# Détection optimale des coins et contours dans des bases d'images volumineuses sur architectures multicœurs hétérogènes

Sidi Ahmed Mahmoudi, Pierre Manneback, Cédric Augonnet, Samuel Thibault

**RenPar'20** / SympA'14 / CFSE 8
Saint-Malo, France, du 10 au 13 mai 2011

# Détection optimale des coins et contours dans des bases d'images volumineuses sur architectures multicoeurs hétérogènes

Sidi Ahmed Mahmoudi[1], Pierre Manneback[1], Cédric Augonnet[2], Samuel Thibault[2]

[1]Université de Mons, UMONS. Faculté Polytechnique, 20, Place du Parc. Mons, Belgique
[2]Equipe RUNTIME, INRIA Bordeaux, LaBRI. 351, cours de la Libération, 33405 Talence, France
{Sidi.Mahmoudi, Pierre.Manneback}@umons.ac.be, {cedric.augonnet, samuel.thibault}@labri.fr

**Résumé**
Les algorithmes de traitement d'images, et en particulier les méthodes d'extraction des caractéristiques des médias deviennent très consommatrices en mémoire et en temps de calcul lors de l'utilisation de grandes bases d'images volumineuses. L'exploitation d'unités de calcul des processeurs graphiques (GPU) a permis de contourner ce problème. Cependant, cette solution n'exploite pas les coeurs CPU multiples dont disposent la majorité des ordinateurs. De plus, cette approche est sérieusement entravée par les coûts de transfert de données entre les mémoires CPU et GPU. Pour faire face à ces contraintes, nous proposons une implémentation hybride et efficace de méthodes de détection des coins et contours, basée essentiellement sur l'exploitation de l'intégralité des ressources de calcul hétérogènes (Multi-CPU/GPU). Cette implémentation permet aussi de concevoir des stratégies d'ordonnancement portables et efficaces, avec une meilleure gestion des données. Des résultats expérimentaux ont été obtenus en utilisant des ensembles d'images (images médicales, images HD), montrant une accélération allant d'un facteur de 6 à 20 par rapport à une implémentation séquentielle sur CPU.

**Mots-clés :** Multicoeur hétérogène, GPU, traitement d'images, détection des coins, détection des contours

## 1. Introduction

Ces dernières années, la fréquence des processeurs centraux (CPU) s'est retrouvée plafonnée, pour des raisons thermiques, à environ 4 GHz. Cette limitation a été contournée par un changement des architectures internes des processeurs. Le gain de performance ne passe plus par une augmentation des fréquences, mais par la multiplication des unités de calcul intégrées dans les processeurs. Ce phénomène se retrouve aussi bien au niveau des processeurs généralistes que des processeurs graphiques, ainsi que dans les tout récents processeurs accélérés (APU), combinant CPU et GPU sur la même puce [2]. Les machines présentent aujourd'hui une architecture complexe associant des processeurs centraux multicoeurs, ainsi que des processeurs graphiques de plus en plus aptes à exécuter des calculs généralistes.
Les processeurs graphiques qui équipent aujourd'hui la majorité des ordinateurs personnels, disposent d'un nombre impressionnant de coeurs. Leur puissance brute a dépassé largement celle des CPU. Si les GPU ont comme premier objectif le rendu d'images 2D/3D et les jeux vidéos, de nombreux chercheurs ont entrepris d'exploiter leur puissance pour réaliser d'autres calculs, habituellement destinés aux CPU.
Les algorithmes de visualisation et de traitement d'images sont à la fois des gros consommateurs de puissance de calcul et de mémoire. L'arrivée du GPU a permis d'implémenter ces algorithmes de manière parallèle sur un grand nombre de coeurs. Ceci est très performant lors du traitement d'image unique, de telle sorte que le résultat du traitement est directement visualisé à partir du GPU, à l'aide de libraires graphiques permettant le rendu d'images 2D/3D telles qu'OpenGL. Cela permet de réduire les coûts de transfert des données entre la mémoire du CPU et celle du GPU. Toutefois, les méthodes appliquées à des ensembles d'images possèdent deux contraintes supplémentaires. La première est l'impossibilité de visualiser plusieurs images résultantes à travers une seule sortie vidéo, ce qui implique de transférer les résultats vers la mémoire centrale. La seconde contrainte est l'augmentation importante du volume de calcul, due au traitement de grands nombres d'images de haute définition HD, et aussi

