Traitements d'Images sur Architectures Parallèles et Hétérogènes

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

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

RÉSUMÉ. *Les algorithmes de traitement d'images présentent des outils nécessaires à de nombreux processus de vision par ordinateur. Ces algorithmes deviennent très consommatrices en temps de calcul lors du traitement de gros volumes d'images de hautes définitions. Nous proposons dans ce travail un schéma de développement permettant une exploitation efficace des architectures parallèles (GPU) et hétérogènes (Multi-CPU/Multi-GPU) afin d'accélérer ces algorithmes. Le schéma proposé permet un ordonnancement efficace des tâches hybrides, tout en assurant une meilleure gestion des espaces mémoires hétérogènes. Nous présentons aussi les implémentations parallèles et hybrides de méthodes de détection des coins et de contours. Des résultats expérimentaux utilisant différents ensembles d'images ont montré une accélération allant de 5 à 25 par rapport à une implémentation CPU.*

ABSTRACT. *Image processing algorithms present a necessary tool for various domains related to computer vision. These algorithms are hampered by their high consumption of computing times when processing large sets of high resolution images. In this work, we propose a development scheme enabling an efficient exploitation of parallel (GPU) and heterogeneous (Multi-CPU/Multi-GPU) platforms, in order to improve performance of image processing algorithms. The proposed scheme enables an efficient scheduling of hybrid tasks and an effective management of heterogeneous memories. We present also parallel and hybrid implementations of edge and corner detection methods. Experimental results showed a global speedup ranging from 5 to 25, when processing different sets of images, by comparison with CPU implementations.*

MOTS-CLÉS : *Calcul Hétérogène, GPU, Traitement d'images, Détection des coins et contours.*

KEYWORDS: *Heterogeneous Computing, GPU, Image processing, Corner and edge detection.*

DOI:10.3166/TSI.31.1-21

1. Introduction

Lors des dernières années, la fréquence des processeurs centraux (CPU) s'est trouvée plafonnée à environ 4 GHz. Une limitation qui a été contournée par un changement des architectures de processeurs à travers la multiplication des unités de calcul. Cette évolution se retrouve aussi bien au niveau des processeurs généralistes que des processeurs graphiques, ainsi que dans les tous récents processeurs accélérés (APU), combinant CPU et GPU sur la même puce (AMD, 2011) et partageant le même espace mémoire. Par ailleurs, les processeurs graphiques possèdent nativement une structure de cœurs massivement parallèles et offrent des puissances brutes de calculs largement supérieures aux CPUs. Si les GPUs ont comme premier objectif le rendu d'images 2D/3D et les jeux vidéo, de nombreux chercheurs ont entrepris d'exploiter leur puissance pour accélérer d'autres traitements, habituellement destinés aux CPUs.

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 des GPUs a permis d'accélérer ces algorithmes par l'exploitation des cœurs GPU multiples en parallèle. Ceci s'avère très performant lors du traitement d'image unique puisque le résultat peut être directement visualisé à partir du GPU, à l'aide de bibliothèques graphiques, telles qu'OpenGL (OpenGL, 2004). Toutefois, les méthodes appliquées sur des images multiples 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 est l'augmentation importante du volume de calcul, due au traitement de grands nombres d'images de haute définition (HD), ainsi qu'à la nature des images bruitées qui demandent plus de traitements. Afin de contourner ces contraintes, nous proposons un schéma de développement permettant d'exploiter efficacement les architectures parallèles (GPU) et hétérogènes (Multi-CPU/Multi-GPU) en utilisant le support StarPU (Augonnet *et al.*, 2009). Ce dernier permet la distribution des tâches sur les ressources hétérogènes, tout en offrant la possibilité de concevoir de nouvelles stratégies d'ordonnancement. En effet, nous employons un ordonnancement des tâches prenant en compte les temps d'exécutions et de transferts des tâches précédentes. Ceci permet d'estimer la durée des tâches lancées afin de les affecter aux ressources (CPU ou GPU) appropriées. Les tâches à forte intensité de calcul seront prioritaires pour un traitement sur GPU, tandis que les tâches demandant peu de calcul seront affectées aux CPU. Nous proposons aussi les implémentations parallèles et hétérogènes des algorithmes de détection des coins et de contours. Ces implémentations seront exploitées dans une application médicale de segmentation de vertèbres (Mahmoudi *et al.*, 2010), ainsi que dans une application de navigation dans des bases de données multimédia (Siebert *et al.*, 2009).

Cet article est organisé comme suit: dans la deuxième section, nous présentons un état de l'art des méthodes de traitement d'images sur GPU, ainsi que les technologies permettant l'exploitation des architectures multi-cœurs hétérogènes. La section 3 présente le schéma proposé pour le traitement d'images sur GPU, ainsi que les implémentations GPU des méthodes de détection des coins et de contours. La quatrième section présente notre modèle de traitement d'images sur architectures hétérogènes.

Les résultats expérimentaux sont présentés dans la section 5. Finalement, la sixième section est consacrée à la conclusion et la proposition de perspectives.

