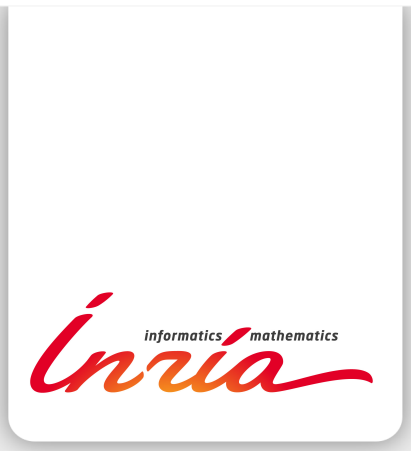


Le modèle de programmation ORWL pour la parallélisation d'une application de suivi vidéo HD sur architecture multi-coeurs

Farouk Mansouri **Jens Gustedt**

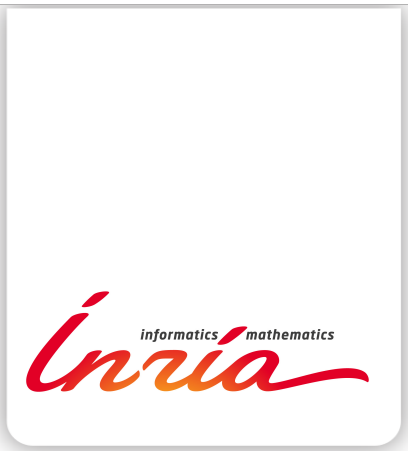
RESEARCH REPORT

N° 8922

June 2016

Project-Teams TADAAM & Camus

ISSN 0249-6399 ISRN INRIA/RR--8922--FR+ENG



Le modèle de programmation ORWL pour la parallélisation d'une application de suivi vidéo HD sur architecture multi-coeurs

Farouk Mansouri Jens Gustedt

Équipes-Projets TADAAM & Camus

Rapport de recherche n° 8922 — June 2016 — 10 pages

Résumé : Grâce à l'évolution des technologies de capture d'image et de vidéo il est possible aujourd'hui de collecter une quantité importante d'information sur le monde observé. En effet, des capteur d'images à résolution HD ou ultra HD peuvent produire plusieurs millions de pixels. Cela permet à des applications de la vidéo surveillance comme le suivi de mobiles de bénéficier de quantité de données supérieure afin de produire de meilleurs résultats. Dans ce contexte les architectures multi-cœurs représentent une bonne solution de calcul. Elles présente des ressources mémoire importantes capables d'accueillir ces données et de les traiter avec les nombreux cœurs les composant. Cependant, pour optimiser leurs performances, le développeur doit gérer plusieurs étapes de programmation. Pour faciliter la programmation de ces architectures, il est possible d'utiliser des modèles de programmation proposant des abstractions sur ces étapes de programmation. Dans cette étude, nous nous intéressons d'implémenter une application de suivi vidéo HD sur une architecture multi-cœurs en utilisant le modèle de programmation à base de tâches ORWL. Ce modèle nous permet de produire une implémentation efficace qui accélère le traitement tout en bénéficiant d'un niveau élevé d'abstraction.

Mots-clés : Modèle de programmation, Suivi vidéo, parallélisme de tâches, parallélisme de données

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

The ORWL programming model for the parallelization on multi-core architectures of a video tracking applcation

Abstract: Due to the evolution of image and video capture technologies it is nowadays possible to collect a large quantity of information about the observed world. Image sensors of HD or ultra HD resolution can provide several millions of pixels. Video tracking applications such as for monitoring moving objects can benefit from that to produce improved results. In this context, multi-core architectures present a valuable solution as computing platforms. They offer large amount of memory to host this data and to treat it simultaneously with all their compute cores. Nevertheless, to optimize their performances, the developer has to deal with several programming phases. To ease the programming of these architectures, it is possible to use programming models that provide abstractions for these phases. With this study, we are implementing a HD video tracking application on a multi-core architecture by using the ORWL programming model. This model allows us to produce an efficient implementation that accelerates the video treatment while at the same time presenting a high level of abstraction.