à la nature des images bruitées qui demandent plus de traitements. Afin de contourner ces contraintes, nous proposons une implémentation hybride et efficace de méthodes de détection des points d'intérêts et de contours. Ces méthodes seront utilisées dans une application médicale de segmentation de vertèbres [3], ainsi que dans une application de navigation dans des bases de données multimédia [4]. L'implémentation proposée permet une exploitation de l'intégralité des ressources de calcul hétérogènes (Multi-CPU/GPU) en utilisant la librairie StarPU [5], qui offre un support exécutif unifié pour les architectures multi-coeurs hétérogènes.
La contribution est organisée comme suit : dans la deuxième section, nous présentons un état de l'art des méthodes de traitement d'images sur GPU, ainsi que des technologies permettant l'exploitation des architecture multi-coeurs et hétérogènes. La troisième section est dévolue à l'utilisation du GPU en traitement d'images, en proposant un modèle basé sur CUDA et OpenGL, tandis que la section 4 décrit l'implémentation GPU de deux méthodes d'extraction des caractéristiques : la détection des contours ansi que la détection des points d'intérêts. La section 5 présente l'implémentation hétérogène (Multi-CPU/GPU) des méthodes citées précédemment. Les résultats expérimentaux sont présentés dans la sixième section. Finalement, la section 7 est consacrée à la conclusion et la proposition de perspectives.

## 2. Etat de l'art

La majorité des algorithmes de traitement d'images contiennent des phases qui consistent en un calcul commun entre les pixels de l'image. Ce type de calcul se prête donc bien à une parallèlisation GPU permettant d'exploiter les unités de traitements (coeurs GPU). Ceci est particulièrement important dans les applications connues par leur forte intensité de calcul, telles que les applications médicales utilisant de gros volumes de données, ou les applications de navigation dans des bases de données multimédia. Dans cette catégorie, Yang et al. ont ont mis en oeuvre plusieurs algorithmes classiques de traitement d'images sur GPU avec CUDA [1]. On trouve aussi dans le projet OpenVIDIA [8] une implémentation de différents algorithmes de vision par ordinateur sur des processeurs graphiques, utilisant OpenGL, Cg [9] et CUDA. Luo et al. ont proposé une implémentation GPU de la méthode d'extraction de contours [10] basée sur le détecteur de Canny [11]. Il existe aussi d'autres méthodes médicales implémentées sur GPU, permettant le calcul du rendu d'images volumineuses [12, 13], ainsi que des méthodes de reconstruction à base d'images IRM [14].
Par ailleurs, il existe différents travaux pour l'exploitation des plateformes multi-coeurs et hétérogènes. Ayguadé et al. ont proposé un modèle de programmation flexible sur plateformes multi-coeurs [15]. StarPU [5] offre un support exécutif unifié pour les architectures multi-coeurs hétérogènes, tout en s'affranchissant des difficultés liées à la gestion des transferts de données. StarPU propose par ailleurs plusieurs stratégies d'ordonnancement efficaces et offre en outre la possibilité d'en concevoir aisément de nouvelles.
Notre contribution porte sur le développement d'approches efficaces non seulement par le choix et la parallélisation d'algorithmes d'extraction de caractéristiques de bases d'images volumineuses, mais aussi par l'adaptation de ces algorithmes pour exploiter au mieux la puissance de calcul des nouvelles architectures parallèles hétérogènes. Les algorithmes implémentés permettent la détection des points d'intérêts, ainsi que l'extraction récursive des contours basée sur le principe de Deriche-Canny, assurant une immunité au bruit de troncature et utilisant un nombre d'opérations réduit. Nous contribuons ainsi à travers ces implémentations à l'accélération des applications médicales telles que la segmentation de vertèbres [3], et aussi des applications de navigation dans des objets multimédia [4].