2. État de l'art

Au-delà du rendu 2D/3D, la majorité des algorithmes de traitement d'images contiennent des phases qui consistent en des calculs similaires entre les pixels de l'image. Ceci rend ces algorithmes bien adaptés à une parallélisation GPU permettant d'exploiter les unités de traitements (cœurs GPU). Ces traitements sont particulièrement importants 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 ce contexte, Yang *et al.* ont mis en œuvre plusieurs algorithmes classiques de traitement d'images sur GPU avec CUDA [1] (Yang *et al.*, 2008). On trouve aussi dans le projet OpenVIDIA (Fung *et al.*, 2005) une collection d'implémentations d'algorithmes de vision par ordinateur sur des processeurs graphiques, utilisant OpenGL, Cg (Mark *et al.*, 2003) et CUDA. Luo *et al.* ont proposé une implémentation GPU (Luo, Duraiswan, 2008) de la méthode d'extraction de contours basée sur le détecteur de Canny (Canny, 1986). Antao *et al.* ont implémenté des algorithmes de traitement linéaire d'images sur GPU sous OpenCL [2] (Antao, Sousa, 2010). Il existe aussi d'autres méthodes médicales implémentées sur GPU, permettant le calcul du rendu d'images volumineuses (Heng, Gu, 2005), (Smelyanskiy *et al.*, 2009), ainsi que des méthodes de reconstruction à base d'images IRM (Schiwietz *et al.*, 2006). Par ailleurs, différents travaux ont été menés pour l'exploitation des plateformes multi-cœurs et hétérogènes. OpenCL propose un standard permettant de programmer sur les plateformes hybrides composées de CPUs et de GPUs (cartes ATI ou NVIDIA). Ayguadé et al. (Ayguadé *et al.*, 2009) ont proposé un modèle de programmation flexible sur plateformes multi-cœurs. StarPU (Augonnet *et al.*, 2009) offre de son côté un support exécutif unifié pour les architectures multi-cœurs hétérogènes, tout en s'affranchissant des difficultés liées à la gestion des transferts de données. Le point fort de StarPU est sa proposition de plusieurs stratégies d'ordonnancement efficaces et portables, en offrant la possibilité d'en concevoir aisément de nouvelles. Il permet également d'exploiter des programmes écrits sous CUDA ou OpenCL. Ces aspects ont motivé son choix dans le cadre de ce travail.

Notre contribution porte sur le développement d'approches efficaces tant sur le choix et la parallélisation GPU d'algorithmes d'extraction de caractéristiques d'images, que par l'adaptation de ces algorithmes pour exploiter au mieux la puissance de calcul des nouvelles architectures parallèles hétérogènes (Multi-CPU/Multi-GPU). Nous contribuons aussi par la réduction des temps de transferts grâce à l'exploitation adaptée des mémoires "texture et partagée" du GPU offrant un accès plus rapide aux pixels d'images. Une autre contribution de ce travail porte sur l'utilisation d'une stratégie

1. NVIDIA CUDA. Disponibles sur Internet à `http://www.nvidia.com/cuda`.
2. Khronos Group. Disponibles sur Internet à `http://www.khronos.org/opencl`.

d'ordonnancement efficace au sein des CPUs et GPUs multiples, prenant en compte les temps de calcul et de transferts estimés pour chaque tâche. Les algorithmes implémentés permettent la détection des points d'intérêts, ainsi que les contours. Nous contribuons ainsi à travers ces implémentations à l'accélération d'une application médicale de segmentation de vertèbres (Mahmoudi *et al.*, 2010), et d'une application de navigation dans des bases d'objets multimédia (Siebert *et al.*, 2009).

3. Traitement parallèle d'images sur GPU

Comme montré dans les sections précédentes, les algorithmes de traitement d'images représentent un excellent champ d'applications pour l'accélération sur GPU. Nous proposons dans cette section un schéma de développement pour le traitement d'images sur processeurs graphiques, ainsi que les implémentations GPU des algorithmes de détection des coins et de contours. Cette section est présentée en trois parties : la première partie décrit le schéma proposé pour le traitement d'images sur GPU. La deuxième partie présente notre implémentation GPU des méthodes de détection des coins et de contours. Les performances obtenues via ces implémentations sont présentées et analysées dans la dernière partie.

3.1. Schéma de développement proposé pour le traitement d'images sur GPU

Nous proposons dans ce paragraphe un schéma de développement pour le traitement d'images sur GPU, permettant le chargement, le traitement et l'affichage d'images sur processeurs graphiques. Notre schéma s'appuie sur CUDA pour les traitements parallèles et OpenGL pour la visualisation des résultats. Ce schéma repose sur quatre étapes principales (Figure 1)

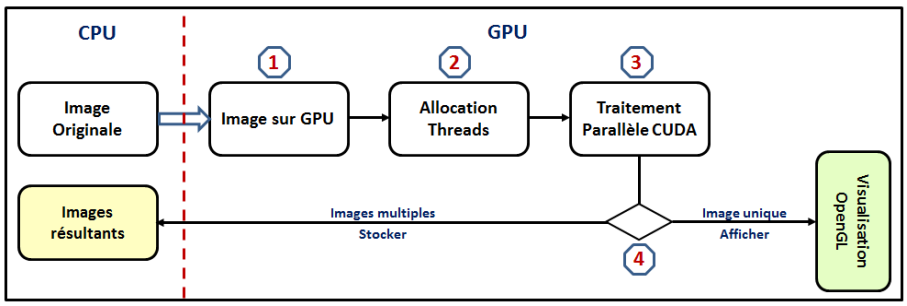


Figure 1. Schéma de développement proposé pour le traitement d'images sur GPU

1. **Chargement des images d'entrée:** le transfert des images d'entrée depuis la mémoire CPU vers la mémoire GPU permet de les traiter sur GPU par la suite.

2. **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. La sélection du nombre de threads dépend du nombre de pixels de l'image.

3. **Traitement parallèle avec CUDA:** les fonctions CUDA (kernels) sont exécutées *N* fois simultanément en utilisant les *N* threads créés lors de l'étape précédente.

4. **Présentation des résultats:** à l'issue des traitements, 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 à la réutilisation de zones mémoires allouées par CUDA. Ceci permet de réduire significativement les coûts de transferts de données. Ce scénario est utile lors du traitement GPU appliqué sur une seule image.

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

Pour une meilleure exploitation des processeurs graphiques, nous proposons d'exploiter les mémoires GPU offrant un accès plus rapide aux données, *i.e.* la mémoire de texture et la mémoire partagée. En effet, nous chargeons l'image d'entrée en mémoire de texture afin d'avoir un accès plus rapide aux pixels. De plus, nous avons chargé pour chaque pixel les valeurs de ses voisins en mémoire partagée. Ceci permet un traitement accéléré des pixels utilisant les valeurs du voisinage.

