# Contribution à l'implémentation des algorithmes de vision avec parallélisme massif de données sur les architectures hétérogènes CPU/GPU

Lhoussein Mabrouk

# THÈSE

Pour obtenir le grade de

**DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES**

**préparée dans le cadre d'une cotutelle entre la Communauté Université Grenoble Alpes et L'Université Cadi Ayyad**

Spécialité : **SIGNAL, IMAGE, PAROLE, TÉLÉCOMS**

Arrêté ministériel : le 6 janvier 2005 – 25 mai 2016

Présentée par

**Lhoussein MABROUK**

Thèse dirigée par **Dominique Houzet** et **Said Belkouch**
codirigée par **Sylvain Huet** et **Abdelkrim Hamzaoui**

préparée au sein des **Laboratoires GIPSA-lab** et **LISA**
dans **les Écoles Doctorales EEATS** et **CED-SI**

# Contribution à l'implémentation des algorithmes de vision avec parallélisme massif de données sur les architectures hétérogènes CPU/GPU.

Thèse soutenue publiquement le **17 juillet 2020**,
devant le jury composé de :

**M. Hassan BELAHRACH**
Professeur, ERA Marrakech, Président
**M. Jean-François NEZAN**
Professeur, INSA Rennes, Rapporteur
**M. Rachid LATIF**
Professeur, ENSA Agadir, Rapporteur
**M. Younes JABRANE**
Professeur, ENSA Marrakech, Rapporteur
**M. Serge WEBER**
Professeur, Institut Jean Lamour Nancy, Examinateur
**M. Dominique HOUZET**
Professeur, Grenoble-INP, Directeur
**M. Said BELKOUCH**
Professeur, ENSA Marrakech, Co-directeur
**M. Sylvain HUET**
Maître de conférences, Grenoble-INP, Encadrant

# Remerciements

« Il n'y a pas de magie à accomplir. Il s'agit vraiment de travail acharné, de choix et de persévérance » Dixit Michelle Obama. Il est indéniable que sans eux tout projet quelconque, et particulièrement de recherche, a peu de chances de porter ses fruits. Néanmoins, de tous ceux qui ont eu l'opportunité de vivre cette expérience, aucun récusera le fait qu'en dépit de nos forces et qualités, nombreuses soient-elles, on ne saurait y'arriver sans l'appui de notre entourage professionnel, personnel et familial, tellement ce parcours est tortueux et ardu. Par ailleurs, je mesure l'inestimable apport de cette enrichissante expérience tant sur le plan intellectuel que sur le plan humain, et me réjouis de l'immense fierté d'avoir contribué, aussi peu soit-il, à l'avancement de la recherche académique. C'est donc tout naturellement, qu'à l'achèvement de cette thèse, je témoigne toute ma gratitude à toute personne qui a contribué de près ou de loin à son aboutissement.

Ainsi, j'adresse mes grands remerciements à **M. Said Belkouch**, qui m'a proposé ce sujet en 2016 et qui, en tant que directeur de thèse à l'ENSA de Marrakech, m'a prodigué des conseils judicieux, m'a fourni des outils méthodologiques indispensables à la conduite de cette recherche, et a toujours été disponible et dévoué. Son concours était précieux de la première réflexion à l'organisation du jury de thèse.

Je ne remercierai jamais assez **M. Dominique Houzet** pour la confiance qu'il m'a accordée en acceptant d'encadrer cette recherche dans le cadre de la cotutelle et pour tout l'intérêt qu'il a porté à mes travaux. Ses orientations et encouragements furent fructueux et stimulants. En outre, il m'a permis de découvrir le monde des GPUs et m'a initié à l'aviation, des moments inestimables qui resteront gravés dans ma mémoire à tout jamais.

Aussi, je témoigne une grande reconnaissance à **M. Sylvain Huet** pour l'aide et le temps considérables qu'il m'a consacré pour : m'installer à Grenoble ; toutes les heures qu'il a consacrées à faire avancer cette recherche, à corriger nos publications, le rapport et

la présentation ; sa disponibilité à chaque fois que je sollicitais son aide.