## 3. Modèle proposé pour le traitement d'images sur GPU

Nous proposons dans cette section un modèle de traitement d'images sur GPU, permettant le chargement, le traitement et l'affichage d'images sur processeurs graphiques. Notre modèle est basé sur CUDA pour les traitements parallèles et OpenGL pour la visualisation des résultats, ce qui permet de réduire les coûts de transfert de données entre la mémoire CPU et la mémoire GPU. Ce modèle repose sur quatre étapes principales (Figure 1) :

1. ***Chargement des images d'entrée :*** Le chargement des images d'entrée depuis la mémoire CPU vers

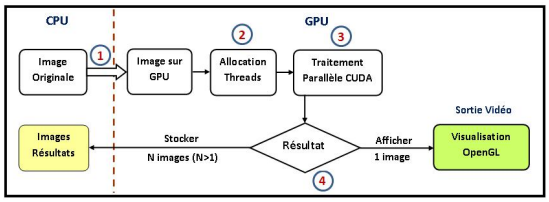



FIGURE 1 – Modèle traitement d'images sur GPU avec CUDA et OpenGL.

la mémoire GPU, permet de les traiter sur GPU par la suite.

2. ***Allocation des threads :*** Une fois que l'image est chargée en mémoire GPU, on choisit le nombre de threads de la grille de calcul GPU, de telle sorte que chaque thread puisse effectuer son traitement sur un ou plusieurs pixels groupés. Cela permet aux threads GPU de traiter les pixels en parallèle. Notons que la sélection du nombre de threads dépend du nombre de pixels de l'image.
3. ***Traitement parallèle avec CUDA :*** Les fonctions CUDA (kernels) sont exécutés $N$ fois en utilisant les $N$ threads sélectionnés lors de l'étape précédente.
4. ***Présentation des résultats :*** Après avoir effectué les traitements parallèles avec CUDA, les résultats peuvent être présentés en utilisant deux scénarios différents :
   - **Visualisation OpenGL** : L'affichage des images de sortie avec la bibliothèque OpenGL permet une visualisation rapide grâce a son utilisation de tampons déjà existants sur GPU. Notre choix d'OpenGL est dû à sa compatibilité avec CUDA, il permet ainsi de réduire les coûts de transfert de données. Ce scénario est utile lorsqu'on applique le traitement GPU sur une seule image.
   - **Transfert des résultats** : La visualisation OpenGL est impossible lors de l'application des traitements GPU sur un ensemble d'images en utilisant une seule sortie vidéo. Dans ce cas, nous devons transférer les images de sortie depuis la mémoire GPU vers la mémoire CPU. Le temps de transfert des images représente un coût supplémentaire pour l'application.

## 4. Mise en oeuvre de détection efficace des coins et contours sur GPU

Les méthodes d'extraction des caractéristiques telles que les coins et les contours représentent des étapes préliminaires à de nombreux processus de vision par ordinateur. Sur base du modèle décrit dans la section 3, nous proposons dans ce paragraphe une implémentation GPU des méthodes de détection des coins ainsi que des contours, permettant d'obtenir à la fois des résultats efficaces au niveau de la qualité des coins et bords extraits, et aussi plus rapides grâce à l'exploitation des coeurs GPU en parallèle. Nous présentons dans un premier temps notre implémentation GPU de la méthode de détection des points d'intérêts basée sur le détecteur de Harris [7]. Ensuite, Nous décrivons notre implémentation GPU de l'approche de détection des contours basée sur le principe de Deriche utilisant les critères de Canny [6].

### 4.1. Extraction des points d'intérêts dans une image sur GPU

Ce paragraphe présente notre implémentation GPU du détecteur des coins utilisant la technique décrite par Bouguet [16], basée sur le principe de Harris. Cette méthode est connue pour son efficacité, due à sa forte invariance à la rotation, à l'échelle, à la luminosité et au bruit de l'image. Nous avons parallélisé cette méthode en implémentant chacune de ses cinq étapes sur GPU (Figure 2.a) :

1. ***Calcul des dérivées et du gradient spatial :*** La première étape est le calcul de la matrice du gradient spatial G pour chaque pixel de l'image I, en utilisant l'équation (2). Cette matrice de 4 éléments (2*2) est calculée sur base des dérivées spatiales $I_x$, $I_y$ calculées suivant l'équation (1).