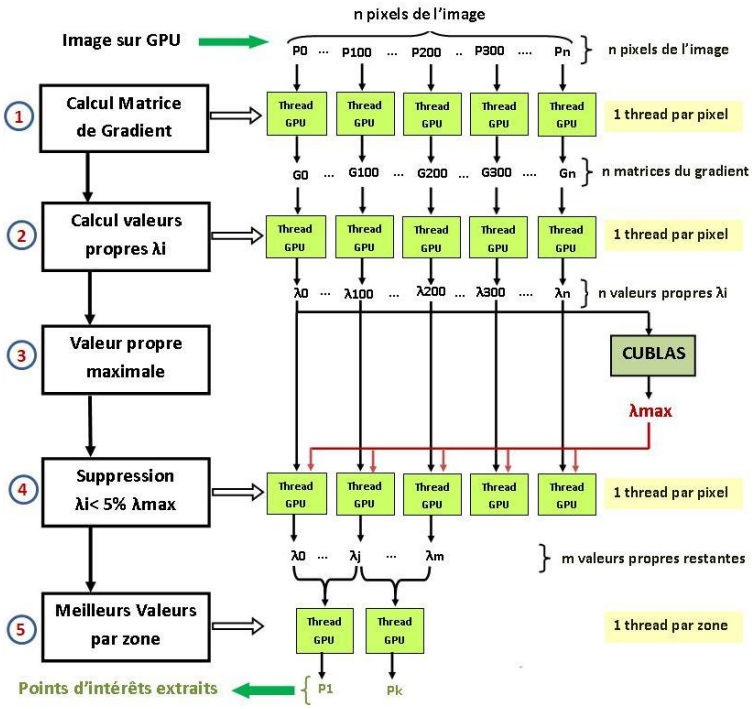


Figure 2. Détection des points d'intérêts (coins) sur GPU

3.2. Mise en œuvre de détection efficace des coins et de contours sur GPU

Les méthodes d'extraction de coins et de contours deviennent très intensives en calcul lors de l'utilisation de bases d'images volumineuses, et aussi lors de l'utilisation d'images bruitées. Sur base du schéma décrit dans la section 3.1, nous proposons l'implémentation GPU de la méthode de détection des points d'intérêts basée sur le détecteur de Harris (Harris, Stephens, 1988). Ensuite, Nous décrivons notre implémentation GPU de l'approche de détection des contours basée sur le principe de Deriche utilisant les critères de Canny (Canny, 1986).

3.2.1. Extraction des points d'intérêts sur GPU

Ce paragraphe présente notre implémentation GPU du détecteur des coins utilisant la technique décrite par Bouguet (Bouguet, 2000), 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. Nous avons parallélisé cette méthode en implémentant chacune de ses cinq étapes sur GPU (Figure 2):

1. ***Calcul des dérivées spatiales:*** la première étape est le calcul de la matrice de dérivées spatiales G pour chaque pixel de l'image I à partir de 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 à partir de l'équation 1. Ensuite, le thread peut calculer la matrice G 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.

2. ***Calcul des valeurs propres :*** à partir de la matrice G, nous calculons les valeurs propres de chaque pixel. Ce dernier sera représenté par deux valeurs propres. Nous gardons ensuite pour chaque pixel 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. ***Recherche de la valeur propre maximale :*** une fois les valeurs propres calculées, nous extrayons la valeur propre maximale. Cette valeur est obtenue sur GPU en faisant appel à la librairie CUBLAS [3].

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 sera 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 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'intérêts.

3.2.2. Détection des contours dans une image sur GPU

Nous décrivons dans ce paragraphe l'implémentation GPU de la méthode de détection des contours basée sur la technique récursive de Deriche (Deriche, 1987). La robustesse au bruit de troncature et le nombre d'opérations réduit 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 :

- calcul des gradients (Gx, Gy);
- calcul de magnitude et de direction du gradient;
- suppression des non maxima;
- seuillage des contours.

Notons que l'étape de calcul des gradients applique un lissage Gaussien récursif avant de filtrer l'image en utilisant les deux filtres de Sobel (Equation 4). Toutefois, 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 (Canny, 1986). L'implémentation GPU proposée pour cette méthode (décrite en détail dans (Mahmoudi *et al.*, 2010)) est basée sur la parallélisation de ses étapes :

1. ***Lissage Gaussien sur GPU :*** la première étape permet de réduire les bruits de l'image originale, en appliquant à chaque pixel de l'image une convolution utilisant le filtre gaussien 3.

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3}$$

3. CUBLAS: Implementation of BLAS (Basic Linear Algebra Subprograms) with CUDA.

L'implémentation GPU du lissage Gaussien est fournie par le SDK CUDA [4]. Cette implémentation parallèle utilise le principe de la méthode de Deriche (Deriche, 1987). L'avantage de cette approche est l'indépendance du temps de calcul par 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.

2. ***Calcul des gradients de Sobel sur GPU***: 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 convolutions G_x et G_y, tel que montre l'équation 4.

$$G_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} * I \qquad G_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} * I \tag{4}$$

L'implémentation GPU de cette étape est développée en utilisant le SDK CUDA. Une mise en œuvre parallèle qui exploite à la fois la mémoire texture et la mémoire partagée du GPU afin d'avoir un accès plus rapide aux données. Cette implémentation GPU 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. ***Calcul de magnitude et de direction du gradient sur GPU :*** une fois que 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 (Equation 5). 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.

$$G = \sqrt{G_x^2 + G_y^2} \qquad \Theta = \arctan\left(\frac{G_x}{G_y}\right) \tag{5}$$

4. ***Suppression des non maxima sur GPU :*** 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.

5. ***Seuillage des contours sur GPU :*** le seuillage représente la dernière étape pour extraire les 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,

4. `http://developer.download.nvidia.com/compute/cuda/sdk/website/samples.html`.

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 (Luo, Duraiswan, 2008). Nous avons étendu cette implémentation par l'utilisation de la mémoire partagée pour avoir un accès plus rapide aux valeurs des pixels connectés.

3.2.3. Analyse des résultats

Nous présentons dans ce paragraphe une analyse des résultats obtenus par les implémentations GPU décrites ci-dessus. Ces résultats sont présentés en deux catégories : traitement d'image unique et traitement d'images multiples.