Il va sans dire que **M. Abdelkrim Hamzaoui** a joué un rôle notable tout au long de ces quatre années de ma thèse. Sa vigilance et ses observations de qualité m'ont grandement facilité la tâche. Toujours soucieux de partager ses connaissances et expériences abondantes avec moi. Je me dois donc saluer très vivement son ample contribution à la réussite de ce travail.

Également, je tiens à exprimer mes vifs remerciements à **M. Hassan Belahrach** pour avoir accepté de présider le jury ; à Messieurs **Rachid Latif**, **Jean François Nezan** et **Younes Jabrane** pour m'avoir fait l'honneur d'être rapporteurs, ainsi que pour toutes leurs remarques constructives qui m'ont permis d'améliorer mon rapport final ; et enfin à Messieurs **Serge Weber, Mounir Ghogho** pour leur acceptation d'assister à la présentation de ce travail et d'enrichir la discussion.

Par ailleurs, je témoigne ma gratitude aux équipes pédagogiques **des laboratoire GIPSA et LISA** à l'égard de l'accueil et des conditions de travail privilégiées qui m'ont été réservés ; à **l'équipe systèmes embarquées** à la **fondation Mascir**, l'équipe du **projet MoViTS**, mais également du **projet Heaven** pour avoir financé mes recherches.

A mes **parents**, mes frères et sœurs **(Younes, Redouane, Soukaina, Ali)**, mais aussi à toute ma famille au grand complet, je dis un grand merci pour leur soutien tout au long de cette thèse.

Je voudrais associer à ces remerciements :
**Mes chers amis** en France et au Maroc avec qui j'ai vécu des souvenirs mémorables, qui m'ont permis d'agrémenter ce travail de longue haleine. **Mes collègues de laboratoire** à Marrakech et à Grenoble pour tous les moments d'échange enrichissants, l'esprit d'entraide et de camaraderie. Enfin, **Mme Laurence pierre, M. Philippe Waille** et **M. Denis Bouhineau**, enseignants à l'UGA, avec lesquels j'ai passé une année en tant qu'ATER très riche et passionnante.

**Résumé** — Le mélange de gaussiennes (MoG) et l'acquisition comprimée (CS) sont deux algorithmes utilisés dans de nombreuses applications de traitement d'image et du son. Leur combinaison, CS-MoG, a été récemment utilisée pour la soustraction de fond dans des applications de détection des objets mobiles. Néanmoins, les implémentations de CS-MoG présentées dans la littérature ne profitent pas pleinement de l'évolution des architectures hétérogènes. Ces travaux de thèse proposent deux contributions pour l'implémentation efficace de CS-MoG sur architectures parallèles hétérogènes CPU/GPU. Ces dernières offrent de nos jours une grande flexibilité de programmation permettant d'optimiser les performances ainsi que l'efficacité énergétique. Notre première contribution consiste à offrir le meilleur compromis accélération-précision sur CPU et GPU. La seconde est la proposition d'une nouvelle approche adaptative de partitionnement de données permettant d'exploiter pleinement l'ensemble des CPUs-GPUs. Quelles que soient leurs performances relatives, cette approche, appelée Curseur de Distribution Optimale de Données (ODDC), vise à assurer un équilibrage automatique de la charge de calcul en estimant la proportion optimale des données qui doit être affectée à chaque processeur, en prenant en compte sa capacité de calcul. Cette approche met à jour ce partitionnement en ligne ce qui permet de prendre en compte toute influence de l'irrégularité du contenu des images traitées. En termes d'objets mobiles, nous ciblons principalement les véhicules dont la détection représente un ensemble de défis, mais afin de généraliser notre approche, nous testons également des scènes contenant d'autres types de cibles. Les résultats expérimentaux, sur différentes plateformes et jeux de données, montrent que la combinaison de nos deux contributions permet d'atteindre 98% des performances maximales possibles sur ces plateformes. Ces résultats peuvent être aussi bénéfiques pour d'autres algorithmes où les calculs s'effectuent d'une manière indépendante sur des petits grains de données.

**Mots clés :** Équilibrage adaptatif de charge, calcul haute performance, plates-formes hétérogènes CPU-GPU, acquisition comprimée et soustraction d'arrière-plan MoG.