$$I_x(x,y) = \frac{I(x+1,y) - I(x-1,y)}{2} \qquad I_y(x,y) = \frac{I(x,y+1) - I(x,y-1)}{2} \tag{1}$$

$$G = \begin{pmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{pmatrix} \tag{2}$$

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 en utilisant l'équation (1). Ensuite, le thread peut calculer le gradient spatial de chaque point de l'image en appliquant l'équation (2). 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. Cela permet d'accéder plus rapidement aux données.

2. ***Calcul des valeurs propres de la matrice du gradient*** : Sur base du gradient calculé par l'équation (2), on calcule les valeurs propres de la matrice G pour chaque pixel. L'implémentation GPU de cette étape est effectuée par le calcul de ces valeurs en parallèle sur les pixels de l'image, en utilisant une grille de calcul GPU contenant un nombre de threads égal au nombre de points de l'image.
3. ***Recherche de la valeur propre maximale*** : Une fois les valeurs propres calculées, on extrait la valeur propre maximale. Cette valeur est extraite sur GPU en faisant appel à la librairie CUBLAS.
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.
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 implémenter cela sur GPU, nous avons affecté à chaque thread GPU un groupe de pixels représentant une zone (10x10 pixels). Chaque thread permet d'extraire la valeur propre maximale dans une zone en utilisant toujours la libraire CUBLAS. Les pixels ayant ces valeurs extraites représentent ainsi les points d'intérêts.

### 4.2. Détection des contours sur GPU

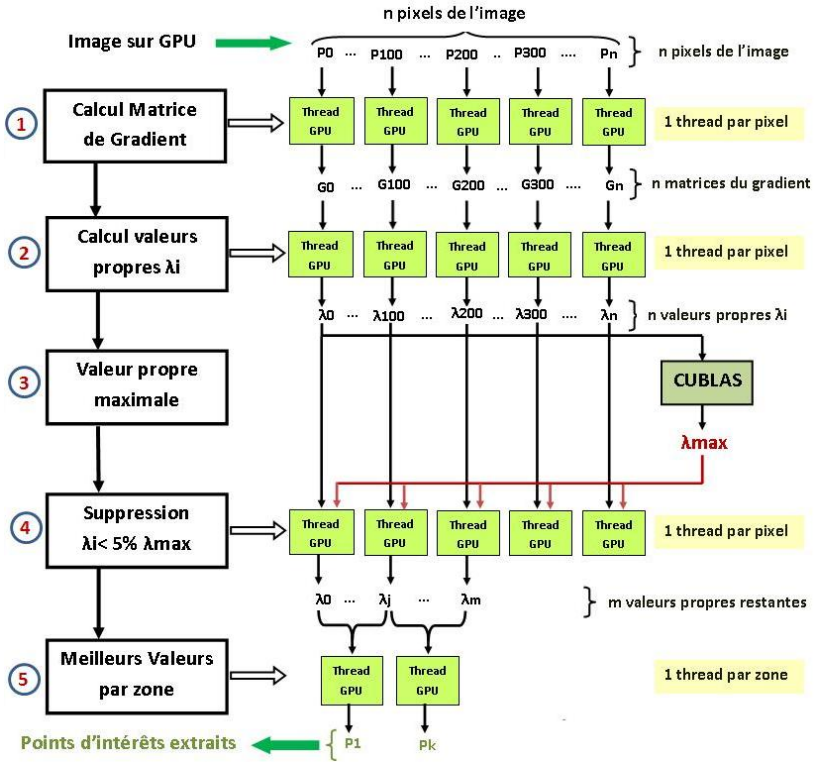
Ce paragraphe présente l'implémentation GPU de la méthode de détection des contours basée sur la technique récursive de Deriche [6]. L'immunité au bruit de troncature et le nombre réduit d'opérations de cette approche la rendent très efficace au niveau de la qualité des contours extraits. Cependant, cette méthode est entravée par les coûts de calcul qui augmentent considérablement en fonction du nombre et de la taille des images utilisées. Cette technique est composée de quatre étapes principales (Figure 2.b). Notons que l'étape du calcul des gradients applique un lissage gaussien récursif avant de filtrer l'image en utilisant les filtres de Sobel [11]. Toutefois, les étapes de calcul de la magnitude et de la direction du gradient, la suppression des non maxima, ainsi que le seuillage sont les mêmes que celles utilisées pour le filtre de Canny. Ces dernières étapes permettent de sélectionner les meilleurs contours sur base des magnitudes et directions des gradients calculés, ainsi que les valeurs de deux seuils choisis en dernière étape. L'implémentation GPU de cette méthode est décrite en détail dans [3]. Cette implémentation est essentiellement basée sur le modèle proposé dans la section 3.