3.2.4. Traitement d'image unique

Une comparaison des temps de calcul entre les implémentations CPU et GPU de la méthode incluant la détection des coins et de contours est présentée dans le tableau 1. Notons que l'utilisation des processeurs graphiques pour le traitement d'image individuelle a permis d'avoir une haute accélération (18,3×), grâce à l'exploitation des unités de calculs GPU en parallèle ainsi que la visualisation rapide via OpenGL. De plus l'utilisation des mémoires partagées et de texture ont permis d'améliorer encore plus les performances (accélération de 22×) par rapport à la solution GPU utilisant la mémoire globale (Tableau 2). En effet, l'accès rapide au voisinage de pixels permet de réduire les temps d'exécutions de kernels (82,5 ms au lieu de 90,2 ms pour une image de 3936×3936). Les temps de chargement d'images sont également réduits grâce à l'utilisation de la mémoire de texture (8,8 ms au lieu de 14,4 ms pour une image de 3936×3936).

3.2.5. Traitement d'images multiples

Lors du traitement d'images multiples sur GPU, les performances (détection des coins et contours) deviennent moins élevées (8,7×) tel que montré dans le tableau 3. Cette réduction des performances est due principalement aux coûts additionnels de transferts d'images entre la mémoire graphique et la mémoire RAM. En effet, les étapes de chargement et de transfert des résultats prennent 42% (14 + 28) du temps total de l'application pour un ensemble de 200 images. Notons aussi que l'augmentation du nombre d'images n'a pas d'effet sur les facteurs d'accélérations puisque les images sont traitées séquentiellement sur GPU. Les traitements parallèles sont appliqués uniquement entre les pixels de la même image. La section prochaine présentera une solution hétérogène pour faire face à ces contraintes. Les expérimentations ont été effectuées sous Linux Ubuntu 11.04 (64 bits) avec CUDA 4.0, utilisant deux plateformes, *i.e.* CPU Bi-cœurs and GPU GTX 280.

- CPU: Bi-cœurs 6600, 2.40 GHz, RAM: 2 Go
- GPU: NVIDA GeForce GTX 280, 240 cœurs CUDA, GRAM: 1 Go

Notons que les implémentations séquentielles sont développées en C via la bibliothèque OpenCV [5] (version 2.1) de traitement d'images sur CPU. L'aspect multi-cœur du CPU est également géré par OpenCV.

Tableau 1. Détection GPU des coins et de contours d'image unique (Mém Globale)

Images	Temps	GPU (Mémoire Globale)				Acc
	CPU	Chargement	Kernels	Vis OpenGL	**Temps Total**	
512×512	**45 ms**	**2** (43%)	**1,9** (40%)	**0,8** (17%)	**4,7 ms**	**9,6**
1024×1024	**141 ms**	**3,1** (27%)	**6,0** (53%)	**2,2** (19%)	**11,3 ms**	**12,5**
2048×2048	**630 ms**	**5,6** (15%)	**23,1** (61%)	**9** (24%)	**37,7 ms**	**16,7**
3936×3936	**2499 ms**	**14,4** (11%)	**90,2** (69%)	**27** (21%)	**131,6 ms**	**18,9**

Tableau 2. Détection des coins et de contours d'image unique (Mém Tex & Partagée)

Images	Temps	GPU (Tex & Shared Mem)				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** (23%)	**6,1** (56%)	**2,2** (20%)	**10,8 ms**	**13,1**
2048×2048	**630 ms**	**4,1** (12%)	**22** (65%)	**8,0** (23%)	**34,1 ms**	**18,5**
3936×3936	**2499 ms**	**8,8** (8%)	**82,5** (73%)	**22** (19%)	**113,3 ms**	**22,1**

Tableau 3. Détection des coins et contours d'images multiples (Mém Tex & Partagée)

Images	Temps	GPU (Tex & Shared Mem)				Acc
Nbr	CPU	Chargement	Kernels	Copie vers CPU	**Temps Total**	
10	**6,00 s**	**0,09** (13%)	**0,45** (63%)	**0,17** (24%)	**0,71 s**	**8,4**
50	**30,3 s**	**0,49** (14%)	**2,13** (60%)	**0,89** (25%)	**3,52 s**	**8,6**
100	**61,5 s**	**0,95** (13%)	**4,24** (59%)	**1,97** (28%)	**7,16 s**	**8,6**
200	**136,1 s**	**2,21** (14%)	**9,00** (58%)	**4,28** (28%)	**15,5 s**	**8,8**

4. Traitement hétérogène d'images multiples

Les implémentations GPU décrites ci-dessus ont permis d'améliorer considérablement les performances de méthodes de traitement d'images individuelles (accélération allant jusqu'à 22×). Toutefois, ces accélérations deviennent moins significatives lors du traitement d'images multiples. Ceci est dû aux coûts de transferts nécessaires pour la sauvegarde des résultats sur la mémoire CPU. Afin d'accélérer ce type de traitements, nous proposons un schéma de développement pour le traitement d'images multiples sur plateformes hétérogènes. Nous décrivons ainsi les stratégies d'ordonnancement employées, permettant une exploitation maximale des ressources (Multi-CPU/Multi-GPU). Cette section est composée de trois parties : la première partie présente le schéma proposé pour les traitements hybrides. La deuxième partie est dévolue

5. Intel. Disponibles sur Internet à `http://opencv.willowgarage.com/wiki/`.

à la description des stratégies d'ordonnancement employées. Une analyse des résultats est décrite dans la troisième partie.

4.1. Schéma proposé pour le traitement hétérogène d'images multiples

Nous proposons dans ce paragraphe un schéma de développement permettant d'exploiter de manière efficace les architectures hétérogènes, offrant une solution plus rapide pour le traitement d'images multiples. Cette solution permet 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 pour la gestion des architectures hétérogènes.

Notons que la librairie StarPU offre un support exécutif unifié pour exploiter les architectures multi-cœurs 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 chargera de lancer les tâches d'exécution sur les données unitaires 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. StarPU propose plusieurs stratégies d'ordonnancement efficaces et offre en outre la possibilité d'en concevoir aisément de nouvelles.

Le schéma 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.

4.1.1. Chargement des images d'entrée

La première étape consiste au 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 1 résume cette étape:

Listing 1 – Chargement des images d'entrée

```
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  Liste = ajouter(Liste, img, img_handle);
6 }
```

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 [6]. Les lignes 3 et 4 permettent d'allouer un tampon (handle) StarPU qui dispose de l'adresse de l'image chargée. La ligne 5 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.

