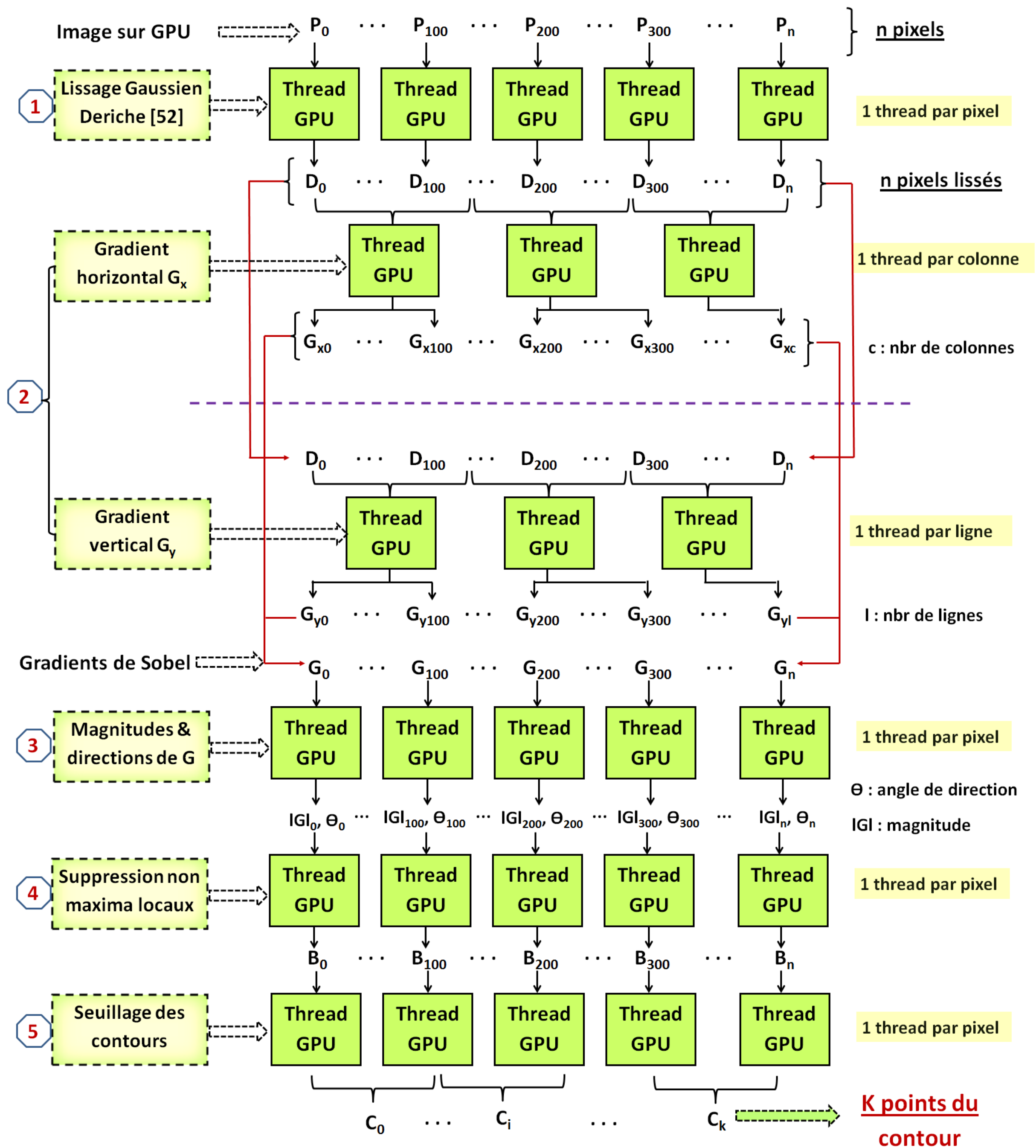


Figure II.9 – Détection GPU des contours basée sur le principe de Deriche-Canny

4 Analyse des résultats

La qualité des résultats est présentée dans la figure II.10 montrant les coins et contours détectés à partir des implémentations décrites ci-avant, en utilisant l'image de « Lena » avec une résolution de 1024×1024. Nous présentons ensuite une analyse des performances de ces résultats en deux catégories : traitement d'images uniques et multiples.

(a) Image originale (b) Détection des coins

(c) Détection des contours (d) Détection des coins et contours

Figure II.10 – Résultats qualitatifs de détection des coins et contours

1. **Performances de traitement d'image unique**

   Une comparaison des temps de calcul entre les implémentations CPU et GPU de la détection des coins et contours est présentée dans les tableaux II.2 et II.3. Ces tableaux présentent les résultats des traitements GPU, visualisés par la bibliothèque de traitement d'image sur CPU « OpenCV », exigeant un transfert des résultats depuis la mémoire GPU vers la mémoire hôte, pouvant coûter jusqu'à 45 % du temps total. Malgré cela, des accélérations significatives, avec facteur de 14 ×, sont obtenues grâce l'exploitation des unités de calculs du GPU en parallèle.

   La visualisation des résultats *via* OpenGL permet d'améliorer ces performances, allant jusqu'à un facteur de 20 × grâce à la réutilisation de tampons CUDA par les fonctions OpenGL, ce qui permet de réduire les temps d'affichage des images de sortie, la visualisation ne coûte plus que 19 % du temps total du traitement (tableaux II.4 et II.5). La figure II.11 montre les gains, en temps, obtenus grâce à la visualisation OpenGL. L'utilisation de la mémoire partagée et de textures a permis d'améliorer encore plus les performances (accélération de 22.6 ×) par rapport à la solution GPU utilisant la mémoire globale (tableaux II.6 et II.7). Cette amélioration est due à l'accès rapide aux valeurs des pixels *via* ces deux mémoires, offrant un chargement et un traitement (kernels) plus rapides des données (figure II.12).

Images	Temps	GPU (Mémoire Globale)				Acc
	CPU	Chargement	Kernels	Vis OpenCV	**Temps Total**	
512×512	**45** *ms*	**2.8** (41 %)	**2.2** (32 %)	**1.9** (28 %)	**6.9** *ms*	**6.52**
1024×1024	**141** *ms*	**4.1** (22 %)	**8.6** (46 %)	**5.9** (32 %)	**18.6** *ms*	**7.58**
2048×2048	**630** *ms*	**8.0** (11 %)	**37.1** (53 %)	**25.4** (36 %)	**70.5** *ms*	**8.94**
3936×3936	**2499** *ms*	**20** (08 %)	**154.8** (62 %)	**76.0** (30 %)	**250.8** *ms*	**9.96**

Tableau II.2 – Détection des coins et contours d'image unique (Vis OpenCV) : GPU FX4800

Images	Temps	GPU (Mémoire Globale)				Acc
	CPU	Chargement	Kernels	Vis OpenCV	**Temps Total**	
512×512	**45** *ms*	**2.0** (34 %)	**1.9** (33 %)	**1.9** (33 %)	**5.8** *ms*	**7.76**
1024×1024	**141** *ms*	**3.1** (21 %)	**6.0** (41 %)	**5.6** (38 %)	**14.7** *ms*	**9.59**
2048×2048	**630** *ms*	**5.2** (10 %)	**24.1** (45 %)	**24** (45 %)	**53.52** *ms*	**11.82**
3936×3936	**2499** *ms*	**12.4** (07 %)	**89.2** (52 %)	**69** (40 %)	**170.6** *ms*	**14.65**

Tableau II.3 – Détection des coins et contours d'image unique (Vis OpenCV) : GPU Tesla M2070

Images	Temps	GPU (Mem. globale)				Acc
	CPU	Chargement	Kernels	Vis OpenGL	**Temps Total**	
512*512	**45** *ms*	**2.8** (47 %)	**2.2** (37 %)	**0.9** (15 %)	**5.9** *ms*	**7.6**
1024*1024	**141** *ms*	**4.1** (26 %)	**8.6** (55 %)	**2.9** (19 %)	**15.6** *ms*	**9.0**
2048*2048	**630** *ms*	**8.0** (14 %)	**37.1** (66 %)	**10.9** (19 %)	**56** *ms*	**11.25**
3936*3936	**2499** *ms*	**20** (10 %)	**154.8** (74 %)	**34.0** (16 %)	**208.8** *ms*	**11.97**

Tableau II.4 – Détection des coins et contours d'image unique (Vis OpenGL) : GPU GPU FX4800

Images	Temps	GPU (Mem. globale)				Acc
	CPU	Chargement	Kernels	Vis OpenGL	**Temps Total**	
512*512	**45** *ms*	**2.0** (43 %)	**1.9** (40 %)	**0.8** (17 %)	**4.7** *ms*	**9.60**
1024*1024	**141** *ms*	**3.1** (27 %)	**6.0** (53 %)	**2.2** (19 %)	**11.3** *ms*	**12.5**
2048*2048	**630** *ms*	**5.2** (14 %)	**24.1** (66 %)	**7** (19 %)	**36.3** *ms*	**17.36**
3936*3936	**2499** *ms*	**12.4** (10 %)	**89.2** (71 %)	**24** (19 %)	**125.6** *ms*	**19.9**

Tableau II.5 – Détection des coins et contours d'image unique (Vis OpenGL) : GPU Tesla M2070

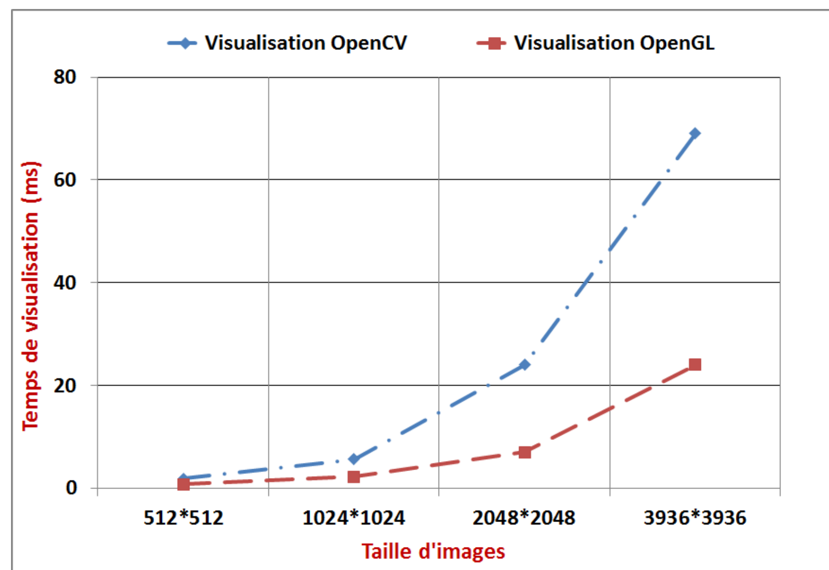


Figure II.11 – Gains, en temps, obtenus grâce à la visualisation OpenGL. GPU FX4800

2. Performances de traitement d'images multiples

Lors du traitement d'images multiples sur processeur graphique, les performances (détection des coins et contours) deviennent moins élevées (allant jusqu'à un facteur 8,7× seulement) tels que montrés dans les tableaux II.8 et II.9. Cette réduction des performances est due principalement aux coûts additionnels de transferts d'images entre la mémoire graphique et la mémoire RAM (environ 30 % du temps total).

Images	Temps	GPU (Mem. tex et Par)				Acc
	CPU	Chargement	Kernels	Vis OpenGL	**Temps Total**	
512*512	**45** *ms*	**2.6** (48 %)	**1.90** (35 %)	**0.9** (17 %)	**5.4** *ms*	**8.3**
1024*1024	**141** *ms*	**3.8** (26 %)	**8.20** (55 %)	**2.9** (19 %)	**14.9** *ms*	**9.5**
2048*2048	**630** *ms*	**7.7** (14 %)	**36.2** (66 %)	**10.6** (19 %)	**54.5** *ms*	**11.6**
3936*3936	**2499** *ms*	**18** (10 %)	**130.2** (71 %)	**34.0** (19 %)	**182.2** *ms*	**13.7**

Tableau II.6 – Détection des coins et contours d'image unique (Mem. tex et Par) : GPU FX4800

Images	Temps	GPU (Mem. tex et Par)				Acc
	CPU	Chargement	Kernels	Vis OpenGL	**Temps Total**	
512*512	**45** *ms*	**1.7** (39 %)	**1.9** (43 %)	**0.8** (18 %)	**4.4** *ms*	**10.2**
1024*1024	**141** *ms*	**2.5** (24 %)	**5.9** (57 %)	**2** (19 %)	**10.4** *ms*	**13.6**
2048*2048	**630** *ms*	**4.7** (14 %)	**21.9** (66 %)	**6.4** (19 %)	**33** *ms*	**19.1**
3936*3936	**2499** *ms*	**9.9** (9 %)	**79.9** (72 %)	**21** (19 %)	**110.8** *ms*	**22.6**

Tableau II.7 – Détection des coins et contours d'image unique (Mem. tex et Par) : Tesla M2070

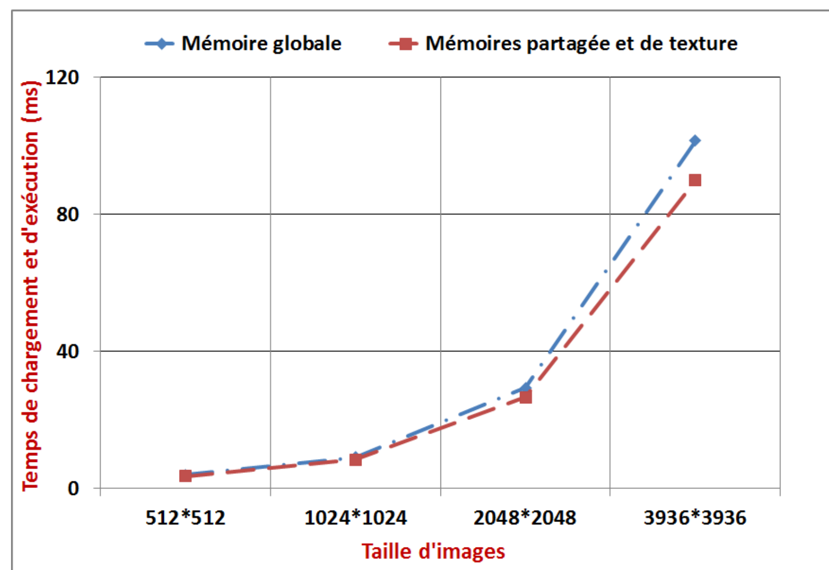


Figure II.12 – Gains obtenus par l'utilisation de la mémoire partagée et de textures. GPU FX4800

L'utilisation de la technique de streaming (4 streams CUDA) permet de réduire les temps de calcul par environ 27 %, offrant ainsi une accélération de 11 ×. Cette amélioration est interprétée par le recouvrement des transferts de données par les kernels d'exécutions sur GPU (tableaux II.10 et II.11). Par conséquent, les temps de copie depuis et vers le processeur graphique sont bien réduits tel que montre la figure II.13. Notons que l'augmentation du nombre d'images n'a pas d'effet sur les facteurs d'ac-

célération puisque les images sont traitées séquentiellement sur GPU. Les traitements parallèles sont appliqués uniquement entre les pixels de la même image.

Images Nbr	Temps CPU	GPU (Mem. tex et Par)				Acc
		Chargement	Kernels	Copie vers CPU	**Temps Total**	
10	**6.00** *s*	**0.09** (11 %)	**0.57** (70 %)	**0.15** (19 %)	**0.81** *s*	**7.4**
50	**30,3** *s*	**0.35** (9 %)	**2.90** (73 %)	**0.73** (18 %)	**3,99** *s*	**7.6**
100	**61.5** *s*	**0.72** (9 %)	**5.50** (71 %)	**1.50** (19 %)	**7.72** *s*	**7,9**
200	**136.1** *s*	**1.44** (9 %)	**11.7** (69 %)	**3.70** (22 %)	**16.8** *s*	**8.1**

Tableau II.8 – Détection des coins et contours d'images multiples (Mem. tex et Par) : GPU FX4800

Images Nbr	Temps CPU	GPU (Mem. tex et Par)				Acc
		Chargement	Kernels	Copie vers CPU	**Temps Total**	
10	**6.00** *s*	**0.08** (11 %)	**0.50** (68 %)	**0.15** (21 %)	**0.73** *s*	**8.2**
50	**30,3** *s*	**0.30** (8 %)	**2.63** (72 %)	**0.73** (20 %)	**3.66** *s*	**8.3**
100	**61.5** *s*	**0.61** (9 %)	**4.84** (70 %)	**1.47** (21 %)	**6.92** *s*	**8.9**
200	**136.1** *s*	**1.30** (8 %)	**10.6** (68 %)	**3.70** (24 %)	**15.6** *s*	**8.7**

Tableau II.9 – Détection de coins et contours d'images multiples (Mem. tex et Par) : GPU Tesla M2070

Images Nbr	Temps CPU	GPU (Streaming)				Acc
		Chargement	Kernels	Copie vers CPU	**Temps Total**	
10	**6.00** *s*	**0.03** (5 %)	**0.57** (86 %)	**0.06** (9 %)	**0.66** *s*	**9.09**
50	**30,28** *s*	**0.13** (4 %)	**2.90** (86 %)	**0.34** (10 %)	**3.37** *s*	**8.99**
100	**61.54** *s*	**0.28** (4 %)	**5.47** (87 %)	**0.56** (9 %)	**6.31** *s*	**9.75**
200	**136.15** *s*	**0.54** (4 %)	**11.63** (86 %)	**1.38** (10 %)	**13.55** *s*	**10.05**

Tableau II.10 – Détection de coins et contours d'images multiples (CUDA streaming) : FX4800

Images Nbr	Temps CPU	GPU (Streaming)				Acc
		Chargement	Kernels	Copie vers CPU	**Temps Total**	
10	**6.00** *s*	**0.03** (5 %)	**0.49** (84 %)	**0.06** (10 %)	**0.58** *s*	**10.34**
50	**30,28** *s*	**0.12** (4 %)	**2.58** (85 %)	**0.33** (11 %)	**3.03** *s*	**09.99**
100	**61.54** *s*	**0.28** (5 %)	**4.79** (85 %)	**0.55** (10 %)	**5.62** *s*	**10.95**
200	**136.15** *s*	**0.50** (4 %)	**10.6** (85 %)	**1.32** (11 %)	**12.4** *s*	**10.98**

Tableau II.11 – Détection des coins et contours d'images multiples (Streaming) : GPU Tesla M2070

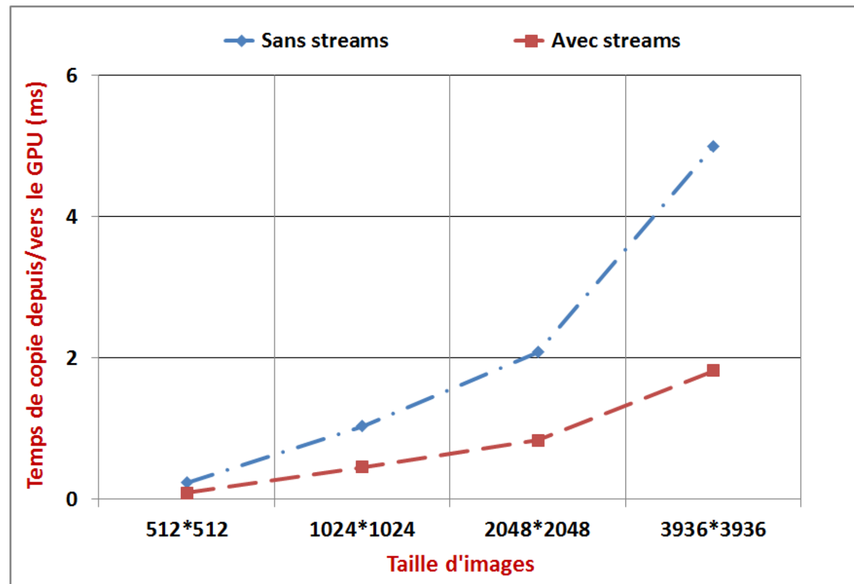


Figure II.13 – Gains obtenus par à l'utilisation de la technique de streaming CUDA. GPU FX4800

Les expérimentations ont été effectuées sous Ubuntu 11.04 avec CUDA 4.0, utilisant différentes plateformes, *i.e.* GPU Quadro FX 4800, GPU Tesla M2070 (Fermi) et CPU Dual core :

- CPU : Intel Dual-core 6600, 2.40 GHz, Mem : 2 GB
- GPU : NVIDIA Quadro FX 4800, 192 CUDA cores, Mem : 1,5 GB
- GPU : NVIDIA TESLA M2070 512 CUDA cores, Mem : 6 GB

5 Conclusion

Nous avons présenté dans ce chapitre la mise en œuvre de certaines méthodes de traitement d'images sur GPU, une mise en œuvre basée sur CUDA pour les traitements parallèles et OpenGL pour la visualisation des résultats. Ceci permet d'avoir des accélérations importantes dans le cas de traitement d'images de haute définition (HD/Full HD). Cependant, les performances des méthodes de traitement de base d'images volumineuse sont sérieusement entravées par les coûts de transferts de données entre mémoires CPU et GPU. On remarque que l'efficacité des GPU est réduite dans le cas du traitement léger de petits volumes de données. Nous proposons ainsi dans le chapitre suivant une solution permettant de faire face à ces contraintes par une exploitation efficace des architectures parallèles et hétérogènes.

Chapitre III

Traitement hétérogène d'images multiples

Sommaire

1 Schéma proposé du traitement hétérogène d'images multiples 75

2 Optimisation des traitements hétérogènes 78

2.1 Ordonnancement efficace des tâches hybrides 79

2.2 Recouvrement des transferts au sein des GPU multiples 80

3 Analyse des résultats 81

3.1 Détection hybride des coins et contours 82

3.2 Performances du calcul hétérogène : ordonnancement statique . . 85

3.3 Performances du calcul hétérogène : ordonnancement dynamique 85

3.4 Performances du calcul hétérogène : CUDA streaming 86

4 Conclusion 88

Les implémentations GPU proposées dans le chapitre précédent ont permis d'atteindre des accélérations allant jusqu'à 22. Ces dernières deviennent moins significatives lors du traitement d'images multiples, puisque le processeur graphique ne permet de paralléliser que les traitements entre pixels, et non entre images. En outre, les applications à faible intensité de calcul (*i.e* images de petite résolution ou à faible taux de calcul par pixel) ne peuvent pas être accélérées sur GPU, elles peuvent même être ralenties. Pour cela, nous proposons un schéma de développement pour le traitement d'images multiples sur plates-formes multicœurs hétérogènes. À partir de ce schéma, nous décrivons les implémentations hybrides de différents algorithmes de traitement d'images cités dans le chapitre précédent. Ce chapitre est composé de trois parties : la première présente le schéma proposé pour les traitements hybrides, la deuxième est dévolue à la description des techniques d'optimisation employées. Une analyse des résultats est décrite dans la troisième partie.

Publications principales liées au chapitre :

1. **Revues internationales :**

   [84] : **S. A. Mahmoudi**, P. Manneback, C. Augonnet, S. Thibault "Traitements d'Images sur Architectures Parallèles et Hétérogènes", *Technique et science informatiques, TSI.* Volume 31 No : 8-9-10/2012. Publié en Décembre 2012.

2. **Conferences internationales :**

   [82] : **S. A. Mahmoudi**, P. Manneback, "Efficient Exploitation of Heterogeneous Platforms for Images Features Extraction", *IPTA 2012 : 3rd International Conference on Image Processing Theory, Tools and Applications*, Istanbul, Turquie. Octobre 2012.

   [83] : **S. A. Mahmoudi**, P. Manneback, C. Augonnet, S. Thibault "Détection optimale des coins et contours dans des bases d'images volumineuses sur architectures multicœurs hétérogènes", *20ème Rencontres Francophones du Parallélisme, RenPar'20*, Saint-Malo, France. Mai 2011.

3. **Workshops internationaux**

   [75] : **S. A. Mahmoudi**, P. Manneback, "Image and video processing on parallel (GPU) and heterogeneous architectures", *2nd Workshop of COST Action IC 0805. Open Network for High-Performance Computing on Complex Environments.* Timisoara, Roumanie. Janvier 2012.

1 Schéma proposé du traitement hétérogène d'images multiples

Comme évoqué dans le chapitre précédent, les implémentations GPU permettent d'avoir des accélérations significatives dans le cas du traitement d'image unique (facteurs d'accélération allant jusqu'à 22,6). Toutefois, ces facteurs deviennent moins importants lors du traitement d'images multiples, ils sont limités à un facteur de 10 seulement. Par conséquent, nous proposons un schéma de développement permettant d'exploiter de manière efficace l'intégralité des ressources des machines hétérogènes (Multi-CPU/Multi-GPU), offrant une solution plus rapide pour le traitement d'images multiples. Cette solution permet également de réduire les transferts entre mémoires CPU et GPU, puisque les images traitées sur CPU ne requièrent aucun transfert entre mémoires. Ce schéma s'appuie sur CUDA pour les traitements parallèles et sur StarPU [8] pour la gestion des plates-formes hétérogènes.

La librairie StarPU offre un support exécutif unifié pour exploiter les architectures multicœurs hétérogènes, tout en s'affranchissant des difficultés liées à la gestion des transferts de données. L'idée principale est de décomposer le traitement en une codelet qui définit le traitement pour des unités de calcul différentes : CPU, GPU et/ou processeur CELL. StarPU se charge de lancer les tâches d'exécution sur les données avec la codelet. Le placement des tâches est défini par les versions d'implémentations de la codelet, les disponibilités des unités de calcul et un ordonnanceur de tâches. Le traitement global des données est réalisé simultanément sur le(s) CPU(s) et le(s) GPU(s). Si nécessaire, StarPU gère les transferts de données entre les différentes mémoires des unités de calcul. Par ailleurs, StarPU propose plusieurs stratégies d'ordonnancement efficaces et offre en outre la possibilité d'en concevoir aisément de nouvelles [10].

Le schéma de développement proposé pour le traitement hybride d'images multiples repose sur trois étapes principales : chargement des images d'entrée, traitement hétérogène d'images, présentation des résultats.

1. **Chargement des images d'entrée :**
   La première étape consiste en un chargement des images d'entrée dans des files d'attente de telle sorte que StarPU puisse appliquer les traitements à partir de ces files. Le listing III.1 résume cette étape.

```
1 for (i = 0; i < n; ++i) {              // n: nombre d'images.
2  img = cvLoadImage(tab_images[i]);
3  starpu_data_handle img_handle;
4  starpu_vector_data_register(&img_handle, 0, img , size);
5  List = add(List, img, img_handle);
6  }
```

Listing III.1 – Chargement des images d'entrée

La ligne 2 permet de charger l'image *i* en mémoire centrale à partir du tableau d'images tab_images (ensemble d'images à traiter). Ce chargement est effectué *via* la fonction « cvLoadImage » de la bibliothèque OpenCV [102]. Les lignes 3 et 4 permettent d'allouer un tampon (handle) StarPU qui dispose de l'adresse de l'image chargée. La cinquième ligne permet d'ajouter l'image ainsi que le tampon StarPU dans une liste chaînée (file d'attente) qui contiendra l'ensemble des images à traiter.

2. **Traitement hétérogène d'images :**

   Une fois les images chargées, le traitement hétérogène est confié à StarPU qui lance les tâches à partir des fonctions implémentées en versions CPU (séquentielle) et GPU (parallèle). Le traitement StarPU se base sur deux structures principales : la codelet et les tâches. La codelet permet de préciser sur quelles architectures le noyau de calcul peut s'effectuer, ainsi que les implémentations associées (listing III.2). Ensuite, les tâches appliquent la codelet sur l'ensemble des images, de telle sorte que chaque tâche est créée et lancée pour traiter une image de la file (listing III.3).

```
1 static starpu_codelet cl ={
2  .where = STARPU_CPU|STARPU_CUDA,  // coeurs CPU et GPU
3  .cpu_func = cpu_impl,             // définir la fonction CPU
4  .cuda_func = cuda_impl,           // définir la fonction GPU
5  .nbuffers = 1                     // nombre de tampons
6  .model = &model_perf;
7   }
```

Listing III.2 – La codelet StarPU

La ligne 2 permet de sélectionner les ressources éligibles (CPU et GPU) pour effectuer le calcul hétérogène. Les lignes 3 et 4 indiquent les fonctions CPU et GPU respectivement. Les fonctions CPU sont développées avec la bibliothèque OpenCV, tandis que les fonctions GPU sont décrites dans le chapitre précédent. La ligne 5 définit le nombre de tampons StarPU utilisés. Nous avons choisi un seul tampon puisque chaque tâche traite une image. La sixième ligne permet d'indiquer le modèle de performances utilisé pour ordonnancer les tâches de manière efficace. Ce modèle est décrit dans la section suivante (III.2).

```
1 while (List != NULL) {
2  task = starpu_task_create();        //créer la tâche
3  task->cl = &cl;                     //définir la codelet
4  task->.handle = List->img_handle;  //définir le tampon
5  task->.mode = STARPU_RW;            //mode Lecture/écriture
6  starpu_task_submit(task);           //soumettre la tâche
7  List = List->next;                  //image suivante
8 }
```

Listing III.3 – Soumission des tâches StarPU à l'ensemble des images

La ligne 1 permet de lancer la boucle de traitement hétérogène tant qu'on n'est pas arrivé au bout de la liste. La ligne 2 est utilisée pour créer une tâche StarPU, les lignes 3 et 4 permettent d'associer à chaque tâche créée la codelet définie ci-avant (listing III.2), ainsi que le tampon StarPU (définit dans le listing III.1) contenant l'image courante à traiter. La cinquième ligne indique le mode lecture/écriture puisque les images seront traitées et modifiées. La soumission de la tâche est effectuée *via* la ligne 6 avant de passer à l'image suivante (ligne 7).

3. **Présentation des résultats :**

   Lorsque toutes les tâches StarPU sont terminées, les résultats des traitements GPU doivent être rapatriés dans les tampons. La mise à jour est assurée par une fonction spécifique de StarPU. Cette fonction se charge aussi de transférer les données depuis la mémoire graphique vers la mémoire centrale dans le cas des traitements exécutés sur GPU.

La figure III.1 résume les différentes étapes décrites ci-dessus pour le traitement hybride d'images multiples. Cette figure montre également l'utilisation de la mémoire partagée et de textures permettant d'assurer un accès plus rapide aux données lors du lancement des tâches sur processeur graphique.

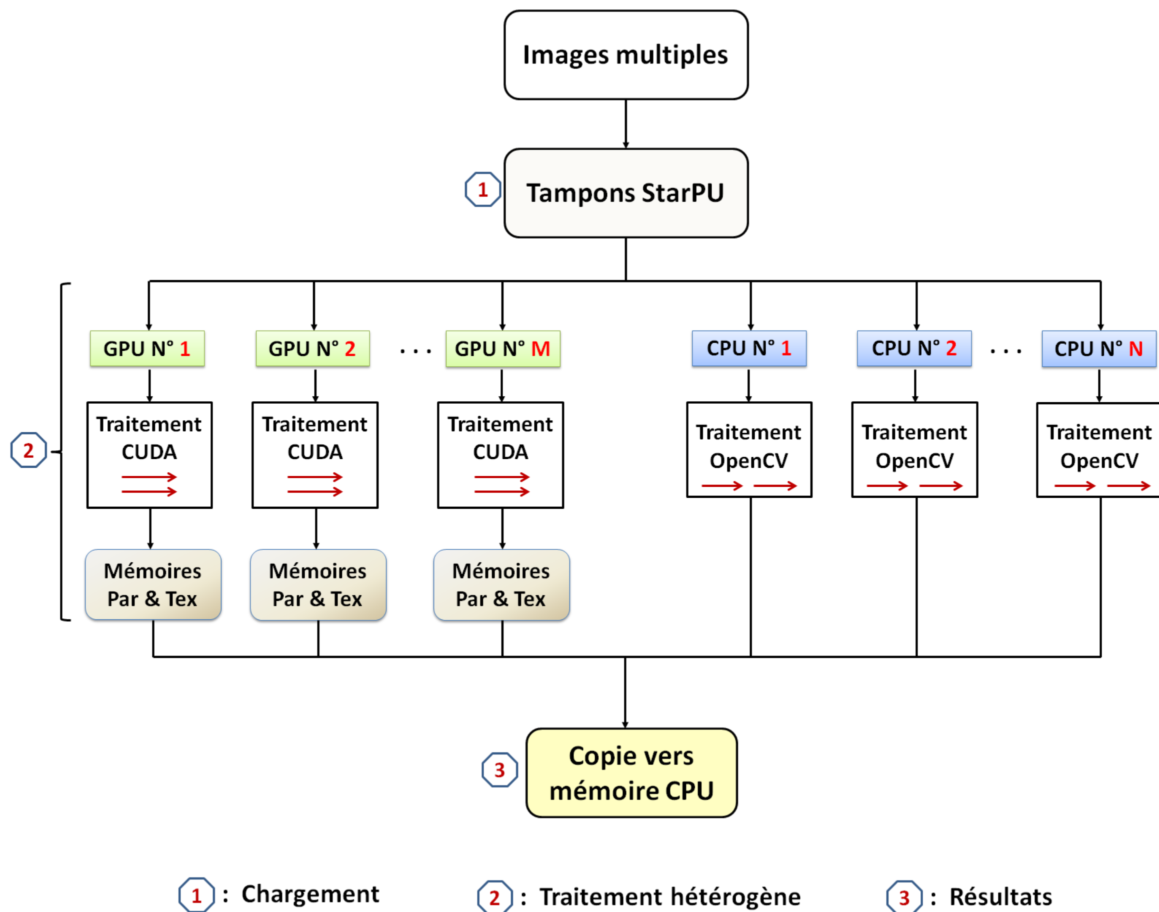


Figure III.1 – Schéma proposé du traitement hétérogène d'images multiples

2 Optimisation des traitements hétérogènes

Pour une meilleure exploitation des unités de calcul hétérogènes, nous proposons d'optimiser les traitements *via* deux techniques différentes :

1. Ordonnancement dynamique des tâches hybrides ;
2. Recouvrement des transferts de données au sein des GPU multiples.

2.1 Ordonnancement efficace des tâches hybrides

L'exploitation des unités de calcul hétérogènes nécessite un ordonnancement efficace de tâches. Celles-ci doivent être lancées dans un ordre assurant une exploitation maximale des cœurs CPU et GPU multiples. Nous présentons dans cette section les deux stratégies d'ordonnancement employées : ordonnancements statique et dynamique.