## 5. Mise en oeuvre de détection efficace des coins et contours sur plateformes hybrides

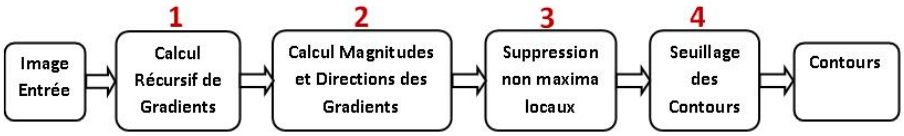
Nous avons présenté dans la section 4 l'implémentation GPU des deux méthodes d'extraction de caractéristiques (extraction de coins et de contours). Cette implémentation est très performante lors du traitement d'image unique, puisque l'image résultante peut être visualisée directement à partir du GPU grâce à la bibliothèque OpenGL. Cependant, si on applique cette implémentation à un ensemble d'images, les résultats ne peuvent plus être visualisés directement à partir du GPU. Dans ce cas, un transfert de données depuis la mémoire graphique vers la mémoire centrale devient nécessaire. Ce transfert devient très coûteux lors de l'utilisation de grandes bases d'images volumineuses. Cette section présente une implémentation hybride de ces méthodes permettant une exploitation de l'intégralité des ressources hétérogènes de calcul, en utilisant le support StarPU. L'avantage majeur de cette implémentation est son exploitation simultanée des coeurs CPU et GPU multiples, permettant d'avoir un traitement plus rapide, avec moins de transfert de données puisque les images traitées sur CPU ne doivent pas être transférées à la fin. Cette implémentation est décrite en trois étapes :

### 5.1. Chargement des images d'entrée

La première étape est le chargement des images d'entrée qui seront adressées par des tampons de telle sorte que StarPU puisse appliquer ces traitements sur ces tampons. Le Listing 1 résume cette étape :




(a) Détection des points d'intérêts (coins) sur GPU




(b) Détection récursive des contours basée sur le principe de Deriche-Canny

FIGURE 2 – Détection des coins et contours sur GPU avec CUDA et OpenGL

Listing 1 – Chargement des images d'entrée avec StarPU

```
1 for (i = 0; i < n; ++i) {                                    // n: nombre d'images.
2   img = cvLoadImage(Input_Image, CV_LOAD_IMAGE_COLOR);
3   starpu_data_handle img_handle;
4   starpu_vector_data_register(&img_handle, 0, (uintptr_t) img , size, sizeof (void *));
5   Liste_chainee = ajouter(Liste_chainee, img, img_handle);
6 }
```

La ligne 2 permet de charger l'image en mémoire centrale, les lignes 3 et 4 permettent d'allouer un tampon (handle) StarPU et le faire pointer vers l'image chargée. La ligne 5 permet de rajouter l'image chargée ainsi que le tampon StarPU dans une liste chaînée qui contiendra l'ensemble des images à traiter.