4.1.2. 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 et GPU. 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 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 en utilisant le CPU ou GPU (Listing 3).

Listing 2 – La codelet StarPU

```
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 };
```

La ligne 2 permet de sélectionner toutes les ressources (CPUs et GPUs) pour effectuer notre calcul hétérogène. Les lignes 3 et 4 indiquent les fonctions CPU et GPU respectivement. Les fonctions CPU sont développées via la bibliothèque OpenCV, tandis que les fonctions GPU sont décrites dans la section précédente. La ligne 5 définit le nombre de tampons StarPU utilisés, nous avons choisi un seul tampon puisque chaque tâche traite une image. La ligne 6 permet d'indiquer le modèle de performances utilisé pour ordonnancer les tâches de manière efficace. Ce modèle sera décrit dans la section 4.2.2.

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

```
1 while (Liste != NULL) {
2  task = starpu_task_create();            //Créer la tâche
3  task->cl = &cl;                         //Définir la codelet
4  task->.handle = Liste->img_handle;      //Définir le tampon
5  task->.mode = STARPU_RW;                //Mode Lecture/écriture
6  starpu_task_submit(task);               //Soumettre la tâche
7  Liste = Liste->next;                    //Passer à l'image suivante
8 }
```

6. Intel. Disponibles sur Internet à `http://opencv.willowgarage.com/wiki/`.

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-dessus (Listing 2), ainsi que le tampon StarPU (définit dans le listing 1) contenant l'image courante à traiter. La ligne 5 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).

4.1.3. Présentation des résultats

Lorsque toutes les tâches StarPU sont terminées, les résultats des traitements sur 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.

4.2. Ordonnancement des tâches hétérogènes

L'exploitation des unités de calcul hétérogènes nécessite un ordonnancement de tâches efficace. En effet, les tâches doivent être lancées dans un ordre assurant une exploitation maximale des cœurs CPUs et GPUs multiples. Nous présentons dans ce paragraphe les deux stratégies d'ordonnancement employées : ordonnancement statique, ordonnancement basé sur l'historique.

4.2.1. 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 les tâches sont soumises de manière asynchrone.

4.2.2. Ordonnancement basé sur l'historique des tâches

Pour atteindre un ordonnancement efficace, il est préférable 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 2, ligne 6). Ce modèle, définit 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 4 décrit notre modèle de performances basé sur l'historique des temps d'exécution des tâches.

Dans notre cas, nous employons l'ordonnanceur dmda (deque model data aware) qui prend en compte à la fois le modèle de performances des exécutions et le temps de transfert des données. En effet, il permet d'ordonnancer les tâches de telle sorte

que le temps de calcul total soit minimum. Notons aussi que les tâches sont lancées de manière asynchrone.

Listing 4 – Modèle de performances utilisé pour l'ordonnancement

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

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 (ce nom est utilisé lors de l'appel du modèle de performances depuis la codelet).

4.3. Analyse des résultats

Nous présentons dans ce paragraphe une analyse des résultats obtenus grâce aux implémentations hétérogènes exploitant à la fois les processeurs centraux et graphiques. En effet, le tableau 4 présente une comparaison des accélérations obtenues via les mises en œuvre hybrides de la méthode incluant la détection des coins de de contours. Notons que l'exploitation des multiples CPUs et GPU a permis d'accélérer significativement les performances (19×) par rapport à une mise en œuvre exploitant un seul processeur graphique. 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 (CPUs et GPUs). Ces résultats sont obtenus en utilisant une simple stratégie d'ordonnancement donnant la même priorité à toutes les tâches tel que décrit dans la section 4.2.1.

Le tableau 5 présente les performances obtenues lors de l'utilisation de la stratégie d'ordonnancement basée sur l'historique des tâches précédentes (décrite à la section 4.2.2). Notons que cet ordonnancement nous a permis d'améliorer les performances. En effet, les accélérations peuvent 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, et vice versa pour les tâches à faible intensité de calcul qui seront exécutés sur CPU. Les expérimentations ont été effectuées sous Linux Ubuntu 11.04 (64 bits) utilisant différentes plateformes, *i.e.* CPU Bi-cœurs and GPU Tesla C1060.

– CPU: Bi-cœurs 6600, 2.40 GHz, RAM: 2 Go
– GPU: Tesla C1060, 240 cœurs CUDA, GRAM: 4 Go

La Figure 3 résume les implémentations proposées dans un modèle général permettant un traitement efficace d'images sur architectures parallèles (GPU) et hétérogènes (Multi-CPU/Multi-GPU). Ce schéma permet de traiter de manière efficace les

Tableau 4. Traitement hybride d'images multiples : ordonnancement statique

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 5. Traitement hybride d'images multiples : ordonnancement dynamique

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 ×	10,34 ×	10,53 ×	11,54 ×	11,76 ×	12,24 ×	13,04 ×
50	1,54 ×	03,40 ×	09,99 ×	10,16 ×	12,06 ×	12,89 ×	15,77 ×	16,92 ×
100	1,56 ×	3,42 ×	10,95 ×	11,23 ×	12,43 ×	13,41 ×	16,86 ×	18,99 ×
200	1,54 ×	3,40 ×	10,98 ×	11,33 ×	13,03 ×	14,66 ×	19,20 ×	22,39 ×

images individuelles et multiples. En effet, les images individuelles sont traitées en parallèle via CUDA et visualisées directement avec OpenGL. Par ailleurs, des traitements hybrides sont appliqués lors du traitement d'images multiples en exploitant simultanément les cœurs CPUs et GPUs multiples.

5. Résultats expérimentaux

Cette section décrit l'exploitation des implémentations parallèles (GPU) et hybrides (Multi-CPU/Multi-GPU) dans deux applications différentes: segmentation des vertèbres et indexation de séquences vidéo.