2.1.1 Ordonnancement statique

Cet ordonnancement se base sur une file d'attente (queue) de tâches ayant la même priorité (0). Chaque nouvelle tâche soumise est mise à la fin de la queue (premier arrivé/-premier servi). Notons que lors du traitement d'images multiples, les tâches sont soumises de manière asynchrone. Aucune dépendance de calcul entre images n'est présente dans ce cas.

Cet ordonnancement peut être efficace lorsque les mêmes instructions (tâches) sont appliquées sur des images ayant la même taille. Cependant, l'application des tâches variées sur des images de différentes tailles nécessite un ordonnancement plus subtil, permettant d'affecter les tâches sur les ressources les mieux adaptées.

2.1.2 Ordonnancement dynamique

Pour obtenir un ordonnancement efficace, il est nécessaire d'avoir une estimation de la durée de calcul de chacune des tâches lancées. Cette estimation peut être effectuée en fournissant un modèle de performances à la codelet (listing III.2, ligne 6). Ce modèle, défini par StarPU, suppose que pour un ensemble d'images de résolutions similaires, les performances resteront très proches. StarPU extrait ensuite la durée moyenne à partir des exécutions précédentes sur les différentes unités de calcul. Cette durée permettra de lancer les tâches sur les processeurs (CPU ou GPU) offrant des temps de calcul plus rapides. Le listing III.4 décrit notre modèle de performances basé sur l'historique des temps d'exécution des tâches.

Dans notre cas, nous employons l'ordonnanceur deque model data aware (dmda) qui prend en compte à la fois le modèle de performances des exécutions et le temps de transfert des données [9]. De ce fait, La durée de tâches estimée prend en compte les temps d'exécution et de transferts des tâches antérieurs (appliquées à des données de taille similaire) afin d'avoir plus de précision. Par conséquent, cet ordonnanceur lance les tâches de telle sorte que le temps de calcul total soit minimum. Notons aussi que les tâches sont lancées

de manière asynchrone.

```
1 static struct starpu_perfmodel mult_perf_model = {
2  .type = STARPU_HISTORY_BASED,          // type du  modèle
3  .symbol= "model_perf"                  // nom du modèle
4 }
```

Listing III.4 – Modèle de performances utilisé pour l'ordonnancement

La ligne 2 permet de définir un modèle de performances basé sur l'historique des temps d'exécution des tâches précédentes. La ligne 3 associe un nom au modèle qui est utilisé lors de l'appel du modèle de performances depuis la codelet.

2.2 Recouvrement des transferts au sein des GPU multiples

L'exploitation des processeurs graphiques multiples nécessite également une bonne gestion des zones mémoires. En effet, la technique de streaming *via* CUDA est employée au sein des GPU multiples. Nous proposons ainsi de créer quatre streams CUDA pour chaque processeur graphique de telle sorte que chaque GPU puisse recouvrir de manière efficace les transferts de données par l'exécution des kernels CUDA. Notre implémentation hybride est basée sur deux étapes seulement (au lieu des trois étapes décrites dans la section III.1) :

1. Chargement des images d'entrée ;
2. Traitements hybrides et transfert des résultats.

2.2.1 Chargement des images d'entrée

Cette étape est identique à celle représentée dans la section III.1.1. Elle consiste simplement à charger les images dans des tampons StarPU afin de les traiter par la suite.

2.2.2 Traitements hybrides et transfert des résultats

Cette étape regroupe les phases de traitement hybride et de transfert des résultats (décrites dans les sections III.1.2 et III.1.3). Ce regroupement de phases est appliqué afin de bénéficier du recouvrement des transferts par les exécutions sur GPU. Chaque fonction GPU sera composée de deux étapes :

1. Lancement des kernels CUDA pour le traitement d'images ;
2. Copie des résultats (images de sortie) vers la mémoire CPU.

Ces deux étapes sont recouvertes par les streams CUDA. Dans ce cas, nous avons aussi sélectionné quatre streams puisqu'ils offrent de meilleures performances (pour les algorithmes d'extraction de caractéristiques d'images) tel que montré dans le figure V.5(b) du chapitre précédent. La figure III.2 illustre les optimisations (ordonnancement dynamique et streaming CUDA) appliquées aux étapes du traitement hybride d'images multiples.

3 Analyse des résultats

Nous présentons dans cette section une analyse des résultats obtenus grâce aux implémentations hétérogènes exploitant à la fois les processeurs centraux et graphiques. Cette analyse permet d'évaluer les implémentations hybrides d'algorithmes de détection des coins et contours, à partir du schéma proposé dans la section III.1. Les traitements sont appliqués sur un ensemble de 200 images (images standards de « Lena » et « Mandrill ») de différentes résolutions présentées comme suit :

- 50 images avec une résolution de 512 × 512 ;
- 50 images avec une résolution de 1024 × 1024 ;
- 50 images avec une résolution de 2048 × 2048 ;
- 50 images avec une résolution de 3936 × 3936.

Les expérimentations ont été effectuées sous Ubuntu 11.04 avec CUDA 4.0, utilisant deux plates-formes différentes , soit GPU Tesla C1060 et CPU Dual core :

- CPU : Intel Dual Core 6600, 2.40 GHz, Mem : 2 GB ;
- GPU : NVIDIA Tesla C1060, 240 CUDA cores, Mem : 4GB.

Cette section est décrite en quatre parties : la première présente en détail les fonctions CPU et GPU utilisées pour appliquer une détection hétérogène des coins et contours d'images multiples. La deuxième partie montre les résultats du traitement hybride à partir d'un ordonnancement statique des tâches. La troisième partie décrit les résultats du même traitement mais à partir d'un ordonnancement dynamique des tâches, tandis que la

quatrième partie est dévolue à la présentation des performances du traitement hétérogène d'images multiples en exploitant la technique du streaming.

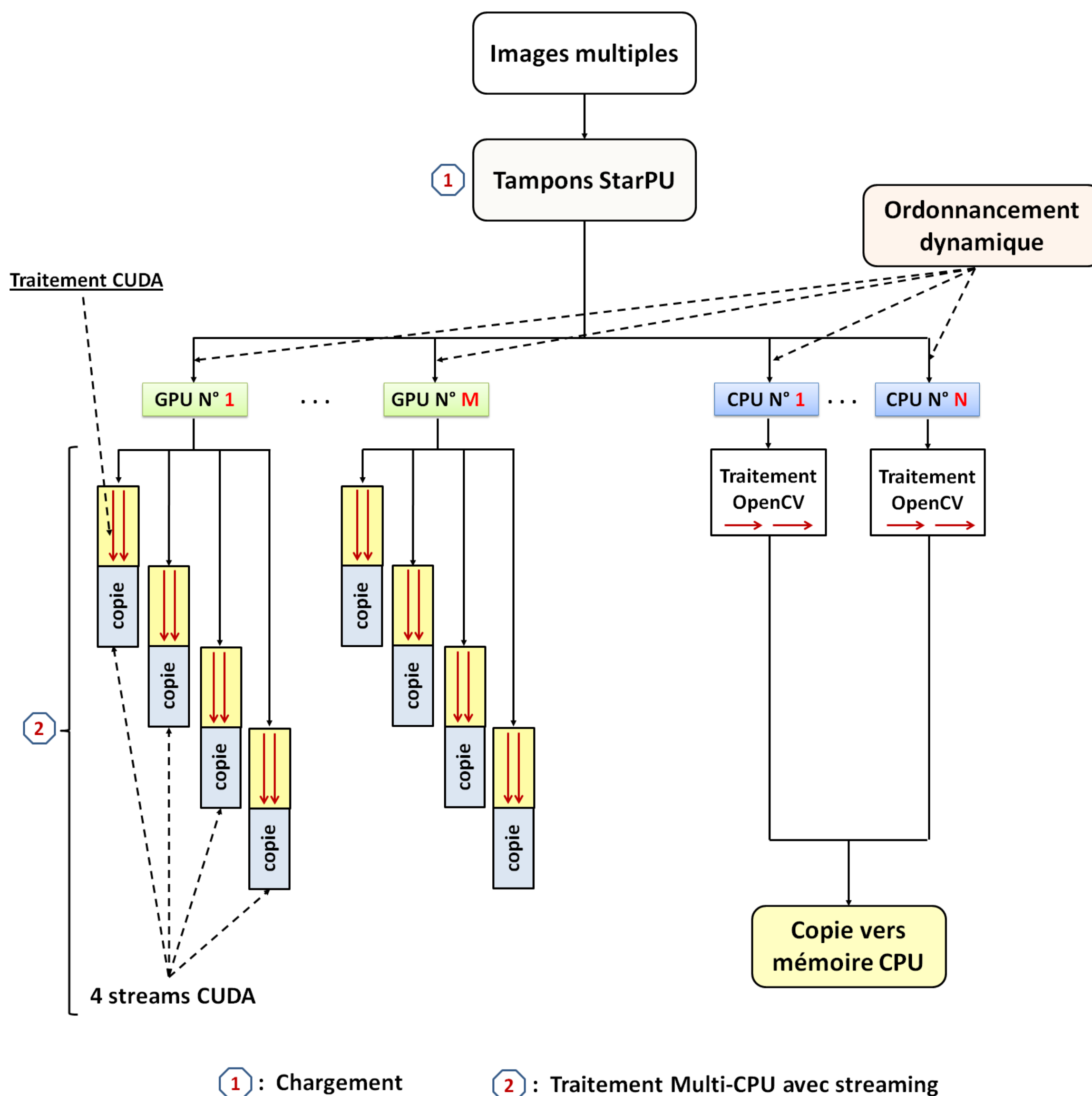


Figure III.2 – Optimisation du traitement hétérogène d'images multiples

3.1 Détection hybride des coins et contours

La mise en œuvre hétérogène de détection des coins et contours est effectuée à base du schéma proposé dans la section III.1. Les étapes de chargement d'images et de soumission des tâches s'effectuent à partir des listings III.1 et III.3 respectivement. Pour lancer les

tâches sur tous les processeurs CPU et GPU disponibles sur la machine, nous utilisons la codelet définie dans le Listing III.2. Cette codelet définit également les fonctions CPU et GPU requises pour le fonctionnement du traitement hétérogène. La fonction CPU (de détection des coins et contours) est décrite dans le listing III.5, tandis que la fonction GPU est décrite dans le listing III.6.

```
1  void cpu_impl(void *descr[], void *_args)
2  {
3  const int size = width * height;                 //taille de l'image
4  uchar* output_data;                              //données de sortie
5  IplImage* input, output;                         //images intermédiaires
6  img_starpu = STARPU_VECTOR_GET_PTR(descr[0]); //récup des pixels
7  input = cvCreateImage(cvSize(width, height));
8      for (int i = 0; i < (size); i++)
9      {
10          input->imageData[i] = img_starpu[i]; //pixels-->image OpenCV
11     }
12 cvCanny(input, output);                          //détection de contours
13 cvGoodFeaturesToTrack(output, points);           //detection de coins
14 output_data = (uchar*) output->imageData;
15     for (int j = 0; j < size; j++)
16         {
17         img_starpu[j] = (float) data[j]          //mise à jour
18         }
19 }
```

Listing III.5 – fonction CPU de détection des coins et contours

Les lignes de 3 à 5 permettent de définir la taille de l'image à traiter ainsi que des pointeurs vers des images intermédiaires utilisées pour le traitement CPU. La sixième ligne permet de récupérer le tableau de pixels (img_starpu) chargé *via* StarPU (Listing III.1). Les lignes de 7 à 11 servent à mettre ce tableau dans une image sous format OpenCV (input), tandis que les lignes 12 et 13 appliquent une détection des coins et contours sur cette image à partir des fonctions de la bibliothèque OpenCV. Les dernières lignes (de 14 à 18) chargent le résultat du traitement CPU dans un tableau (output_data) qui servira à mettre à jour le tableau de pixels d'origine (img_starpu).

```
1 void cuda_codelet(void *descr[], void *_args)   //size : width*height
2 {
3 img_starpu = STARPU_VECTOR_GET_PTR(descr[0]);   //récup image chargée
4 cudaMalloc(img_edge, size);                     //image de contours
5 cudaMalloc(img_corner, size);                   //image de coins
6 dim3 block(BLOCK_WIDTH, BLOCK_HEIGHT, 1);       //taille des blocs
7 dim3 grid((w/block.x), (h/block.y),1);          //nombre de blocs
8 gpu_edge<<<grid, block>>>(img_starpu, img_edge);//détection contours
9 gpu_corner <<<grid, block>>>(img_edge, img_corner);//détection coins
10 cudaMemcpy(img_starpu, img_corner,DeviceToDevice);
11 cudaFree(img_edge);                            //libéartion mém gpu
12 cudaFree(img_corner);                          //libération mém gpu
13 }
```

Listing III.6 – fonction GPU de détection des coins et contours

La ligne 3 permet de récupérer l'image chargée *via* StarPU (img_starpu) afin de la traiter (sur GPU) par la suite. Les lignes 4 et 5 servent à allouer deux espaces mémoire sur GPU, ces derniers contiendront les résultats de détection des coins et de contours. La sixième ligne définit le nombre de threads GPU par blocs (multiple de 16 dépendant du type de la carte graphique), tandis que la ligne 7 permet de calculer le nombre de blocs dans la grille de calcul GPU. Ce nombre est obtenu en divisant la taille de l'image par le nombre de threads dans chaque bloc. La huitième ligne fait appel à la fonction GPU de détection des contours (décrite dans la section II.3) pour l'appliquer à l'image chargée (img_starpu). De même, la neuvième ligne fait appel à la fonction GPU de détection des coins afin de l'appliquer sur l'image des contours obtenue (img_edge). La dixième ligne effectue une copie de l'image résultante (img_corner) vers l'image d'origine "img_starpu" afin de la mettre à jour.

Les résultats des traitements CPU ou GPU seront récupérés ultérieurement (dans le programme principal : main) par l'appel d'une fonction StarPU "starrpu_data_acquire (tampon, STARPU_R)" appliquant une mise à jour des tampons StarPU utilisés.

3.2 Performances du calcul hétérogène : ordonnancement statique

Ce paragraphe présente les résultats d'exploitation des multiples CPU et GPU, en utilisant une stratégie simple d'ordonnancement donnant la même priorité à toutes les tâches (premier arrivé premier servi) tel que décrit dans la section III.2.1.1. Ces résultats (tableau III.1) montrent des accélérations significatives allant jusqu'à un facteur de 19 par rapport à une mise en œuvre exploitant un seul processeur graphique (accélération d'un facteur de 10). Notons aussi que ces accélérations augmentent en fonction du nombre d'images traitées puisque les traitements sont appliqués en parallèle à la fois entre pixels (cœurs d'un GPU) et images (CPU et GPU).

Images	2CPU	8CPU	1GPU	1GPU-2CPU	2GPU	2GPU-4CPU	4GPU	4GPU-8CPU
Nbr	**Acc**	**Acc**	**Acc**	**Acc**	**Acc**	**Acc**	**Acc**	**Acc**
10	1,32 ×	3,53 ×	10,34 ×	10,53 ×	11,54 ×	11,54 ×	12,24 ×	12,77 ×
50	1,54 ×	3,40 ×	09,99 ×	10,09 ×	12,06 ×	12,41 ×	15,77 ×	16,19 ×
100	1,56 ×	3,42 ×	10,95 ×	11,05 ×	12,43 ×	12,93 ×	16,86 ×	17,34 ×
200	1,54 ×	3,40 ×	10,98 ×	11,30 ×	13,03 ×	14,20 ×	17,94 ×	19,79 ×

Tableau III.1 – Détection des coins et contours d'images multiples : ordonnancement statique

3.3 Performances du calcul hétérogène : ordonnancement dynamique

Le tableau III.2 présente les performances obtenues lors de l'utilisation de la stratégie d'ordonnancement basée sur l'historique des tâches précédentes (décrite à la section III.2.1.2). Cet ordonnancement permet d'améliorer les performances, qui pouvent atteindre un facteur de 22 au lieu de 19 (obtenu avec un simple ordonnancement). Cette amélioration est due à une meilleure exploitation des ressources. Les tâches demandant plus de calcul sont prioritaires pour un traitement sur GPU, alors que les tâches à faible intensité de calcul sont exécutées sur CPU.

Notons que l'utilisation d'un ordonnancement dynamique a permis d'améliorer les performances dans le cas des traitements Multi-CPU/Multi-GPU uniquement. Les performances du traitement Multi-GPU ne sont pas améliorées *via* cet ordonnancement puisqu'on dispose du même type de ressources et par conséquent le choix des unités de calcul

n'est plus nécessaire. De même, si on applique un traitement Multi-CPU, un simple ordonnancement statique sera suffisant.

Images	2CPU	8CPU	1GPU	1GPU-2CPU	2GPU	2GPU-4CPU	4GPU	4GPU-8CPU
Nbr	**Acc**	**Acc**	**Acc**	**Acc**	**Acc**	**Acc**	**Acc**	**Acc**
10	01,32 ×	03,53 ×	09,09 ×	10,53 ×	11,54 ×	11,76 ×	12,24 ×	13,04 ×
50	1,54 ×	03,40 ×	08,99 ×	10,16 ×	12,06 ×	12,89 ×	15,77 ×	16,92 ×
100	1,56 ×	3,42 ×	09,75 ×	11,23 ×	12,43 ×	13,41 ×	16,86 ×	18,99 ×
200	1,54 ×	3,40 ×	10,05 ×	11,33 ×	13,03 ×	14,66 ×	17,94 ×	22,39 ×

Tableau III.2 – Détection des coins et contours d'images multiples : ordonnancement dynamique

3.4 Performances du calcul hétérogène : CUDA streaming

Le tableau III.3 montre les performances obtenues grâce l'exploitation de la technique de streaming *via* CUDA. Celle-ci permet de recouvrir les transferts de données par les exécutions de kernels (section III.2.2) sur les processeurs graphiques multiples. Les streams CUDA nous ont permis de réduire les temps de calcul, offrant une accélération d'un facteur de 27. En outre, le recouvrement des transferts de données par les kernels d'exécutions a permis d'améliorer les performances dans les cas des traitements Multi-GPU et Multi-CPU/Multi-GPU. Le streaming est donc utile dans tous les cas exploitant un ou plusieurs processeurs graphiques. La figure III.3 montre les gains, en temps, obtenus grâce à l'utilisation des streams CUDA. Les traitements sont accélérés d'environ 22 % par rapport à un traitement qui n'exploite pas les streams CUDA.

Images	2CPU	8CPU	1GPU	1GPU-2CPU	2GPU	2GPU-4CPU	4GPU	4GPU-8CPU
Nbr	**Acc**	**Acc**	**Acc**	**Acc**	**Acc**	**Acc**	**Acc**	**Acc**
10	01,32 x	03,53 x	10,34 x	10,53 x	12,00 x	12,24 x	12,77 x	13,64 x
50	1,54 x	03,40 x	09,99 x	10,44 x	12,56 x	13,46 x	16,64 x	17,81 x
100	1,56 x	3,42 x	10,95 x	11,35 x	12,74 x	13,95 x	19,85 x	21,08 x
200	1,54 x	3,40 x	10,98 x	11,94 x	13,55 x	15,06 x	22,28 x	27,01 x

Tableau III.3 – Détection des coins et contours d'images multiples : CUDA streaming (4 streams)

Ces résultats montrent également que les facteurs d'accélération augmentent en fonction du nombre d'images traitées. Ceci est dû au traitement parallèle des images à deux niveaux :

1. Une parallélisation de bas niveau qui s'exprime par le portage de l'application sur GPU (traitements parallèles intra-image).
2. Une parallélisation de haut niveau (traitements inter-images) exploitant à la fois les CPU et les GPU, chaque cœur traitant un sous-ensemble d'images.

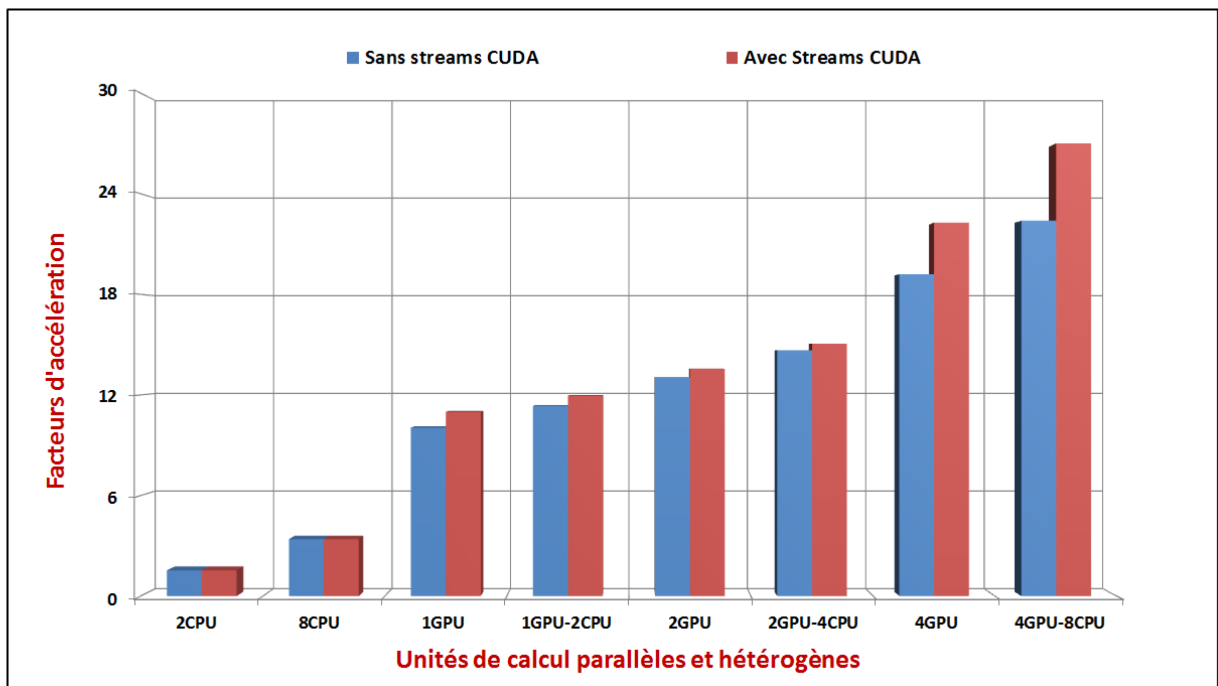


Figure III.3 – Gains, en temps, obtenus grâce au streaming CUDA

À partir de la comparaison des résultats présentés dans les trois tableaux ci-dessus, nous pouvons proposer des directives d'optimisation offrant une exploitation efficace des ressources hétérogènes (Multi-CPU/Multi-GPU) lors du traitement d'images multiples. Le tableau III.4 montre les optimisations proposées par rapport au type de ressources utilisées (GPU, Multi-GPU, Multi-CPU/Multi-GPU). Les traitements Multi-GPU utilisent un ordonnancement statique car toutes les ressources sont similaires. Au contraire, les traitements Multi-CPU/Multi-GPU nécessitent un ordonnancement dynamique basé sur l'historique des tâches, afin de mieux affecter les ressources (CPU ou GPU) aux tâches.

	GPU	Multi-GPU	Multi-CPU/Multi-GPU
Ordonnancement statique	Non	Oui	Non
Ordonnancement dynamique	Non	Non	Oui
Streaming CUDA	Oui	Oui	Oui

Tableau III.4 – Recommandations d'optimisation du traitement d'images multiples

4 Conclusion

Nous avons présenté dans ce chapitre un schéma de développement pour le traitement efficace d'images multiples sur architectures parallèles et hétérogènes (Multi-CPU/Multi-GPU). Nous avons également proposé d'ordonnancer les tâches hybrides de façon dynamique à partir de l'estimation des temps de calcul et de transfert des tâches précédentes. Les tâches exigeant un calcul intensif seront prioritaires pour un traitement sur GPU. Les tâches à faible taux de calcul seront prioritaires pour un traitement sur CPU. Par ailleurs, nous avons présenté l'intérêt du recouvrement des transferts de données par les kernels d'exécutions sur les GPU multiples, en utilisant la technique de streaming *via* CUDA.

Chapitre IV

Traitement Multi-GPU de vidéo Full HD

Sommaire

1 Détection d'objets en temps réel sur GPU 91
1.1 Soustraction de l'arrière-plan sur GPU 91
1.2 Extraction de silhouette sur GPU 94
1.3 Analyse des résultats 96
2 suivi de mouvements en parallèle sur GPU 97
2.1 Mesure du flot optique 97
2.2 Algorithme de suivi de mouvements 99
2.3 Suivi de mouvements sur GPU uniques ou multiples 104
2.4 Analyse des résultats 106
3 Conclusion 114

Les méthodes de traitement de vidéos sont au même niveau que les méthodes de traitement d'images, elles présentent de très bons champs d'applications pour l'accélération sur GPU. Cependant, le traitement de vidéo exige de respecter la contrainte de temps réel (25 fps), qui devient hors de portée des CPU, lors de l'utilisation de vidéos de haute définition (HD/Full HD). Notre contribution porte sur l'exploitation efficace des processeurs graphiques uniques ou multiples afin de faire face aux contraintes citées ci-dessus. Notre intérêt s'est porté en particulier sur les méthodes d'analyse, de détection et de suivi de mouvements, qui présentent le cœur de nombreuses applications de vision par ordinateur, telles que la détection d'évènements [41], la reconnaissance d'objets ainsi que le suivi de robots [101]. Ce chapitre est présenté en deux parties : la première décrit notre mise en œuvre sur GPU de différentes méthodes de détection d'objets en mouvement, tandis que la seconde décrit notre implémentation Multi-GPU d'une méthode de suivi de mouvements à partir des vecteurs du flot optique.

Publications principales liées au chapitre :

1. **Revue internationale (en soumission) :**

   S. A. Mahmoudi, M. Kierzynka, P. Manneback, K. Kurowski "Real-time motion tracking using optical flow on multiple GPUs", *Bulletin of the Polish Academy of Sciences*. Soumis le 10/12/2012.

2. **Chapitre de livre (en soumission) :**

   S. A. Mahmoudi, Erencan Ozkan, Pierre Manneback, Souleyman Tosun "Efficient exploitation of Multi-CPU/Multi-GPU platforms for High Definition image and video processing", *Complex HPC book, Wiley books*. Soumis le 05/08/2012.

3. **Workshops internationaux :**

   [76] : **S. A. Mahmoudi**, H.Sharif, N.Ihaddadene, C.Djeraba, "Abnormal Event Detection in Real Time Video", *First International Workshop on Multimodal Interactions Analysis of Users in a Controlled Environment*. Crete, Greece. Novembre 2008.

   [74] : **S. A. Mahmoudi**, P. Manneback, " GPU-based real time tracking in high definition videos", *3rd Workshop of COST 0805. Open Network for High-Performance Computing on Complex Environments*. Gênes, Italie. 17-18 Avril 2012.

4. **Publications nationales :**

   [86] : M. Mancas, R. B. Madkhour, **S. A. Mahmoudi**, T. Ravet, "VirTrack : Tracking for Virtual Studios", *QPSR of the numediart research program, volume 3, No. 1, pp. 1-4*, Mars 2010.

   [85] : M. Mancas, M. Bagein, N. Guichard, S. Hidot, C. Machy, **S. A. Mahmoudi**, X. Siebert, "AVS : Augmented Virtual Studio", *QPSR of the numediart research program, Vol. 1, No. 4*, Decembre 2008.

1 Détection d'objets en temps réel sur GPU

La détection d'objets (ou de personnes) en mouvement représente une étape nécessaire à de nombreux processus de suivi et d'analyse d'objets. Généralement, la détection de mouvement ne requiert pas de traitement intensif puisqu'elle est basée sur des techniques basiques de soustraction de background ou de différences entre images successives. Toutefois, le respect de la contrainte du traitement temps réel des vidéos HD ou Full HD ne peut pas être garanti à partir d'un traitement séquentiel sur CPU. Un traitement efficace sur GPU peut résoudre ce problème en appliquant un calcul parallèle sur les pixels. Cette solution permet également de simplifier l'application des méthodes de suivi de mouvements à partir de données déjà traitées (objets détectés) sur GPU. Nous présentons dans cette section les implémentations GPU proposées pour l'extraction du background ainsi que la détection de silhouette avec une analyse détaillée des résultats obtenus.

1.1 Soustraction de l'arrière-plan sur GPU

La soustraction de l'arrière-plan permet d'éliminer les objets statiques dans une scène et de se concentrer uniquement sur les objets en mouvement. Cette méthode est basée sur la soustraction de l'image du fond (background) à toutes les images (frames) de la séquence vidéo. L'implémentation GPU de cette méthode peut être résumée en trois étapes (figure IV.1) : lecture et sauvegarde du background, soustraction du background et visualisation OpenGL.

1. **Lecture et sauvegarde de l'arrière-plan :** la première étape permet d'effectuer la lecture de la première image (background) de la vidéo à l'aide de la bibliothèque OpenCV [102]. Une fois l'image chargée, elle est copiée en mémoire GPU afin de pouvoir l'utiliser lors des prochaines étapes.
2. **Soustraction de l'arrière-plan :** cette étape permet le chargement et la lecture des autres images de la vidéo (à partir de la deuxième image), chaque image doit être chargée en mémoire GPU. Ensuite, on lui soustrait l'image du background stockée lors de la première étape.
3. **Visualisation OpenGL :** une fois la soustraction effectuée, on peut visualiser l'image résultante à l'aide de la bibliothèque OpenGL [103]. Après visualisation de chaque image, nous revenons à l'étape de soustraction de background pour appliquer le même processus aux trames restantes de la vidéo.

La figure IV.2(a) présente le résultat de l'application de la soustraction du background sur GPU. Elle montre ainsi les véhicules en mouvement détectés après élimination de l'arrière-plan (background) puisque la caméra est immobile dans ce cas. La figure IV.2(b) montre le résultat de la soustraction du fond bleu représentant l'arrière-plan afin d'extraire la silhouette de la personne uniquement.