**Abstract** — Mixture of Gaussians (MoG) and Compressive Sensing (CS) are two common algorithms in many image and audio processing systems. Their combination, CS-MoG, was recently used for mobile objects detecting through background subtraction. However, the implementations of CS-MoG presented in previous works do not take full advantage of the heterogeneous architectures evolution. This thesis proposes two contributions for the efficient implementation of CS-MoG on heterogeneous parallel CPU/GPU architectures. These technologies nowadays offer great programming flexibility, which allows optimizing performance as well as energy efficiency. Our first contribution consists in offering the best acceleration-precision compromise on CPU and GPU. The second is the proposition of a new adaptive approach for data partitioning, allowing to fully exploiting the CPUs-GPUs. Whatever their relative performance, this approach, called the Optimal Data Distribution Cursor (ODDC), aims to ensure automatic balancing of the computational load by estimating the optimal proportion of data that has to be affected to each processor, taking into account its computing capacity. This approach updates the partitioning online, which allows taking into account any influence of the irregularity of the processed images content. In terms of mobile objects, we mainly target vehicles whose detection represents some challenges, but in order to generalize our approach, we test also scenes containing other types of targets. Experimental results, on different platforms and datasets, show that the combination of our two contributions makes it possible to reach 98% of the maximal possible performance on these platforms. These results can also be beneficial for other algorithms where calculations are performed independently on small grains of data.

**Keywords :** Adaptive workload balancing, High performance computing, Heterogeneous CPU-GPU platforms, Compressive sensing and MoG background subtraction.

**ملخص** --- يعتبر مزيج الدوال الغاوسية "MoG"، وكذا الاستشعار الضغطي أو الاستشعار بالإسقاط "CS"، تقنيتان شائعتان في العديد من أنظمة معالجة الصور والصوت. تم دمج هاتين الخوارزميتين مؤخرًا من أجل إزالة الخلفية من الفيديوهات و بالتالي تحديد موضع الأجسام المتحركة داخل المشاهد. ومع ذلك، لم يتم بعد استغلال نتيجة هذا الدمج للاستفادة من تطور أنظمة الحوسبة المتوازية. نقترح، من خلال هذا العمل، استراتيجية فعالة لتشغيل هاتين التقنيتين على منصات الحوسبة غير المتجانسة من نوع "CPU-GPU"، و ذلك من خلال إسهامين. الأول هو ضمان أفضل سرعة ودقة يمكن الوصول إليهما لهذه الخوارزمية الناتجة، بشكل منفصل، على كل وحدة معالجة رقمية. بفضل الفهم و التحليل المعمق لمختلف الأسباب التي تحول دون الوصول إلى هذه الغاية، تمكنا من الحصول على نتائج أكثر فعالية من تلك المنشورة سابقا. مساهمتنا الثانية تكمن في اقتراح تقنية سميناها مؤشر توزيع البيانات الأمثل"ODDC"، وهي طريقة جديدة لتقسيم البيانات بشكل تلقائي و تكيفي من أجل استغلال وحدات المعالجة غير المتجانسة في وقت واحد على أي منصة رقمية. ويهدف هذا المؤشر إلى ضمان موازنة كمية الحسابات تلقائيًا من خلال تقدير حجم البيانات الأمثل الذي يجب تخصيصه لكل وحدة معالجة، مع الأخذ بعين الاعتبار قدرتها الحاسوبية. علاوة على ذلك، تضمن طريقتنا تحديث هذا التقسيم حتى أثناء التشغيل لمراعاة أي اختلال قد يسببه عدم انتظام محتوى البيانات. فيما يخص الأجسام المتحركة، فإننا نستهدف بشكل أساسي العربات التي يشكل تحديد مواقعها بعض التحديات، ولكن من أجل تعميم تقنيتنا، فإننا نختبر أيضًا المشاهد التي تحتوي على أنواع أخرى من الأجسام. تظهر النتائج التجريبية التي قمنا بها، على منصات مختلفة ومجموعات بيانات، أن الجمع بين هاذين الإسهامين يتيح الوصول إلى 98 بالمئة من الأداء القصوي للمنصات المستهدفة. يمكن أن تكون هذه النتائج مفيدة أيضًا للخوارزميات الأخرى التي تعتمد على إجراء العمليات الحسابية بشكل مستقل على جزيئات صغيرة من البيانات.