5.1. Calcul hétérogène pour la 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 et de segmenter automatiquement les vertèbres. Une solution a été proposée dans (Benjelloun *et al.*, 2010) basée sur les modèles de forme active (Active Shape Model) (Cootes *et al.*, 1995), (Cootes *et al.*, 1998) suivant plusieurs étapes (Figure 4). Cette méthode est caractérisée par la faible variation du niveau de gris ainsi que le grand volume de données (images multiples) à traiter. A partir de notre modèle proposé ci-dessus (Figure 3), nous appliquons des traitements hétérogènes (Multi-CPU/Multi-GPU) aux étapes les plus intensives de l'application. Ces étapes consistent aux phases de détection des contours et d'extraction des points d'intérêts.

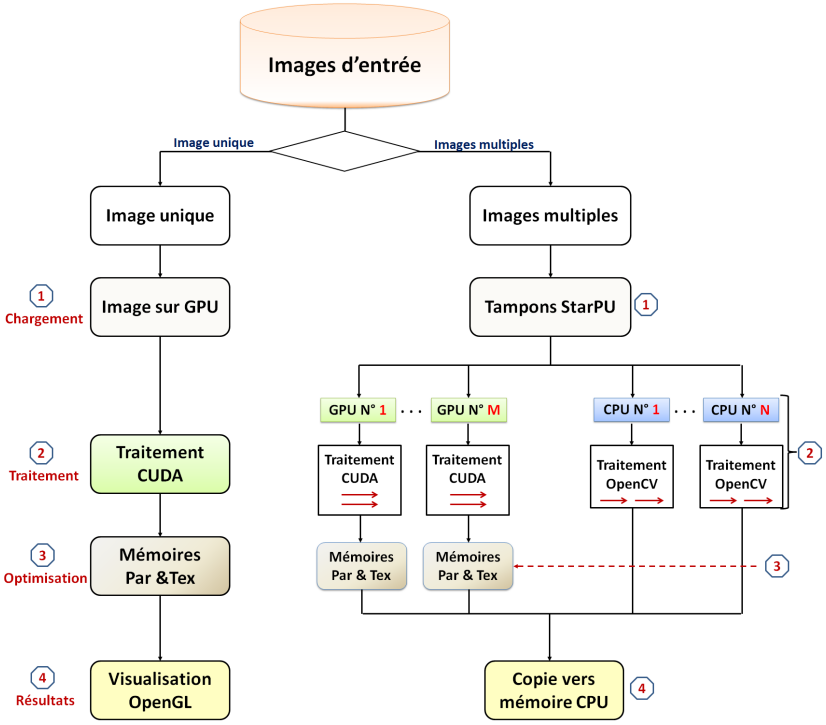


Figure 3. Schéma de développemnt pour le traitement d'images sur architectures parallèles et hétérogènes

La qualité des résultats de segmentation des vertèbres reste identique puisque la procédure est toujours la même. Toutefois, l'exploitation des plateformes parallèles et hétérogènes a permis de diminuer significativement les temps de calcul. Ceci autorise d'appliquer la méthode sur des plus grands ensembles d'images de hautes définitions. Par conséquent, les résultats de l'application médicale peuvent être plus précis (formes de vertèbres détectés). La figure 6(a) montre la comparaison des temps de calcul entre les implémentations séquentielles (CPU), parallèles (GPU) et hybrides des étapes de détection des coins et de contours. La figure 6(b) montre les accélérations obtenues grâces à ces implémentations. Ceux-ci ont permis d'avoir un gain de 50 % (1,5 minute) par rapport au temps total de l'application (3 minutes) utilisant un ensemble de 200 images avec une résolution de 1472x1760.

5.2. Calcul hétérogène pour l'indexation de séquences vidéo

Le but de cette application est de fournir un nouvel environnement d'indexation et de navigation dans des bases de séquences vidéo. Cette méthode 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

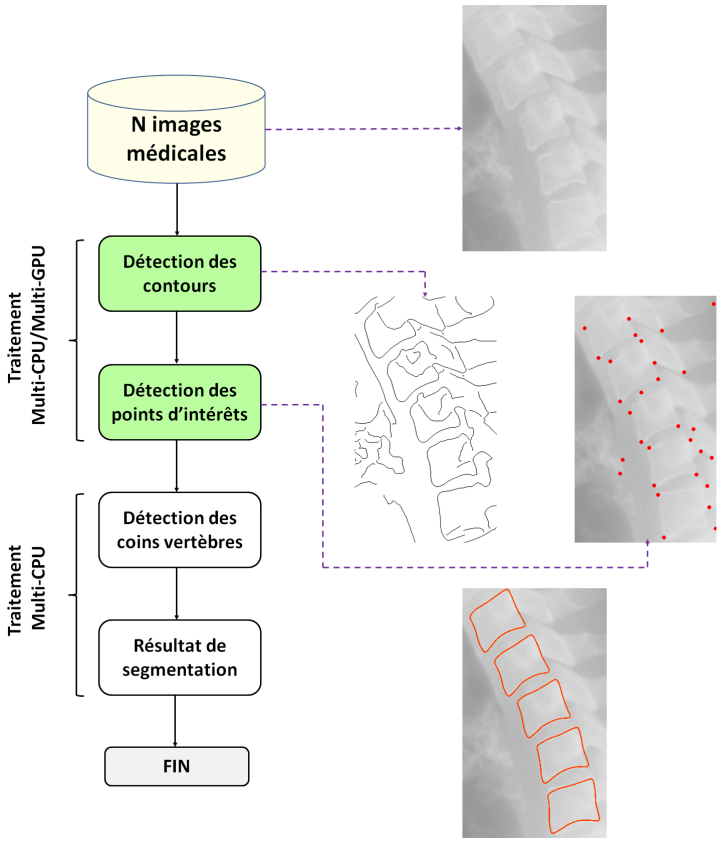
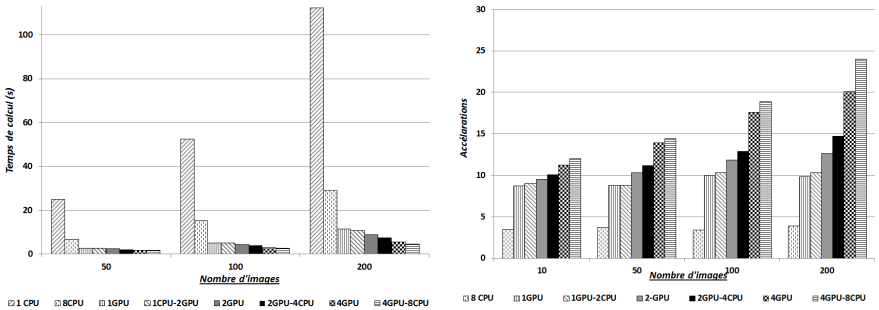