Key-words: programming model, video tracking, task parallelism, data parallelism

1 Introduction

Les architectures multi-cœurs sont en continuelle évolution. Elle intègrent de plus en plus de cœurs de calcul partageant des ressources mémoire importantes. Ces caractéristiques permettant d'atteindre de grandes performance sont adéquates pour les applications traitant d'importantes quantités de données avec des algorithmes à forte complexité et sous des contraintes de temps restreintes. Les applications du domaine du traitement d'images et de la vidéo haute définition (HD) sont un bon exemple d'application gourmandes en ressources. En effet, les capteurs d'aujourd'hui produisent des images avec un nombre élevé de pixels (16 millions dans une résolution 4K). Cela permet d'obtenir une meilleurs descriptions du monde réel mais induit de traiter de grandes quantités d'information. Les architectures multi-cœurs représentent certes une solution pour ces traitements coûteux en puissance de calcul et en stockage mémoire. Cependant, leur programmation de façon efficace reste une tâche complexe pour les développeurs applicatifs. Pour surmonter cet obstacle plusieurs modèles de programmation ont été conçus et implantés, avec pour missions principales de faciliter la programmation parallèle et d'exploiter les performances des architectures. Dans ce papier nous présentons un travail de recherche visant à étudier l'implémentation d'une application de traitement de vidéo HD sur une architecture multi-cœurs en exploitant un modèle de programmation à base de tâches nommé ORWL. Ce modèle, décrit dans la section 2, propose des abstractions sur la décomposition, la gestion des communications et des synchronisations de tâches. La section 3 comporte une description de l'application de suivi vidéo et de ses algorithmes. Nous présentons ensuite dans la section 4 l'implémentation de cette application avec ORWL, nos contributions et optimisations du modèle et de son environnement de support. Finalement, dans la section 5 nous montrons les résultats obtenus de l'exécution de cette implémentation sur une architecture multi-cœurs.

2 Le modèle de programmation ORWL

Le modèle de programmation ORWL pour "Ordered Read-Write Locks" [4] est un concept de programmation orienté gestion des ressources partagées. En effet, dans un environnement parallèle, les données, les espaces mémoire, les niveaux de cache ou les entrées-sorties sont des ressources partagées par plusieurs processus ou threads de calcul. Dans ce contexte, le modèle propose la modélisation de l'application par des tâches (abstrayant de processus ou de threads) et la gestion des accès occurrents à une ressource par le biais d'une fille FIFO stockant les requêtes (ressource demandée, allouée, libérée) émises par ces tâches. Le manager de la FIFO gère les priorités des requêtes et verrouille la ressource pour certains tâches ou l'affecte en lecture ou en écriture aux tâches adéquats. Ce mécanisme est proposé à l'utilisateur sous la forme d'un environnement de programmation à base de bibliothèque C proposant plusieurs abstractions sur les étapes de programmation. L'utilisateur doit utiliser

des primitives *orwl_task* pour décomposer son application en plusieurs tâches interdépendantes. Les ressources partagées par les tâches sont décrites par les primitives *orwl_location*. Les connexions en lecture ou en écriture des tâches aux ressources sont décrites par des les primitives *orwl_handle*. Ainsi, l'utilisateur n'a pas besoin de manipuler des processus ou threads ni de gérer les synchronisations et les communications entre eux en manipulant des verrous.

En outre, ORWL présente plusieurs propriétés garantissant la cohérence de données, la vivacité d'applications itératives et l'exécution événementielle avec une gestion décentralisée.

3 L'application de suivi vidéo

L'application de suivi vidéo est une application de traitement d'images qui consiste à suivre des objets mobiles dans le temps en utilisant une ou plusieurs caméras. Cette application a connu un grand engouement ces derrières années grâce à la vidéo surveillance pour la sécurité des espaces publiques, le contrôle du trafic ou l'interaction homme-machine. Pour effectuer un suivi des objets mobiles dans une vidéo, plusieurs approches algorithmiques ont été explorées [10].

Dans notre étude, nous nous intéressons à traiter des vidéos haute définition en utilisant l'algorithme de suivi basé sur une détection des mobiles par extraction du fond [15]. Ces étapes sont détaillées dans l'algorithme 1. Cet algorithme permet d'obtenir des résultats intéressants mais reste sensible à la taille des images ce qui limite son utilisation dans un contexte sans stockage "streaming". Pour résoudre cette problématique, nous nous intéressons à l'exploitation du parallélisme dans les architectures multi-cœurs. Notre objectif est d'accélérer l'algorithme et améliorer le débit du traitement vidéo.