**الكلمات المفاتيح**: الموازنة التكيفية لكمية الحسابات، الحوسبة عالية الأداء، منصات CPU-GPU غير المتجانسة، الاستشعار بالإسقاط، وإزالة الخلفية باستعمال مزيج الدوال الغاوسية MoG.

# Table des figures

**Table des figures**

# Liste des tableaux

# Table des sigles et acronymes

**AAA** *Adéquation Algorithme Architecture*

**ALU** *Arithmetic–Logic Unit*

**API** *Application Programming Interface*

**ARM** *Advanced RISC Machines*

**AVX** *Advanced Vector Extensions*

**BGS** *Background Subtraction*

**BKGD** *Background*

**CC** *Charge de Calcul*

**CCL** *Connected-Component Labeling*

**CNRST** *Centre National pour la Recherche Scientifique et Technique*

**CPU** *Central Processing Unit*

**CS** *Compressive Sensing*

**CS-MoG** *Compressive Sensing - Mixture of Gaussians*

**CU** *Control Unit*

**CUDA** *Compute Unified Device Architecture*

**CvMoG1** *OpenCV MoG first version*

**CvMoG2** *OpenCV MoG second version*

**DCT** *Discrete Cosine Transform*

**DDC** *Data Distribution Cursor*

**DSP** *Digital Signal Processor*

**DWT** *Discret Wavelet Transform*

**FFT** *Fast Fourier Transform*

**FDPG** *Fonction Densité de Probabilité Gaussienne*

| | |
|---|---|
| **FPGA** | *Field-Programmable Gate Array* |
| **FRGD** | *Foreground* |
| **GCC** | *GNU Compiler Collection* |
| **GMM** | *Gaussian Mixture Model* |
| **GPU** | *Graphics Processing Unit* |
| **HEAVEN** | *HEterogenous Architectures : Versatile Exploitation and programiNg* |
| **HSA** | *Heterogeneous System Architecture* |
| **HW** | *HardWare* |
| **IBM** | *International Business Machines* |
| **ICC** | *Intel C++ Compiler* |
| **ITS** | *Intelligent Transportation Systems* |
| **KDE** | *Kernel Density Estimation* |
| **MaxD** | *Maximal Dynamicity* |
| **MD** | *Mahalanobis Distance* |
| **MedD** | *Medium Dynamicity* |
| **MF** | *Median Filter* |
| **MinD** | *Minimal Dynamicity* |
| **MoG** | *Mixture of Gaussians* |
| **MoViTS** | *Moroccan Video Intelligent Transport System* |
| **NVCC** | *NVIDIA CUDA Compiler* |
| **NVVP** | *NVIDIA Visual Profiler* |
| **ODDC** | *Optimal Data Distribution Cursor* |
| **OpenCL** | *Open Computing Language* |
| **OpenCV** | *Open source Computer Vision library* |
| **OpenMP** | *Open Multi-Processing* |

| | |
|---|---|
| **OS** | *Operating System* |
| **PCA** | *Principal Component Analysis* |
| **PPM** | *Pixel Persistence Map* |
| **PU** | *Processing Unit* |
| **QF** | *Quantification* |
| **RS** | *Relative Speedup* |
| **RT** | *Real Time* |
| **RVB** | *Rouge Vert Bleu* |
| **SIMD** | *Single Instruction Multiple Data* |
| **SM** | *Streaming Multi- processors* |
| **SSE** | *Streaming SIMD Extensions* |
| **SW** | *SoftWare* |
| **TBB** | *Threading Building Blocks* |

# Table des matières

# Introduction générale

L'évolution de la civilisation humaine était liée, depuis l'antiquité, à la capacité de voyager entre des lieux lointains et dispersés en moins de temps. Les moyens de transport primitifs étaient lents et difficiles à utiliser, puisqu'ils reposaient principalement sur l'effort musculaire de l'homme ou d'animaux entraînés. L'évolution des systèmes de transport a commencé juste après l'invention de la roue, il y a environ 5000 ans, à base de laquelle les premiers chariots ont été conçus pour transporter les charges et les personnes. Entre la fin du dix-huitième siècle et le début du dix-neuvième siècle, le premier véhicule à moteur a été inventé [1]. À partir de cette époque, les systèmes de transport ont connu une grande évolution accompagnée d'un développement important des infrastructures.