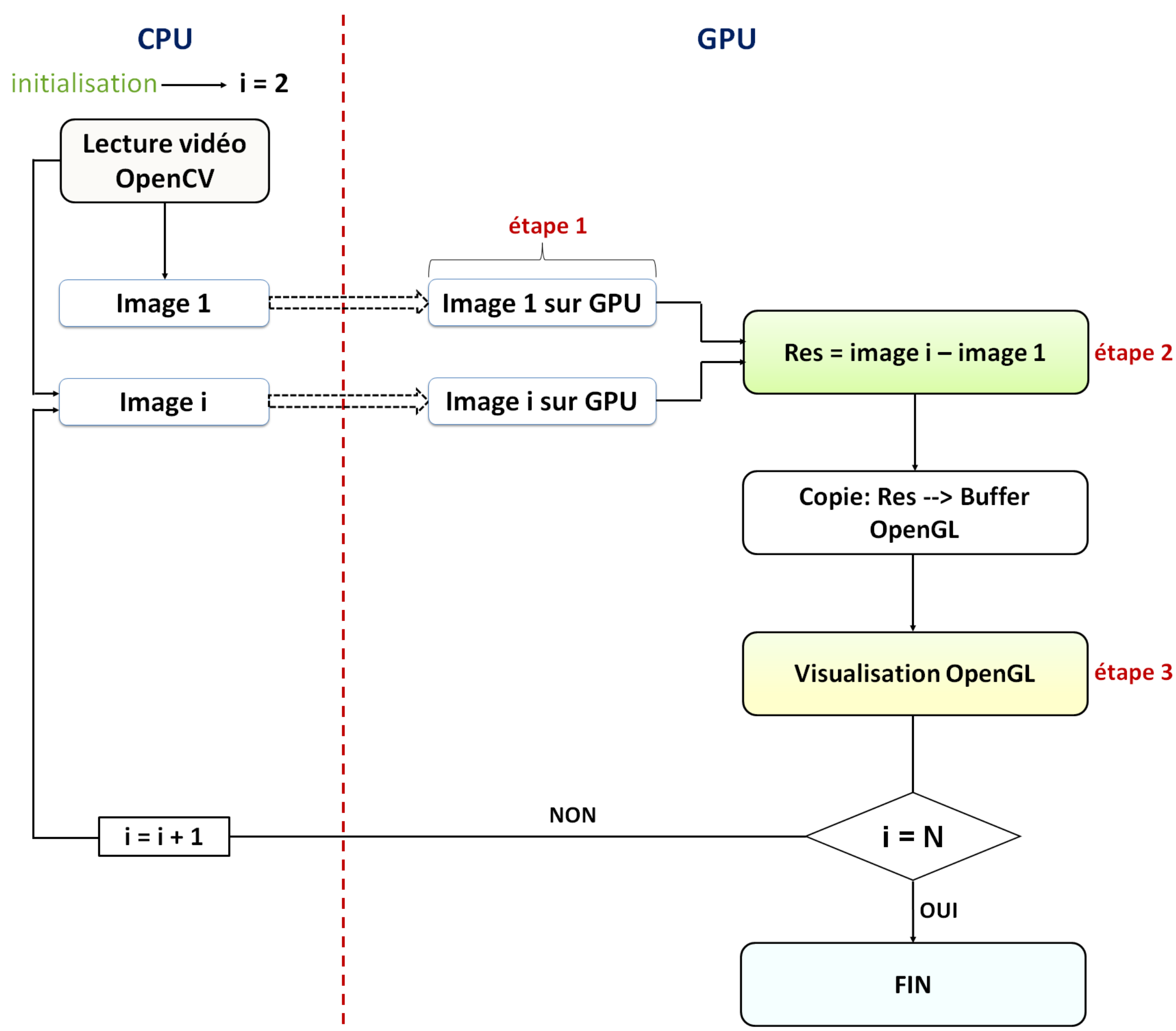
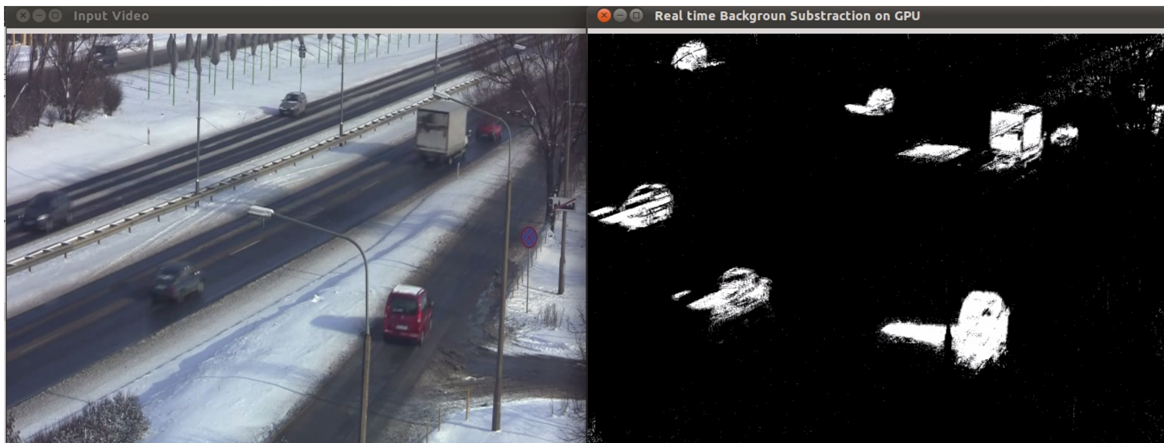
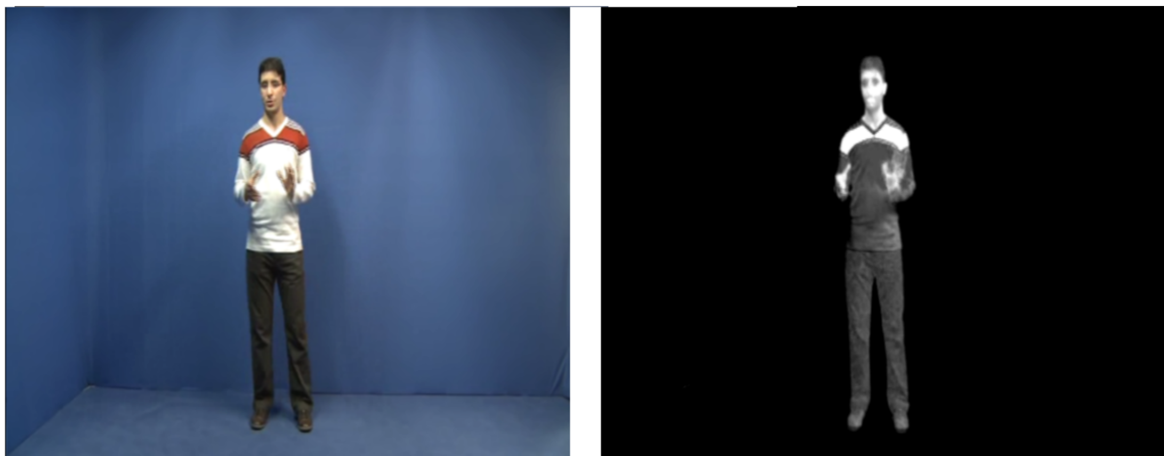


Figure IV.1 – Etapes de soustraction du background sur GPU d'une vidéo de N (1. . N) trames



(a) Soustraction de l'arrière plan dans une autoroute



(b) Soustraction du fond bleu (background)

Figure IV.2 – Soustraction de l'arrière plan sur processeur graphique

1.2 Extraction de silhouette sur GPU

La différence entre images représente une technique simple et efficace pour extraire les silhouettes ainsi que les objets en mouvement dans les séquences vidéo. Cette méthode s'appuie sur le calcul de la différence entre chaque paire de trames consécutives de la vidéo. L'implémentation GPU proposée de cette méthode peut être résumée en trois étapes (figure IV.3) : lecture et chargement de vidéo, différence entre trames, visualisation OpenGL.

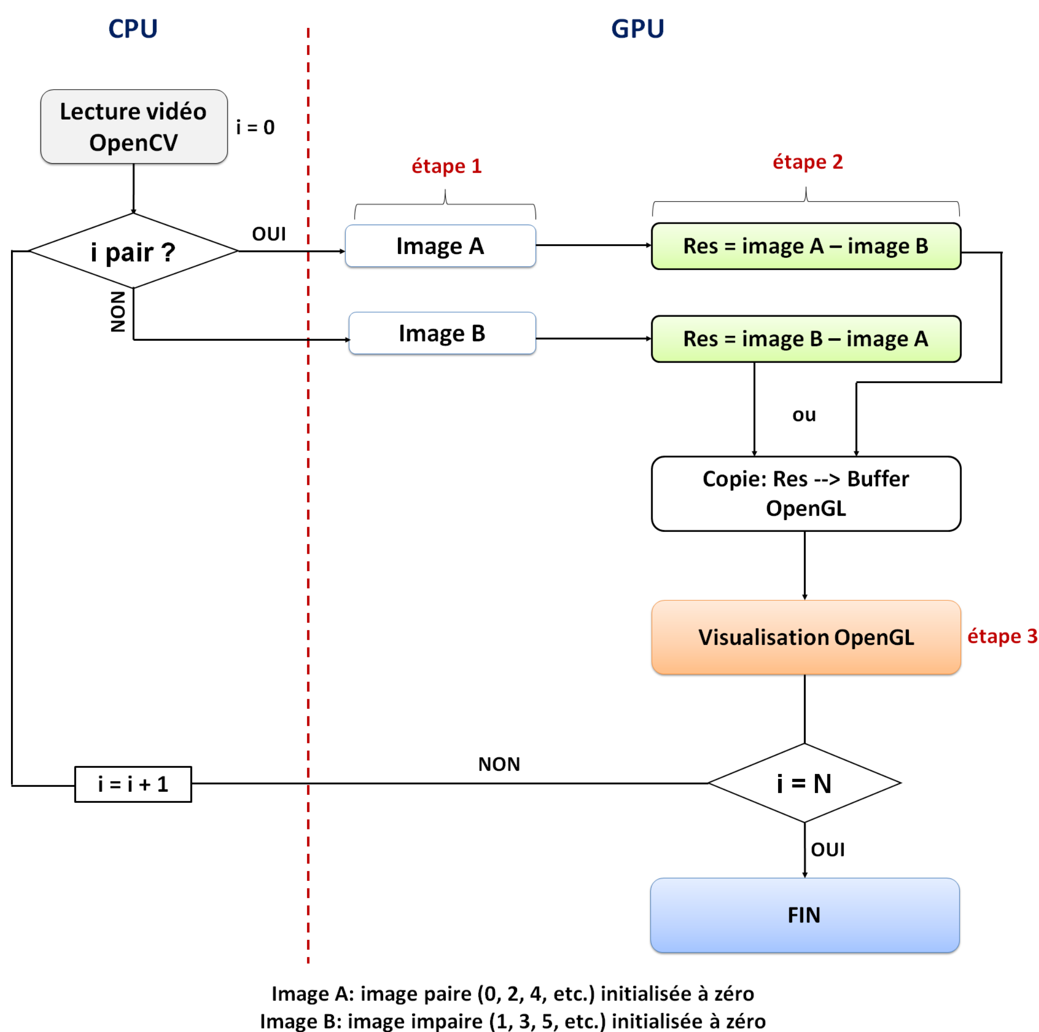


Figure IV.3 – Etapes d'extraction de silhouettes sur GPU

1. **Lecture et chargement de vidéo :** la première étape permet d'effectuer la lecture de la vidéo sur CPU à l'aide de la bibliothèque OpenCV [102]. Ensuite, nous copions les deux premières images de la vidéo pour les utiliser lors de la prochaine étape.

2. **Différence entre trames :** cette étape permet d'effectuer la soustraction entre les deux images chargées sur GPU lors de la première étape. Cette soustraction permet de détecter chaque mouvement apparaissant dans la vidéo et de le suivre par la suite.
3. **Visualisation OpenGL :** une fois la soustraction effectuée, nous pouvons visualiser l'image résultante à l'aide de la bibliothèque OpenGL. L'image résultante montre la silhouette de chaque nouvel objet apparaissant dans la vidéo. Après visualisation de chaque image résultante, nous revenons à la première étape pour recharger de nouvelles images sur GPU. Nous appliquons ensuite le même processus de différence de trames sur le reste des images composant la séquence vidéo.

La figure IV.4 montre le résultat d'extraction de silhouettes (différence entre trames) sur GPU. Elle montre ainsi deux silhouettes extraites représentant deux personnes en mouvement. Pour améliorer la qualité des résultats obtenus, nous avons appliqué un seuillage dans les deux cas (soustraction de l'arrière-plan et extraction de silhouette). Un seuil de 200 est utilisé pour présenter les résultats avec deux valeurs uniquement, noir ou blanc. Ainsi, si un pixel à une valeur supérieure au seuil (par exemple 220), il prendra la valeur de 255 (blanc), sinon, il prendra la valeur 0 (noir). La deuxième silhouette (en haut à droite de la figure IV.4) n'a pas été détectée entièrement suite à notre utilisation d'un seuil de valeur relativement faible (200) par rapport au niveau de gris d'un pixel blanc (255). Toutefois, ceci permet une meilleure élimination des bruits telle que montre la figure IV.4.

Figure IV.4 – Extraction de silhouettes sur GPU

1.3 Analyse des résultats

Les figures IV.5 et IV.6 présentent la comparaison des temps de calcul, en termes du nombre de fps, entre les implémentations séquentielles (CPU) et parallèles (GPU) des méthodes de soustraction d'arrière-plan (background) et d'extraction de silhouette respectivement. Notons une grande amélioration des performances obtenue grâce à l'exploitation des processeurs graphiques. Ces accélérations ont permis un traitement de vidéos de haute définition en temps réel.

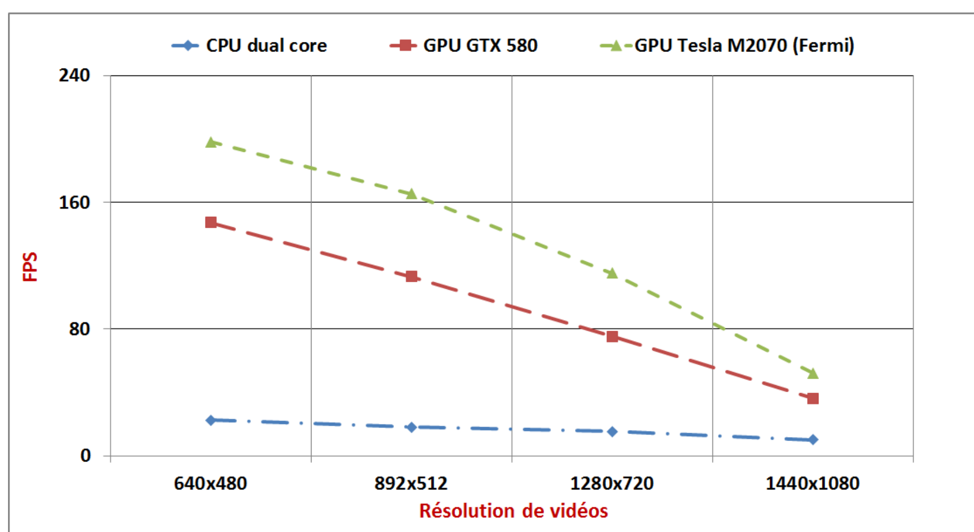


Figure IV.5 – Performances de soustraction d'arrière-plan (background) sur GPU

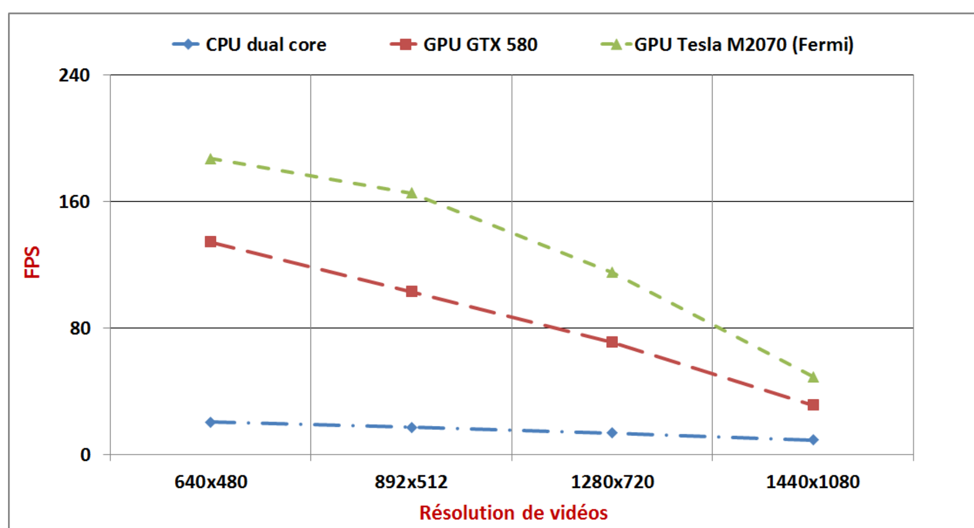


Figure IV.6 – Performances d'extraction de silhouettes sur GPU

2 suivi de mouvements en parallèle sur GPU

Les algorithmes de suivi de mouvements peuvent être présentés par l'estimation des déplacements (vitesse et direction) de caractéristiques d'une image (frame) par rapport à l'image précédente dans une séquence vidéo. Ces algorithmes présentent un outil nécessaire à de nombreuses applications telles que le codage et la compression de vidéos, la détection d'évènements [76] ainsi que l'indexation de vidéos [6]. Ces applications s'appuient sur différentes techniques telles que l'estimation du flot optique [43], les descripteurs SIFT [68], la mise en correspondance de blocs [14]. Nous proposons dans cette section une méthode GPU et Multi-GPU de suivi de mouvements à base des vecteurs de flot optique. Ces derniers permettent de suivre différents types d'objets (personnes, voitures, etc.) et de mouvements (petits ou grands) dans différents contextes (bruits, foule, etc.). Cette section est présentée en quatre parties : la première définit le principe des mesures du flot optique. La deuxième partie décrit notre algorithme proposé pour le suivi de mouvements à base du flot optique, tandis que la troisième partie est dévolue à la présentation de notre mise en œuvre parallèle de cet algorithme, exploitant à la fois un ou plusieurs processeurs graphiques. Enfin, une analyse des résultats obtenus est présentée dans la dernière partie.

2.1 Mesure du flot optique

Le flot optique représente le champ de vitesse apparent observé entre deux images successives dans une scène. Pour chaque pixel de coordonnées (x, y), le flot optique détermine sa position à l'instant suivant. Les méthodes de calcul du flot optique sont basées sur l'hypothèse d'éclairage constant [52]. Dans cette hypothèse, on suppose que les mesures d'image (par exemple la luminosité) dans une petite région restent constantes dans un court laps de temps, malgré que leur situation peut changer. Cette hypothèse peut être représentée par l'équation suivante :

$$I(x+u, y+v, t+1) = I(x, y, t). \tag{IV.1}$$

telque :

- I(x,y,t) : niveau de gris du pixel (x,y) à l'instant t ;
- I(x+u,y+v,t+1) : niveau de gris du pixel (x+u,y+v) à l'instant t+1 ;
- u,v : déplacements selon les axes horizontaux et verticaux.

À partir de cette hypothèse, on aboutit à l'équation de contraintes suivante :

$$\frac{dI}{dt}(x, y, t) = 0 \tag{IV.2}$$

$$\Longleftrightarrow \frac{\delta I}{\delta x}\frac{\delta x}{\delta t} + \frac{\delta I}{\delta y}\frac{\delta y}{\delta t} + \frac{\delta I}{\delta t} = 0 \tag{IV.3}$$

$$\Longleftrightarrow Ix.u + Iy.v + It = 0 \tag{IV.4}$$

L'équation (4) présente la contrainte à respecter pour le mouvement. Toutefois, nous ne disposons que d'une unique équation pour déterminer deux inconnues u et v (les deux composantes du vecteur de mouvement au point considéré). Ce phénomène est connu sous le nom de « aperture problem » [88]. Afin de déterminer ces deux inconnues, toutes les méthodes de calcul du flot optique effectuent une ou plusieurs hypothèses supplémentaires par rapport à la nature du champ de mouvement, permettant ainsi d'obtenir des contraintes additionnelles. Dans ce contexte, deux grandes familles de méthodes se distinguent :

1. Les méthodes locales initiées par Lucas et Kanade [70] supposant que le mouvement d'un pixel est constant dans son voisinage. Ceci permet d'assurer une efficacité de suivi de petits mouvements avec une meilleure robustesse aux bruits.
2. les méthodes globales, ou variationnelles, introduites par Horn et Schunck [52], imposant une régularité spatiale au flot optique recherché. Cette méthode suppose que les pixels voisins doivent avoir une vitesse similaire, c.à.d. le flot optique présente une variation progressive. Cependant, cette méthode est entravée par sa faible efficacité en cas de petits mouvements.

La méthode Lucas-Kanade présente la technique la plus utilisée pour le calcul du flot optique grâce à son utilisation d'une approche locale offrant une meilleure précision des résultats. Cette méthode offre également une meilleure robustesse aux bruits et un suivi adapté à différents types de mouvements (lents, rapides, etc.). Par conséquent, nous proposons d'exploiter cette approche pour l'algorithme de suivi de mouvements proposé dans ce travail.

2.2 Algorithme de suivi de mouvements

Avant de présenter nos implémentations GPU et Multi-GPU de la méthode de suivi de mouvements à base du flot optique, nous décrivons l'algorithme séquentiel proposé. Ce dernier s'appuie sur trois étapes principales : détection des coins, suivi des coins détectés, élimination des régions statiques.

2.2.1 Détection des coins

La première étape consiste à détecter les points d'intérêt (coins) afin de pouvoir les suivre dans les images suivantes dans la vidéo. Pour ce faire, nous avons exploité la technique de Bouguet [17] utilisant le principe de Harris [49], offrant des résultats efficaces grâce à son invariance à la translation, à la rotation, au changement d'échelle, etc. La détection des coins permet de réduire l'espace d'entrée en gardant les points les plus importants pour simplifier le suivi par la suite. Le processus de détection des points d'intérêt est décrit en détail dans la section 3.2 du deuxième chapitre.

2.2.2 Suivi des coins détectés

Une fois les coins détectés, nous pouvons les suivre dans les images suivantes *via* les mesures du flot optique. Nous exploitons la technique de Lucas-Kanade connue par son efficacité et sa robustesse aux bruits [70]. Elle peut être présentée en sept étapes : construction des pyramides, calcul des nouvelles coordonnées de pixels, calcul du gradient spatial local, calcul des dérivées temporelles, mesure du flot optique, correction d'estimation et propagation du résultat.

1. **Etape 1 : construction des pyramides**

   La première étape consiste à créer les pyramides d'images I et J tel que I représente l'image initiale et J l'image à l'instant suivant. Ensuite, une conversion vers le niveau de gris est appliquée à ces deux images. Ces dernières représentent le niveau 0 des pyramides, les autres niveaux sont obtenus par un sous-échantillonnage des images en appliquant un filtre gaussien. Chaque niveau de la pyramide est représenté par les mêmes images qu'au niveau inférieur mais avec une résolution inférieure (moitié). Les définitions des images aux différents niveaux sont calculées comme suit :

   $$Largeur^L = \frac{Largeur^{L-1} + 1}{2} \quad \text{(IV.5)}$$

$$Hauteur^L = \frac{Hauteur^{L-1} + 1}{2} \quad \text{(IV.6)}$$

Une fois les niveaux de pyramides construits, on lance une boucle itérant autant de fois qu'il y a de niveaux. Cette boucle commence par la plus petite image (niveau plus haut), et se termine par l'image originale (niveau 0). Le but de cette boucle est de propager le vecteur de déplacement entre les différents niveaux de pyramides. Ce vecteur, initialisé à zéro, sera calculé lors des étapes suivantes. Le nombre de niveaux de pyramides est défini en fonction de la résolution des images et des vitesses maximales.

2. **Etape 2 : calcul des nouvelles coordonnées de pixels**

   À chaque passage de niveau (L), les coordonnées de coins doivent être recalculées. Ce calcul peut se faire via l'équation (IV.7).

$$x^L = \frac{x}{2^L} \qquad y^L = \frac{y}{2^L} \quad \text{(IV.7)}$$

   Par exemple, le pixel ayant les coordonnées (204, 100) au niveau 0 aura les coordonnées (102, 50) au niveau 1 et (51, 25) au niveau 2.

3. **Etape 3 : calcul du gradient spatial local**

   Cette étape est consacrée au calcul de la matrice du gradient spatial G pour chaque pixel (point d'intérêt) de l'image I, en utilisant l'équation (IV.10). Cette matrice de quatre éléments (2×2) est calculée à partir des dérivées spatiales I_x, I_y calculées suivant les équations (IV.8) et (IV.9).

$$I_x(x, y) = \frac{I^L(x + 1, y) - I^L(x - 1, y)}{2} \quad \text{(IV.8)}$$

$$I_y(x, y) = \frac{I(x, y + 1) - I(x, y - 1)}{2} \quad \text{(IV.9)}$$

   La matrice du gradient spatial est calculée dans une fenêtre centrée sur le point à analyser. Sa taille dépend du type et de la taille d'image. En général, on choisit une taille de 9×9 [17]. Notons que la taille de la fenêtre est de : (2w +1)×(2w+1).

$$G = \sum_{x_i=x^L-w}^{x^L+w} \sum_{y_i=y^L-w}^{y^L+w} \begin{bmatrix} I_x^2(x_i, y_i) & I_x(x_i, y_i)I_y(x_i, y_i) \\ I_x(x_i, y_i)I_y(x_i, y_i) & I_y^2(x_i, y_i) \end{bmatrix} \quad \text{(IV.10)}$$

4. **Etape 4 : lancement de la boucle itérative et calcul des dérivées temporelles**

   Cette étape consiste à lancer une boucle itérant tant que la mesure du flot optique (qui sera calculée lors de l'étape prochaine) entre deux itérations n'est pas descendue sous un certain seuil de précision. Le nombre d'itérations maximal est défini par l'utilisateur. Ensuite, un calcul des dérivées temporelles est effectué à partir de l'image J (image à l'instant suivant). Ce calcul est réalisé à base de l'équation suivante :

$$I_t(x,y) = I^L(x,y) - J^L(x+g_x+v_x, y+g_y+v_y) \tag{IV.11}$$

   Cette dérivée effectue la soustraction entre chaque point (coin) de l'image I par son point correspondant estimé à l'image J. Les valeurs g_x et g_y, initialisées à zéro, représentent les estimations du déplacement qui seront propagées d'un niveau de pyramide à l'autre (étape 7). Les valeurs v_x , v_y représentent les corrections d'estimation du déplacement, qui sont calculées et propagées à l'intérieur de la boucle itérative à l'étape 6.

5. **Etape 5 : mesure du flot optique**

   Le flot optique est calculé à partir de la matrice du gradient G (calculé en étape 3) et le vecteur de décalage $\bar{b}$. Ce dernier est représenté par la somme des dérivées du déplacement estimé tel que décrit à l'équation 4.

$$\bar{b} = \sum_{x_i=x^L-w}^{x^L+w} \sum_{y_i=y^L-w}^{y^L+w} \begin{bmatrix} I_t(x_i,y_i)I_x(x_i,y_i) \\ I_t(x_i,y_i)I_x(x_i,y_i) \end{bmatrix} \tag{IV.12}$$

   La mesure du flot optique $\bar{n}$ est obtenue en multipliant l'inverse de la matrice du gradient spatial par le vecteur de décalage $\bar{b}$ (équation (IV.13)).

$$\bar{n} = G^{-1}\bar{b} \tag{IV.13}$$

6. **Etape 6 : correction d'estimation et fin de la boucle itérative**

   Durant cette étape, une correction d'estimation (assignation (IV.14)) est appliquée avant de la propager à l'itération suivante de la boucle. Notons que v_x et v_y sont initialisés à zéro.

$$v_x = v_x + n_x \qquad v_y = v_y + n_y \tag{IV.14}$$

Deux cas de figures permettent de sortir de la boucle lancée à l'étape 4. Le premier cas est l'arrivé à la dernière itération de la boucle. Le deuxième cas se présente lorsque la correction mesurée devient inférieure à un certain seuil de précision.

7. **Etape 7 : propagation du résultat et fin de boucle pyramidale**

   Cette étape consiste à propager les résultats vers le niveau inférieur en utilisant les équations suivantes (IV.15) :

$$g_x = 2(g_x + v_x) \qquad\qquad g_y = 2(g_y + v_y) \tag{IV.15}$$

   Lorsqu'on atteint le niveau de pyramide le plus bas (image originale), la boucle de niveaux (lancée à l'étape 1) peut être arrêtée. Le vecteur $\bar{g}$ représente le flot optique final du point analysé.

Le résultat du suivi de mouvements à partir du flot optique peut être présenté par un ensemble de n vecteurs, comme montré dans l'équation (IV.16) :

$$\Omega = \{\omega_1 \ ... \ \omega_n \mid \omega_i = (x_i, y_i, v_i, \alpha_i)\} \tag{IV.16}$$

avec :

- x_i : coordonnée x du coin i ;
- y_i : coordonnée y du coin i ;
- v_i : vitesse du coin i ;
- α_i : direction (angle) du mouvement au coin i.

2.2.3 Elimination des régions statiques

La dernière étape permet d'éliminer les coins n'ayant pas fait de mouvement ainsi que les bruits. Les vecteurs de flot optique présentant une vitesse nulle sont considérés comme objets statiques et sont éliminés. Les vecteurs qui possèdent des valeurs de vitesse et de direction très différentes par rapport à leurs voisins sont considérés comme bruits et seront également éliminés.

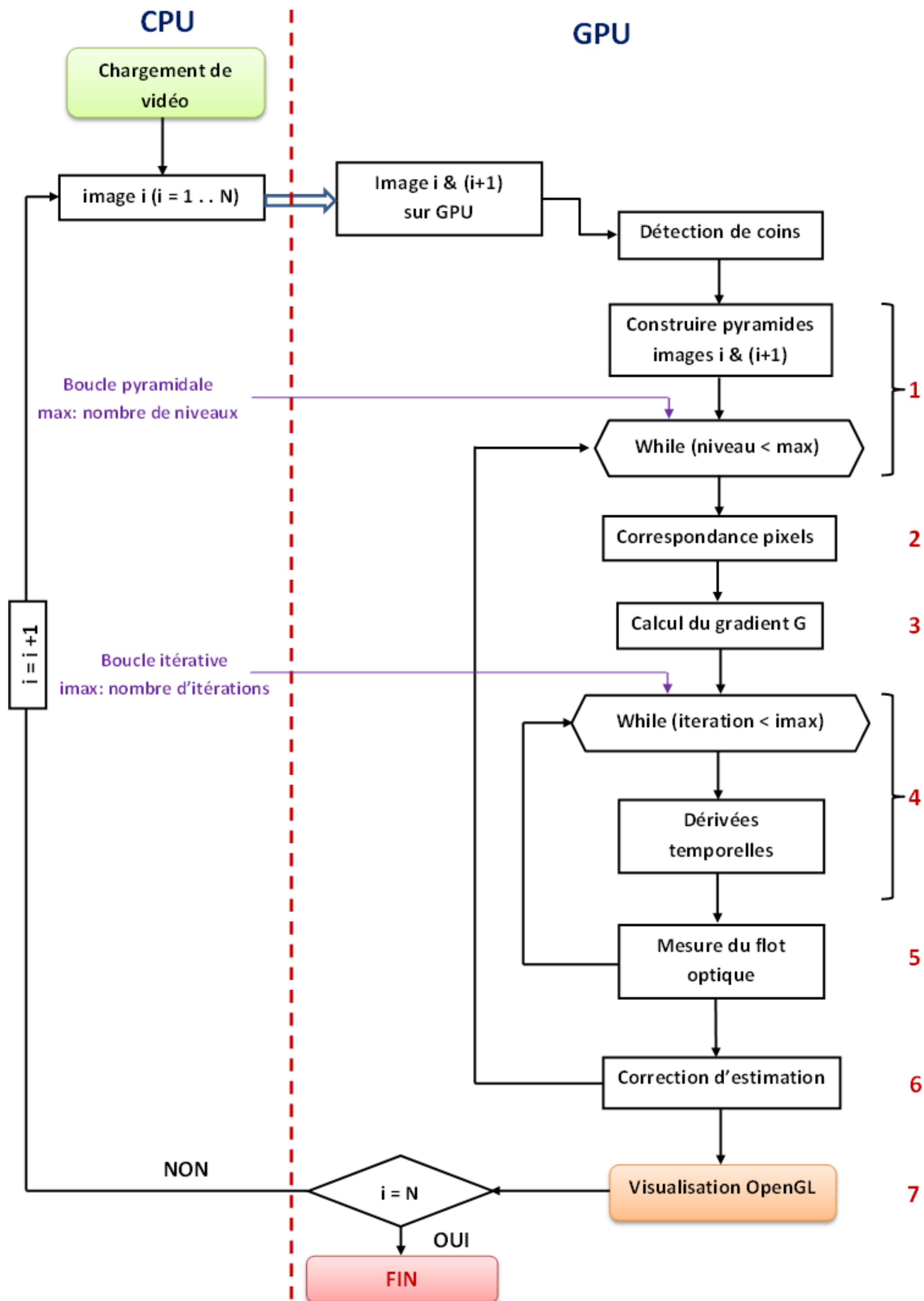


Figure IV.7 – Étapes d'implémentation pyramidale de la mesure du flot optique sur GPU

2.3 Suivi de mouvements sur GPU uniques ou multiples

Nous présentons dans ce paragraphe la mise en œuvre parallèle de l'algorithme de suivi de mouvements proposé ci-avant. Cette implémentation est décrite en deux parties : la première présente la mise en œuvre exploitant un seul processeur graphique, tandis que la seconde décrit notre approche de suivi de mouvements exploitant plusieurs GPU.