4 Implémentation ORWL

Dans cette section nous nous intéressons à implémenter l'application de suivi vidéo décrite précédemment sur une architecture multi-cœurs en se basant sur ORWL. Comme présenté dans la section 2, ce modèle permet de facilement décomposer les applications sous forme de tâches dépendantes et de bénéficier d'abstractions sur la gestion des communications et de la synchronisation des threads. Nous l'utilisons pour produire une implémentation qui accélère le débit

Algorithm 1 Application de Suivi

Input: A video r_vid of size $w \cdot l$
Output: The tracking of each object
1: **for** $r\_im \leftarrow$ image number i of r_vid **do**
2: $r\_fg \leftarrow Foreground\_extraction(r\_im)$
3: $e\_fg \leftarrow Erosion(r\_fg)$
4: $d\_fg \leftarrow Dilatation(e\_fg)$
5: $ccl\_fg \leftarrow Connected\_Compnent\_labeling(d\_fg)$
6: $track\_fg \leftarrow Tracking(d\_fg, r\_im)$
7: **end for**

du traitement de la vidéo en combinant deux types de parallélisme, le parallélisme de tâche en "pipeline" et le parallélisme de donnée en "split-merge".

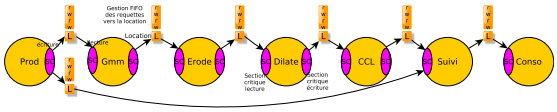


FIGURE 1 – Illutration de l'implémentation DFG de l'application de suivi vidéo avec ORWL

L'application de suivi décrite dans l'algorithme 1 étant une application itérative avec des traitements répétitifs, elle est modélisée sous forme de graphe de flot de donnée (DFG) [8]. Les nœuds du graphe représentent les fonctions de l'algorithme et les arcs représentent les échanges de données entre les fonctions. Cette modélisation permet d'exploiter le parallélisme de tâches où chaque fonction est traitée dés que ses données d'entrée sont disponibles. Comme illustré dans la figure 1, nous implémentons ce modèle dans ORWL en représentant chaque nœud du graphe par une tâche itérative traitant des données d'entrée à chaque itération et produisant des données de sortie. Pour gérer les dépendances entre les tâches dans le modèle ORWL nous utilisons les "locations" et les "handles". Chaque tâche dispose d'une "location" reliée par un "handle" d'écriture pour toute dépendance sortante (données de sortie) et d'un "handle" de lecture pour chaque dépendance entrante (donnée d'entrée) attachée à la "location" de la tâche précédente. A l'exécution, chaque tâche ORWL attend dans des sections critiques de lecture pour récupérer les données d'entrée à partir des "locations" des tâches précédentes. Elle traite ensuite ces données indépendamment des autres tâches puis attend dans ses sections critiques d'écriture pour transférer les données produites dans ses propres "locations". Ainsi, les tâches ORWL sont exécutés en parallèles par différents threads et traitent plusieurs images en exploitant le mode "pipeline".

4.1 Optimisations de l'implémentation ORWL

L'implémentation ORWL de l'algorithme de suivi telle que décrite précédemment permet d'exploiter du parallélisme de tâches en traitant plusieurs itérations en même temps. Cependant, plusieurs limitations la caractérise réduisant ses performances. Parmi ces limitations nous citons : (1) Le non passage à l'échelle causé par la profondeur maximale du pipeline. (2) Les goulots d'étranglement causés par les tâches les plus coûteuses (gmm, ccl). (3) Les temps d'attente liés à la copie mémoire des données dans les sections critiques et à des écriture bloquantes dans certaines dépendances. Dans ce qui suit nous détaillons et présentons des optimisations utilisées pour améliorer cette implémentation.

La primitive split-merge Cette optimisation vise a supprimer les goulots d'étranglement qui ralentissent le pipeline en exploitant le parallélisme de donnée.

Pour cela nous décomposons les tâches les plus coûteuses, à savoir la tâche GMM et CCL en plusieurs sous tâches traitant chacune une partie de l'image.

Cette optimisation permet de réduire le temps de traitement d'une tâche en répartissant le traitement des données sur plusieurs autres cœurs de calcul puis de regrouper les sous résultats vers la tâche principale. Concernant la tâche GMM, cette recomposition ne nécessite pas de traitement particulier. En effet, l'algorithme "Gaussian Mixture-Background Extraction" [16, 17] traitant chaque pixel indépendamment, il est possible de traiter des sous parties de l'image puis de concaténer les résultats pour reconstituer l'avant plan de l'image. En revanche, dans la tâche CCL, l'algorithme "Connected Component Labeling" [7] se base sur un voisinage proche (4,8 pixels) pour détecter les "blobs". Pour ce cas nous avons implémenté un algorithme de recomposition et de mise à jour des "blobs" à partir des blobs détectés dans des parties indépendantes de l'image initiale. Dans la figure 2 nous montrons une illustration des composantes ORWL (location, handle, sous tâches) utilisées pour effectuer la décomposition "split-merge" de la tâche GMM. L'idée principale est que l'image est répartie à travers des sous-tâches en utilisant des "locations" pour synchroniser la lecture/écriture des données.