### 5.2. Traitement hétérogènes d'images avec StarPU

Une fois les images chargées dans les tampons, on peut les traiter avec StarPU de manière hétérogène en utilisant les fonctions CPU et GPU implémentées dans les sections précédentes. Le traitement StarPU se base sur deux structures principales : la codelet et la tâche. La codelet permet de préciser sur quelles architectures le noyau de calcul peut s'effectuer, ainsi que les implémentations associées (Listing 2 : ligne 1-6). Les tâches StarPU consistent à appliquer la codelet sur l'ensemble des images. Dans notre cas, une

tâche est créée et lancée tant qu'on n'est pas arrivé au bout de la liste chaînée (chaque tâche traite une image). L'ordonnanceur de StarPU distribue les tâches sur les coeurs multiples hétérogènes selon des stratégies efficaces. StarPU se charge aussi de transférer les données depuis la mémoire centrale vers la mémoire graphique lorsqu'on applique des tâches GPU. (Listing 2 : lignes :7-14). Notons que dans notre cas les tâches StarPU traitent les images de manière indépendante (traitements asynchrones).

Listing 2 – Soumission des tâches StarPU à l'ensemble des images

```
1  static starpu_codelet cl ={
2    .where = STARPU_CPU|STARPU_CUDA,      // utilisation des coeurs CPU et GPU
3    .cpu_func = cpu_impl,                 // définir la fonction CPU
4    .cuda_func = cuda_impl,               // définir la fonction GPU en CUDA.
5    .nbuffers = 1                         // nombre de tampons.
6  };
7  while (Liste != NULL) {
8          struct starpu_task *task = starpu_task_create();       //Créer la tâche StarPU
9          task->cl = &cl;                                        //Associer la codelet à la tâche
10         task->buffers[0].handle = liste->A_handle;              //Indiquer le tampon à traiter
11         task->buffers[0].mode = STARPU_RW;                     //Mode Lecture/écriture
12         starpu_task_submit(task);                              //Soumettre la tâche StarPU
13         Liste = Liste->next;                                   //Passer à l'image suivante
14 }
```

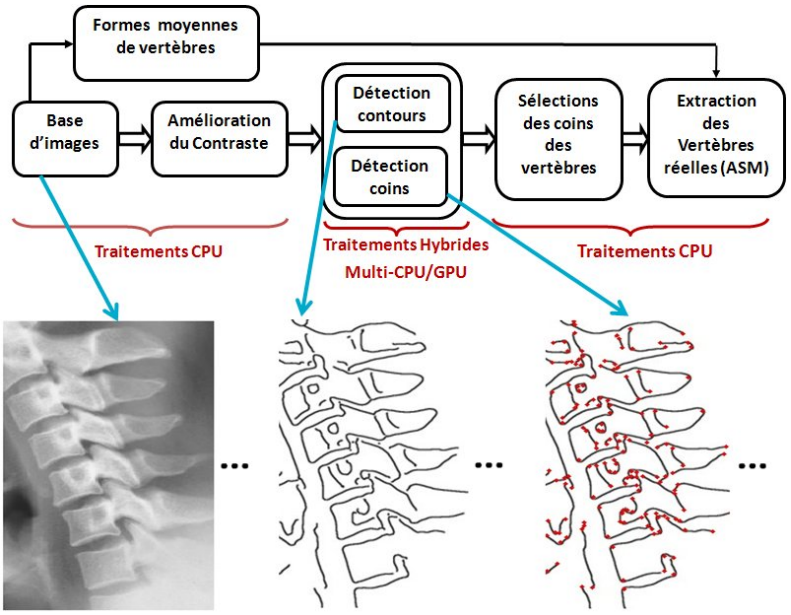
### 5.3. Mise à jour et récupération des résultats

Une fois que toutes les tâches StarPU sont lancées et appliquées sur l'ensemble des images, les résultats peuvent être récupérés. Cependant, il faut mettre à jour les tampons StarPU afin de prendre en compte les traitements effectuées par les coeurs CPU et GPU. La mise à jour est assurée par la fonction StarPU "starpu_data_acquire". Cette fonction se charge aussi de transférer les données depuis la mémoire graphique vers la mémoire centrale si les traitements étaient effectués sur GPU.