2.3.1 suivi de mouvements sur un GPU

L'implémentation GPU de cette méthode est basée sur le même principe que celui utilisé précédemment pour la mise en œuvre de la soustraction de l'arrière-plan et de l'extraction de silhouettes. Les images de la vidéo sont chargées dans la mémoire du processeur graphique afin de les traiter en parallèle sous CUDA. Une fois les traitements finis, les résultats sont affichés à l'aide de la bibliothèque graphique OpenGL. L'implémentation GPU de notre approche de suivi de mouvements est basée sur la parallélisation de ses étapes décrites dans la section 2.2 : détection des coins, suivi des coins et élimination des régions statiques.

1. **Détection des coins sur GPU**

   La détection des coins sur GPU est appliquée sur toutes les images de la vidéo. Cette détection est effectuée à partir de l'implémentation GPU du détecteur de Harris décrite dans la section 3.2 du deuxième chapitre.

2. **Suivi des coins sur GPU**

   Le suivi des coins est effectué *via* les vecteurs du flot optique tel que décrit dans la section précédente. Les sept étapes décrites ci-dessus (section 2.2) sont appliquées en parallèles *via* CUDA de telle sorte que chaque thread GPU applique une instruction (parmi les étapes) sur un coin (point d'intérêt), détecté préalablement, d'une image de la vidéo. Le nombre de threads GPU utilisés est égal au nombre de points d'intérêt extraits lors de la première étape. La figure IV.7 résume l'implémentation GPU des différentes étapes de mesure du flot optique appliqué à un ensemble de points d'intérêt. Pour l'optimisation des calculs sur GPU, les étapes de calcul du gradient, de la dérivée temporelle ainsi que la mesure du flot optique exploitent la mémoire partagée du GPU afin d'avoir un accès rapide aux valeurs de pixels voisins. De même, les images traitées sont chargées dans la mémoire de texture pour assurer un accès plus rapide aux pixels d'image.

3. **Elimination des régions statiques sur GPU**

   L'élimination des régions statiques s'effectue tous simplement en n'affichant que les vecteurs qui présentent une vitesse supérieure à zéro. La visualisation de la vidéo de sortie est effectuée par OpenGL.

La particularité de notre implémentation est son efficacité due à deux facteurs majeurs. Le premier est sa robustesse aux bruits grâce au principe pyramidal de la méthode Lucas-Kanade. Le deuxième facteur est la précision des résultats obtenus du fait que les vecteurs du flot optique sont appliqués uniquement aux points d'intérêt extraits préalablement (au lieu de les appliquer sur l'intégralité des pixels d'images). Ceci permet également de réduire significativement les temps de calcul. La figure IV.8 présente les vecteurs du flot optique extraits à partir des points d'intérêt d'une trame de vidéo Full HD. Les vecteurs sont présentés par des flèches montrant les directions et vitesses du flot optique. En effet, les vitesses sont extraites à partir de la grandeur des vecteurs du flot optique détectés (flèches). Notons que les flèches montrées sur les objets statiques (arbres et bâtiment) sont dus au mouvement de la caméra.

Figure IV.8 – Vecteurs de flot optique extraits à partir des points d'intérêt

2.3.2 Suivi de mouvements sur GPU multiples

Les algorithmes de traitement de vidéo peuvent appliquer un traitement hautement parallèle entre les pixels des images de la vidéo. Ceci est particulièrement vrai dans les méthodes de détection des coins et de suivi de mouvements présentés ci-dessus. L'implémentation GPU a permis dans un premier temps d'accélérer ces techniques en exploitant les unités de calcul en parallèle. Par ailleurs, nous pouvons atteindre des performances plus importantes par l'exploitation efficace des systèmes composés de plusieurs processeurs graphiques. Ce type de systèmes Multi-GPU est de plus en plus rencontré à l'heure actuelle. Notre programme Multi-GPU de suivi de mouvements commence par détecter le nombre de processeurs graphiques disponibles avant de les initialiser. Pour la phase de détection des coins, l'image courante de la vidéo est chargée sur chaque GPU. Bien que cela semble redondant, les données de l'image entière sont nécessaires dans une phase ultérieure. Toutefois, chaque image est divisée, selon l'axe horizontal y, en sous-images de taille identique. Le nombre de sous-images est égale au nombre de GPU. Ensuite, chaque processeur graphique est chargé de détecter les coins dans la sous-image correspondante.

Lors de l'étape du calcul du flot optique *via* la technique Lucas-Kanade, chaque GPU extrait les niveaux de pyramides à base des formules présentées dans la section 2.2.2.1. Ensuite, chaque processeur graphique applique un suivi des coins détectés auparavant via la technique Lucas-Kanade décrite dans la section précédente. Après fin de calcul pour chaque image, nous copions les résultats de suivi (coordonnées de déplacement des coins) vers le GPU qui dispose d'une sortie vidéo. Ce dernier permettra de visualiser le résultat *via* la bibliothèque graphique OpenGL.

2.4 Analyse des résultats

Nous présentons dans cette section une analyse des résultats de l'approche de suivi de mouvements proposée sur processeurs graphiques. Ces résultats sont présentés en trois parties : la première montre une comparaison des temps de calcul de différentes implémentations CPU et GPU de la technique Lucas-Kanade de calcul du flot optique. Nous présentons ensuite les accélérations obtenus lors de l'exploitation des processeurs graphiques multiples pour la détection des points d'intérêt ainsi que le calcul du flot optique. La troisième partie est dévolue à la présentation des performances de calcul en nombre d'images traitées par seconde de notre approche de suivi de mouvements incluant toutes ses étapes.

2.4.1 Comparaison des temps de calcul du flot optique sur GPU

Afin d'évaluer notre implémentation sur GPU, nous proposons de comparer ses performances avec les nouvelles mises en œuvre de calcul du flot optique dans la littérature. Dans ce contexte, on trouve la bibliothèque OpenCV qui propose dans sa version 2.4, sortie en 2012 [102], des versions CPU et GPU de la technique Lucas-Kanade. La méthode Lucas-Kanade de calcul du flot optique s'appuie sur trois paramètres principaux *i.e.* le nombre de niveaux de pyramides, le nombre d'itérations dans chaque niveau et enfin la taille de la fenêtre utilisée pour le calcul du gradient. Ces paramètres impactent également la charge de calcul requise par la méthode. Pour avoir une comparaison pertinente, nous mesurons les temps de calcul de la méthode en changeant à chaque fois les valeurs d'un paramètre. Les valeurs standards de ces paramètres sont :

- nombre de niveaux de pyramides : 4 ;
- nombre d'itérations : 3 ;
- taille de fenêtre : 5×5.

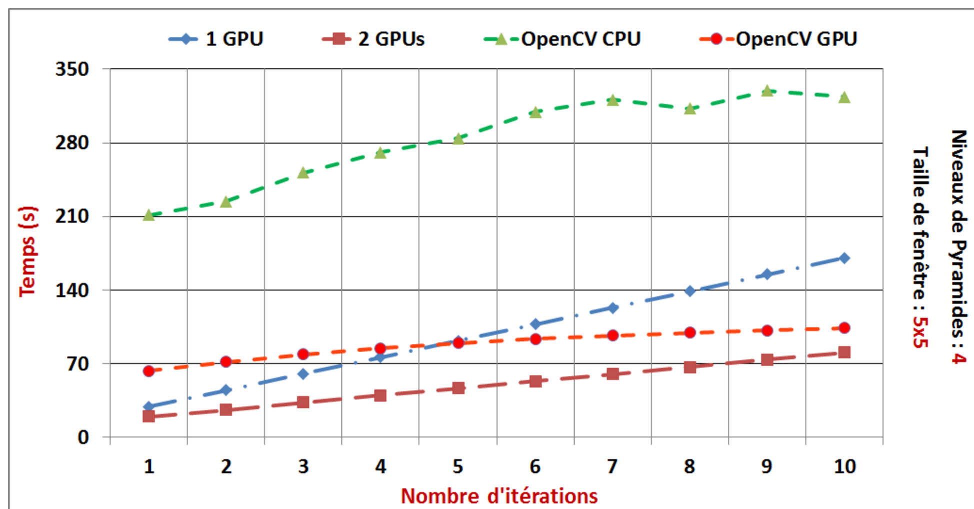


Figure IV.9 – Évolution du temps de calcul du flot optique par rapport au nombre d'itérations

La séquence vidéo utilisée pour les tests dispose de 4850 images avec une résolution, Full HD, de 1920 × 1080 pixels. Les résultats présentent le temps en secondes nécessaire à chaque programme pour exécuter l'algorithme de calcul du flot optique uniquement. Ce temps ne prend pas en compte les étapes de décodage de la vidéo, sa conversion en niveaux de gris ainsi que sa visualisation. Afin d'assurer la même quantité de travail pour chaque mise en œuvre, notre implémentation du détecteur des points d'intérêt est utilisé dans chaque cas.

Les expérimentations ont été effectuées sous Ubuntu 11.04 (64-bit) avec CUDA 4.0, utilisant les plates-formes suivantes : CPU Dual core, GPU GTX 580 :
- CPU : Intel Core 2 Quad Q8200, 2.33GHz, Mem : 8 GB ;
- GPU : 2×NVIDIA GeForce GTX 580, Mem : 1.5 GB.

La figure IV.9 présente une comparaison entre les différentes implémentations de calcul du flot optique pour une vidéo Full HD. Ces résultats montrent l'évolution du temps de calcul de chaque approche en fonction du nombre d'itérations dans chaque niveau pyramidal. Notons que l'implémentation OpenCV sur CPU est la plus lente, tandis que notre approche exploitant deux processeurs graphiques est la plus rapide. Toutefois, les temps de calcul sur CPU (OpenCV) augmentent de façon moins importante que notre implémentation. En effet, la version OpenCV peut arrêter le calcul si la nouvelle itération n'améliore pas la qualité du résultat. Cette fonctionnalité n'est pas prise en compte dans notre approche. Néanmoins, pour la valeur la plus courante de ce paramètre *i.e.* 3, notre mise en œuvre exploitant 1 ou 2 GPU est plus rapide que ceux d'OpenCV (versions CPU et GPU).

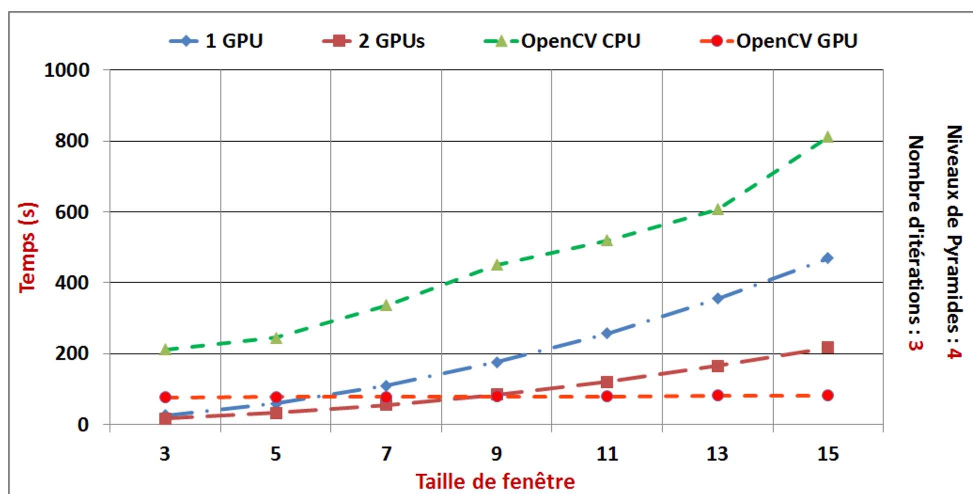


Figure IV.10 – Évolution du temps de calcul du flot optique par rapport à la taille de la fenêtre

La figure IV.10 montre l'influence de la taille de la fenêtre utilisée pour le calcul du gradient sur les temps d'exécutions des différentes implémentations. Notons que pour des petites fenêtres, notre approche GPU est plus rapide. Toutefois, la version GPU d'OpenCV est plus rapide lors de l'utilisation de grandes fenêtres puisque ses temps gardent la même valeur en changeant la taille de la fenêtre. Ceci et du à l'utilisation de plus de ressources sur GPU et aussi à une meilleure gestion de la mémoire cache.

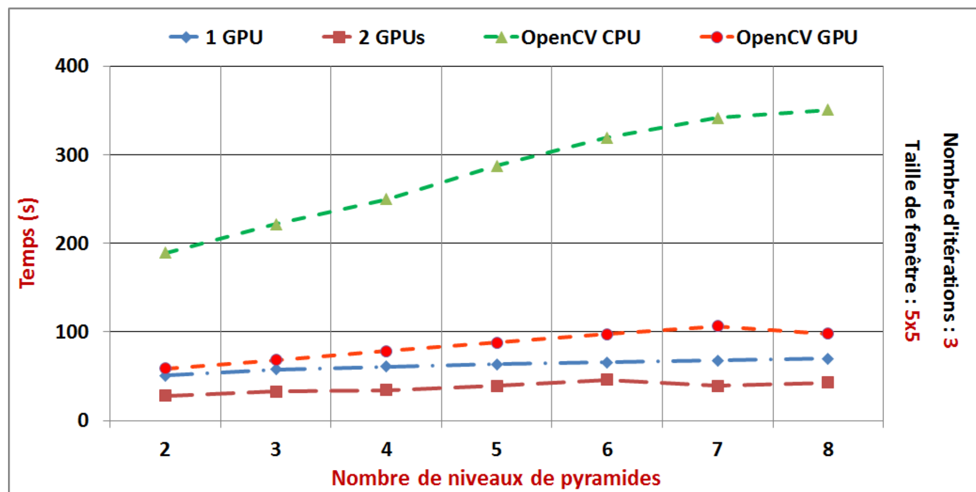


Figure IV.11 – Évolution du temps de calcul du flot optique par rapport au nombre de niveaux de pyramides

La figure IV.11 montre, à son tour, l'influence du nombre de niveaux pyramidales sur les temps de calcul des différentes implémentations. Notons que les deux implémentations OpenCV(CPU et GPU) sont moins rapides. Notre approche présente de meilleures performances quel que soit le nombre de niveaux de pyramides.

En résumé, nous pouvons conclure qu'il n y a pas de mise en œuvre qui surpasse nettement les autres. On note que la version GPU d'OpenCV est plus rapide lors de l'utilisation de grandes fenêtres, tandis que notre approche est plus rapide lors de l'augmentation des niveaux de pyramides. Ceci est particulièrement intéressant dans le cas du traitement de vidéos de haute définition. Toutefois, nous pouvons constater que les processeurs graphiques sont bien adaptés à ce type d'algorithmes puisque toutes les versions GPU sont plus rapides que leurs équivalents sur CPU. Par ailleurs, les performances des implémentations présentées ci-dessus dépendent fortement des valeurs de paramètres utilisés, et par conséquent des différents cas d'utilisation.

2.4.2 Résultats d'exploitation des processeurs graphiques multiples

Notre approche de suivi de mouvements permet également d'exploiter plusieurs processeurs graphiques simultanément. Cette section présente les résultats de cette exploitation à partir de deux tests principaux. Dans le premier, nous mesurons les accélérations obtenues pour la détection des points d'intérêt sur GPU multiples, tandis que les accélérations de la méthode Lucas-Kanade sont présentées dans le second test. Ces deux tests ont été effectués sur une vidéo Full HD (1920×1080) de 4850 frames. Les accélérations présentées sont me-

surées par rapport à une implémentation exploitant un seul GPU. Notons que ces résultats ne sont pas comparés avec les versions CPU et GPU d'OpenCV puisque ces dernières ne supportent pas de traitement Multi-GPU.

Les expérimentations ont été effectuées sous Ubuntu 11.04 (64-bit) avec CUDA 4.0, utilisant les plates-formes suivantes : CPU Xeon E5405 Dual core, GPU Tesla S1070 :

- CPU : 2×Intel Xeon E5405, 2.00GHz, Mem : 16 GB ;
- GPU : NVIDIA Tesla S1070 avec 4 GPU, Mem : 4×4 = 16 GB.

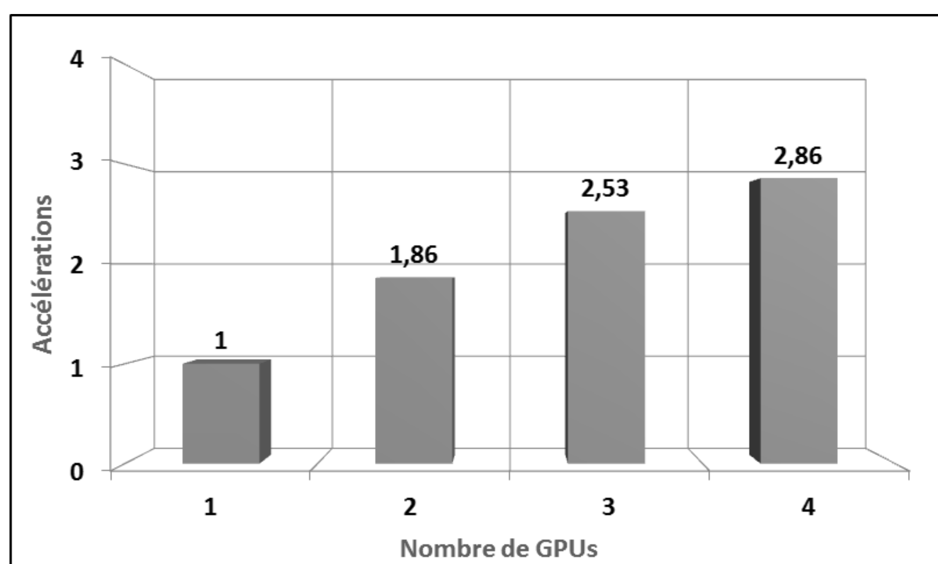


Figure IV.12 – Performances de détection des points d'intérêt sur processeurs graphiques multiples

La figure IV.12 montre que les accélérations obtenues *via* l'exploitation de GPU multiples pour la détection des coins sont presque linéaires jusqu'à trois GPU. Cependant, l'utilisation d'un GPU supplémentaire n'améliore que légèrement les performances. Ceci peut être interprété par la charge de travail du GPU qui devient trop faible pour exploiter au mieux les processeurs graphiques. La figure IV.13 montre les accélérations obtenues par l'implémentation Multi-GPU de la méthode Lucas-Kanade, en fonction du nombre de pixels détectés et suivis. Notons que ces accélérations augmentent en fonction du nombre de coins suivis puisqu'un grand nombre de ces derniers requiert plus de calcul. Par conséquent, on peut exploiter plus d'unités de calcul de chaque GPU de façon parallèle. L'accélération maximale est atteinte pour le plus grand nombre de pixels. Elle a une valeur de 3.28, 2.65, et 1.8 lors de l'utilisation de 4, 3 et 2 GPU respectivement. De ce fait, nous pouvons conclure que l'exploitation des processeurs graphiques multiples est bien adaptée pour l'accélération des algorithmes de suivi de mouvements à base de vecteurs du flot optique.

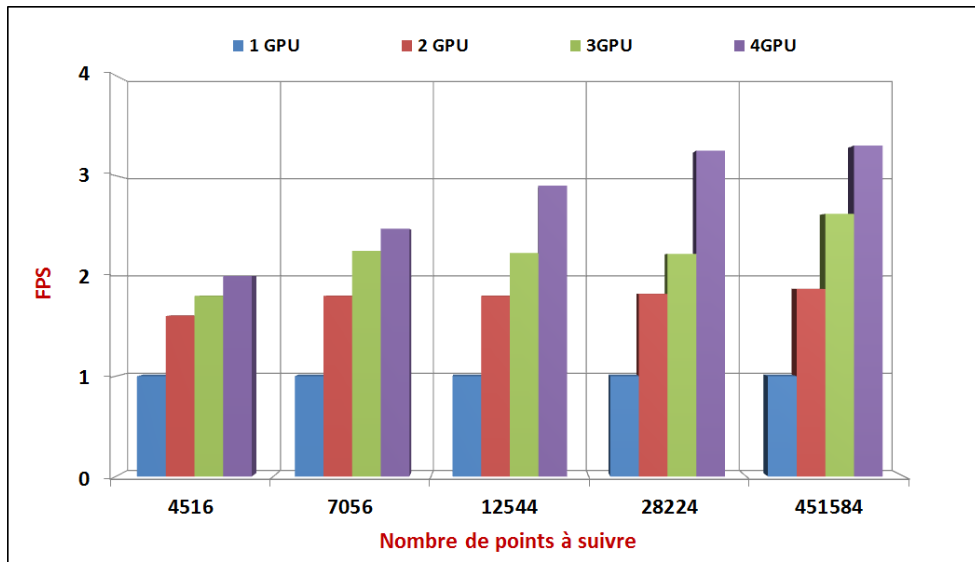


Figure IV.13 – Performances de calcul du flot optique (Lucas-Kanade) sur processeurs graphiques multiples

2.4.3 Performances en nombre d'images par secondes (fps)

Afin de mieux évaluer les implémentations GPU et Multi-GPU de la méthode de suivi de mouvements proposée, nous mesurons les performances en nombre d'images traitées par secondes. Cette mesure inclut toutes les étapes *i.e.* décodage de la vidéo, conversion en niveau de gris, détection des coins et finalement le calcul du flot optique. Les résultats expérimentaux sont effectués sur deux séquences vidéo : une sous format Full HD (4850 trames de résolution 1920 × 1080) et une autre sous format 4K (1924 trames de résolution 3840 × 2160). Notons que l'implémentation GPU de la méthode Lucas-Kanade s'appuie sur notre propre mise en œuvre de détection des coins, tandis que les implémentations CPU et GPU d'OpenCV s'appuient sur la même technique de détection des points d'intérêt d'OpenCV. Les résultats présentent une comparaison entre les trois solutions. Pour avoir une comparaison équitable, nous utilisons les mêmes valeurs de paramètres :

- nombre de niveaux de pyramides : 4 ;
- nombre d'itérations : 3 ;
- taille de fenêtre : 7×7.

Les expérimentations ont été également effectuées sous Ubuntu 11.04 (64-bit) avec CUDA 4.0, utilisant les plates-formes suivantes : CPU Dual core Quad Q8200, GPU GTX 580 :

- CPU : Intel Core 2 Quad Q8200, 2.33GHz, Mem : 8 GB ;
- GPU : NVIDIA GeForce GTX 580, 1.5 GB.

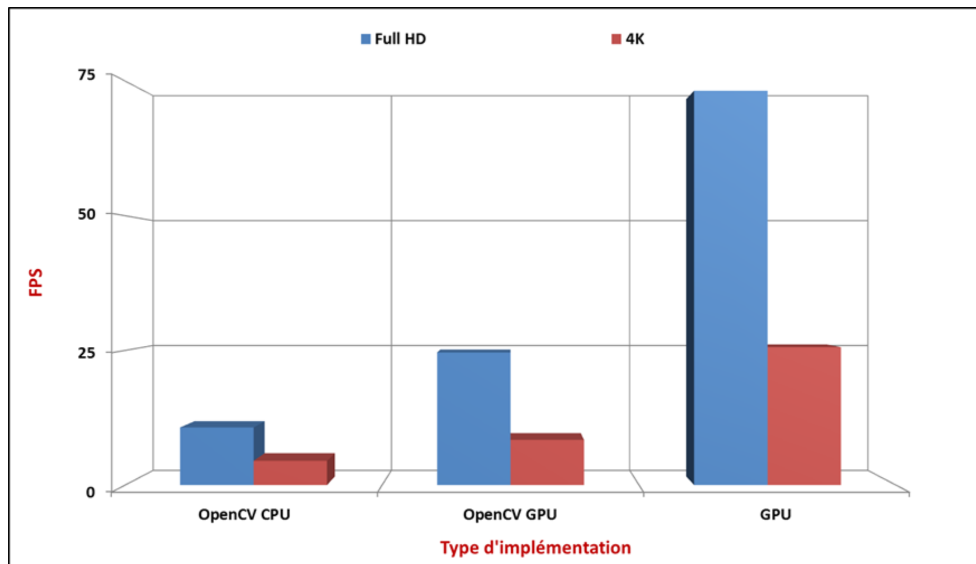


Figure IV.14 – Performances (en fps) des implémentations GPU sur vidéos Full HD et 4K

La figure IV.14 présente les performances des implémentations citées ci-dessus en termes de nombre d'images traitées par seconde. La méthode CPU d'OpenCV présente la solution la plus lente, tandis que la version GPU d'OpenCV permet de l'accélérer légèrement avec un facteur allant de 1.9 à 2.3. Toutefois, notre mise en œuvre GPU est plus rapide, et présente la seule solution permettant un traitement en temps réel de vidéos Full HD ou 4K. Pour une évaluation plus détaillée des résultats, nous présentons les figures IV.15 et IV.16 montrant la répartition des temps en pourcentage de chaque étape des différentes implémentations pour les vidéos Full HD et 4K. On peut distinguer trois étapes principales :

1. Détection des coins : technique de Harris ;
2. Calcul du flot optique : technique Lucas-Kanade ;
3. Etapes restantes : décodage et lecture vidéo, copie des données entre mémoire CPU et GPU, visualisation des résultats.

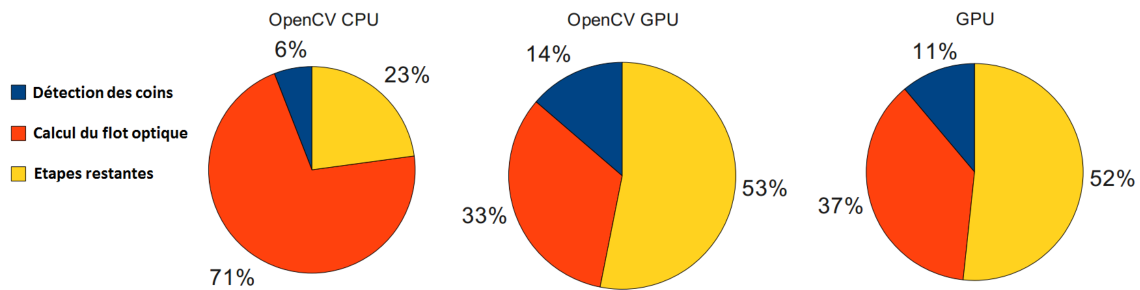


Figure IV.15 – Répatition des temps pris par chaque étape des implémentations proposées sur une vidéo Full HD

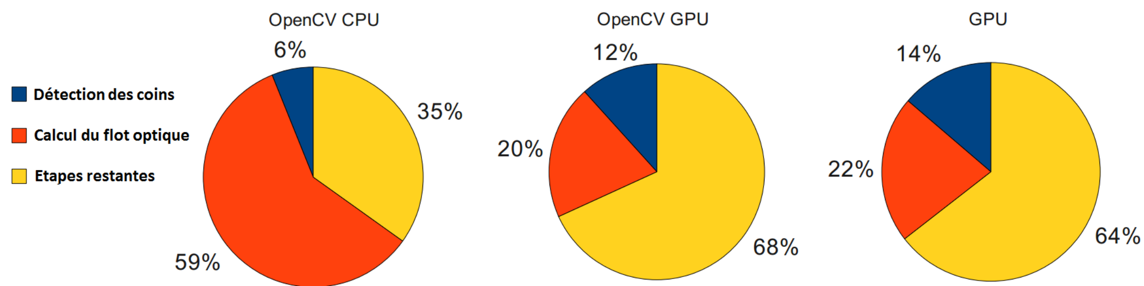


Figure IV.16 – Répatition des temps pris par chaque étape des implémentations proposées sur une vidéo Full 4K

Les deux implémentations GPU présentent une distribution identique des temps pris par chaque étape. En effet, les deux approches sont entravées par des temps de transfert importants entre mémoires CPU et GPU. Toutefois, ces étapes sont plus lentes dans la version GPU d'OpenCV par rapport à notre mise en œuvre.

Notons que la qualité des résultats est identique entre les implémentations CPU et GPU d'OpenCV puisqu'ils se basent sur le même principe. Toutefois, notre implémentation GPU offre une meilleure élimination des bruits grâce la suppression des vecteurs de flot optique ayant une vitesse très proche du zéro. De même, les vecteurs qui présentent des vitesses ou directions très différentes par rapport à leurs voisins sont supprimés. Ils sont considérés comme bruits dans ce cas également.

3 Conclusion

Nous avons présenté dans ce chapitre la mise en œuvre de méthodes de détection et de suivi de mouvements sur GPU, soit une mise en œuvre basée sur CUDA pour les traitements parallèles et OpenGL pour la visualisation des résultats. Une implémentation Multi-GPU a été également proposée pour le suivi de mouvements à partir des vecteurs de flot optique, qui sont appliqués sur les points d'intérêt détectés par la méthode de Harris. En outre, les mises en œuvre proposées sont bien adaptées aux nouvelles architectures de processeurs graphiques Fermi [25], et plus particulièrement lors de l'accès aux mémoires GPU pour une exploitation efficace de leurs espaces cache. Les résultats expérimentaux montrent que notre approche GPU peut être largement plus rapide que la version GPU d'OpenCV dans plusieurs cas (en fonction des valeurs de paramètres). De plus, la scalabilité de nos résultats lors de l'exploitation de processeurs graphiques multiples offre les meilleurs taux de calcul en nombre d'images traitées par secondes. Notre approche permet de traiter des séquences vidéo Full HD ou 4K en temps réel.

Chapitre V

Modèle de traitement hétérogène d'objets multimédias

Sommaire

1 Modèle de traitement hétérogène d'objets multimédias . . . 117

1.1 Traitement CPU/GPU d'image individuelle 117

1.2 Traitement hétérogène d'images multiples 122

1.3 Traitement hétérogène de vidéos multiples 122

1.4 Traitement GPU/Multi-GPU de vidéo en temps réel 123

2 Premier cas : segmentation des vertèbres 127

3 Deuxième cas : indexation de vidéos 132

4 Troisième cas : détection de mouvements avec caméra mobile 134

5 Conclusion 139

Nous proposons dans ce chapitre un modèle de traitement d'objets multimédias, exploitant de manière efficace les architectures parallèles (GPU) et hétérogènes (Multi-CPU/Multi-GPU). Ce modèle permet de choisir les ressources à utiliser (CPU ou/et GPU) ainsi que les méthodes à appliquer, selon la nature des médias à traiter et la complexité des algorithmes, tout en assurant un ordonnancement efficace des tâches. Le modèle proposé s'appuie sur les résultats d'implémentations parallèles et hybrides présentés dans les chapitres précédents, permettant de traiter différents types de médias. Ce chapitre est constitué de quatre parties : la première présente le modèle proposé, la deuxième décrit son utilisation pour une application de segmentation des vertèbres [77]. La troisième partie présente l'utilisation du modèle pour une application d'indexation d'images et de vidéos [126]. Enfin, la quatrième partie décrit l'application de notre modèle pour la détection de mouvements en temps réel à partir d'une caméra mobile [136].

Publications principales liées au chapitre :

1. **Revue internationale :**

   [66] : F. Lecron, **S. A. Mahmoudi**, M. Benjelloun, S. Mahmoudi and P. Manneback "Heterogeneous Computing for Vertebra Detection and Segmentation in X-Ray Images", *International Journal of Biomedical Imaging : Parallel Computation in Medical Imaging Applications*, Article ID 640208, 12 pages, 2011. doi :10.1155/2011/640208.

2. **Conférences internationales :**

   [78] : **S. A. Mahmoudi**, F. Lecron, P. Manneback, M. Benjelloun, S. Mahmoudi, "Efficient Exploitation of Heterogeneous Platforms for Vertebra Detection in X-Ray Images", *Biomedical Engineering International Conference (Biomeic'12).* Tlemcen, Algérie, octobre 2012.

   [77] : **S. A. Mahmoudi**, F. Lecron, P. Manneback, M. Benjelloun, S. Mahmoudi, "GPU-Based Segmentation of Cervical Vertebra in X-Ray Images", *Workshop HPCCE. IEEE International Conference on Cluster Computing.* Heraklion, Grèce, septembre 2010.

3. **Publications nationales :**

   [124] : S. Dupont, C. Frisson, **S. A. Mahmoudi**, X. Siebert, J. Urbain, T. Ravet, "MediaBlender : Interactive Multimedia Segmentation and Annotation", *QPSR of the numediart research program, Vol 4, No. 1*, Mars 2011.

   [87] : M. Mancas, J.Tilmanne, R.Chessini, S.Hidot, C.Machy, **S. A. Mahmoudi**, T.Ravet, "MATRIX : Natural Interaction Between Real and Virtual Worlds", *QPSR of the numediart research program, Vol. 1, No. 5*, Janvier 2009.