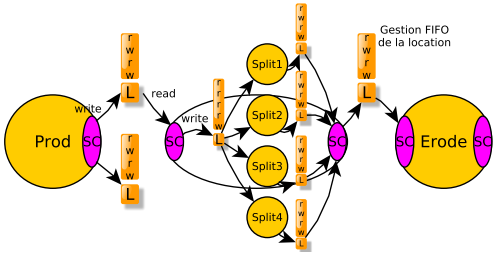


FIGURE 2 – Illustration de l'impléentation de l'optimisation Split-merge dans le modèle ORWL

La file FIFO de données
Dans cette optimisation nous nous intéressons à résoudre la problématique des attentes causés par les écritures bloquantes. En effet, dans le modèle ORWL les lectures sont protégées en verrouillant les écritures ce qui crée des écritures bloquantes. Ces blocages ne génèrent pas de goulet d'étranglement sur des dépendances courtes se trouvant sur le chemin critique car c'est le flux de traitement qui régule le pipeline. Cependant, des dépendances longues en dehors du chemin critique peuvent ralentir le flux du traitement si elles comportent une écriture bloquante. Or, notre application modélisée avec le modèle DFG comporte une dépendance longue *Producteur-Suivi*. Cette dépendance retarde la tâche *Producteur* au rythme de la tâche *Suivi* car son écriture est bloquée sur la lecture de cette dernière.

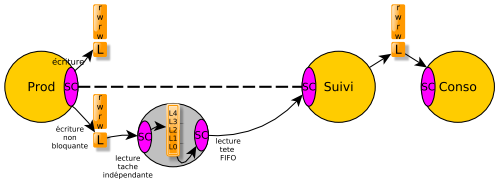


FIGURE 3 – Implémentation de l'optimisation d'écriture FIFO non blocante dans le modèle ORWL

Pour une implémentation plus optimale, la tâche *Producteur* doit s'exécuter répétitivement et sans blocage en écriture pour alimenter le pipeline avec les images à traiter. Pour cela nous introduisons une optimisation basée sur une fille FIFO qui permet de stocker les données produites sans blocage et de poursuivre son traitement. Ainsi, la tâche consommatrice lit la tête de la FIFO à chaque itération. Dans la figure 4 nous illustrons les primitives ORWL utilisées pour implémenter cette optimisation. Une tâche indépendante est insérée dans la dépendance *Producteur-Suivi*. Elle gère une fille FIFO dans laquelle elle stocke les données produites par la tâche *Producteur*. A chaque requête de lecture de la tâche *Suivi*, la tête de la fille est défilée et allouée a cette dernière.

Le multi-buffering et l'échange de pointeurs L'optimisation suivante vise a réduire les temps d'attente des tâches ORWL causés par les copies de données effectuées dans les sections critiques. En effet, le modèle ORWL permet de garantir la cohérence des données par des verrouillages écriture-lecture. Ainsi, les tâches de lecture (consommateurs) ne peuvent lire une donnée tant que la copie effectuée par la tache d'écriture (producteur) ne soit terminée. Inversement, le modèle protège également la donnée non encore copiée par tout les lecteurs (consommateur) contre une écriture par le producteur qui écraserai cette dernière. Cependant, cette protection comporte un coût non négligeable du aux copies mémoires effectuée en concurrence dans les sections critiques qui peux retarder le pipeline. Pour résoudre cette problématique nous adoptons une politique de multi-bufferig en remplaçant la copie de donnée par un échange de pointeur. Ainsi, chaque itération est traitée dans des buffers indépendants et les tâches échangent les pointeurs des buffers via le mécanisme de verrouillage écriture-lecture du modèle ORWL garantissant une cohérence de donnée. Les pointeurs sont alloués progressivement et désalloués à la fin de chaque itération.

5 Expérimentations et résultats

Dans cette section nous présentons les résultats du débit de traitement obtenus par l'exécution de l'implémentation de l'application de suivi avec ORWL et différentes optimisations.