Aujourd'hui, nous vivons dans des grandes métropoles bondées de véhicules sophistiqués et utilisés dans différentes missions. Ainsi, ces métropoles souffrent de problèmes liés à l'organisation et la gestion de la circulation de ces véhicules. La congestion du trafic, la pollution et le nombre important d'accidents représentent des défis majeurs auxquels des solutions innovantes doivent être proposées afin d'assurer une meilleure mobilité urbaine. Cet aspect a attiré notre attention vue l'augmentation spectaculaire du nombre de véhicules dans le monde ces dernières années. En effet, ce nombre aura augmenté de 2400 unités avant même de finir la lecture de cette introduction (environ 150 unités/minute selon [2]). La réponse à un tel besoin, nécessite l'intégration de nouvelles idées et technologies dans l'environnement existant pour avoir des systèmes de transport plus efficaces. Ces technologies doivent être suffisamment développées pour prévoir, mesurer et gérer les situations critiques liées au trafic routier d'une façon autonome, précise et rapide. Ainsi, l'interaction avec un tel environnement dynamique nécessite le traitement numérique de grandes quantités de données dans de petits intervalles de temps. Ces données représentent souvent des scènes de circulation qu'il faut comprendre et analyser grâce à un dispositif de calcul. Il s'agit alors de traiter automatiquement un ensemble d'images capturées régulièrement sur

des voies routières : nous parlons de la vision par ordinateur ou le traitement d'image.

Indépendamment du trafic routier, le traitement d'image a connu une grande évolution tant au niveau algorithmique que matériel. Quelques années en arrière, cette notion était liée uniquement à l'amélioration esthétique de la photographie. Toutefois, à l'heure actuelle, elle est omniprésente dans des secteurs critiques, à savoir la médecine, la surveillance, la robotique, l'agriculture et bien d'autre. Il suffit de citer l'exemple des caméras de surveillance qui jouent un rôle important dans la sécurité des biens et des individus, les smartphones utilisés par 76% de la population des pays développés en 2018 [3], la nouvelle génération de voitures intelligentes où la vision est composante principale [4], ou même les sondes spatiales et les robots planétaires qui ont des systèmes de vision avancés comme le rover Discovery qui se trouve sur Mars en ce moment [5].

Assurer la sécurité et la flexibilité de la circulation routière ne peut être garanti qu'à travers des algorithmes sophistiqués implémentés sur des calculateurs numériques performants. Ces algorithmes peuvent être classés en trois catégories selon le compromis performance-précision [6], [7]. Dans certains cas, la précision des opérations est plus prioritaire comme dans le cas de de la reconnaissance des plaques d'immatriculation dans des images radar de haute résolution [8]. Dans d'autres cas, la vitesse du traitement est primordiale en particulier quand on est sous la contrainte du temps réel. Prenons comme exemple, une caméra de surveillance qui doit générer une alarme lors de la détection d'une anomalie sur une section d'autoroute [9]. Plus important encore, nous nous intéressons dans cette thèse à l'étude d'un cas intermédiaire, mais considéré plus critique, vu qu'il nécessite à la fois la précision et la vitesse du traitement. C'est le cas des algorithmes de segmentation des scènes routières utilisés pour détecter des zones d'intérêt (régions, véhicules, objets etc.). Cette tâche nécessite l'exécution d'un ensemble d'opérations massives et précises dans des petits intervalles de temps. De ce fait, l'intégration de tels algorithmes dans un système de gestion de trafic nécessite une optimisation et puis une implémentation efficace. Pour cela, notre étude ne concerne pas uniquement l'optimisation algorithmique, mais surtout la méthodologie de portage sur les architectures de calcul hétérogènes CPU/GPU ciblées.