1 Modèle de traitement hétérogène d'objets multimédias

À partir des résultats d'implémentations proposées dans les sections précédentes, nous proposons un modèle permettant de traiter efficacement quatre types de médias : image individuelle, images multiples, vidéos multiples et vidéo en temps réel.

1.1 Traitement CPU/GPU d'image individuelle

Ce type de méthodes applique des traitements sur une seule image, et affiche le résultat immédiatement sur écran. Ce traitement est utilisé par exemple pour la détection du visage, l'extraction des contours et des régions, l'élimination de bruits, etc. Le temps de calcul de ces méthodes augmente en fonction de la complexité des algorithmes appliqués, et de la taille des données traitées. La nature de ces traitements est bien adaptée à une parallélisation sur GPU, s'ils requièrent suffisamment de calcul permettant de profiter pleinement de la puissance des processeurs graphiques. Ceci s'effectue par le lancement de plusieurs tâches simultanément sur un nombre maximal d'unités de calcul de la carte graphique. Toutefois, ces algorithmes ne peuvent pas être accélérés si la quantité de travail requise est très faible. Dans ce cas, les temps de calcul peuvent même être plus importants que leurs équivalents sur CPU. Une forte dépendance entre les tâches présente également un handicap si on veut exploiter les processeurs graphiques de manière efficace.

Afin d'assurer une exploitation efficace des processeurs graphiques, nous proposons d'estimer dans un premier temps la complexité des algorithmes de traitement d'images. Le but de cette estimation est d'affecter le processeur central aux méthodes à faible intensité de calcul, et le processeur graphique aux méthodes ayant une forte intensité. Nous proposons ainsi d'appliquer un traitement CPU ou GPU d'image individuelle en trois étapes : estimation de complexité de calcul, choix de ressource adaptée, application du traitement.

1.1.1 Estimation de complexité de calcul

Nous proposons d'estimer la complexité temporelle d'algorithmes de traitement d'images à partir de trois paramètres : fraction de parallélisation, taux de calcul par pixel, taux de calcul par image.

1. **Fraction de parallélisation f :** la loi d'Amdahl [46] propose une estimation de l'accélération théorique maximale pouvant être atteinte avec plusieurs processeurs. Cette loi suppose que si f est la fraction du programme pouvant être parallélisée, l'accélération maximale S qui peut être obtenue utilisant N processeurs est :

$$S \leq \frac{1}{1 - f + \frac{f}{N}} \tag{V.1}$$

2. **Taux de calcul par pixel T_{pix} :** comme évoqué dans les chapitres précédents, les processeurs graphiques permettent d'accélérer les algorithmes de traitement d'images grâce à l'exploitation des unités de calcul que possèdent les GPU. Ces accélérations deviennent plus significatives lors de l'application de traitements intensifs, puisque les processeurs graphiques sont particulièrement adaptés aux calculs massivement parallèles. Le nombre d'opérations appliquées à chaque pixel présente ainsi un élément important pour l'estimation de la complexité temporelle du calcul.
3. **Taux de calcul par image T_{img} :** ce facteur est calculé simplement en multipliant la taille de l'image par le taux de calcul par pixel T_{pix} décrit ci-dessus.

Images	**f**	T_{pix}	T_{img}	**Acc**
256 × 256	0.55	6.1	$4.0 * 10^5$	**00.87** ↘
512 × 512	0.81	6.1	$1.6 * 10^6$	**05.88** ↗
1024 × 1024	0.86	6.1	$6.4 * 10^6$	**12.01** ↗
2048 × 2048	0.88	6.1	$2.6 * 10^7$	**16.98** ↗
3936 × 3936	0.88	6.1	$9.4 * 10^7$	**19.85** ↗

Tableau V.1 – Évolution des accélérations en fonction des paramètres de complexité estimés

Le tableau V.1 présente l'évolution des facteurs d'accélération en fonction des paramètres de complexité estimés. Les mesures ont été obtenues pour la méthode de détection des coins et contours d'une image sur GPU (décrite dans la section 3 du chapitre II). La fraction de parallélisation f présente le pourcentage du temps pris par la partie de code pouvant être parallélisée (boucle traitant les pixels) par rapport au temps total. La partie restante $(1 - f)$ présente les opérations d'initialisation, chargement et affichage de l'image. Par exemple, pour une application qui présente une répartition des temps comme suit :

– temps d'initialisation : 0.25 s ;
– temps du traitement (boucle) T_{calcul} : 0.50 s ;
– temps d'affichage : 0.25 s.

La fraction de parallélisation f correspondante est égale à $\frac{T_{calcul}}{T_{total}} = 0.5/1 = 0.5$. Le taux de calcul par pixel T_{pix} présente le nombre moyen d'opérations appliquées pour chaque étape de la détection des coins et contours. Notons que de grandes valeurs de ces paramètres permettent d'obtenir de bonnes accélérations. Cependant, les calculs peuvent même être ralentis lorsque les paramètres de complexité estimés présentent de faibles valeurs.

1.1.2 Choix de ressource adaptée

À partir des paramètres d'estimation de complexité décrits ci-dessus, nous pouvons avoir de bonnes directives pour le choix de la ressource de calcul, soit CPU ou GPU. En effet, nous calculons un facteur de complexité f_c *via* la multiplication du facteur de parallélisation f, le taux de calcul par pixel ainsi que la taille de l'image tel que décrit dans l'équation (V.2).

$$f_c = f \quad \times T_pix \quad \times \quad size \tag{V.2}$$

Tel que :
- f : facteur de parallélisation ;
- T_{pix} : taux de calcul par pixel ;
- $size$: taille de l'image.

Un seuil est utilisé ensuite pour choisir la ressource (CPU ou GPU). Si f_c est inférieur au seuil S, l'image sera traitée sur CPU puisque la méthode appliqué ne nécessite pas assez de traitement. Sinon, si f_c est supérieur au seuil S, l'image sera traitée sur GPU afin d'accélérer le temps de calcul. Le calcul GPU est effectué à partir du schéma proposé dans le chapitre II, s'appuyant sur CUDA pour les traitements parallèle et OpenGL pour la visualisation rapide du résultat. Le choix du seuil est fait à partir de nos résultats expérimentaux montrant que les traitements GPU ne deviennent plus rapides (par rapport au CPU) que lorsqu'on traite une image de taille supérieure à 400×400 appliquant au moins 10 opérations par pixel (exemple de détection des contours sur GPU) avec un facteur de parallélisation f minimal de 0.5. De ce fait, la valeur minimal du facteur de complexité S nécessitant un traitement GPU est de : 800000 (équation (V.1)).

$$S = 0.5 \times 10 \times (400 \times 400) = 800000. \tag{V.3}$$

En outre, si le processeur graphique est choisi pour effectuer le traitement requis, nous pouvons calculer d'autres paramètres permettant d'estimer le taux d'efficacité du traitement GPU. Pour cela, nous proposons de calculer les taux d'utilisation de mémoire partagée et de texture.

1. **Taux d'utilisation de mémoire partagée du GPU T_{par} :** lors de l'application des traitements sur GPU, la mémoire partagée offre un accès plus rapide aux pixels de l'image (lecture/écriture) par rapport à la mémoire globale. Par conséquent, une utilisation maximale de la mémoire partagée offre de meilleures performances.
2. **Taux d'utilisation de mémoire de textures du GPU T_{text} :** lors de l'application des traitements sur GPU, la mémoire texture offre, à son tour, un accès plus rapide aux pixels de l'image (lecture uniquement) par rapport à la mémoire globale. Cette mémoire est également bien adaptée pour l'accès aux tableaux de pixels 2D, permettant de faciliter les opérations de filtrage. Par conséquent, une utilisation maximale de la mémoire de texture offre de meilleures performances.

Images	T_{par}	T_{tex}	**Acc**
512 × 512	0	0	**05.88**
512 × 512	0.37	0	**06.16** ↗
512 × 512	0.37	0.25	**06.34** ↗
3936 × 3936	0	0	**19.85**
3936 × 3936	0.37	0	**20.99** ↗
3936 × 3936	0.37	0.25	**22.10** ↗

Tableau V.2 – Évolution des accélérations en fonction des paramètres de complexité estimés

Le tableau V.2 montre une amélioration des performances grâce à l'exploitation de la mémoire partagée et de texture. Le taux d'utilisation de la mémoire partagée (ou de texture) est calculé par la division du nombre d'accès à la mémoire partagée (ou de texture) par rapport au nombre d'accès à la mémoire globale. Nous pouvons distinguer qu'une application bien adaptée à l'utilisation des mémoires partagée et de texture (accès aux voisins, traitement par groupe, etc.) peut être accélérée significativement grâce à l'accès rapide aux données. Notre estimation du facteur de complexité permet un meilleur choix des ressources. Toutefois, ce facteur peut être plus précis en prenant en compte d'autres paramètres (nombre de branchements, dépendance de tâches, etc.). Le calcul du seuil peut être également amélioré par la prise en compte du taux de dépendance entre calcul des pixels voisins.

1.1.3 Application du traitement

La dernière étape consiste à appliquer des traitements séquentiels si le processeur central est sélectionné. Le CPU est mieux adapté pour les algorithmes appliquant de faibles quantités de calcul sur des petits volumes de données (images). Par ailleurs, le processeur graphique offre de très bonnes performances lors de l'application des traitements intensifs sur des gros volumes de données. Ceci permet d'exploiter efficacement la puissance de calcul du GPU. La figure V.1 résume le calcul CPU/GPU proposé pour le traitement d'image individuelle.

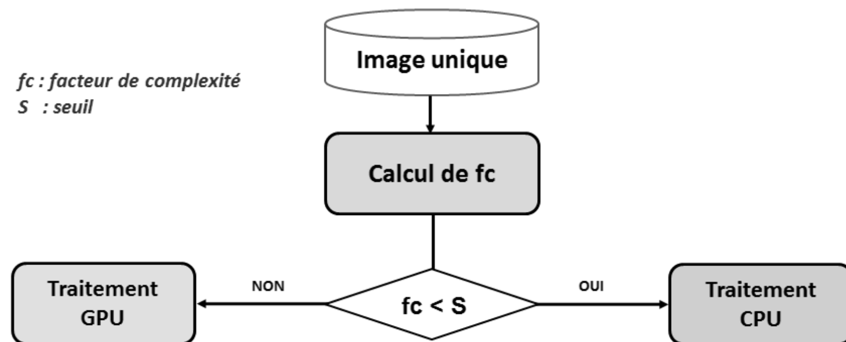


Figure V.1 – Traitement CPU/GPU d'image individuelle

Par ailleurs, les optimisations des calculs GPU dans ce cas (traitement d'image unique) consistent en deux points principaux :
- exploitation de la mémoire partagée et de texture ;
- recouvrement des transferts par les kernels *via* le streaming CUDA.

Remarque :

Les paramètres décrits ci-dessus offrent une estimation efficace des complexités temporelles des algorithmes de traitement d'images. Cependant, cette estimation peut présenter de mauvaises indications si l'algorithme séquentiel analysé n'est pas optimisé. En effet, un algorithme mal codé peut présenter plusieurs contraintes d'évaluation tellles que :

- **Nombre d'instructions élevé :** un code non-optimisé peut faire appel à des opérations mathématiques supplémentaires par rapport à un code optimisé. Par conséquence, les taux de calcul par pixel et par image auront des valeurs élevées mais non significatives ;

- **Nombre de lignes de code élevé :** un code non-optimisé utilise généralement plus de lignes de codes par rapport à un code optimisé. Par conséquent, la fraction de parallélisation f peut prendre des valeurs élevées mais non significatives.

De ce fait, on peut distinguer que les paramètres d'estimation de complexité doivent être extraits à partir d'une version optimisée afin d'éviter d'avoir de mauvaises estimations.

1.2 Traitement hétérogène d'images multiples

Ce type de méthodes consiste à appliquer des traitements sur un ensemble d'images afin d'extraire des caractéristiques communes entre elles. Ces traitements sont utilisés dans plusieurs applications telles que l'indexation d'images, la segmentation de vertèbres des images médicales, etc. L'exploitation des processeurs graphiques pour ces méthodes offre une amélioration relativement faible des performances tel que décrit dans le chapitre II. Ceci est dû à l'étape de présentation des résultats nécessitant un transfert coûteux des images traitées depuis la mémoire GPU vers la mémoire RAM du CPU. Nous proposons dans ce cas d'appliquer des traitements hétérogènes exploitant l'intégralité des ressources hybrides. Ces implémentations appliquent une stratégie d'ordonnancement dynamique permettant une exploitation maximale des unités de calcul (Multi-CPU/Multi-GPU). De plus, nous exploitons la technique de streaming *via* CUDA afin de recouvrir les transferts de données par les exécutions des kernels. Cette mise en œuvre a été décrite en détail dans le troisième chapitre.

Les optimisations des calculs hybrides dans ce cas se résument en trois points :
- exploitation des mémoires partagée et de texture.
- recouvrement des transferts par les kernels *via* le streaming CUDA.
- ordonnancement dynamique (basé sur l'historique) des tâches hétérogènes.

1.3 Traitement hétérogène de vidéos multiples

Ce type de méthodes est appliqué sur un ensemble de séquences vidéo afin d'en extraire des caractéristiques significatives. Celles-ci peuvent être exploitées dans de nombreuses applications telles que le calcul de similarité entre mouvements, l'indexation, la navigation ainsi que la classification de vidéos. Le traitement en temps réel (25 images traitées et affichées par seconde) n'est pas exigé dans ce cas. Le traitement appliqué sur un ensemble

de vidéos peut être vu comme un traitement d'ensemble d'images (une séquence vidéo est toujours représentée par une succession d'images). Par conséquent, nous proposons d'appliquer des traitements hétérogènes sur l'ensemble des images constituant les différentes séquences vidéo. Ceci est effectué en suivant le même principe de traitement hybride d'images multiples présenté dans la section précédente. Les optimisations employées sont également similaires.

1.4 Traitement GPU/Multi-GPU de vidéo en temps réel

Ce type de méthodes permet de traiter une séquence vidéo (ou une capture de caméra) en temps réel, de telle sorte que le résultat soit visualisé immédiatement. Pour respecter cela, les traitements doivent être effectués très rapidement afin d'assurer un taux de rafraichissement de 25 fps. Ces méthodes sont très utilisées dans les domaines de vidéo surveillance, de détection d'évènements, d'études des comportements, etc. Nous proposons dans ce cas d'exploiter la puissance des processeurs graphiques uniques ou multiples (pas de traitement hybride), en s'appuyant sur CUDA pour les traitements parallèles et OpenGL pour la visualisation des trames (images) de la vidéo traitée. Les plates-formes hétérogènes n'étaient pas exploitées dans ce cas à cause de l'aspect temps réel exigeant un traitement et un affichage ordonnancé des trames traitées (lors des traitements hybrides, l'ordonnanceur lance les tâches selon leurs temps de calcul estimé à partir des tâches précédentes et de la disponibilité des ressources). L'application des traitements GPU ou Multi-GPU permet un affichage rapide des résultats grâce à l'utilisation de buffers OpenGL déjà existants sur GPU. Ceci permet une réduction significative des temps de transfert de données entre mémoires CPU et GPU. Cette mise en œuvre est décrite en détail dans le quatrième chapitre.

En outre, nous avons exploité les cartes de capture Serial Digital Interface (SDI) [99] permettant une acquisition directe du flux vidéo sur le processeur graphique. Ceci permet d'appliquer des traitements plus rapides (n'exigeant aucun transfert entre CPU et GPU), et de satisfaire la contrainte du traitement temps réel de vidéos HD ou Full HD multiples. Les cartes SDI permettent l'acquisition de plusieurs flux vidéos (jusqu'à quatre) simultanément puisqu'elles possèdent quatre entrées SDI tel que montré dans la figure V.2. Cette acquisition permet par la suite d'appliquer des traitements Multi-GPU de vidéos multiples. Chaque GPU s'occupera de traiter un flux vidéo.

Les optimisations des traitements GPU dans ce cas se résument en trois points :

- exploitation de la mémoire partagée et de texture ;
- capture directe de flux vidéo multiples sur GPU *via* les cartes SDI ;
- traitement Multi-GPU de vidéos multiples.

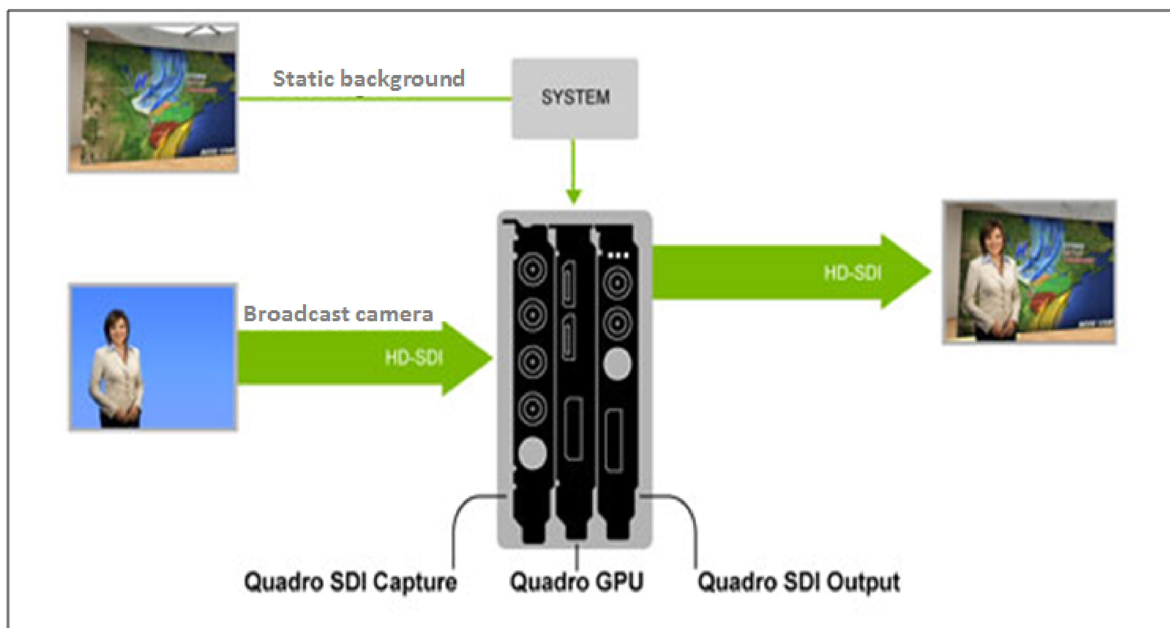


Figure V.2 – Acquisition directe de flux vidéo sur GPU avec les cartes de captures SDI [99]

La figure V.3 résume le modèle proposé pour le traitement de différents types d'objets multimédias (image unique, images multiples, vidéos multiples et vidéo en temps réel). Cette figure montre les étapes (chargement, traitement, optimisation et présentation du résultat) de chaque traitement appliqué sur un type d'objets exploitant les architectures parallèles (GPU) et hétérogènes (Multi-CPU/Multi-GPU). Le tableau V.3 montre ainsi les étapes clés des traitements appliqués par notre modèle sur les différents types de médias.

Les expérimentations (pour les 3 cas d'utilisation) ont été effectuées sous Ubuntu 11.04 (64-bit) avec CUDA 4.0, en utilisant les deux plates-formes suivantes : CPU Xeon E5405 Dual core, GPU Tesla S1070 :

- CPU : 2×Intel Xeon E5405, 2.00GHz, Mem : 16 GB ;
- GPU : NVIDIA GeForce GTX 580, Mem : = 1.5 GB.

Etape/Type	**Image unique**	**Images multiples**	**Vidéos multiples**	**Vidéo (temps réel)**
Chargement	Copie vers GPU	Copie vers tampons StarPU	Copie vers tampons StarPU	Capture SDI
Traitement	Parallèle GPU	Hétérogène Multi-CPU-GPU	Hétérogène Multi-CPU-GPU	Parallèle Multi-GPU
Optimisation	Mém Par & Tex CUDA streaming	Mém Par & Tex CUDA streaming Ord dynamique	Mém Par & Tex CUDA streaming Ord dynamique	Mém Par & Tex Capture multiple Synchronisation
Résultat	Visualisation OpenGL	Copie vers Mém CPU	Copie vers Mém CPU	Visualisation OpenGL

Tableau V.3 – Étapes de traitements d'images et de vidéos à partir du modèle proposé

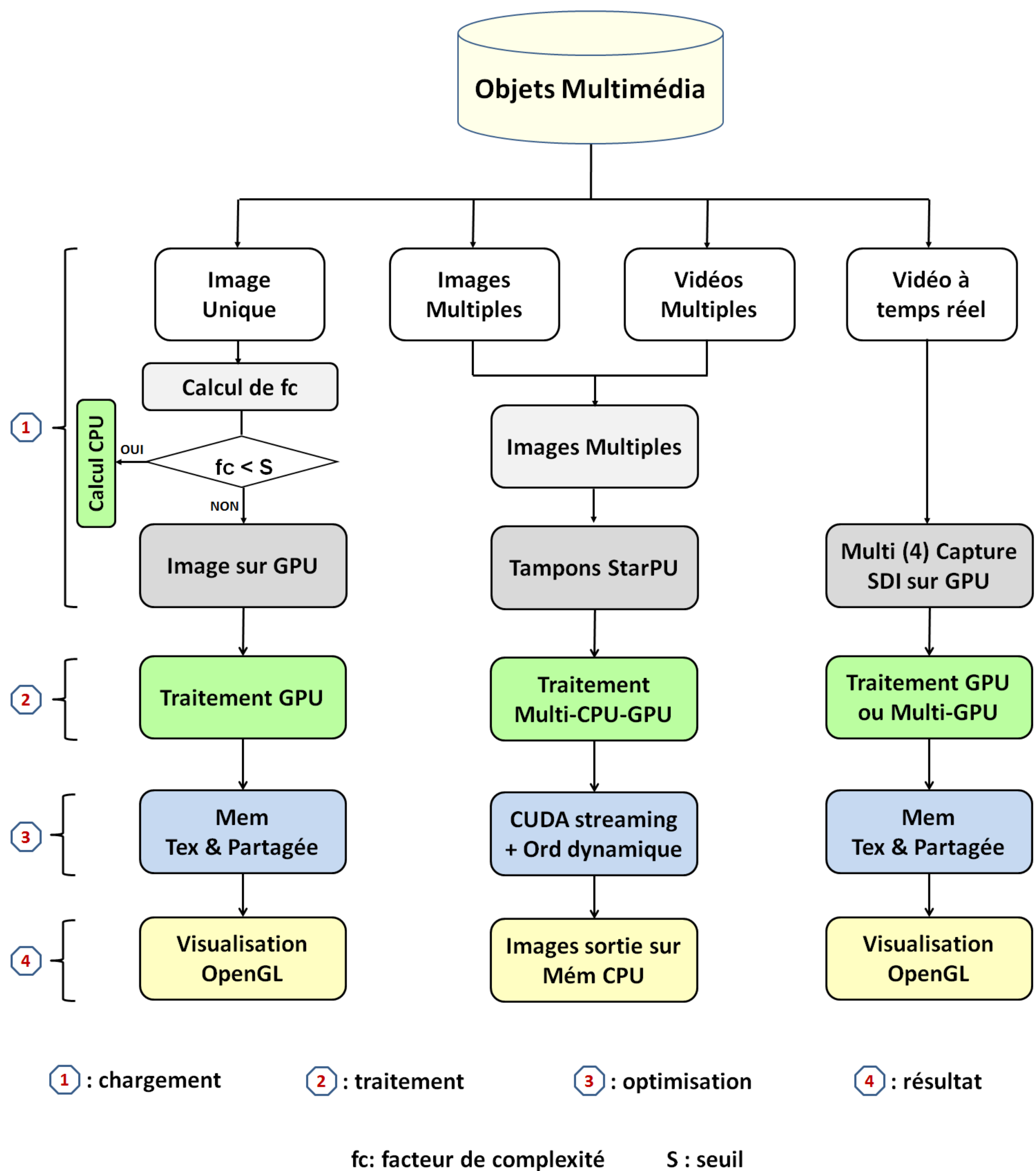


Figure V.3 – Modèle de traitement d'objets multimédias sur architectures parallèles et hétérogènes

2 Premier cas : segmentation des vertèbres

Le contexte de cette application est l'analyse de la mobilité de la colonne vertébrale dans des images X-ray. L'objectif est d'extraire automatiquement les vertèbres. Une solution en plusieurs étapes a été proposée dans [13] (section 4.1 du premier chapitre). Cette méthode est caractérisée par la faible variation du niveau de gris ainsi que le grand volume de données (images multiples) à traiter. Nous proposons d'accélérer cette méthode en exploitant les architectures parallèles (GPU) et hétérogènes. Une estimation de complexité de chaque étape est effectuée, dans un premier lieu, afin de distinguer les étapes ayant une forte et faible intensité de calcul. Cette estimation est effectuée à partir des paramètres décrits dans la section 1.1.1 de ce chapitre. Les étapes d'apprentissage et de modélisation ne sont pas évaluées puisqu'elles sont lancées une seule fois avant de commencer le processus de détection des vertèbres. Les étapes de localisation et de segmentation des vertèbres sont également non évaluées puisqu'elles présentent une forte dépendance entre tâches, ce qui constitue un handicap pour la parallélisation GPU.

Par ailleurs les étapes d'égalisation d'histogramme, de détection des contours et d'extraction des points d'intérêt (à partir des polygones) peuvent être bien adaptées à une parallélisation GPU, elles présentent un faible taux de dépendance entre tâches. Le tableau V.4 présente les paramètres de complexités estimés pour ces étapes. Notons que l'étape de détection des contours est la plus intensive, elle est également la seule étape qui présente un facteur de complexité f_c supérieur au seuil S. Par conséquent, et à partir du modèle proposé dans la section précédente, nous appliquons un traitement Multi-CPU/Multi-GPU de l'étape la plus intensive, soit la détection des contours. Un traitement Multi-CPU est appliqué aux étapes nécessitant peu de calcul, soit l'égalisation d'histogramme et la détection des coins de vertèbres (figure V.4). Le traitement Multi-CPU est appliqué de la même manière que le traitement Multi-CPU/Multi-GPU (décrit dans le chapitre 3) en indiquant seulement dans la codelet que les ressources utilisées sont des processeurs centraux.

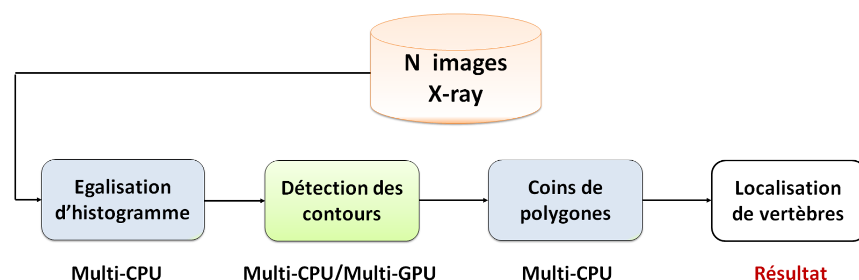
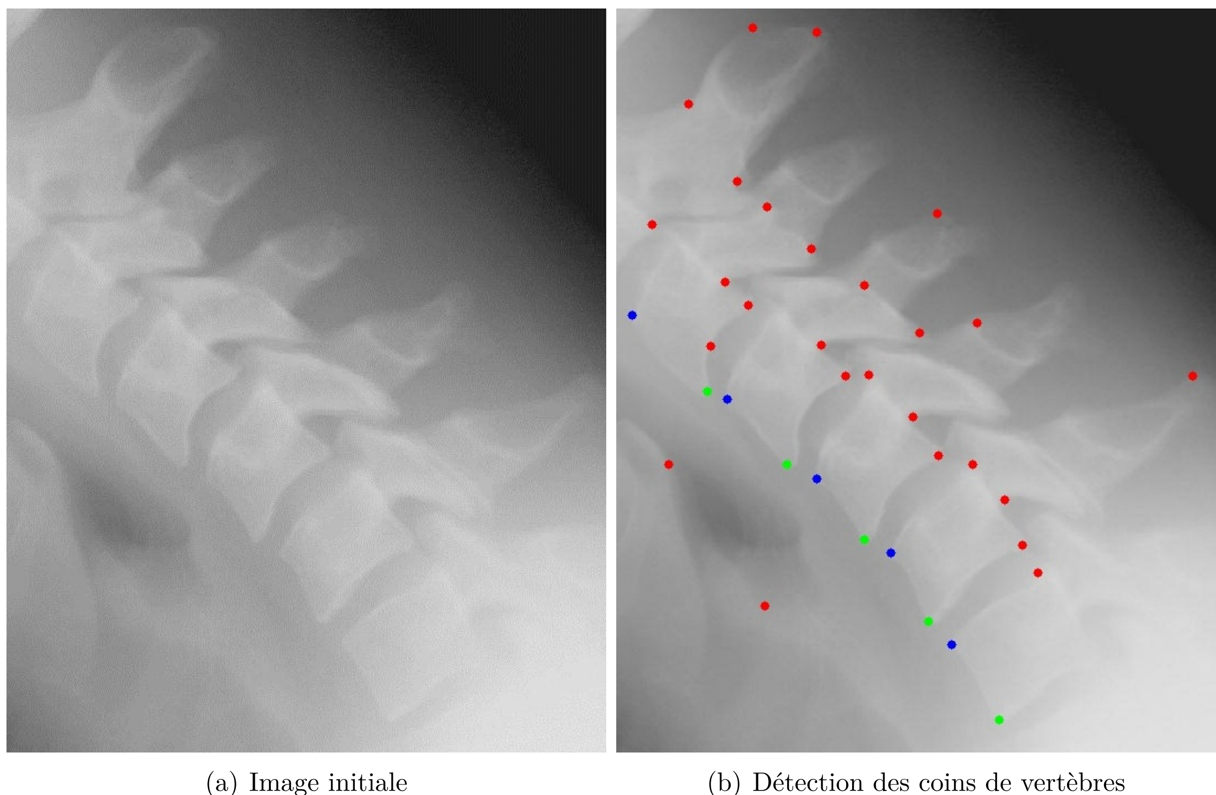


Figure V.4 – Calcul Multi-CPU/Multi-GPU pour la détection des vertèbres

Étapes	Taille-image	Sous-étapes	T_{pix}/sous-étape	T_{pix}/étape	f	f_c	$f_c > S$?
Égalisation	512×512	Egalisation	< 4	< 4	0.68	<713031	**Non**
Détection contours	512×512	Calcul-Gradient	25	30	0.88	6920601	**Oui**
		Direction-Grdient	1				
		Magnitude-Gradient	1				
		Non-max supp	2				
		Seuillage	1				
Coins polygones	512×512	Coins polygones	1	1	0.61	159907	**Non**

Tableau V.4 – Estimation de la complexité temporelle des étapes de détection des vertèbres

Nous présentons dans un premier temps une évaluation qualitative de la méthode proposée (la solution séquentielle a déjà été validée et publiée dans [77]). Les données utilisées dans ce travail sont des radiographies des vertèbres C3 à C7. Cet échantillon d'images provient la base de données NLM accessible gratuitement dans [129]. Les images présentent des versions numérisées de films radiographiques (X-ray) de personnes âgées de 25 à 74 ans. Ces images ont été recueillies entre 1976 et 1980. Une illustration du résultat visuel de notre approche de détection des coins de vertèbres dans une image X-ray est présentée à la figure V.5.



(a) Image initiale (b) Détection des coins de vertèbres

Figure V.5 – Réultats qualitatifs de détection des verèbres

D'une part, la qualité des résultats de segmentation des vertèbres lors de l'exploitation des architectures Multi-CPU/Multi-GPU reste identique puisque la procédure est toujours la même. D'autre part, l'exploitation de ces plates-formes a permis de diminuer significativement les temps de calcul. Ceci permet d'appliquer la méthode sur des plus grands ensembles d'images de haute définition. Par conséquent, les résultats de l'application médicale peuvent être plus précis (formes de vertèbres détectés).