L'échantillon vidéo traité comportent 500 frames d'une résolution de 1280x720 pixels. Le FPS d'une implémentation est calculé comme moyenne de 3 exécutions de l'échantillon. L'architecture multi-cœurs utilisée est composée de 20 nœuds NUMA Intel Xeon CPU E7-8837 cadencés à 2.67GHz contenant 8 cœurs chacun. L'interconnexion entre les sockets est NUMAlink 6 de fabrication SGI avec un débit maximal de 6.7 GB/s. Le débit de suivi obtenu pour une implémentation séquentielle sur un seul cœur est d'environ 7.5 FPS. Avec l'implémentation ORWL en mode "pipeline" optimisée mais sans décomposition des tâches GMM et CCL nous obtenons une accélération d'environ 2.4 x sur 10 cœurs en plaçant une tâche par cœur. En rajoutant différents facteurs de décomposition "split-merge" permettant d'augmenter le nombre de tâches dans l'implémentation "S-M NoBind" nous obtenons une accélération jusqu'à 11.5x pour 36 tâches/cœurs.

Il est à noter que le modèle ORWL ne propose de pas de politique de placement et confie cette opération au système. A titre expérimental, nous avons attaché les threads réalisant les tâches ORWL sur les cœurs de la machine dans l'implémentation "S-M Bind". La politique de placement est naïve (intuitive) est consiste à regrouper par socket les threads qui échange beaucoup de données. Cela permet d'améliorer les résultats pour obtenir jusqu'à environ 30 % comme constaté sur l'implémentation à 28 tâches. Ceci montre un intérêt de pousser cette étude de localité dans le modèle ORWL.

6 État de l'art

Les modèle de programmation à base de tâches comme StarPU [1], TBB data-flow [13] ou OpenMP 4.0 [11] sont un bon compromis entre la complexité des modèles bas niveaux à l'image de Pthread [9], MPI [12] ou CUDA [14] et l'opacité des modèles à directives *pragma* dans OpenMP [3], OpenACC [6] ou PGAS [5]. Ils permettent, d'un coté, de facilement exprimer les applications sous forme de graphe de tâches acyclique et orienté (DAG) en bénéficiant d'abstraction sur les étapes d'implémentation : la décomposition en parties parallèles, la synchronisation, la communications et le placement. D'un autre coté, ils donne assez de contrôle sur l'architecture pour garantir des performances élevées. Le modèle ORWL se positionne dans cette catégorie représentant le calcul parallèle sous forme d'accès concurrents à plusieurs types de ressources. Il propose une gestion décentralisé et événementielle des requêtes d'allocation émises par les tâches. Ce mécanisme permet notamment de modéliser un DAG de taches partageant des buffers (ressources mémoire). En outre, il permet de modéliser un accès partagé à d'autre ressources comme les Caches, les périphériques d'entrée-sortie ou le nombre de cœurs utilisés. A ce titre, il porte une différence par rapport aux modèles à base de tâche cités au début du paragraphe. D'un point de vu exécution, il présente également la caractéristique de traiter un graphe de tâche à taille fixe à l'inverse des outils comme StarPU ou TBB qui génèrent dynamiquement les tâches augmentant la taille du DAG à traiter. Cela fait qu'il est plus adéquat pour modéliser certaines applications statiques comme les DFG

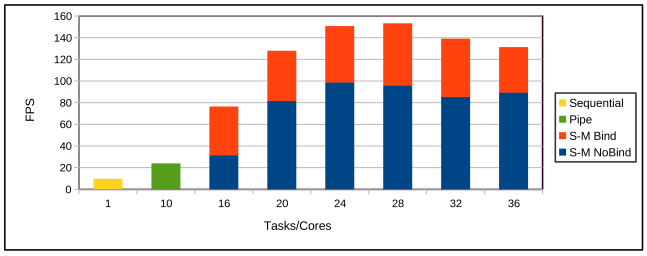


FIGURE 4 – Comparaison FPS des implémentations de l'application de suivi vidéo HD

synchrones et itératifs [8].

7 Conclusion

Dans ce papier nous avons présenté une parallélisation d'une application de suivi vidéo d'objets mobiles dans une vidéo HD sur architecture multi-cœurs. Pour cette implémentation nous avons utilisé le modèle de programmation ORWL proposant une abstraction sur la décomposition en tâches, la synchronisation et la communication entre les tâches. Avec cet environnement nous avons modélisé l'implémentation du modèle graphe de flots de donnée (DFG) représentant notre application. En outre, nous avons introduit plusieurs optimisations permettant de contrecarré des problèmes de ralentissements. Nous avons testé cette implémentation avec différents niveaux d'optimisation ce qui a produit un maximum d'accélération d'environ 14x par rapport à une exécution séquentielle. Pour nos travaux futur, nous nous intéressons à étudier le placement des threads sur les cœurs en se basant sur les caractéristiques de l'architectures récoltées par l'environnement HWLOC [2] et sur les caractéristiques de l'application.