L'analyse d'une scène routière passe généralement par trois étapes [10] : 1) le prétraitement qui concerne les processus de filtrage, d'amélioration et de conversion d'images. 2) la segmentation qui consiste à partitionner une image en sous-ensembles pour pouvoir en extraire les éléments qui la composent (routes, véhicules, objets etc.). 3) l'interprétation qui consiste à exécuter un ensemble de processus cognitifs pour comprendre les événements se passant dans l'image. Le prétraitement est un processus qui s'applique de la même façon sur tous les éléments de l'image indépendamment de son contenu. Ainsi le temps d'exécution de cette étape dépend uniquement de la taille des images traitées. Par contre, les méthodes de segmentation représentent une complexité algorithmique variable qui dépend du contenu de l'image. En ce qui concerne la mobilité urbaine, la segmentation vise généralement à éliminer les zones qui représentent moins d'intérêt. Cela va permettre aux algorithmes d'interprétation de l'image de se focaliser uniquement sur les segments contenant des objets mobiles sur route, et plus particulièrement les véhicules. Nous parlons alors de la détection de véhicules [11]. Le processus de détection de véhicules est commun aux applications de gestion du trafic routier. Néanmoins, sa mise en œuvre n'est pas triviale vu qu'il faut répondre, dans la majorité des cas, aux deux contraintes suivantes :

— La première est liée au temps du traitement. En effet, une caméra transmet généralement plusieurs dizaines d'images brutes de haute résolution qu'il faut segmenter chaque seconde. Ce temps de traitement doit être suffisamment optimisé afin de permettre au système d'exécuter le reste de la chaîne algorithmique tout en répondant à la contrainte du temps réel.

— La deuxième consiste à ce que la détection des véhicules doit être suffisamment précise vue qu'elle se situe à l'entrée de la phase d'interprétation de l'image. Ainsi, si le résultat de détection n'est pas précis, l'interprétation de la scène sera moins fiable.

La réponse à ces contraintes ne peut être garantie qu'à travers trois aspects complémentaires : le bon choix de l'algorithme, le choix d'architecture de calcul et la mise en adéquation de cet algorithme avec l'architecture ciblée. En ce qui concerne le choix de l'algorithme, nous ciblons comme technique pertinente la soustraction de fond (Background Subtraction

[12]), classée dans la troisième catégorie d'algorithmes citée précédemment, et dont le but est de détecter les changements dans les séquences d'images prisent généralement avec une caméra fixe. Cette technique repose sur la représentation de l'arrière-plan d'une image par un modèle pertinent. Grace à ce modèle, on peut continuer facilement la détection tout au long de la séquence d'images. Un état de l'art approfondi sur les méthodes de soustraction de fond les plus connues dans la littérature nous a amené à nous focaliser sur l'implémentation de la méthode MoG [13] : cette méthode représente un bon compromis entre la précision et le temps de traitement. De plus, la technique de l'acquisition comprimée ou Compressive Sensing (CS) [14]) a été proposée pour accélérer la version séquentielle de MoG. Il s'agit de multiplier l'image par une matrice de projection afin de réduire la taille des données traitées tout en conservant le maximum d'informations. Ainsi, CS-MoG est la version la plus accélérée et la plus récente de MoG. Toutefois, CS-MoG n'a pas encore été optimisé pour tirer parti de l'évolution des architectures de calcul parallèle hétérogènes.

Les fabricants de semi-conducteurs proposent depuis quelques années des systèmes à architectures hétérogènes (Heterogeneous System Architecture HSA), afin de répondre aux besoins croissants de puissance de calcul. Ces dispositifs de calcul sont considérés comme les plus dominants et les plus efficaces pour le traitement d'image sur terrain, vu qu'ils disposent de caractéristiques et atouts complémentaires. Ils peuvent contenir des unités centrales de traitement CPU, des processeurs graphiques GPU, ou encore des DSPs et FPGAs. Dans nos travaux, nous ciblons les systèmes contenant CPU et GPU, qui constituent la combinaison la plus efficace et la plus courante de nos jours. Malgré cette évolution technologique d'une part, et la disponibilité d'algorithme suffisamment précis comme CS-MoG d'autre part, l'absence d'une méthode efficace permettant de paralléliser ce type d'algorithmes sur des architectures hétérogènes a motivé la proposition de ce sujet de thèse. L'originalité de notre approche porte sur le fait que dans la majorité des systèmes actuels, seuls les GPU sont exploités pour effectuer des calculs de charge élevée, alors que les CPU restent souvent inactifs pendant ce temps. Ainsi, une exploitation parallèle de ces deux unités de traitement (Processing Units PUs) peut être bénéfique en termes de perfor-

mances. Pour atteindre cet objectif, sans altérer la précision des résultats, nos contributions concernent deux aspects :