Les accélérations obtenues grâce au traitement hétérogène sont présentées dans le tableau V.5. Ce tableau montre également l'avantage de l'exploitation des techniques d'optimisation du traitement Multi-CPU/Multi-GPU proposées dans notre modèle. L'exploitation des mémoires partagée et de texture (MP & MT) permettent d'atteindre une accélération de 19,79. L'ordonnancement dynamique de tâches hybrides (section 2.1.2 du chapitre III) offre, à son tour, une amélioration des performances, permettant d'atteindre une accélération de 22,39. Cette dernière est encore améliorée lors de l'application du recouvrement des kernels d'exécutions sur GPU multiples par les transferts de données *via* la technique de streaming CUDA. Ces résultats ont été obtenus en utilisant un ensemble de 200 images X-ray avec une résolution de 1476 × 1680.

plates-formes	**Implémentation basique**	**Exploitation de MP & MT**	**Ordonnancement dynamique**	**streaming CUDA**
1GPU/2CPU	08,43 ×	09,26 ×	10,33 ×	11,65 ×
2GPU/4CPU	13,11 ×	14,20 ×	14,96 ×	15,81 ×
4GPU/8CPU	16,93 ×	19,79 ×	22,39 ×	**28,06** ×

Tableau V.5 – Performances du calcul hétérogène et d'optimisations appliquées

Le tableau V.6 présente une comparaison entre les versions CPU, GPU et hétérogènes des étapes de la détection de vertèbres proposée. Les implémentations parallèles et hybrides sont appliquées pour la détection des contours (la plus intensive), tandis que les étapes d'égalisation d'histogramme et de détection des coins de vertèbres (moins intensives) sont lancées sur les CPUs multiples. Le tableau V.6 montre une amélioration significative des performances, et cela grâce à l'exploitation efficace de processeurs graphiques et centraux de manière simultanée.

Etapes	1CPU	8CPU		1GPU/8CPU				4GPU/8CPU			
				1GPU		8CPU		8CPU		4GPU/8CPU	
	Temps(T)	T	Acc	T	Acc	T	Acc	T	Acc	T	Acc
Egalisation histogramme	62.10 s	15.44 s	4.02×	/		15.44 s	4.02×	15.44 s	4.02×	/	
Détection contours	135.8 s	39.06 s	3.48×	15.80 s	08.60×	/		/		4.84 s	28.06×
Détection coins (poly)	46.12 s	11.51 s	4.01×	/		11.51 s	4.01×	11.51 s	4.01×	/	
Temps total	T	T	Acc	T		Acc		T		Acc	
	244.02 s	66.01 s	3.70 x	42.75 s		5.71 x		**31.79** s		**7.68 x**	

Tableau V.6 – Détection des vertèbres sur plates-formes Multi-CPU/Multi-GPU pour un ensemble de 200 images (1476×1680)

3 Deuxième cas : indexation de vidéos

Le but de cette application est de fournir un nouvel environnement d'indexation et de navigation dans des bases d'objets multimédias (images, vidéos), permettant une recherche par requête. Video Cycle [126] présente la partie concernée par l'indexation de séquences vidéo. Cette dernière s'appuie sur le calcul de similarité entre les différentes vidéos par l'extraction des caractéristiques des trames (images) composant chaque séquence. Cette application ne requiert pas de traitement temps réel et de visualisation de vidéo à la fin, puisque l'objectif est d'extraire des informations permettant une navigation efficace dans des bases de vidéos. Toutefois, cette méthode est pénalisée par l'augmentation significative du temps de calcul, lors de l'utilisation de gros ensembles d'images et de vidéos. À partir du modèle proposé, nous appliquons des traitements hétérogènes pour l'étape de calcul la plus consommatrice en temps de calcul, soit la détection de contours. Cette étape permet de fournir des informations pertinentes pour la détection des zones de mouvements qui seront exploités ensuite *via* les moments de Hu [53]. Ces derniers sont également décrits dans la première annexe.

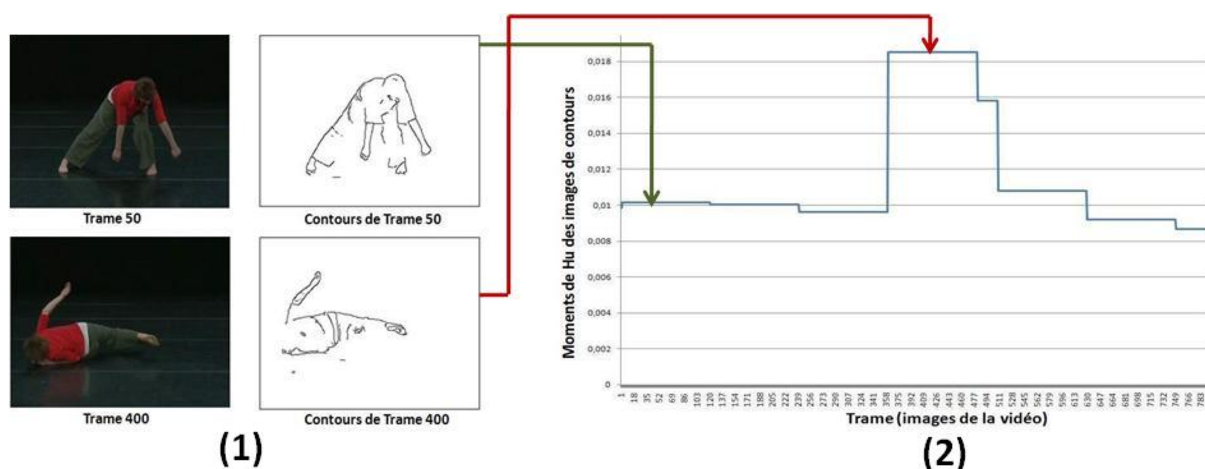
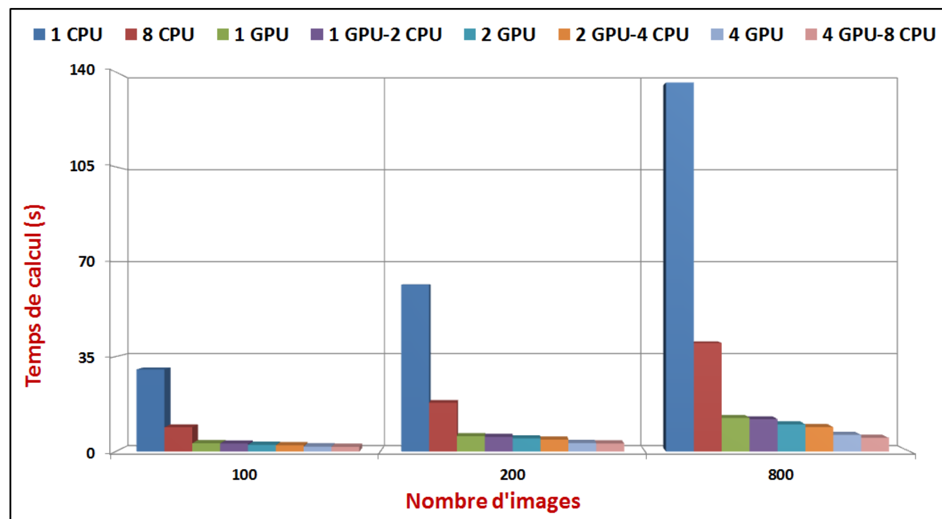


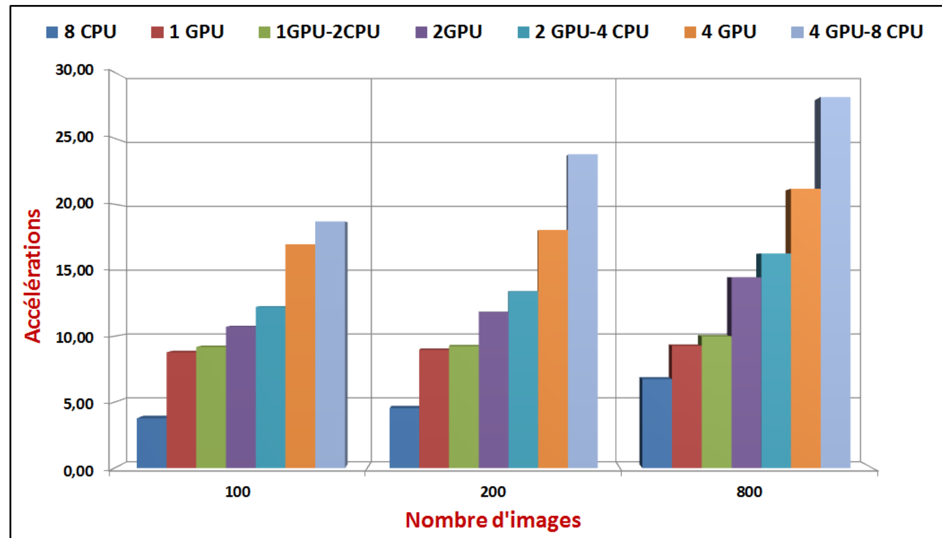
Figure V.6 – Calcul hétérogène pour l'indexation de vidéos

La figure V.6 montre les contours extraits de manière hybride à partir des trames d'une vidéo de danse. Cette figure montre aussi les moments de Hu extraits à partir de ces contours. Notons que notre implémentation hétérogène a permis d'accélérer le calcul de similarité entre séquences vidéo. En effet, la figure V.7(a) présente la comparaison des temps de calcul entre les mises en en œuvres CPU, GPU et hybrides (à partir du modèle proposé) de la méthode de détection des contours. La figure V.7(b) montre les accélérations

obtenues grâce à ces implémentations permettant un gain de 60 % (3 minutes) par rapport au temps total de l'application (environ 5 minutes), lors du traitement de 800 trames d'une séquence vidéo (1080 × 720). Notons que les expérimentations étaient effectuées à partir de vidéos de danseurs filmées dans le cadre du projet artistique « DANCERS » [31]. Ce dernier a comme objectif de fournir une base de données audiovisuelle de danseurs professionnels. Ces données ont été présentées en 2009 à la Biennale de Charleroi-Danses en Belgique [1].



(a) Temps de détection des coins et contours



(b) Accélérations correspondantes

Figure V.7 – Performances du calcul hétérogène appliqué à l'indexation de vidéos (VideoCycle)

1. www.charleroi-danses.be

4 Troisième cas : détection de mouvements avec caméra mobile

Le but de cette application est de détecter les mouvements lors de l'utilisation d'une caméra mobile. Cette section est composée de trois parties : la première décrit le processus séquentiel de l'application, tandis que la deuxième partie présente la version GPU proposée à partir du modèle proposé ci-avant (section 1). Les résultats expérimentaux sont présentés ensuite dans la troisième partie.

1. **Processus séquentiel :**

   L'implémentation séquentielle de cette application [136], proposée par nos collaborateurs de l'Université d'Ankara en Turquie, s'appuie sur deux étapes principales : estimation du mouvement de la caméra et détection du mouvement d'objets en scène.

   (a) **Estimation du mouvement de la caméra :**

   Le mouvement de la caméra est estimé à partir du calcul des vecteurs du flot optique permettant de suivre les points d'intérêt extraits préalablement pour chaque image de la vidéo. Ensuite, les vecteurs du flot optique présentant des vitesses et directions très proches sont sélectionnés. Ils présentent les mouvements estimés du déplacement de la caméra. Cette estimation permet d'extraire la région commune entre deux trames (images) successives. Ceci s'effectue par la soustraction des parties disparues de l'image (trame) de départ, ainsi que les parties apparues dans l'image suivante (trame + 1) suite au mouvement de la caméra. La soustraction s'effectue à partir des valeurs de vitesse et de direction de la caméra estimés à partir des mesures du flot optique

   (b) **Détection du mouvement :**

   Une fois le mouvement de la caméra estimé, nous pouvons détecter les objets en mouvement dans la scène. Ceci s'effectue en appliquant une soustraction entre images successives, qui permet de localiser les mouvements apparus. Ces derniers seront présentés sous formes de groupe de pixels blancs connectés. Notons qu'un seuil est utilisé afin d'éliminer les bruits. Pour plus de détail, nous invitons le lecteur à consulter la section 4.3 du premier chapitre.

Cependant, cette implémentation CPU ne permet pas d'appliquer un traitement en temps réel en raison du temps de calcul élevé des étapes de détection des coins, de

calcul du flot optique et de soustraction entre frames. Les traitements séquentiels sont limités à 4 fps (trames par secondes) lors de l'utilisation d'une vidéo de résolution 640 × 480. Nous proposons ainsi d'accélérer cette application à partir de notre modèle, décrit dans la section 1 de ce chapitre, proposant un traitement GPU ou Multi-GPU de vidéos nécessitant un traitement en temps réel.

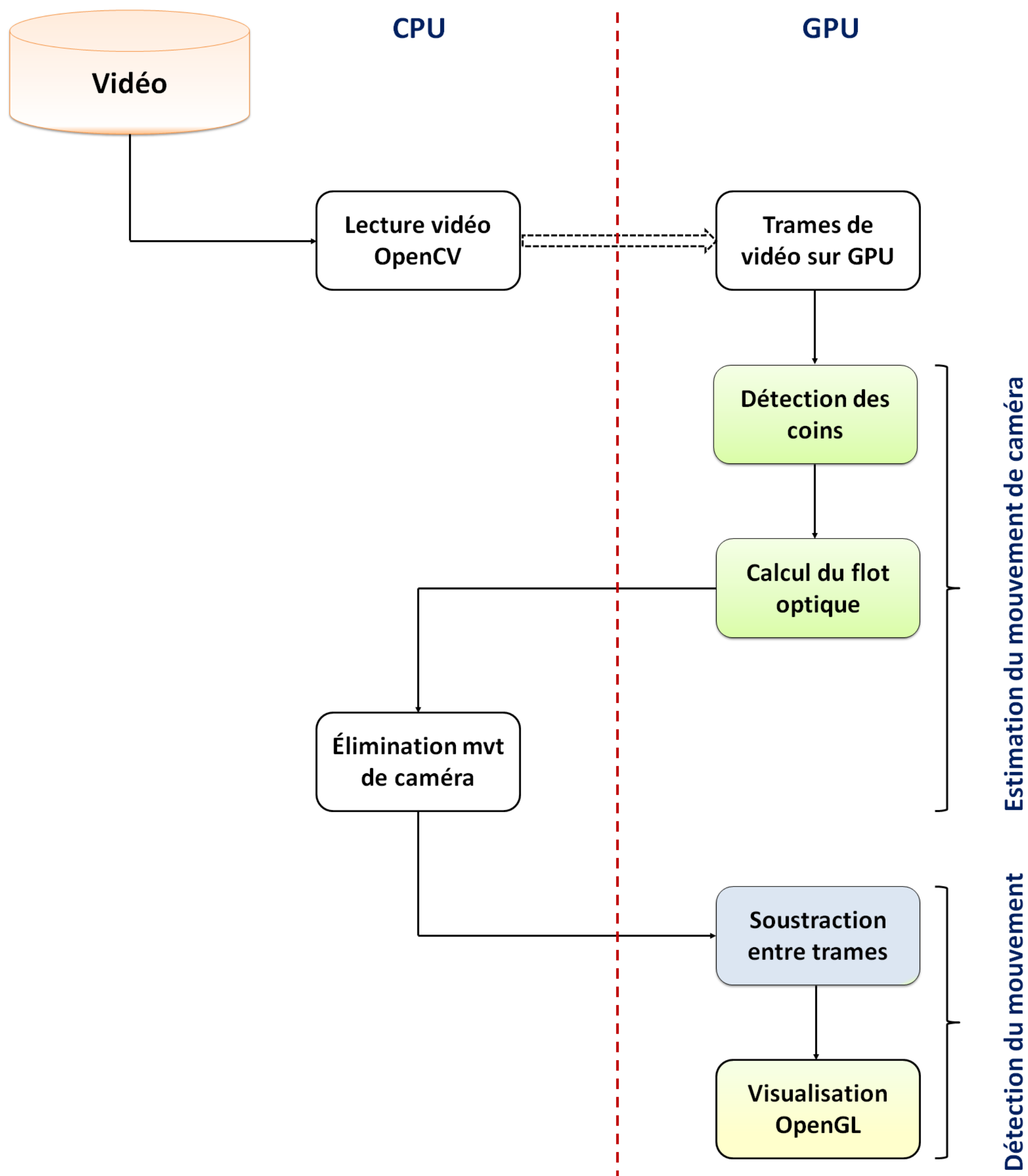


Figure V.8 – Processus de détection de mouvements avec caméra mobile sur GPU

2. **Processus parallèle sur GPU :**

   Afin d'accélérer l'application décrite ci-dessus, nous proposons une mise en œuvre sur GPU permettant d'exploiter ses unités de calcul en parallèle. Les étapes nécessitant un taux de calcul important sont portées sur processeur graphique. Ces étapes se présentent comme suit :

   (a) **Détection des points d'intérêt :** cette détection est appliquée à chaque image de la séquence vidéo. L'implémentation GPU est décrite en détail dans la section 3.2 du deuxième chapitre.

   (b) **Calcul du flot optique :** le calcul du flot optique est appliqué aux coins détectés pour chaque image de la vidéo. L'implémentation GPU de cette étape est décrite en détail dans la section 2.3 du quatrième chapitre.

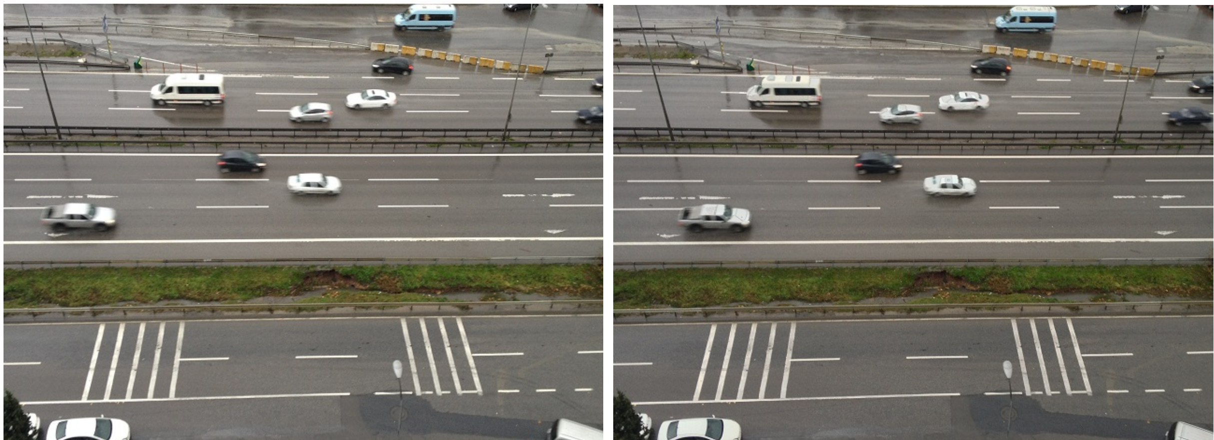
   (c) **soustraction d'images :** cette étape applique une soustraction entre chaque couple d'images successives afin de détecter les mouvements, après avoir détecté et éliminé ceux de la caméra. La version GPU de cette étape est décrite en détail dans la section 1.2 du quatrième chapitre.

   La figure V.8 résume le processus proposé pour la détection de mouvements lors de l'utilisation d'une caméra mobile. Cette figure montre également les étapes portées sur GPU afin d'accélérer le traitement et atteindre un calcul en temps réel.

3. **Résultats expérimentaux :**

   La qualité du résultat du traitement GPU reste identique au traitement CPU puisque nous utilisons le même processus de calcul. Les figures V.9(a) et V.9(b) montrent deux images prises avec deux positions différentes de la caméra. La figure V.9(c) présente le résultat d'élimination de la partie apparue dans la première image suite au mouvement de la caméra (en bas à gauche), tandis que la figure V.9(d) présente le résultat d'élimination de la partie apparue dans la deuxième image suite au mouvement de la caméra (en haut à droite). La taille de la partie soustraite pour les deux images est calculée à partir des vecteurs du flot optique. La figure V.10(a) montre la partie commune extraite entre les deux images. Les cadres, rouge et jaune, présentent les images de départ (trame i) et d'arrivée (trame $i+1$), tandis que le cadre blanc présente le résultat de l'extraction de la partie commune entre les deux images. La figure V.10(b) présente le résultat de la soustraction entre les deux images, montrant les mouvements des voitures détectés lors de l'utilisation d'une caméra mobile.

(a) Image 1, capturée avec une caméra mobile (b) Image 2, capturée avec une caméra mobile

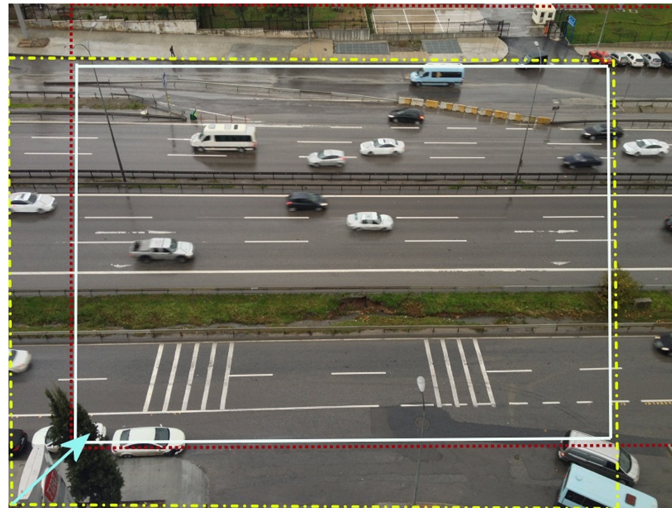


(c) Soustraction du mouvement de caméra (image 1) (d) Soustraction du mouvement de caméra (image 2)

Figure V.9 – Estimation et élimination du mouvement de la caméra

Résolution	CPU	GPU	Accélération
512×512	5,4 fps	109 fps	20,18 ×
1280×720	3,2 fps	72 fps	22,50 ×
1920×1080	2,1 fps	53 fps	25,24 ×

Tableau V.7 – Performances GPU de la détection de mouvements à partir d'une caméra mobile



(a) Partie commune extraite entre images capturées par une caméra mobile

(b) Détection de mouvements lors de l'utilisation d'une caméra mobile

Figure V.10 – Estimation et élimination du mouvement de la caméra

Par ailleurs, l'exploitation des processeurs graphiques a permis d'améliorer considérablement les performances de détection de mouvements avec une caméra mobile. Cette amélioration a permis d'atteindre un traitement en temps réel de vidéos de haute définition. Le tableau V.7 présente une comparaison en termes du nombre d'images traitées par secondes (fps) entre les implémentations CPU et GPU. Ce tableau présente des accélérations allant de 20 à 25 grâce à l'exploitation efficace des processeurs graphiques.

5 Conclusion

Ce chapitre résume les implémentations proposées pour le traitement d'images et de vidéos sur architectures parallèles et hétérogènes. Ces mises en œuvre sont présentées par un modèle adapté au traitement de quatre types d'objets multimédias : image individuelle, images multiples, vidéos multiples et vidéo à temps réel. Une estimation de la complexité temporelle est également proposée pour le traitement d'image individuelle. Ceci permet d'exploiter le processeur de manière efficace, tout en évitant de lui affecter des tâches à faible intensité de calcul. Afin de valider le modèle proposé, nous avons proposé trois cas d'utilisation appliquant des traitements sur trois types de médias :

– Traitement hétérogène d'images multiples : segmentation des vertèbres (premier cas) ;
– Traitement hétérogène de vidéos multiples : indexation de vidéos (deuxième cas) ;
– Traitement Multi-GPU de vidéo en temps réel : détection de mouvements en temps réel avec une caméra mobile (troisième cas).

Les résultats expérimentaux montrent des accélérations significatives de ces trois applications. Ces accélérations sont comprises entre 7 et 30 par rapport à une implémentation séquentielle sur un CPU dual-core.

Conclusion générale et perspectives

Le travail présenté dans ce rapport de thèse est lié aux domaines de programmation parallèle, de calcul haute performance et de vision par ordinateur. Plus particulièrement, nous nous adressons à l'exploitation des nouvelles architectures parallèles (GPU) et hétérogènes (Multi-CPU/Multi-GPU) pour accélérer les algorithmes de traitement d'images et de vidéos de haute définition (HD/Full HD).

À travers ce rapport, nous avons présenté dans le premier chapitre les différents outils et techniques, publiées dans la littérature scientifique, permettant la programmation de ces nouvelles architectures et plus précisément pour le traitement de gros volumes d'images et de vidéos. Cependant, ces techniques sont entravées par différents facteurs indispensables pour obtenir une approche complète. Ces facteurs peuvent être résumés en cinq points comme suit :

1. **Réduction des coûts de transfert entre mémoires CPU et GPU :** ces transferts présentent le premier handicap lors du traitement d'images et de vidéo sur processeurs graphiques. Une réduction maximale des temps de ces transferts est nécessaire pour obtenir de meilleures performances ;
2. **Gestion des mémoires de machines hétérogènes (Multi-CPU/Multi-GPU) :** l'utilisation de ces architectures nécessite une gestion efficace de leurs espaces mémoires afin d'assurer un accès rapide aux données ;
3. **Scalabilté de performances :** l'utilisation de GPU ou/et CPU multiples nécessite d'affecter de manière efficace les ressources aux tâches, afin d'atteindre des performances évoluant en fonction du nombre de processeurs (CPU ou/et GPU) utilisées ;
4. **Choix adapté de ressources (CPU ou/et GPU) :** en fonction du type du traitement et de données à traiter ;

5. **Respect de la contrainte du du traitement en temps réel :** lors de l'utilisation de vidéos de haute définition (HD/Full HD).

Pour respecter ces contraintes, nous avons proposé un modèle d'exploitation efficace des architectures parallèles (GPU) et hétérogènes (Multi-CPU/Multi-GPU) pour le traitement de différents types d'objets multimédias, soit image individuelle, images multiples, vidéos multiples et vidéo en temps réel. Le modèle a comme objectif d'affecter efficacement les ressources (CPU ou/et GPU) à partir de la nature des traitements appliqués et des objets multimédias. Le modèle s'appuie sur les implémentations CPU, GPU et hétérogènes décrites dans les chapitres 2, 3 et 4. Notons que les versions CPU sont fournies par la bibliothèque OpenCV [102]. Les principales contributions proposées avec ce modèle se résument dans les cinq points suivants :

1. **Traitement parallèle intra-image :** ceci consiste à porter de manière efficace les différents algorithmes de traitement d'images et de vidéos sur GPU. Ce calcul s'appuie sur CUDA pour les traitements parallèles et OpenGL pour la visualisation des données. Le nombre de threads CUDA lancés est calculé automatiquement en fonction de la taille des images utilisées. Cette contribution est décrite en détail dans le deuxième chapitre.
2. **Traitement hétérogène inter-image :** ceci consiste à exploiter à la fois les multiples processeurs centraux (CPU) et graphiques (GPU), chaque cœur (CPU ou GPU) traitant un sous-ensemble d'images. L'ordonnancement des tâches est appliqué dynamiquement de telle sorte que les tâches nécessitant de grands calculs (temps d'exécution et de transfert élevés) soient prioritaires pour un traitement parallèle sur GPU. À l'inverse, les tâches nécessitant peu de calcul seront prioritaires pour un traitement séquentiel sur CPU. Cette contribution est décrite en détail dans le troisième chapitre.
3. **Gestion efficace des mémoires GPU :** ceci consiste à réduire les coûts de transfert entre mémoires CPU et GPU. L'accès aux mémoires GPU est accéléré grâce à l'exploitation des mémoires « partagée et de texture » du processeur graphique. Les étapes de chargement et de traitement d'images sont accélérées avec le recouvrement des transferts de données par les kernels d'exécutions sur des processeurs graphiques uniques ou multiples (CUDA streaming). Finalement, la visualisation des résultats est accélérée grâce à l'utilisation de buffers OpenGL déjà existants sur GPU. Cette contribution est décrite en détail dans les chapitre 2 et 3.

4. **Traitement GPU/Multi-GPU de vidéo Full HD en temps réel :** ceci consiste à appliquer des traitements GPU ou Multi-GPU sur des vidéos HD ou Full HD en temps réel. Les algorithmes mis en œuvre permettent de détecter les coins et de calculer les mesures du flot optique, utilisées dans les applications de suivi de mouvements en temps réel. Ces mises en œuvre s'appuient sur les traitements CUDA ainsi que les optimisations décrites ci-avant, offrant des performances augmentant en fonction du nombre de GPU utilisés. En outre, ils sont bien adaptés aux nouvelles architectures des processeurs graphiques Fermi [25] et plus particulièrement lors de l'accès aux mémoires GPU pour une exploitation efficace de leurs espaces caches. Cette contribution est décrite en détail dans le quatrième chapitre.
5. **Choix efficace des ressources :** ceci consiste à affecter les unités de calcul (CPU ou/et GPU) selon la nature des méthodes à appliquer et de données (image individuelle, images multiples, vidéos multiples, vidéo à temps réel) à traiter. Un facteur de complexité est également estimé afin d'évaluer la quantité de calcul nécessaire pour les méthodes de traitement d'image individuelle. Ceci permet d'exploiter les processeurs graphiques dans les applications qui requièrent suffisamment de calcul pour profiter de la puissance des GPU. Cette contribution est décrite en détail dans le cinquième chapitre.

Ces contributions sont exploitées dans trois applications traitant différents type d'objets multimédias. La première consiste à segmenter les vertèbres dans un ensemble d'images X-ray multiples [78], tandis que la deuxième est dévolue à l'indexation et la navigation dans un ensemble de séquences vidéo multiples [82]. À partir du modèle proposé, nous appliquons des traitements hétérogènes à ces deux applications, ce qui offre des accélérations allant de de 4 à 30 par rapport à une implémentation séquentielle sur CPU.

Le troisième cas d'application de ces contributions est une méthode de détection de mouvements en temps réel lors de l'utilisation de caméra mobile. À partir du modèle proposé, nous appliquons des traitements GPU ou Multi-GPU de la vidéo, ce qui permet d'atteindre un traitement de vidéos Full HD en temps réel. Ceci est 25 fois plus rapide que son équivalent sur CPU.

Comme perspectives, nous envisageons d'améliorer le choix de ressources (CPU ou/et GPU) par **l'estimation d'un facteur de complexité plus précis**. En plus des paramètres de fraction de parallélisation f et taux de calcul par pixel et image, ce facteur devra prendre en compte des paramètres supplémentaires tels que le ratio de temps entre calcul et accès mémoire, le graphe de dépendances de tâches, le nombre d'instructions de contrôle et de branchement. Ceci devrait permettre de mieux définir les tâches pouvant bénéficier de la puissance des processeurs graphiques.

Par conséquent, nous pourrions **améliorer la stratégie d'ordonnancement employée au sein des unités de calcul hétérogènes (Multi-CPU/Multi-GPU)**. Cette stratégie prendra en compte le type de données ainsi que le facteur de complexité décrit ci-avant, afin d'affecter les tâches aux ressources adéquates. Les tâches nécessitant un calcul intensif seront prioritaires pour un traitement GPU, tandis que les tâches demandant peu de calcul seront affectées aux processeurs centraux. Le but est d'augmenter au maximum le taux d'occupation des ressources (CPUs et GPUs). Le nombre de ressources utilisées devrait être défini automatiquement en fonction de la complexité du calcul, afin d'éviter l'utilisation inutile des processeurs et de réduire la consommation d'énergie. L'évolution de l'occupation de chaque ressource (CPU et GPU) sera présentée en fonction des traitements appliqués en temps réel.

Nous envisageons aussi d'**analyser la consommation d'énergie** lors du lancement des applications de traitement d'objets multimédias. Le but est de lancer les calculs sur les ressources offrant une **consommation minimale d'énergie tout en gardant les mêmes performances**. L'utilisateur pourrait également donner la priorité à l'aspect de consommation d'énergie par rapport aux performances de calcul et *vice versa*.

Une autre perspective de ce travail est de **développer un système intelligent de suivi de mouvements**. Ce système présente une amélioration de la méthode présentée dans le quatrième chapitre. L'objectif est d'assurer une détection et un suivi de mouvements en temps réel de vidéos de haute définition. Le nombre de GPU utilisés devrait être défini automatiquement de telle sorte que l'aspect temps réel doit toujours être respecté. La présence de plusieurs objets en mouvement dans la scène peut en effet provoquer une augmentation du temps de calcul et par conséquent le non-respect de l'aspect temps réel. Dans ce cas, le système devra faire appel à un ou plusieurs GPU disponibles dans la machine. Ce système pourrait également utiliser les cartes de capture SDI permettant de

lire quatre vidéos simultanément, et de les traiter ensuite séparément de telle sorte que chaque processeur graphique puisse appliquer son propre traitement avant de visualiser tous les résultats (quatre vidéos traitées en temps réel) sur le même écran (quatre fenêtres). Ce processus pourrait être d'une grande utilité dans les systèmes de vidéosurveillances. Notons que ce travail est déjà lancé en collaboration avec le laboratoire PSNC (Poznan Supercomputing and Networking Center) à Poznan en Pologne.