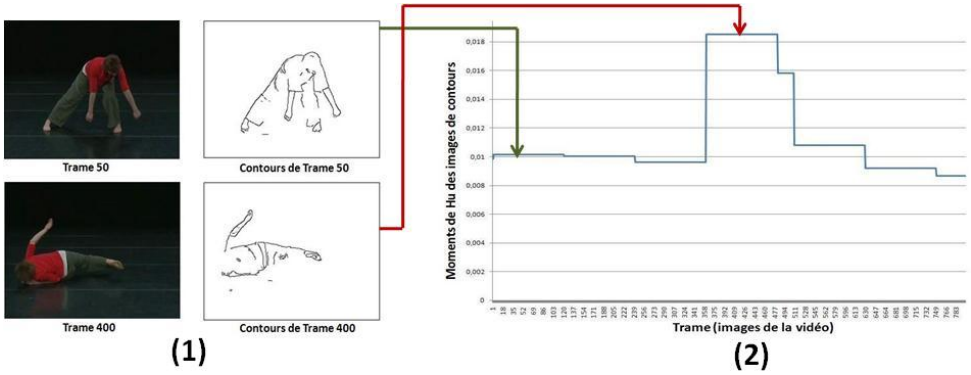
## 6. Résultats Expérimentaux

Les implémentations décrites dans les sections précédentes sont exploitées dans un premier temps pour une application médicale de segmentation de vertèbres basée sur les modèles de forme active (Active Shape Model) [3]. Cette application est caractérisée par la faible variation du niveau de gris et le grand volume de données à traiter (grande base d'images HD). Nous proposons ainsi d'accélérer les étapes les plus intensives (détection des coins, détection des contours) de cette application, en exploitant les implémentations parallèles et hétérogènes (Figure 3.a).

La qualité de la segmentation des vertèbres reste identique puisque la procédure est toujours la même. Cependant, l'exploitation des plateformes parallèles et hétérogènes a permis diminuer significativement le temps de calcul. Ceci autorise d'appliquer la méthode de segmentation de vertèbres sur des plus grands ensembles d'images ayant des résolutions plus hautes. Par conséquent, les résultats de l'application médicale peuvent être plus précis (formes de vertèbres détectées). La figure 3.c montre la comparaison des temps de calcul entre les implémentations séquentielles (CPU), parallèles (GPU) et hybrides de la méthode regroupant la détection des coins ainsi que des contours, appliquée sur un ensemble de 200 images. La figure 3.c montre les accélérations obtenues grâce à ces implémentations. Les accélérations obtenues sont dues à deux facteurs principaux : le premier est l'exploitation des coeurs GPU permettant d'appliquer des traitements parallèles à l'intérieur des images (entre pixels). Le deuxième facteur est l'exploitation simultanée des multiples coeurs CPU et GPU permettant d'appliquer des traitements parallèles entre les images de telle sorte que chaque coeur CPU ou GPU traite un sous-ensemble d'images. Par ailleurs, nous avons exploité ces implémentations hybrides dans une application de navigation et de calcul de similarité entre séquences vidéo (vidéos de danse)[4]. Cette application est basée sur le l'extraction des caractéristiques des trames (images) composant chaque séquence vidéo. Parmi ces caractéristiques, on trouve les contours qui permettent de fournir des informations pertinentes pour détecter les zones de mouvement en utilisant les moments de Hu [17]. La Figure 3.b montre les contours extraits de manière hybride à parir des trames de la vidéo, elle montre aussi les moments de Hu extraits à partir

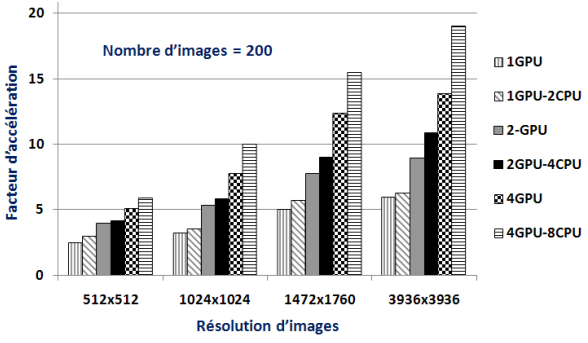



(a) Etapes de segmentation des vertèbres




(b) 1. Détection hybide de contours des trames 2. Moments de Hu des contours extraits


(c) Temps de détection hybride des coins + contours




(d) Taux d'accélération de détection des coins + contours

FIGURE 3 – Segmentation des vertèbres sur architectures hétérogènes.

de ces contours. Cette détection hybride des contours nous a permis d'accélérer le processus de calcul de similarité entre séquences vidéo.