1. Le premier consiste à garantir une accélération optimale des blocs (appelés kernels) de CS-MoG sur nos processeurs. Pour CPU, les boucles de calcul sont réparties entre les cœurs physiques en utilisant le multithreading [15]. Cette répartition assure qu'aucun cœur ne reste en repos avant la fin des calculs. De plus, l'exécution à l'intérieur de chaque cœur, est optimisée en utilisant la technique de vectorisation [16]. Il s'agit d'exécuter une même instruction simultanément sur un ensemble d'entrées grâce au fait que les registres de calcul des CPU récents sont suffisamment longs (jusqu'à 512 bit [17]). Pour le GPU, les kernels sont optimisés en jouant, d'une part, sur la mise en place de schémas d'accès aux éléments de données, qui optimisent les accès à la mémoire, et d'autre part, sur l'organisation des threads et le masquage du temps de transfert des données de la mémoire CPU vers GPU. Les résultats obtenus grâce à ces améliorations permettent d'atteindre de meilleurs compromis "précision/performances temporelles" que d'autres implémentations présentées dans la littérature.

2. Notre deuxième contribution est la proposition du Curseur de Distribution Optimale des Données (Optimal Data Distribution Cursor : ODDC), une nouvelle approche de partitionnement adaptatif des données permettant d'exploiter entièrement l'ensemble des éléments de calcul d'une plateforme hétérogène CPU/GPU. Il vise à assurer l'équilibrage automatique de la charge de calcul (CC) en estimant la taille optimale des portions de données qu'il faut attribuer à chaque processeur, en prenant en compte sa capacité de calcul. De plus, notre méthode assure une mise à jour du partitionnement au moment de l'exécution afin de prendre en compte toute influence de l'irrégularité du contenu des données.

La présence d'un cadre de recherche et d'application constitue une autre motivation importante pour la réalisation de cette thèse. En effet, il s'agit d'un partenariat entre deux projets et ainsi d'une cotutelle entre deux universités : le projet MoViTS et L'université

Cadi Ayyad au Maroc, et le projet HEAVEN et l'Université Grenoble Alpes en France.

Ce mémoire de thèse est organisé en 5 chapitres de la façon suivante :

— Le chapitre $I$ est consacré à la présentation détaillée du contexte de cette thèse. Ensuite, une description approfondie de CS-MoG est donnée ainsi qu'un état de l'art sur l'évolution de cette technique. Nous définissons les blocs fonctionnels de cet algorithme et caractérisons leurs structures mathématiques ainsi que la configuration des différents paramètres. Nous détaillons également l'influence de ces paramètres algorithmiques sur la précision des résultats.

— Dans le chapitre $II$ nous nous intéressons à la présentation des architectures hétérogènes CPU/GPU. Nous commençons par la présentation de leurs caractéristiques fonctionnelles, et ensuite nous fournissons des guides pratiques pour l'optimisation d'un algorithme donné sur ces architectures. Un état de l'art détaillé sur les implémentations de cet algorithme CS-MoG sur CPU et GPU est présenté avant de conclure ce chapitre, ainsi que les différentes méthodes proposées pour l'exploitation simultanée de ces deux processeurs.

— le chapitre $III$ est dédié à l'implémentation des différents kernels constituant CS-MoG sur les architectures multi-cœur CPU. Nous décrivons nos optimisations basées sur deux aspects complémentaires : le multithreading et la vectorisation. Nous terminons ce chapitre avec la présentation des résultats obtenus et nous les confrontons avec ceux de l'état de l'art.

— Dans le chapitre $IV$, nous présentons, de la même façon que pour le CPU, notre stratégie d'optimisation de CS-MoG sur les processeurs graphiques GPU. Une comparaison avec l'état de l'art montre et les gains de performance obtenus.