Bibliographie

[1] AMD, "Ati stream technology," 2008. [Online]. Available : http://www.amd.com/US/PRODUCTS/TECHNOLOGIES/STREAM-TECHNOLOGY/Pages/stream-technology.aspxl 4

[2] ——, "The future brought to you by amd introducing the amd apu family." *AMD Fusion^{TM}Family of APUs*, 2011. [Online]. Available : http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx/ 2

[3] E. L. Andrade, S. Blunsden, and R. B. Fisher, "Hidden markov models for optical flow analysis in crowds," in *Proceedings of the 18th International Conference on Pattern Recognition*, ser. ICPR '06, vol. 1. Washington, DC, USA : IEEE Computer Society, 2006, pp. 460–463. [Online]. Available : http://dx.doi.org/10.1109/ICPR.2006.621 32, 169

[4] T. Apodaca and D. Peachey, "Writing renderman shaders," *Siggraph '92 course notes (course21)*, 1992. 12

[5] Apple, "Grand Central Dispatch. A better way to do multicore," 2009. [Online]. Available : http://developer.apple.com/technologies/mac/snowleopard/gcd.html 28

[6] E. Ardizzone and M. La Cascia, "Video indexing using optical flow field," *Int. Conf. on Image Processing, ICIP-96, Lausanne, Switzerland*, pp. 16–19, 1996. 97

[7] H. Asada and M. Brady, "The curvature primal sketch," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8(1), pp. 2–14, 1986. 162

[8] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," *In Concurrency and Computation : Practice and Experience, Euro-Par 2009, best papers issue*, pp. 863–874, 2009. 5, 21, 75

[9] C. Augonnet, "Scheduling Tasks over Multicore machines enhanced with Accelerators : a Runtime SystemŠs Perspective," PhD Thesis, Université Bordeaux 1, 2011. 79

[10] C. Augonnet, J. Clet-Ortega, S. Thibault, and R. Namyst, "Data-Aware Task Scheduling on Multi-Accelerator based Platforms," in *The 16th International Conference on Parallel and Distributed Systems (ICPADS)*, Shanghai, China, Dec. 2010. [Online]. Available : http://hal.inria.fr/inria-00523937 75

[11] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Orti, "An Extension of the StarSs Programming Model for Platforms with Multiple GPUs," *Proceedings of the 15th International Euro-Par Conference on Parallel Processing. Euro-Par'09*, pp. 851–862, 2009. 23

[12] P. Bellens, J. M. Pérez, R. M. Badia, and L. J, "CellSs : a programming model for the Cell BE architecture," *SC 2006 Conference, Proceedings of the ACM/IEEE*, pp. 5–15, 2006. 23

[13] M. Benjelloun, S. Mahmoudi, and F. Lecron, "A New Semi-Automatic Approach For X-Ray Cervical Images Segmentation Using Active Shape Model," in *Proceedings of the 3rd International Conference on Bio-inspired Systems and Signal Processing*, 2010, pp. 501–506. 127

[14] M. Benmoussat and A. Belbachir, "diamond search algorithm for block based motion estimation," *IEEE Transactions On Image Processing*, 2003. 32, 97, 167

[15] N. Birkbeck, M. Sofka, and S. K. Zhou, "Fast boosting trees for classification, pose detection, and boundary detection on a GPU," *Computer Vision and Pattern Recognition Workshop CVPRW, IEEE Computer Society Conference*, pp. 36–41, 2011. xiii, 31

[16] D. Blythe, "The direct3d 10 system," in *ACM SIGGRAPH 2006 Papers*, ser. SIGGRAPH '06. New York, NY, USA : ACM, 2006, pp. 724–734. [Online]. Available : http://doi.acm.org/10.1145/1179352.1141947 13

[17] J. Y. Bouguet, "Pyramidal Implementation of the Lucas Kanade Feature Tracker, Description of the algorithm," *Intel Corporation Microprocessor Research Labs.* 5, 60, 99, 100, 162

[18] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679–714, 1986. 29, 30, 63, 165

[19] CAPS, "The Many-core Programming Company." [Online]. Available : http://www.caps-entreprise.com/ 24

[20] CAPS, CRAY, PGI, and NVIDIA, "OpenACC : Directives for Accelerators." [Online]. Available : http://www.openacc-standard.org/ 24

[21] K. Chang, "Computation for bilinear interpolation," *Introduction to Geographic Information Systems, 5th ed. New York, NY : McGraw-Hall. Print*, 2009. 12

[22] D. Chetverikov, J. Liang, J. Komuves, and R. Haralick, "Zone classification using texture features," *Proceedings of the 13th International Conference on Pattern Recognition*, vol. 3, pp. 676–680, 1996. 30

[23] T. F. Cootes and C. J. Taylor, "Active Shape Models - ' Smart Snakes '," in *Proceedings of the British Machine Vision Conference.* Springer-Verlag, 1992, pp. 266–275. 39

[24] T. F. Cootes, C. J. Taylor, D. H. Cooper, and J. Graham, "Active shape models-their training and application," *Computer vision and image understanding*, vol. 61, no. 1, pp. 38–59, 1995. 39

[25] N. Corporation, "Nvidia gf100. world's fastest gpu delivering great gaming performance with true geometric realism." [Online]. Available : http://www.nvidia.com/object/GTX_400.html 10, 12, 114, 143

[26] ——, "Second-generation unified gpu architecture for visual computing," *Technical Brief. NVIDIA GeForce GTX 200 GPU Architectural Overview*, 2008. 10, 12

[27] ——, "Nvidia geforce gtx 580 gpu datasheet," 2010. [Online]. Available : http://www.geforce.com/Active/en_US/en_US/pdf/GTX-580-Web-Datasheet-Final.pdf xvii, 9

[28] CRAY, "The supercomputing Company." [Online]. Available : http://www.cray.com/Home.aspx 24

[29] F. Cupillard, A. Avanzi, F. Bremond, and M. Thonnat, "Video understanding for metro surveillance. Networking, Sensing and Control," *IEEE International Conference on Networking, Sensing and Control*, vol. 1, pp. 186–191, 2004. 33, 169

[30] P. Da Cunha Possa, S. Mahmoudi, N. Harb, and C. Valderrama, "A new self-adapting architecture for feature detection," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2012. 50

[31] Dancers, "Interactive video database of professional dancers," *A project of the company Bud Blumenthal*, p. 85, 2009. [Online]. Available : http://www.dancersproject.com 133

[32] C.-D. Daniel, M. Dominik, S. Andreas, P. Sabine, and S. F. Achilleas, "Performance evaluation of image processing algorithms on the GPU," *Journal of Structural Biology, 164(1)*, pp. 153–160, 2008. 29

[33] A. del Bimbo and P. a. J. L. C. Nesi, "Optical flow computation using extended constraints," *IEEE transaction on image processing*, pp. 720–739, 1996. 32

[34] R. Deriche, "Using Canny's criteria to derive a recursively implemented optimal edge detector," *Internat. J. Vision,Boston*, pp. 167–187, 1987. 5, 62

[35] R. Deriche and T. Blaszka, "Recovering and characterizing im-age features using an efficient model based approach," *In Proceedings of the Conference on Computer Vision and Pattern Recognition , New York, USA*, pp. 530Ű–535, 1993. 162

[36] D. H. Douglas and T. K. Peucker, "Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature," *Cartographica : The International Journal for Geographic Information and Geovisualization*, vol. 10, no. 2, pp. 112–122, 1973. 41, 162

[37] S. Dupont, N. d'Alessandro, T. Dubuisson, C. Frisson, R. Sebbe, and J. Urbain, "Audio cycle," in *QPSR of the numediart research program*, T. Dutoit and B. Macq, Eds., vol. 1, no. 4. numediart Research Program on Digital Art Technologies, 12 2008, pp. 119–127. [Online]. Available : http://www.numediart.org/docs/numediart_2008_s04_p1_report.pdf 43

[38] I. ET International, “SWARM (SWift Adaptive Runtime Machine),” 2011. [Online]. Available : http://www.etinternational.com/index.php/products/swarmbeta/ xiii, 26, 27

[39] X. F and M. K, “Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware,” *IEEE Nucl. Sci., 52*, pp. 654–663, 2005. 30

[40] C. Farrugia, Horain, and Y. Alusse, *Multimedia and Expo, 2006 IEEE International Conference on Digital Object Identifier.* 30

[41] A. Fonseca, L. Mayron, D. Socek, and O. Marques, “Design and implementation of an optical flow-based autonomous video surveillance system,” *Proceedings of the IASTED*, pp. 209–214, 2008. 89

[42] J. Fung, S. Mann, and C. Aimone, “OpenVIDIA :Parallel gpu computer vision.” *In Proc of ACM Multimedia*, pp. 849–852, 2005. 30

[43] J. Gibson, “The perception of the visual world,” *Houghton Mifflin*, 1950. 97

[44] C. Goodall, “Procrustes Methods in the Statistical Analysis of Shape,” *Journal of the Royal Statistical Society. Series B*, vol. 53, no. 2, pp. 285–339, 1991. 39

[45] GPU4VISION, “GPU4VISION,” 2010. [Online]. Available : http://www.gpu4vision.org xiii, 13

[46] A. Grama, A. Gupta, G. Karypis, and V. Kumar, “Introduction to Parallel Computing,” *second ed. Pearson Education Limited*, 2003. 36, 118

[47] K. GROUP, “Khronos group : Open standards for media authoring and acceleration,” 2000. [Online]. Available : http://www.khronos.org/about/ 18

[48] T. P. Group, “PGI Optimizing Fortran, C and C++ Compilers & Tools.” [Online]. Available : http://www.pgroup.com/ 24

[49] C. Harris, “A combined corner and edge detector,” *In Alvey Vision Conference*, pp. 147–152, 1988. 5, 29, 99, 162

[50] Y. Heng and L. Gu, “GPU-based Volume Rendering for Medical Image Visualization,” *Proceedings of the 2005 IEEE Engineering in Medicine and Biology 27th Annual Conference Shanghai, China*, pp. 5145–5148, 2005. 31

[51] R. Horaud, T. Skordas, and F. Veillon, "Finding geometric and relational structures in an image," *In Proceedings of the 1st Euro-pean Conference on Computer Vision, Antibes, France*, 1986. 162

[52] B. K. P. Horn and B. G. Schunk, "Determining Optical Flow," *Artificial Intelligence*, vol. 2, pp. 185–203, 1981. 32, 97, 98, 167, 168, 169

[53] M. K. Hu, "Visual Pattern Recognition by Moment Invariants," *In : IRE Transactions on Information Theory IT-8*, pp. 179–187, 1962. 45, 132, 165, 166

[54] J. Huang, S. Ponce, S. Park, Y. Cao, , and F. Quek, "Gpu-accelerated computation for robust motion tracking using cuda framework," *In Proceedings of the IET International Conference on Visual Information Engineering*, 2008. 33

[55] N. Ihaddadene and C. Djeraba, "Real-time crowd motion analysis," *In Proceedings of the 19th International Conference on Pattern Recognition (ICPR '08)*, 2008. 33, 169

[56] S. Indu, M. Gupta, and A. Bhattacharyya, "Vehicle Tracking and Speed Estimation Using Optical Flow Method," *International Journal of Engineering Science and Technology*, vol. 3(1), 2011. 32, 169

[57] Intel, "Intel core^{TM}i7-980 processor, (12m cache, 3.33 ghz, 4.8 gt/s intel qpi)," *Your Source for Intel Product Information*, 2011. [Online]. Available : http://ark.intel.com/products/58664 xvii, 9

[58] S. Jean-Luc and M. Fionn, "Astronomical Image and Data Analysis," *Springer, Internatinal Publisher*, 2006. 29

[59] I. Jolliffe, "Principal Component Analysis," *Series : Springer Series in Statistics. 2nd ed.*, 2002. 29

[60] K. K, R. Mukherjee, M. S. Rehman, S. Patidar, P. J. Narayanan, and K. Srinathan, "A performance prediction model for the cuda gpgpu platform," *International Conference on High Performance Computing HiPC*, pp. 463–472, 2009. 35

[61] Z. Kalal, K. Mikolajczyk, and J. Matas, "Face-tld : Tracking-learning-detection applied to faces," *IEEE International Conference on Image Processing, ICIP*, 2010. 33, 169

[62] J. Kessenich, D. Baldwin, and R. Rost, “The opengl shading language,” *Computer graphics. 2*, 2008. 13

[63] Khronos-Group, “The Open Standard for Parallel Programming of Heterogeneous Systems,” 2009. [Online]. Available : http://www.khronos.org/opencl 4, 13

[64] J. Kim, H. Kim, J. H. Lee, and J. Lee, “Achieving a single compute device image in opencl for multiple gpus,” in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, ser. PPoPP ’11. New York, NY, USA : ACM, 2011, pp. 277–288. [Online]. Available : http://doi.acm.org/10.1145/1941553.1941591 30

[65] B. Kitt, B. Ranft, and H. Lategahn, “Block-matching based optical flow estimation with reduced search space based on geometric constraints,” *13th International IEEE Annual Conference on Intelligent Transportation Systems, Madeira Island, Portugal*, pp. 1104–1109, 2010. 171

[66] F. Lecron, S. A. Mahmoudi, M. Benjelloun, S. Mahmoudi, and P. Manneback, “Heterogeneous Computing for Vertebra Detection and Segmentation in X-Ray Images,” *International Journal of Biomedical Imaging. Volume 2011*, pp. 1–12, 2011. 38, 116

[67] D. G. Lowe, “Distinctive image features from scale-invariant keypoints,” *International Journal of Computer Vision (IJCV)*, vol. 60(2), pp. 91–110, 2004. 167

[68] ——, “Programming Massively Parallel Processors. A Hands-on Approach,” *IJCV*, pp. 91–110, 2004. 32, 97

[69] D. G. lowe, “Object recognition from local scale-invariant features,” *Proceedings of the International Conference on Computer Vision, ICCV*, pp. 1150–11 570, 1999. xv, 169, 170

[70] B. Lucas and T. Kanade, “An iterative image registration technique with an application to stereo vision,” *Proceedings of Imaging Understanding Workshop*, pp. 121–130, 1981. 98, 99

[71] B. D. Lucas and T. Kanade, “An iterative image registration technique with an application to stereo vision,” 1981, pp. 674–679. 168

[72] Y. Luo and R. Duraiswani, “Canny Edge Detection on NVIDIA CUDA,” *Proceedings of the Workshop on Computer Vision on GPUS, CVPR*, 2008. 30, 64

[73] S. Mahmoudi, S. Frémal, M. Bagein, and P. Manneback, "Calcul intensif sur gpu : exemples en traitement d'images, en bioinformatique et en télécommunication," in *CIAE 2011 : Colloque d'informatique, automatique et électronique*, 2011, pp. 1–7. 50

[74] S. Mahmoudi and P. Manneback, "Gpu-based real time motion tracking in high definition videos," in *Proceedings of the 3rd Workshop of COST action IC805. Open Network for High-Performance Computing on Complex Environments, Gênes, Italie*, 2012. 90

[75] ——, "Image and video processing on parallel (gpu) and heterogeneous architectures," in *Proceedings of the 2nd Workshop of COST 0805 Open Network for High-Performance Computing on Complex Environments, Timisoarah, Roumanie*, 2012. 74

[76] S. Mahmoudi, S A Haidar, N. Ihaddadene, and C. Djerabe, "Abnormal event detection in real time videos," *First International Workshop on Multimodal Interactions Analysis of Users a Controlled Environment*, 2008. 90, 97

[77] S. A. Mahmoudi, F. Lecron, P. Manneback, B. Mohammed, and M. Saïd, "GPU-Based Segmentation of Cervical Vertebra in X-Ray Images," *Proceeding of the workshop HPCCE. In Conjunction with IEEE Cluster*, pp. 1–8, 2010. 6, 115, 116, 129

[78] ——, "Efficient Exploitation of Heterogeneous Platform for Vertebra Detection in X-Ray Images," *Biomedical Engineering International Conference (BIOMEIC'12) October 10-11,2012, Tlemcen, Algérie*, pp. 1–6, 2012. 116, 143

[79] S. A. Mahmoudi and P. Manneback, "Traitements d'images sur gpu sous cuda et opengl : application à l'imagerie médicale," in *CIGIL09, Calcul intensif et grilles informatiques, USTL, Lille, France*, 2009. 50

[80] ——, "Le traitement d'objets multimédias sur gpu," in *Seconde journée scientifique du pôle hainuyer, Mons, Belgique*, 2010. 50

[81] S. A. Mahmoudi, P. Manneback, F. Lecron, M. Benjelloun, and S. Mahmoudi, "Parallel image processing with cuda and opengl," *1 st Workshop of COST 0805. Open Network for High-Performance Computing on Complex Environments, Lisbon. Portugal*, 2009. 50

[82] S. A. Mahmoudi and P. Manneback, “Efficient Exploitation of Heterogeneous Platforms for Images Features Extraction,” *3rd International Conference on Image Processing Theory, Tools and Applications, IPTA'12. Istanbul, Turquie*, pp. 1–6, 2012. 74, 143

[83] S. Mahmoudi, P. Manneback, C. Augonnet, and S. Thibault, “Détection optimale des coins et contours dans des bases d'images volumineuses sur architectures multicœurs hétérogènes,” in *20ème Rencontres francophones du parallélisme (RenPar'20)*, 2011. 74

[84] S. Mahmoudi, P. Manneback, C. Augonnet, S. Thibault *et al.*, “Traitements d'images sur architectures parallèles et hétérogènes,” *Technique et Science Informatiques, Revue des sciences et technologies de l'information*, 2012. 74

[85] M. Mancas, M. Bagein, N. Guichard, S. Hidot, C. Machy, S. Mahmoudi, and X. Siebert, “Augmented virtual studio,” *QPSR of the numediart research program*, vol. 1, no. 4, 2008. 90

[86] M. Mancas, R. Ben Madhkour, S. Mahmoudi, and T. Ravet, “Virtrack : Tracking for virtual studios : Virtrack : Tracking for virtual studios,” *Quarentlt Progress Scientific Report. numediart project*, vol. 3, 2010. 90

[87] M. Mancas, J. Tilmanne, R. Chessini, S. Hidot, C. Machy, S. Mahmoudi, and T. Ravet, “Matrix : Natural interaction between real and virtual worlds,” *QPSR of the numediart research program. Ed. by Thierry Dutoit and Benoît Macq*, vol. 2, p. 1, 2009. 116

[88] R. Mann and M. S. Langer, “Optical snow and the aperture problem,” in *International Conference on Pattern Recognition, ICPR*, 2002, pp. 264–267. 98, 168

[89] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, “Cg : A system for programming graphics hardware in a C-like language,” *ACM Transactions on Graphics 22*, pp. 896–907, 2003. 29, 30

[90] J. Marzat, Y. Dumortier, and A. Ducrot, “Real-time dense and accurate parallel optical flow using CUDA,” *In Proceedings of WSCG*, pp. 105–111, 2009. 33

[91] M. Midhun, K. C. Neethu, and J. Preetha, "Real-time face tracking with GPU acceleration," *High Performance Computing Group, Network Systems and Technologies (P) Ltd*, 2008. xiii, 34

[92] Y. Mizukami and K. Tadamura, "Optical Flow Computation on Compute Unified Device Architecture," *In Proceedings of the 14th International Conference on Image Analysis and Processing*, pp. 179–184, 2007. 33

[93] F. Mokhtarian and A. Mackworth, "Scale-based description and recognition of planar curves and two-dimensional shapes," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8(1), pp. 34–43, 1986. 162

[94] H. Moravec, "Obstacle avoidance and navigation in the real world by a seeing robot rover," *Technical Report CMU-RI-TR-3, Carnegie-Mellon University, Robotics Institute*, 1980. 162

[95] J.-M. Morel, "Is sift sclae invariant ?" *Inverse Problems and Imaging*, vol. 5(1), 2011. [Online]. Available : http://www.aimSciences.org xv, 171

[96] NVIDIA, "NVIDIA CUDA SDK code samples." [Online]. Available : http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html 63

[97] ——, "CUBLAS," 2007. [Online]. Available : http://developer.download.nvidia.com/compute/cuda/1_0/CUBLAS_Library_1.0.pdf 61

[98] ——, "NVIDIA CUDA," 2007. [Online]. Available : http://www.nvidia.com/cuda. 4, 13, 30

[99] ——, "Quadro sdi capture," 2009. [Online]. Available : http://www.nvidia.com/object/product_quadro_sdi_capture_us.html xv, 123, 124

[100] nVidia Corporation, "nvida cuda programming guide version 3.2. in : Cuda zone," 2010. [Online]. Available : http://www.nvidia.com/object/cuda_develop.html xiii, 3

[101] N. Ohnishi and A. Imiya, "Dominant plane detection from optical flow for robot navigation," *Pattern Recognition Letters*, vol. 27(9), pp. 1009–1021, 2006. 89

[102] OpenCV, "OpenCV computer vision library." [Online]. Available : http://opencv.willowgarage.com/wiki/ 30, 76, 91, 94, 107, 142

[103] OpenGL, "OpenGL Architecture Review Board : ARB vertex program. Revision 45." 2004. [Online]. Available : http://oss.sgi.com/projects/ogl-sample/registry/ 5, 13, 30, 91

[104] L. Parida, D. Geiger, and R. Hummel, "Junctions : Detection, classification, and reconstruction," *IEEE Transactions on Pattern Analysis and Machine Intelligence , 20(7)*, pp. 687–698, 1998. 162

[105] K. Park, S. Nitin, and H. L. Man, "Design and Performance Evaluation of Image Processing Algorithms on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, pp. 1–14, 2011. 36

[106] S. M. Pizer, E. P. Amburn, J. D. Austin, R. Cromartie, A. Geselowitz, T. Greer, B. Ter Haar Romeny, J. B. Zimmerman, and K. Zuiderverld, "Adaptive histogram equalization and its variations," *Computer Vision, Graphics, and Image Processing*, vol. 39, no. 3, pp. 355–368, 1987. 40

[107] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77(2), pp. 257–286, 1989. 169

[108] C. G. Rafael and E. W. Richard, "Digital Image Processing. Prentice-Hall, 2nd edition," *Library of Congress Cataloging-in-Pubblication Data*, 2002. 29

[109] J. M. Ready and C. N. Taylor, "GPU Acceleration of Real-time Feature Based Algorithms," *IEEE Workshop on Motion and Video Computing*, pp. 8–16, 2007. 33

[110] K. Rohr, "Recognizing corners by fitting parametric models," *International Journal of Computer Vision, 9(3)*, pp. 213–230, 1992. 162

[111] E. Rollins, "Real-time ray tracing with nvidia cuda gpgpu and intel quad-core." [Online]. Available : http://ericrollins.home.mindspring.com/ray/cuda.html. 36

[112] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, Salt Lake City, UT, USA*, pp. 20–23, 2008. 35

[113] A. Samuel and S. Leonel, "Exploiting SIMD extensions for linear image processing with OpenCL," *28th IEEE International Conference on Computer Design,* , pp. 425–430, 2010. 30

[114] D. Schaa and D. Kaeli, "Exploring the multiple gpu design space," *In Proc. of the IEEE International Parallel and Distributed Processing Symposium IPDPS*, 2009. 35

[115] T. Schiwietz, T. Chang, P. Speier, and R. Westermann, "MR image reconstruction using the GPU," *Image-Guided Procedures, and Display. Proceedings of the SPIE*, pp. 646–655, 2006. 31

[116] C. Schmid, R. mohr, and C. Bauckhage, "Evaluation of interest point detectors," *International Journal of Coputer Vision (IJCV)*, vol. 37, no. 2, pp. 151–172, 2000. 162

[117] SGI, "Silicon Graphics. OpenGL-the industry's foundation for high performance graphics," 1992. [Online]. Available : http://www.sgi.com/products/software/opengl/ 13

[118] Y. sheng Chen, Y. ping Hung, and C. shann Fuh, "Fast block matching algorithm based on the winner-update strategy," *IEEE Transactions on Image Processing*, vol. 10, pp. 1212–1222, 2001. 171

[119] X. Siebert, S. Dupont, P. Fortemps, and D. Tardieu, "Media Cycle : Browsing and Performing with Sound and Image libraries." *in QPSR of the numediart research program*, pp. 19–22, 2009. xiii, 6, 38, 43, 44

[120] C. Sigg and M. Hadwiger, "Fast third-order texture filtering," *In GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation, Matt Pharr (ed.), Addison-Wesley ; chapter 20*, pp. 313–329, 2005. 12

[121] ——, "Nvidia geforce 8800 gpu architecture overview," *NVIDIA Technical Brief TB-02787-001*, vol. 01, 2006. 10, 12

[122] S. N. Sinha, J.-M. Fram, M. Pollefeys, and Y. Genc, "Gpu-based video feature tracking and matching," *EDGE, Workshop on Edge Computing Using New Commodity Architectures*, 2006. 33

[123] M. Smelyanskiy, D. Holmes, J. Chhugani, A. Larson, and al, “Mapping high-fidelity volume rendering for medical imaging to CPU, GPU and many-core architectures,” *IEEE Transactions on Visualization and Computer Graphics, 15(6)*, pp. 1563–1570, 2009. 31

[124] D. Stéphane, F. Christian, U. Jérôme, M. Sidi, and X. Siebert, *MediaBlender : Interactive Multimedia Segmentation.* {QPSR} of the numediart research program, 2011, vol. 4. 116

[125] N. Sundaram, T. Brox, and K. Keutzer, “Dense point trajectories by gpu-accelerated large displacement optical flow,” 2010. [Online]. Available : http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-104.html 34

[126] D. Tardieu, R. Chessini, J. Dubois, S. Dupont, S. Hidot, B. Mazzarino, A. Moinet, X. Siebert, G. Varni, and A. Visentin, “Video navigation tool : Application to browsing a database of dancers’ performances,” in *QPSR of the numediart research program*, T. Dutoit and B. Macq, Eds., vol. 2, no. 3. numediart Research Program on Digital Art Technologies, 9 2009, pp. 85–90. [Online]. Available : http://www.numediart.org/docs/numediart_2009_s07_p2_report.pdf xiii, 45, 115, 132

[127] M. Teague, “Image analysis via general theory of moments,” *Optical Society of America*, vol. 70, pp. 920–930, 1980. 166

[128] E. Tejedor, M. Farrreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta, “ClusterSs : A Task-Based Programming Model for Clusters,” *The 20th International ACM Symposium on High-Performance Parallel and Distributed Computing, San Jose, California, USA*, 2011. 23

[129] N. U. S. National Library of Medecine, “Nhanes ii x-ray images.” [Online]. Available : http://archive.nlm.nih.gov/proj/ftp/ftp.php 129

[130] G. Vincent, “Estimation de mouvement subpixélique par blocs adaptée ţa la couleur avec modèle de mouvement,” *Rapport de Stage, DEA Image Vision. Universite de Nice - Sophia Antipolis*, 2004. xv, 172, 173, 174

[131] K. W, “Ieee standard 754 for binary floating-point arithmetic,” *Lecture Notes on the Status of IEEE 754. University of California Berkeley, CA 94720-1776*, 1997. 19

[132] B. Welch and K. Jones, in *Practical Programming in Tcl and Tk, 4th ed*, 2003. 14

[133] T. Xiang and S. Gong, “Video behaviour proling and abnormality detection without manual labelling,” *Proceedings IEEE International Conference on Computer Vision (ICCV), Beijing, China*, 2005. 33

[134] Z. Yang, Y. Zhu, and Y. pu, “Parallel Image Processing Based on CUDA,” *International Conference on Computer Science and Software Engineering. China*, pp. 198–201, 2008. 29

[135] W. Zhang, “A preliminary study of OpenCL for accelerating CT reconstruction and image recognition,” *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*, pp. 4059–4063, 2009. 30

[136] E. ÖZKAN, “Background subtraction for real time video frames,” *Master thesis project, Ankara University, Computer Engineering Department. Turkey*, 2012. xiii, 6, 38, 46, 47, 48, 115, 134

Annexe 1 : Extraction de caractéristiques d'images

Nous présentions dans cette section les principales techniques d'extraction de caractéristiques d'images utilisées dans ce travail : détection des points d'intérêt, extraction des contours et calcul des moments de Hu.

1 Détection des points d'intérêts

La détection des points d'intérêt (ou coins) est une étape nécessaire pour de nombreuses applications de traitement d'images et de vidéos. Les coins sont définis par des doubles discontinuités de la fonction d'intensité, de réflectance et de profondeur dans plusieurs directions dans une image. Ces discontinuités peuvent être interprétées par la présence de différents types de points d'intérêt tels que les jonctions en T ou les points de fortes variations de texture. Le point fort des points d'intérêt est leur présence dans la majorité des images contrairement aux contours ou formes. Ces coins offrent une meilleure description des images, ils sont également invariant aux à la rotation, à la translation, au changement d'échelle, etc. Dans la littérature, on trouve trois catégories de méthodes de détection des coins :

1. Détection à base des contours ;
2. Détection à base de la fonction d'intensité ;
3. Détection à base de modèles.

1.1 Détection à base de contours

Ce type de méthodes applique une détection des contours avant d'extraire les points d'intérêt. Ceci s'effectue par la recherche des points de courbure maximale le long des contours. Dans certains cas, une approximation polygonale [36] est appliquée avant d'extraire les points d'intersection. Dans ce contexte, Asada *et al.* [7] ont proposé une approche de détection des coins d'objets 2D à partir de courbes planes. Les changements de courbures sont considérés comme points d'intérêt. Mokhtarian *et al.* [93] ont développé une approche similaire utilisant les points d'inflexion d'une courbe plane. Les auteurs dans [51] ont proposé d'extraire des segments à partir des contours, l'intersection de segments présente les points d'intérêt.

1.2 Détection à base de la fonction d'intensité

Ce type de méthodes s'appuie sur le calcul de la fonction d'intensité présentée par la variation du niveau de gris entre les pixels de l'image. Dans ce contexte, Harris et Stephens [49] ont développé un détecteur de coins connu par sa robustesse grâce à son invariance à la rotation, à la translation, au changement d'échelle et aux bruits de l'image. Bouguet [17] a proposé une implémentation simplifiée du filtre de Harris suivant plusieurs étapes. Un prédécesseur du détecteur de Harris a été décrit par Moravec [94], où la discontinuité se réfère au déplacement des plaques (ou régions).

1.3 Détection à base de modèles

Ce type de méthodes s'appuie sur un modèle paramétrique prédéfini. Rohr *et al.* [110] ont proposé un modèle de jonction convolué par un filtre gaussien. Les paramètres de ce modèle (angle entre les coins de la lettre L et l'axe x, les niveaux de gris, etc.) sont ajustés avec l'image afin d'extraire les points d'intérêt. Cette méthode a été améliorée par Deriche et Blaszka dans [35] appliquant un lissage afin de mieux éliminer les bruits de l'image. Ils offrent également une convergence plus rapide grâce à l'utilisation de régions d'image plus grandes. Les auteurs dans [104] ont présenté une approche de détection des jonctions à partir d'une description optimale du signal de l'image.

Les méthodes de détection à base de la fonction d'intensité sont les plus utilisées dans le domaine pour la détection des points d'intérêt, puisqu'ils ne requièrent pas d'informations au préalable telles que les contours, types de coins, etc. On trouve dans [116] une

comparaison des techniques citées ci-avant, constatant que le détecteur de Harris offre les meilleures performances. Ce dernier présente également une solution efficace et robuste pour détecter les points à suivre dans une séquence vidéo.