Figure 4. Étapes de segmentation des vertèbres



(a) Temps de détection des coins et de contours (b) Accélérations obtenues

Figure 5. Performances de détection des coins et contours sur architectures hybrides

traitement temps réel et de visualisation à la fin puisque l'objectif est d'extraire des informations permettant une navigation efficace dans des bases de vidéos. Toutefois, cette méthode est entravée par l'augmentation significative du temps de calcul lors de l'utilisation de gros ensembles d'images et de vidéos. A partir du modèle proposé, nous appliquons des traitements hétérogènes à l'étape de calcul la plus consommatrice en temps de calcul, *i.e.* détection des contours. Cette étape permet de fournir des informations pertinentes pour la détection des zones de mouvements qui seront exploitées ensuite via les moments de Hu (Hu, 1962).

La figure 6 montre les contours extraits de manière hybride à partir des trames de la vidéo (vidéo de danse). Cette figure montre aussi les moments de Hu extraits à partir de ces contours. Ces moments sont utilisés pour décrire les zones de mouvement à partir des contours. Notons que notre implémentation hétérogène a permis d'accélérer le calcul de similarité entre séquences vidéo. En effet, la figure 8(a) présente la comparaison des temps de calcul entre les mise en œuvres CPU, GPU et hybrides de la méthode de détection des contours. La figure 8(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.

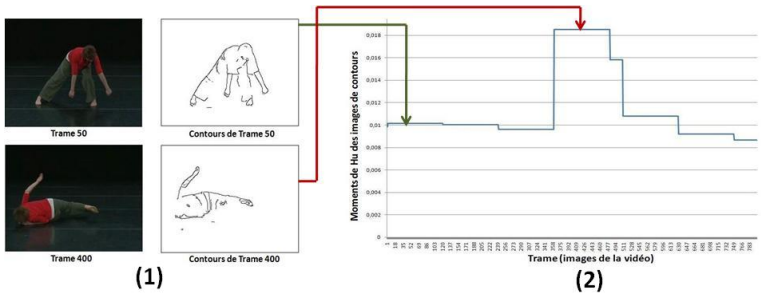
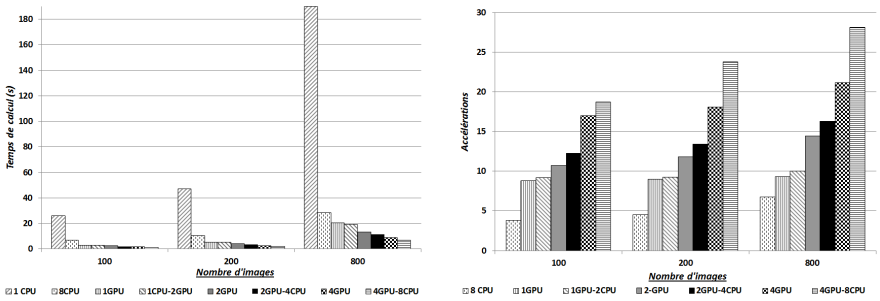


Figure 6. 1. Détection hybride de contours 2. Moments Hu des contours extraits

6. Conclusion

Nous avons proposé dans ce travail un modèle permettant l'exploitation efficace des architectures parallèles (GPU) et hétérogènes (Multi-CPU/Multi-GPU) pour le traitement d'images uniques et multiples. Le modèle a comme objectif d'affecter efficacement les ressources (CPU ou/et GPU) aux tâches traitant les images. Une exploitation du modèle proposé est présentée dans la sixième section dans deux applications : la première consiste à segmenter les vertèbres, tandis que la deuxième permet d'indexer et de naviguer dans des séquences vidéo. Les résultats expérimentaux montrent des gains allant de 5 à 25 par rapport à une implémentation séquentielle sur CPU. Ces accélérations sont dues à quatre facteurs principaux :



(a) Temps de détection hybride des contours (b) Performances de détection des contours

Figure 7. Performances de détection des contours sur architectures hétérogènes

1. **Traitement parallèle intra-image** : ceci consiste à porter de manière efficace des algorithmes d'extraction de caractéristiques (coins et contours) d'images sur GPU. Ce portage est réalisé en appliquant un traitement CUDA parallèle entre pixels. La troisième section décrit en détail ce point.

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. Ce point est décrit en dans la section 4.

3. **Choix efficace des ressources** : ceci consiste à affecter les unités de calcul (CPU ou/ GPU) selon la taille et le nombre d'images à traiter. En effet, lors des traitements hétérogènes (Multi-CPU/Multi-GPU), nous appliquons un ordonnancement basé sur l'estimation des temps de calcul et de transfert requis par chaque tâche. Ceci permet d'augmenter significativement le taux d'occupation des ressources.

4. **Gestion efficace des mémoires GPU** : ceci consiste à réduire les coûts de transferts entre mémoires CPU et GPU. En effet, les implémentations proposées exploitent pleinement les mémoires "partagée et de texture" assurant un accès plus rapide aux pixels d'images.

Comme perspectives, nous envisageons d'étendre le modèle proposé pour traiter les séquences vidéo (bases de vidéos, vidéo à temps réel). Nous envisageons aussi de calculer un facteur de complexité des méthodes appliquées prenant en compte plusieurs paramètres tels que le nombre d'opérations par pixel, ratio entre calcul et accès mémoire, taux de dépendance de tâches, etc. Ceci permet de mieux définir les tâches pouvant bénéficier de la puissance des processeurs graphiques. Une autre perspective de ce travail est de développer une nouvelle stratégie d'ordonnancement au sein des unités de calcul hétérogènes. Cette stratégie devra prendre en compte le facteur de complexité cité ci-dessus afin d'affecter au mieux les tâches à forte intensité aux processeurs graphiques. 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).