Les expérimentations ont été menées sur les plates-formes suivantes :
– **CPU :** Dual Core 6600, 2.40 GHz, 2 GB RAM of Memory.
– **GPU :** NVIDIA Tesla C1060, 240 CUDA cores, 4GB of Memory.

## 7. Conclusion

Nous avons proposé dans ce travail d'exploiter les architectures parallèles (GPU) et hétérogènes (Multi-CPU/GPU) dans le domaine du traitement de bases d'images volumineuses dans le cas d'applications basées sur l'extraction des caractéristiques, telles que la segmentation de vertèbres dans des images médicales, ainsi que dans la navigation et le calcul de similarité entre séquences vidéo. Les résultats expérimentaux montrent des gains allant d'un facteur de 6 à 20 par rapport à une implémentation classique sur CPU. Ces gains sont dus essentiellement aux traitements parallèles exploitant des plateformes hétérogènes. Ces traitements permettent de manipuler les ensembles d'images en deux niveaux de parallélisme : le premier niveau est le traitement parallèle entre les pixels de l'image (intra-image) grâce à l'exploitation des coeurs GPU, tandis que le deuxième niveau est le traitement parallèle entre les images (inter-images) de telle sorte que chaque coeur CPU ou GPU traite un sous-ensemble d'images. Comme perspective, nous envisageons de concevoir un modèle plus général pour le traitement d'objets multimédia (images et vidéos HD, etc.) sur plateformes parallèles et hétérogènes. Ce modèle devra choisir automatiquement les ressources à utiliser (CPU et/ou GPU) ainsi que les méthodes (séquentielles, parallèles, hybrides) à appliquer, selon la nature des médias à traiter.

## Bibliographie


1. 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.
2. AMD Fusion$^{TM}$Family of APUs. The Future brought to you by AMD introducing the AMD APU Family. 2011. [Online]. Available : http ://sites.amd.com/us/fusion/apu/Pages/fusion.aspx
3. S.A. Mahmoudi, F. Lecron, P. Manneback, M. Benjelloun and S.Mahmoudi. GPU-Based Segmentation of Cervical Vertebra in X-Ray Images. Workshop HPCCE. In Conjunction with IEEE Cluster. Greece, pp. 1-8, 2010.
4. 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. volume 2, n°1, 2009.
5. C. Augonnet and al, StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In Concurrency and Computation : Practice and Experience. Euro-Par 2009. best paper. pp. 863-874.
6. R. Deriche. Using Canny's criteria to derive a recursively implemented optimal edge detector. Internat. J. Vision. Boston. pp. 167-187, 1987.
7. C. Harris, M. Stephens. A combined corner and edge detector. In Alvey Vision Conference. pp 147-152. 1988.
8. J. Fung, S. Mann, and al. OpenVIDIA :Parallel gpu computer vision.ACM Multimedia. pp 849-852. 2005.
9. 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.
10. Y. Luo and R. Duraiswani. Canny Edge Detection on NVIDIA CUDA. Proceedings of the Workshop on Computer Vision on GPUS. CVPR. 2008.
11. J. Canny. A computational approach to edge detection. IEEE Transactions on Pattern Analysis and Machine Intelligence. vol. 8. no. 6. pp 679-714. 1986.
12. 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.
13. M. Smelyanskiy and al. Mapping high-fidelity volume rendering for medical imaging to CPU, GPU and many-core architectures. IEEE Transactions on Visualization and Computer Graphics. pp 1563-1570. 2009.
14. 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.
15. E.Ayguadé and al. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In Euro-Par'09 : Proceedings of the 15th International Euro-Par Conference on Parallel Processing. pp 851-862.
16. Bourget J.Y. Pyramidal Implementation of the Lucas Kanade Feature Tracker, Description of the algorithm. Intel Corporation Microprocessor Research Labs. 2000.
17. Ming K. Hu. Visual Pattern Recognition by Moment Invariants. In : IRE Transactions on Information Theory IT-8 (1962). pp 179-187. P. :20.