Par conséquent, nous proposons de décrire cette méthode de façon plus détaillée. La première étape de ce détecteur consiste au calcul des dérivées spatiales, horizontale I_x et verticale I_y, à partir des équations (1) et (2). Ces dernières permettent de calculer la matrice M décrite dans l'équation (6) obtenue à son tour à partir des équations (3), (4) et (5).

$$I_x(u,v) = \frac{\delta I}{\delta x}(u,v) \approx \frac{f(u+1) - f(u-1)}{2} \tag{1}$$

$$I_y(u,v) = \frac{\delta I}{\delta y}(u,v) \approx \frac{f(u+1) - f(u-1)}{2} \tag{2}$$

$$A = I_x^2 * \omega \tag{3}$$

$$B = I_y^2 * \omega \tag{4}$$

$$C = (I_x I_y) * \omega \tag{5}$$

$$M = \begin{bmatrix} A & C \\ C & B \end{bmatrix} \tag{6}$$

Tel que ω présente l'opérateur de lissage gaussien. La dernière étape est consacrée au calcul de la réponse de l'opérateur de Harris tel que montré dans l'équation (7).

$$R = Det(M) - K(Trace(M))^2 \tag{7}$$

Notons que R présente des valeurs positives pour les coins, de valeurs négatives pour les contours et de petites valeurs dans les régions planes. La valeur de k est fixée entre 0.04 et 0.06. Par ailleurs, cette étape peut également être résolue par l'analyse des valeurs propres de la matrice M

2 Détection des contours

Les contours correspondent aux points de l'image présentant un changement brutal de l'intensité lumineuse. En général, si la luminosité d'un pixel présente une différente significative par rapport aux pixels voisins, on peut constater la présence d'une arrête (ou contour). Afin de détecter ces changements, le gradient local ∇ de l'image I est généralement appliqué, qui représente la base de nombreux opérateurs de détection des contours. Le gradient ∇I (équation 8) est calculé à partir des dérivées spatiales, du premier ordre, de l'image I le long des deux axes (u, v).

$$\nabla I(u,v) = \begin{bmatrix} \frac{\delta I}{\delta u}(u,v) \\ \frac{\delta I}{\delta v}(u,v) \end{bmatrix} \tag{8}$$

La magnitude du gradient ∇I est calculée à partir de l'équation (9).

$$|\nabla I(u,v)| = \sqrt{\left(\frac{\delta I}{\delta u}(u,v)\right)^2 + \left(\frac{\delta I}{\delta v}(u,v)\right)^2} \tag{9}$$

Par ailleurs, il existe également des opérateurs permettant l'approximation du gradient tels que ceux proposés par Sobel et Prewitt. Ces derniers utilisent des filtres linéaires pour calculer les gradients dans chaque direction (x et y). Les équations (10) et (11) présentent les filtres proposés par Sobel et Prewitt respectivement.

$$H_x^S = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad H_y^S = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{10}$$

$$H_x^P = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \qquad H_y^P = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{11}$$

L'application de ces deux filtres pour une image d'entrée permet d'obtenir les gradients G_x et G_y tel que montré dans les équations (12) et (13).

$$G_x(u,v) = H_x * I \tag{12}$$

$$G_y(u,v) = H_y * I \tag{13}$$

À partir de ces gradients, nous pouvons calculer la magnitude du gradient (équation 14) ainsi que sa direction (15).

$$|G(u,v)| = \sqrt{(G_x(u,v))^2 + (G_y(u,v))^2} \tag{14}$$

$$\Phi(u,v) = tan^{-1}\left(\frac{G_y(u,v)}{G_x(u,v)}\right) \tag{15}$$

Afin d'améliorer la détection des contours, plusieurs algorithmes étaient proposés. Le détecteur de Canny présente l'un des algorithmes les plus populaires grâce à son efficacité due à son faible taux d'erreur (faux contours) et sa bonne localisation des bords [18]. Le filtre de Canny consiste en trois étapes : lissage, calcul des gradients et localisation des bords. Lors de l'étape du lissage, l'algorithme de Canny applique un filtre passe-bas gaussien afin d'éliminer les bruits de l'image. Ensuite, le gradient spatial est calculé pour chaque pixel de l'image lissée. Ce calcul peut s'effectuer à partir des opérateurs de Sobel ou de Prewitt, ou simplement à partir du calcul des dérivées spatiales du premier ordre (équations (1) et (2)) le long des axes horizontaux et verticaux. L'étape de localisation des bords est composée à son tour de deux phases : la suppression des non-maxima locaux et le seuillage, offrant une élimination efficace des faux contours.

3 Calcul des moments de Hu

Les moments de Hu fournissent des attributs de caractérisation de formes très puissants. Un moment est décrit par une somme pondérée des pixels en fonction de leur position dans l'image. En 1962, Hu [53] a proposé un ensemble de sept moments invariants aux translations, rotations et changements d'échelle. Ils permettent de décrire la forme à l'aide de propriétés statistiques ce qui les rend d'une grande utilité pour la description des formes en

vue d'une classification ou d'une indexation. Ces moments sont décrits dans l'équation (16).

$$\begin{aligned}
\Phi_1 &= \mu_{20} + \mu_{02} \\
\Phi_2 &= (\mu_{20} - \mu_{02})^2 + 4\mu_{11}^2 \\
\Phi_3 &= (\mu_{30} - 3\mu_{12})^2 + (3\mu_{21} - \mu_{03})^2 \\
\Phi_4 &= (\mu_{30} + \mu_{12})^2 + (\mu_{21} + \mu_{03})^2 \\
\Phi_5 &= (\mu_{30} - 3\mu_{12})(\mu_{30} + \mu_{12})[(\mu_{30} + \mu_{12})^2 - 3(\mu_{21} + \mu_{03})^2] + (3\mu_{21} - \mu_{03}) \\
&\quad (\mu_{21} + \mu_{03})[(3\mu_{30} + \mu_{12})^2 - (\mu_{21} + \mu_{03})^2] \\
\Phi_6 &= (\mu_{20} - \mu_{02})[(\mu_{30} + \mu_{12})^2 - (\mu_{21} + \mu_{03})^2] + 4\mu_{11}(\mu_{30} - 3\mu_{12})(\mu_{21} + \mu_{03}) \\
\Phi_7 &= (3\mu_{21} - \mu_{03})(\mu_{30} + \mu_{12})[(\mu_{30} + \mu_{12})^2 - 3(\mu_{21} + \mu_{03})^2] - (\mu_{30} - 3\mu_{12}) \\
&\quad (\mu_{21} + \mu_{30})[3(\mu_{30} + \mu_{12})^2 - (\mu_{21} + \mu_{03})^2]
\end{aligned} \tag{16}$$

La particularité des moments de Hu est leur invariance aux transformations géométriques. Ces moments sont décrites en détail dans [53] et [127].

Annexe 2 : Méthodes de suivi de mouvements

Les méthodes de suivi de mouvements consistent à estimer les déplacements (vitesse et direction) de points ou régions spécifiques d'une trame (image) de la vidéo par rapport à la trame précédente. Ils présentent un outil important pour de nombreuses applications telles que l'analyse de comportements humains, la détection d'évènements, l'indexation de séquences vidéo, etc. Ces méthodes s'appuient sur différentes techniques telles que les mesures du flot optique [52], les descripteurs SIFT [67] ainsi que la mise en correspondance d'objets (block matching) [14].

4 Mesure du flot optique

Dans ce travail, nous nous sommes focalisés sur les techniques de calcul du flot optique puisqu'elles offrent des informations pertinentes, permettant de suivre différents types d'objets (personnes, véhicules, etc.) dans différentes situations (présence de bruits, petits mouvements, etc.). Le flot optique représente le champ de vitesse apparent observé entre deux images successives dans une scène. Pour chaque pixel de coordonnées (x, y), le flot optique détermine sa position à l'instant suivant. Les méthodes de calcul du flot optique sont basées sur l'hypothèse d'éclairage constant [52]. Dans cette hypothèse, on suppose que les mesures d'image (par exemple la luminosité) dans une petite région restent les mêmes, malgré que leur situation peut changer. Cette hypothèse peut être représentée par l'équation (1) :

$$I(x + u, y + v, t + 1) = I(x, y, t). \quad (1)$$

telque :
- I(x,y,t) : niveau de gris du pixel (x,y) à l'instant t ;
- I(x+u,y+v,t+1) : niveau de gris du pixel (x+u,y+v) à l'instant t+1 ;
- u,v : déplacement selon les axes horizontaux et verticaux.

À partir de cette hypothèse, on aboutit à l'équation de contraintes suivante :

$$\frac{dI}{dt}(x, y, t) = 0 \tag{2}$$

$$\Longleftrightarrow \frac{\delta I}{\delta x}\frac{\delta x}{\delta t} + \frac{\delta I}{\delta y}\frac{\delta y}{\delta t} + \frac{\delta I}{\delta t} = 0 \tag{3}$$

$$\Longleftrightarrow Ix.u + Iy.v + It = 0 \tag{4}$$

L'équation (4) présente la contrainte à respecter pour le mouvement. Toutefois, nous ne disposons que d'une unique équation pour déterminer deux inconnues u et v (les deux compsantes du vecteur de mouvement au point considéré). Ce phénomène est connu sous le nom de « *aperture problem* » [88]. Afin de déterminer ces deux inconnues, toutes les méthodes de calcul du flot optique effectuent une ou plusieurs hypothèses supplémentaires par rapport à la nature du champ de mouvement, permettant ainsi d'obtenir des contraintes additionnelles.

Dans ce contexte, Horn et Schunck [52] ont introduit une contrainte globale de lissage, permettant d'estimer le flot optique dans toute l'image. Leur but est de minimiser les distorsions dans le flot optique, ils préfèrent ainsi les solutions qui présentent plus de lissage. En effet, la méthode proposée par Horn et Schunck suppose que les pixels voisins doivent avoir une vitesse de mouvement similaire, ce qui veut dire que le flot optique présente une variation progressive. Cependant, cette approche est entravée par sa faible efficacité dans le cas de petits mouvements.

Lucas et Kanade [71] ont développé une méthode locale d'estimation du flot optique supposant que ce dernier soit constant dans un voisinage local. Ceci permet de résoudre l'équation de flot optique (4) pour tous les pixels dans le voisinage considéré. La méthode proposée par Lucas et Kanade est également connue par sa robustesse aux bruits par rapport aux autres techniques. Toutefois, cette approche peut présenter des erreurs (en termes des vecteurs de flot optique) lors du traitement de régions uniformes.

L'algorithme proposé par Lucas et Kanade est le plus utilisé pour l'estimation du flot optique, puisqu'il présente une approche locale offrant des résultats plus précis dans de nombreuses de situations. Il offre également une meilleure robustesse aux bruits. Par conséquent, nous avons exploité la méthode Lucas-Kanade dans ce travail.

Dans la littérature, on trouve plusieurs techniques basées sur les mesures du flot optique, et qui sont appliquées sur différents problèmes. Les auteurs dans [56] ont proposé une méthode de suivi de véhicules à partir du flot optique, l'objectif est d'estimer les vitesses de véhicules automatiquement dans une séquence vidéo capturée par une caméra fixe. L'approche de Horn et Schunck [52] est exploitée pour effectuer cette estimation. Andrade *et al.* [3] ont développé une méthode d'analyse de comportements normaux afin de détecter les évènements anormaux. Cette solution combine les techniques de calcul du flot optique, les modèle de Markov cachés [107] ainsi que l'analyse en composante principale afin de détecter les scénarios d'urgence dans la foule. Les auteurs dans [61] ont proposé une méthode de suivi de visages basée sur trois étapes : suivi, apprentissage et détection. Par ailleurs, [29] ont présenté des techniques de suivi de mouvements à partir de caméras mobiles. On trouve également des travaux [55] de détection des évènements anormaux dans des vidéos de foule à partir de l'analyse des mouvements au lieu de suivre les objets un par un.

Ces travaux présentent seulement une partie des applications possibles pouvant exploiter les mesures du flot optique. En effet, on peut constater un fort besoin et une grande utilité de ce type d'implémentations dans le domaine de vision par ordinateur. Ceci explique notre intérêt aux techniques de calcul du flot optique, et plus particulièrement notre implémentation Multi-GPU offrant un suivi de mouvements en temps réel dans des vidéos Full HD ou 4K.

5 Descipteurs SIFT

Les descripteurs SIFT, que l'on peut traduire en français par « transformation de caractéristiques visuelles invariante à l'échelle », présentent une technique très utilisée dans le domaine de vision par ordinateur. Ces descripteurs, développés par David Lowe [69] en 1999, permettent de détecter et d'identifier les objets similaires entre images différentes.

Les descripteurs SIFT sont calculés à partir d'informations numériques issues de l'analyse locale d'une image. Cette analyse permet de caractériser le contenu visuel de l'image de façon indépendante par rapport à l'échelle, la luminosité et la position de la caméra. Par exemple, la distance entre deux roues différentes doit être plus grande que la distance entre deux roues identiques, vues sous des angles différents. La robustesse de ces descripteurs SIFT les rend très utilisés dans les applications de suivi et de reconnaissance de mouvements.

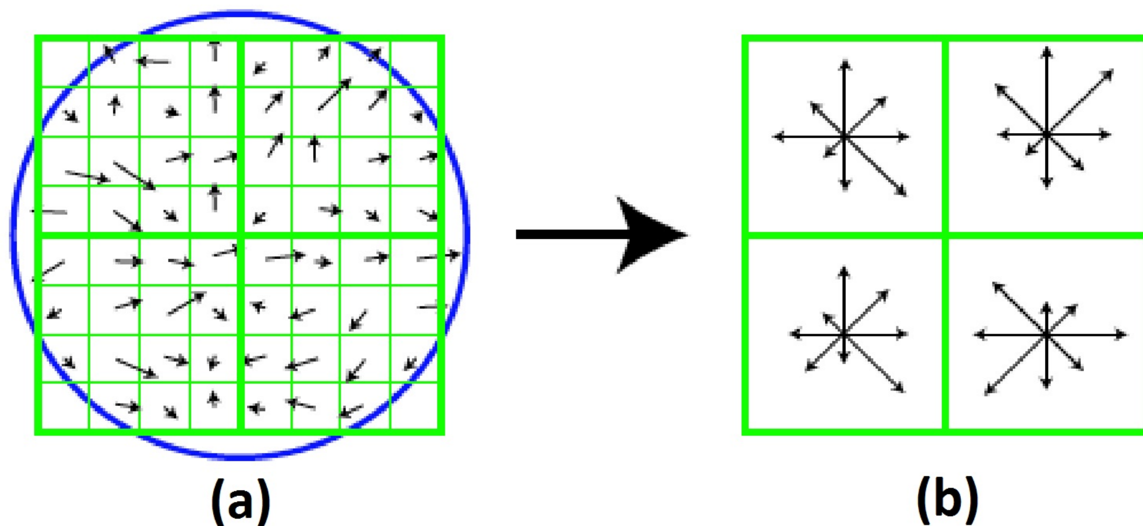


Figure A.1 – (a). Gradients de l'image. (b). Descripteurs de points clés [69]

Les descripteurs SIFT s'appuient sur trois paramètres principaux : les gradients, les poids et l'histogramme.

1. **Les gradients :** la première étape de SIFT consiste à calculer l'amplitude et l'orientation des gradients de pixel de l'image (flèches dans la figure A.1.(a)).
2. **Les poids :** une fonction de poids gaussienne est ensuite appliquée à l'amplitude des gradients afin de réduire les petits changements dans la position de la fenêtre (cercle dans la figure A.1.(a))
3. **Histogramme :** La région est découpée en une grille de 4*4 cellules. Sur chaque cellule et pour chaque orientation possible, l'histogramme contient la somme des amplitudes des gradients (figure A.1.(b)). Le descripteur présente ainsi la concaténation des histogrammes de chacune des cellules.

La figure A.2 montre un exemple d'utilisation des descripteurs SIFT pour la reconnaissance d'objets dans deux images avec deux positions différentes.

Figure A.2 – Correspondance d'objets avec les descripteurs SIFT [95]

6 Mise en correspondance d'objets (block matching)

Les auteurs dans [65, 118] ont proposé , à leur tour, un algorithme de mise en correspondance entre blocs afin d'estimer les mouvements dans une séquence vidéo. Le but de cette méthode consiste à détecter le mouvement apparu entre deux images successives dans une région (ou bloc). Chaque bloc de la trame courante est mis en correspondance avec un bloc de la trame suivante (bloc sélectionné parmi un ensemble de régions voisines). Cette mise en correspondance est basée sur le calcul des distances entre les valeurs des niveaux de gris de deux blocs successives. La solution offrant une somme totale (des distances) minimale présente alors la meilleure mise en correspondance. Leur principal avantage est la simplicité de leur mise en œuvre. Nous décrivons ici le principe de son fonctionnement.

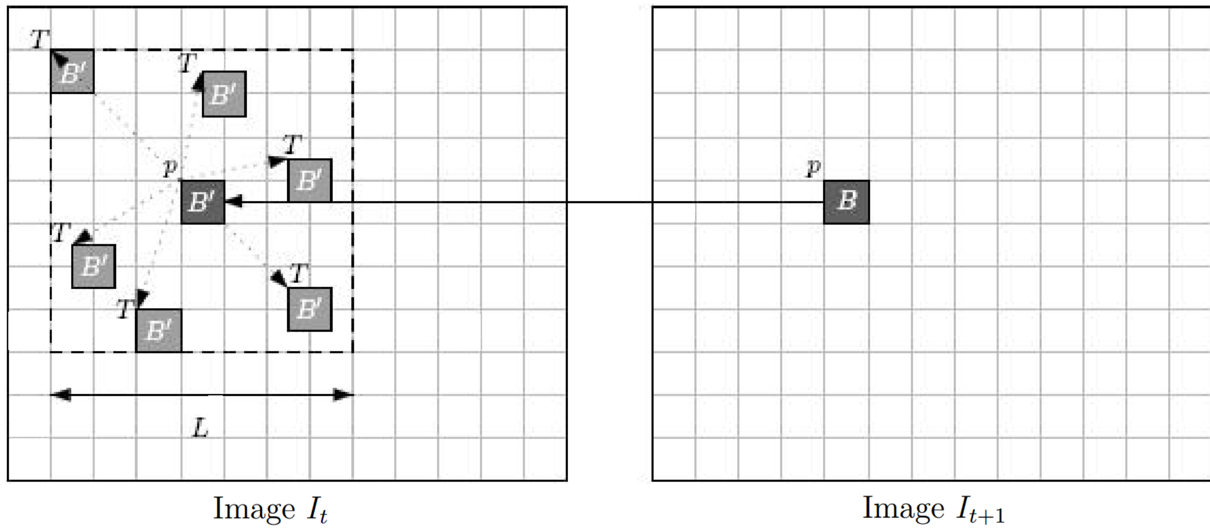


Figure A.3 – Estimation par blocs du vecteur de mouvement entre deux images successives [130]

6.1 Principe de mise en correspondance d'objets

Considérons deux images successives de niveau de gris I_1 et I_2 respectivement aux instants t_1 et t_2, d'une même séquence vidéo. Chaque image est représentée par un tableau de pixels à deux dimensions. L'estimation du mouvement s'effectue par blocs de pixels. En effet, pour chaque bloc de l'image I_2, la méthode consiste à trouver le bloc de l'image I_1 qui lui correspond le mieux. La mesure de correspondance est réalisée en calculant l'écart entre les niveaux de gris des deux blocs. Le bloc correspondant est celui qui minimise cet écart au sens d'une norme à choisir. Cette technique d'estimation du mouvement est illustrée à la figure A.3

6.1.1 Critères de similtitude

Différents critères peuvent être retenus pour déterminer si deux blocs sont similaires ou non. Nous pouvons citer principalement deux techniques :

1. La somme des différences au carré (Sum of Squared Difference (SSD)) : pour deux blocs B et B' ($B \in I_1$, $B' \in I_2$), de taille (N×N), SSD vaut :

$$SSD = \sum_{i=1}^{N} \sum_{j=1}^{N} (B(_{i,j}) - B'(_{i,j}))^2 \tag{5}$$

Pour un bloc B donné, le bloc B' le plus similaire est celui qui minimise SSD. Ainsi, tous les pixels de chaque bloc possèdent le même vecteur de mouvement.

2. Un autre critère tout aussi valable peut être utilisé, à savoir la somme en valeur absolue des différences (Sum of Absolute Difference (SAD)) :

$$SAD = \sum_{i=1}^{N} \sum_{j=1}^{N} |(B(_{i,j}) - B'(_{i,j})|^2 \quad (6)$$

6.1.2 Fenêtre de recherche

Pour estimer la translation associée à chaque bloc, il est nécessaire de fixer la taille L de la fenêtre de recherche. Dans la plupart des cas, aucune information a priori n'est disponible pour estimer le mouvement. La fenêtre de recherche est donc souvent centrée sur le bloc dans I_2 tandis que la taille L est choisie arbitrairement en tenant compte de l'amplitude maximale des mouvements attendus dans la scène et des limites imposées par les dimension de l'image elle-même.

6.1.3 Méthodes de recherche

Plusieurs méthodes de test au sein de la fenêtre peuvent être envisagées dans les algorithmes de mise en correspondance par bloc. Parmi ceux-ci, citons-en deux assez connues : la recherche exhaustive et la recherche en 3 pas [130]. La recherche exhaustive consiste, comme son nom l'indique, à effectuer une recherche de correspondance sur l'intégralité de la fenêtre de recherche définie. Cet algorithme est le plus est connu par sa simplicité et précision, mais il est aussi moins rapide puisqu'il effectue un calcul très intensif. De ce fait, plusieurs autres méthodes dites rapides ont vu le jour et, parmi celles-ci, la « recherche en 3 pas » a été un précurseur dans le domaine. Ainsi, plutôt que de tester tous les blocs, l'algorithme en 3 pas parcourt la fenêtre de recherche en se rapprochant pas à pas du minimum global. La figure A.4 illustre les trois étapes effectuées par l'algorithme pour trouver la meilleure correspondance entre deux blocs. Dans un premier temps, le bloc centré ainsi que les huit blocs situés à une distance de quatre pixels du pixel central sont testés. Le bloc qui minimise le critère est conservé et sera pris comme étant la nouvelle origine. La deuxième étape consiste à réaliser la même opération mais en considérant que les huit blocs situés à une distance de deux pixels. Notons que le bloc situé à la nouvelle origine n'est pas testé car la valeur du critère a déjà été calculée sur ce bloc. Enfin, l'algorithme réitère le même processus lors de la troisième étape mais avec un déplacement d'un seul pixel.

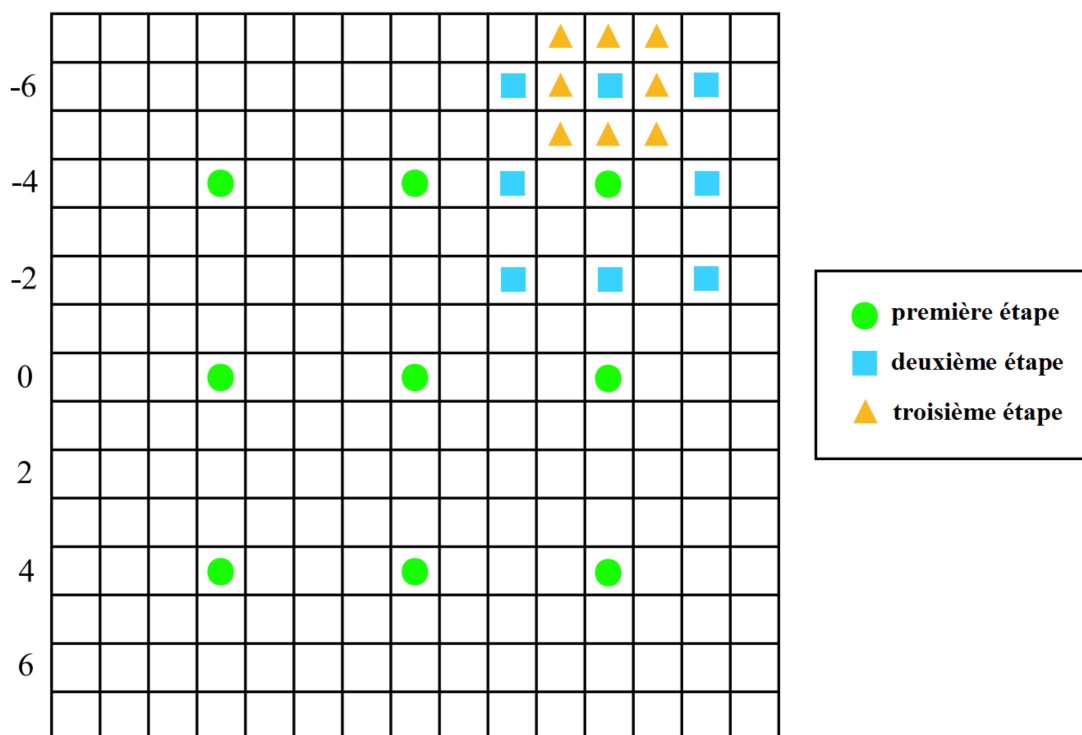


Figure A.4 – Algorithme de recherche en 3 pas [130]

Bien que légèrement moins précis que la recherche exhaustive, l'avantage de cet algorithme est qu'il trouve une correspondance en effectuant un nombre de tests considérablement réduit.

Annexe 3 : publications

7 Revues internationales

F. Lecron, **S. A. Mahmoudi**, M. Benjelloun, S. Mahmoudi and P. Manneback "Heterogeneous Computing for Vertebra Detection and Segmentation in X-Ray Images", *International Journal of Biomedical Imaging.* Accepté : Juin 2011.

S. A. Mahmoudi, P. Manneback, C. Augonnet, S. Thibault "Traitements d'Images sur Architectures Parallèles et Hétérogènes", *Technique et science informatiques, TSI.* Volume 31 No : 8-9-10/2012. Publié en Décembre 2012.

En soumission :

P. D. Possa, **S. A. Mahmoudi**, N. Harb, C. Valderrama "A Challenging/Adaptable FPGA-Based Architecture for Feature Detection", *IEEE Transactions on Computers*, Soumis le 30/10/2012.

S. A. Mahmoudi, M. Kierzynka, P. Manneback, K. Kurowski "Real-time motion tracking using optical flow on multiple GPUs", *Bulletin of the Polish Academy of Sciences.* Soumis le 14/12/2012.

8 Chapitre de livre (en soumission) :

S. A. Mahmoudi, Erencan Ozkan, Pierre Manneback, Souleyman Tosun "Efficient exploitation of Multi-CPU/Multi-GPU platforms for high definition image and video processing", *Wiley complex HPC book, Parallel and Distributed Computing.* Soumis le 05/08/2012.

9 Conférences internationales

S. A. Mahmoudi, H.Sharif, N.Ihaddadene, C.Djeraba, "Abnormal Event Detection in Real Time Video", *First International Workshop on Multimodal Interactions Analysis of Users in a Controlled Environment.* Crete, Greece. 2008.

S. A. Mahmoudi, F. Lecron, P. Manneback, M. Benjelloun, S. Mahmoudi, "GPU-Based Segmentation of Cervical Vertebra in X-Ray Images", *Workshop HPCCE. IEEE International Conference on Cluster Computing*, Crete, Greece. Septembre 2010.

S. A. Mahmoudi, S. Frémal, M. Bagein, P. Manneback, "Calcul intensif sur GPU : exemples en traitement d'images, en bioinformatique et en télécommunication", *CIAE 2011 : Colloque d'Informatique, Automatique et Electronique*, Casablanca, Maroc. Mars 2011.

S. A. Mahmoudi, P. Manneback, C. Augonnet, S. Thibault "Détection optimale des coins et contours dans des bases d'images volumineuses sur architectures multicœurs hétérogènes", *20ème Rencontres Francophones du Parallélisme, RenPar'20*, Saint-Malo, France. Mai 2011.

P. D. Possa, **S. A. Mahmoudi**, N. Harb, C. Valderrama "A New Self-Adapting Architecture for Feature Detection", *FPL 2012 : 22 nd International Conference on Field Programmable Logic and Applications*, Oslo, Novège. Août 2012.

S. A. Mahmoudi, F. Lecron, P. Manneback, M. Benjelloun, S. Mahmoudi, "Efficient Exploitation of Heterogeneous Platforms for Vertebra Detection in X-Ray Images", *Biomedical Engineering International Conference (Biomeic'12).* Tlemcen, Algérie. Octobre 2012.

S. A. Mahmoudi, P. Manneback, "Efficient Exploitation of Heterogeneous Platforms for Images Features Extraction", *IPTA 2012 : 3rd International Conference on Image Processing Theory, Tools and Applications*, Istanbul, Turquie. Octobre 2012.

10 Workshops nationaux et internationaux

S. A. Mahmoudi, P. Manneback, "Parallel Image Processing with CUDA and OpenGL", *Network for High-Performance Computing on Complex Environments.* Lisbon, Portugal. COST ACTION IC 805, WG Meeting. Octobre 2009.

S. A. Mahmoudi, P. Manneback, "Traitements d'images sur GPU sous CUDA et OpenGL : application à l'imagerie médicale", *Journées CIGIL : Calcul Intensif et Grilles Informatiques à Lille.* Lille, France. Décembre 2009.

S. A. Mahmoudi, Pierre Manneback, "Le traitement d'objets multimédias sur GPU", *Seconde journée scientifique du pôle hainuyer.* Mons, Belgique, Mai 2010. Communication orale.

S. A. Mahmoudi, P. Manneback, " Multimedia Processing on Parallel Heterogeneous Architectures (Multi-CPU/Multi-GPU)", *Groupe de Contact FNRS "Calcul Intensif.* Liège, Belgique. Avril 2011.

S. A. Mahmoudi, P. Manneback, "Image and video processing on parallel (GPU) and heterogeneous architectures", *2nd Workshop of COST 0805. Open Network for High-Performance Computing on Complex Environments.* Timisoara, Roumanie. Janvier 2012.

S. A. Mahmoudi, P. Manneback, " GPU-based real time tracking in high definition videos", *3rd Workshop of COST 0805. Open Network for High-Performance Computing on Complex Environments.* Gênes, Italie. Avril 2012.

11 Publications nationales

M. Mancas, M. Bagein, N. Guichard, S. Hidot, C. Machy, **S. A. Mahmoudi**, X. Siebert, "AVS : Augmented Virtual Studio", *QPSR of the numediart research program, Vol. 1, No. 4*, Decembre 2008.

M. Mancas, J.Tilmanne, R.Chessini, S.Hidot, C.Machy, **S. A. Mahmoudi**, T.Ravet, "MATRIX : Natural Interaction Between Real and Virtual Worlds", *QPRS of the numediart research program, Vol. 1, No. 5*, Janvier 2009.

M. Mancas, R. B. Madkhour, **S. A. Mahmoudi**, T. Ravet, "VirŋTrack : Trackŋing for Virŋtual Studios", *QPSR of the numediart research program, volume 3, No 1, pp. 1-4*, Mars 2010.

S. Dupont, C. Frisson, **S. A. Mahmoudi**, X. Siebert, J. Urbain, T. Ravet, "MediaBlender : Interactive Multimedia Segmentation and Annotation", *QPSR of the numediart research program, volume 3*, Mars 2011.

12 Communications orales

S. A. Mahmoudi, P. Manneback, "Parallel Image Processing with CUDA and OpenGL", *Network for High-Performance Computing on Complex Environments*. Lisbon, Portugal. COST ACTION IC 805, WG Meeting. Octobre 2009.

S. A. Mahmoudi, P. Manneback, "Traitements d'images sur GPU sous CUDA et OpenGL : application à l'imagerie médicale", *Journées CIGIL : Calcul Intensif et Grilles Informatiques à Lille*. Lille, France. Décembre 2009.

S. A. Mahmoudi, Pierre Manneback, "Le traitement d'objets multimédias sur gpu", *Seconde journée scientifique du pôle hainuyer*. Mons, Belgique, Mai 2010.

13 Posters

S. A. Mahmoudi, "Calcul Intensif GPGPU", *Matinée des chercheurs, MDC2009*, Mons, Belgique. Mars 2009. Poster

S. A. Mahmoudi, "Traitement Multimédia sur GPU", *Nouveaux outils mathématiques pour le traitement d'images et la vision par ordinateur*. Centre CNRS, Toulouse, France. juin 2009. Poster.

S. A. Mahmoudi, Pierre Manneback, "Calcul intensif sur gpu : application au traitement d'objets multimédia", *Grascomp Day, Graduate School in Computing Science of the Belgium's French Community*. Ulg, Liège. Belgique. Mars 2010.Poster.

S. A. Mahmoudi, "Traitement d'objets multimédia sur architectures hétérogènes", *Matinée des chercheurs, MDC2011*, Mons, Belgique. Mars 2011. Poster.

S. A. Mahmoudi, P. Manneback, "Traitement d'images et de vidéos sur Architectures Hétérogènes", *Ecole d'été Francophone de Traitement d'Image sur GPU*, GIPSA-lab, Départment d'Images Signal. Grenoble, France. Juillet 2011. Poster.

S. A. Mahmoudi, P. Manneback, "Traitement d'images et de vidéos sur Architectures Hétérogènes", *Ecole d'été Francophone de Traitement d'Image sur GPU*, INRIA. Grenoble,France. Juin 2011. Présentation orale (invité).

S. A. Mahmoudi, P. Manneback, "For a Better Exploitation of Heterogeneous Architectures (Multi-CPU/Multi-GPU) in Multimedia Processing Algorithms", *Grascomp's Day*, ULB, Brussels, Belgique. Novembre 2011. Poster.

14 Visites scientifiques et Collaboration

Séjour de 3 semaines (10 au 28 mai 2010) au laboratoire LABRI (INRIA. Bordeaux. France), "Multimedia Processing on Mixed Platforms (CPU-GPU)". Dans le cadre de notre collaboration sous le projet européen COST IC 805 (Short Mission).

Séjour de 17 jours du 4 au 21 Décembre 2011, au laboratoire PSNC (Poznan Supercomputing and Networking Center. Poznan. Pologne, "GPU-Based Real Time Motion Tracking on High Definition (HD) videos". Dans le cadre de notre collaboration sous le projet européen COST IC 805 (Short Mission).

Collaboration avec l'université d'Ankara en Turquie *via* l'action européenne IC805. Accueil du Chercheur Erencan Ozkan pour une durée de deux semain dans notre laboratoire entre le 15 et 30 Juin 2012.