Remerciements

Les auteurs tiennent à remercier la Communauté Française de Belgique, au travers du soutien du projet Arc-OLIMP (Optimization for Live Interaction Multimedia Processing), convention AUWB-2008-12. Les auteurs remercient également l'action européenne IC 805 2009-2013.

Bibliographie

AMD. (2011). The future brought to you by amd introducing the amd apu family. *AMD Fusion™ Family of APUs*. Consulté sur http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx/

Antao S., Sousa L. (2010). Exploiting SIMD extensions for linear image processing with OpenCL. *28th IEEE International Conference on Computer Design,*, p. 425–430.

Augonnet C., Thibault S., Namyst R., Wacrenier P.-A. (2009). StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *In concurrency and computation: practice and experience, Euro-Par 2009, best papers issue,*, p. 863–874.

Ayguadé E., Badia R., Igual F.-D., Labarta J., Mayo R., Q-O E. (2009). 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,*, p. 851–862.

Benjelloun M., Mahmoudi S., Lecron F. (2010). 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*, p. 501–506.

Bouguet J.-Y. (2000). Pyramidal implementation of the lucas kanade feature tracker. *Intel corporation microprocessor research labs*.

Canny J. (1986). A computational approach to edge detection. *IEEE transactions on pattern analysis and machine intelligence*, vol. 8, n° 6, p. 679–698.

Cootes T. F., Edwards G. J., Taylor C. J. (1998). Active appearance models. In *5th european conference on computer vision*, p. 484–498. Springer-Verlag.

Cootes T. F., Taylor C. J., Cooper D. H., Graham J. (1995). active shape models-their training and application. *Computer vision and image understanding,*, p. 38–59.

Deriche R. (1987). Using Canny's criteria to derive a recursively implemented optimal edge detector. *Internat. J. vision, Boston*, p. 167–187.

Fung J., Mann S., Aimone C. (2005). Openvidia : parallel gpu computer vision. *In proc of ACM multimedia,*, p. 849–852.

Harris C., Stephens M. (1988). A combined corner and edge detector. *Proceedings of the 4th Alvey vision conference*, vol. 15, p. 147–151.

Heng Y., Gu L. (2005). GPU-based volume rendering for medical image visualization. *Proceedings of the 2005 IEEE engineering in medicine and biology 27th annual conference. Shanghai, China,*, p. 5145–5148.

Hu M.-K. (1962). Visual pattern recognition by moment invariants. *In : IRE transactions on information theory IT-8,*, p. 179-187.

Luo Y., Duraiswan R. (2008). Canny edge detection on nvidia cuda. *Proceedings of the workshop on computer vision on GPUs, CVPR*.

Mahmoudi *et al.* (2010). GPU-based segmentation of cervical vertebra in X-Ray images. *Proceeding of the workshop HPCCE. In conjunction with IEEE Cluster,*, p. 1–8.

Mark W.-R., Glanville R.-S., Akeley K., Kilgard M.-J. (2003). Cg: A system for programming graphics hardware in a C-like language. *ACM transactions on graphics 22,*, p. 896–907.

OpenGL. (2004). OpenGL architecture review board: ARB vertex program. Revision 45. Consulté sur http://oss.sgi.com/projects/ogl-sample/registry/

Schiwietz T., Chang T., Speier P., Westermann R. (2006). MR image reconstruction using the GPU. *Image-guided procedures, and display. proceedings of the SPIE,*, p. 646–655.

Siebert *et al.* (2009). Media Cycle: browsing and performing with sound and image Libraries. *in QPSR of the numediart research program,*, p. 19–22.

Smelyanskiy M., Holmes D., Chhugani J., Larson A., al. (2009). Mapping high-fidelity volume rendering for medical imaging to CPU, GPU and many-core architectures. *IEEE transactions on visualization and computer graphics, 15(6),*, p. 1563–1570.

Yang Z., Zhu Y., Pu Y. (2008). Parallel image processing based on CUDA. *International conference on computer science and software engineering. China,*, p. 198–201.

Reçu le 16/09/2011
Accepté le 21/06/2012

Sidi Ahmed Mahmoudi. *est chercheur-doctorant à l'Université de Mons (Belgique). Après avoir obtenu un diplôme d'ingénieur à l'Université de Tlemcen (Algérie) et un Master de recherche à la Faculté Polytechnique de Tours (France), il a commencé une Thèse sur l'exploitation efficace des plateformes parallèles (GPU) et hétérogènes (Multi-CPU/Multi-GPU) pour l'accélération du traitement d'objets multimédia de hautes définitions*

Pierre Manneback. *est Professeur au Département d'Informatique et Gestion à la Faculté Polytechnique de l'Université de Mons, Belgique. Il y enseigne notamment l'informatique parallèle et l'informatique temps-réel. Il y dirige une équipe de recherche en informatique haute performance, intégrée dans l'Institut de Recherche en Traitement de l'Information et en Science Informatique InforTech. Il a été ou est partenaire de projets de recherche ou réseaux européens et régionaux dans les domaines de l'HPC. Il est expert régulier pour l'évaluation de projets ANR, DoE ou européens*

Cédric Augonnet. *est ingénieur chez NVIDIA, à Santa Clara (USA). Après avoir obtenu une License à l'École Normale Supérieure de Lyon et un Master à l'Université de Bordeaux, il a effectué une Thèse à l'Université de Bordeaux sur la conception de supports exécutifs destinés à l'ordonnancement de tâches sur des machines multicoeurs équipées d'accélérateurs. Ses travaux actuels portent sur la gestion de données et sur les évolutions du modèle de programmation de CUDA*

Samuel Thibault. *est maître de Conférence à l'Université de Bordeaux. Après une thèse en ordonnancement de threads sur machine à organisation hiérarchique, il a développé pendant son post-doctorat chez XenSource des environnements d'exécution virtualisés légers. Enfin, de retour à Bordeaux, il a encadré la thèse de Cédric Augonnet portant sur un support d'éxécution de tâches sur machine hétérogènes*