Références

[1] Augonnet (C.), Thibault (S.), Namyst (R.) et Wacrenier (P.-A.). – StarPU : A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. – In *Euro-Par 2009*, Delft, Netherlands, août 2009.

[2] Broquedis (F.), Clet-Ortega (J.), Moreaud (S.), Furmento (N.), Goglin (B.), Mercier (G.), Thibault (S.) et Namyst (R.). – hwloc : A generic framework for managing hardware affinities in HPC applications. – In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pp. 180–186, février 2010.

[3] Chapman (B.), Jost (G.) et Pas (R. v. d.). – *Using OpenMP : Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. – The MIT Press, 2007.

[4] Clauss (P.-N.) et Gustedt (J.). – Iterative computations with ordered read–write locks. *Journal of Parallel and Distributed Computing*, vol. 70, n° 5, 2010, pp. 496 – 504.

[5] De Wael (M.), Marr (S.), De Fraine (B.), Van Cutsem (T.) et De Meuter (W.). – Partitioned global address space languages. *ACM Comput. Surv.*, vol. 47, n° 4, mai 2015, pp. 62 :1–62 :27.

[6] Hart (A.), Ansaloni (R.) et Gray (A.). – Porting and scaling OpenACC applications on massively-parallel, GPU-accelerated supercomputers. *The European Physical Journal Special Topics*, vol. 210, n° 1, 2012, pp. 5–16.

[7] He (L.), Chao (Y.), Suzuki (K.) et Wu (K.). – Fast connected-component labeling. *Pattern Recognition*, vol. 42, n° 9, 2009, pp. 1977 – 1987.

[8] Lee (E. A.) et Messerschmitt (D. G.). – Synchronous data flow. *Proceedings of the IEEE*, vol. 75, n° 9, septembre 1987, pp. 1235–1245.

[9] Nichols (B.), Buttlar (D.) et Farrell (J. P.). – *Pthreads Programming.* – Sebastopol, CA, USA, O'Reilly & Associates, Inc., 1996.

[10] Ojha (S.) et Sakhare (S.). – Image processing techniques for object tracking in video surveillance – a survey. – In *Pervasive Computing (ICPC), 2015 International Conference on*, pp. 1–6, janvier 2015.

[11] OpenMP Architecture Review Board. – *OpenMP Application Program Interface (version 4.0).* – Rapport technique, juillet 2013.

[12] Pacheco (P. S.). – *Parallel Programming with MPI.* – San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 1996.

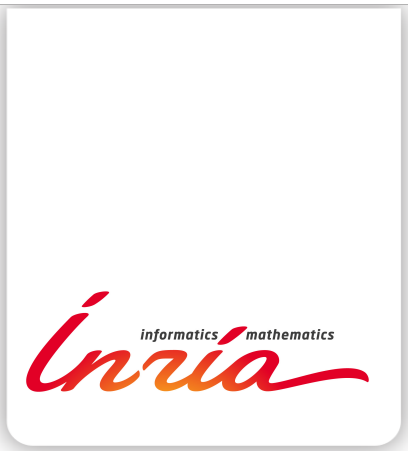
[13] Reinders (J.). – *Intel Threading Building Blocks.* – Sebastopol, CA, USA, O'Reilly & Associates, Inc., 2007, first édition.

[14] Sanders (J.) et Kandrot (E.). – *CUDA by Example : An Introduction to General-Purpose GPU Programming.* – Addison-Wesley Professional, 2010, 1st édition.

[15] Yang (T.), Pan (Q.), Li (J.) et Li (S. Z.). – Real-time multiple objects tracking with occlusion handling in dynamic scenes. – In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05), vol. 1*, pp. 970–975, juin 2005.

[16] Zivkovic (Z.). – Improved adaptive Gaussian mixture model for background subtraction. – In *Proceedings of the 17th International Conference on Pattern Recognition (ICPR 2004), vol. 2*, pp. 28–31, août 2004.

[17] Zivkovic (Z.) et van der Heijden (F.). – Efficient adaptive density estimation per image pixel for the task of background subtraction. *Pattern Recognition Letters*, vol. 27, n° 7, 2006, pp. 773 – 780.



RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399