— Nous présentons dans le chapitre $V$ notre approche appelée ODDC pour l'équilibrage des CCs entre CPU et GPU, tout en exploitant les implémentations des chapitres précédents. Nous présentons le modèle mathématique de l'ODDC et sa structure fonctionnelle. Les résultats expérimentaux obtenus sur 4 plateformes et 5 jeux de données (datasets) sont présentés à la fin de ce chapitre.

— Nous finissons ce manuscrit par une conclusion où nous résumons les travaux réalisés dans le cadre de cette thèse et dressons le bilan des résultats obtenus. Un ensemble de perspectives est également présenté.

# Liste des publications

Ce travail de thèse a donné lieu aux publications suivantes :

— Mabrouk, L., Huet, S., Houzet, D., Belkouch, S., Hamzaoui, A., Zennayi, Y. : Efficient adaptive load balancing approach for compressive background subtraction algorithm on heterogeneous CPU–GPU platforms. J Real-Time Image Proc. (2019).

— Mabrouk, L., Huet, S., Houzet, D., Belkouch, S., Hamzaoui, A., Zennayi, Y. : Efficient parallelization of GMM background subtraction algorithm on a multi-core platform for moving objects detection. In : 2018 4th International Conference on Advanced Technologies for Signal and Image Processing (ATSIP). pp. 1–5 (2018) (**Best paper award**).

— Mabrouk, L., Huet, S., Belkouch, S., Houzet, D., Zennayi, Y., Hamzaoui, A. : Performance and Scalability Improvement of GMM Background Segmentation Algorithm on Multi-core Parallel Platforms. In : Proceedings of the 1st International Conference on Electronic Engineering and Renewable Energy. pp. 120–127. Springer, (2018)

— Mabrouk, L., Houzet, D., Huet, S., Belkouch, S., Hamzaoui, A., Zennayi, Y. : Single Core SIMD Parallelization of GMM Background Subtraction Algorithm for Vehicles Detection. In : 2018 IEEE 5th International Congress on Information Science and Technology (CiSt). pp. 308–312 (2018)

Chapitre 1

# Algorithmes de soustraction de fond : contexte & état de l'art.

## Sommaire

## 1.1 Introduction

De nombreuses applications, dans le domaine de la vision par ordinateur et du traitement d'image, ciblent uniquement les informations concernant les changements dans la scène, car les régions d'intérêt d'une image sont souvent des objets mobiles (humains, voitures, texte, etc.). Les caméras de surveillance utilisées dans les domiciles par exemple n'ont pas intérêt à enregistrer de longues heures ou jours de scènes vides, mais uniquement lorsqu'un mouvement est détecté. Cela permet non seulement d'économiser l'espace de stockage multimédia, mais facilite également la localisation du moment de l'anomalie dans la vidéo. En outre, cela offre également la possibilité de déclencher une alarme ou d'envoyer une notification en temps réel au propriétaire. Pour la mobilité urbaine, la prise de décisions en temps réel est essentielle. Les systèmes de surveillance doivent nécessairement être capables de détecter instantanément les véhicules et les anomalies liées à la circulation.

Ce chapitre est dédié à la présentation détaillée du cadre théorique et pratique de cette thèse. Dans un premier temps nous présentons les projets MoViTS et HEAVEN. Puis nous passons à l'état d'art sur les techniques de détection de véhicules utilisant la soustraction de fond. Après, nous détaillons la couche mathématique des différentes phases de CS-MoG et les résultats qu'elles génèrent.

## 1.2 Projets ciblés : MoViTS et HEAVEN

Cette thèse s'inscrit, d'une part, dans le cadre du projet MoViTS (Moroccan Video Intelligent Transport System [10]) visant la gestion de la mobilité urbaine des grandes villes. Il s'agit d'un projet de recherche financé par le Centre National pour la Recherche Scientifique et Technique (CNRTS) au Maroc. Son objectif est de développer un système permettant de planifier un schéma de circulation, d'évaluer, de prévoir et de réguler en temps réel le trafic routier, et enfin de contrôler les infractions. Notre contribution dans le projet consiste à proposer aux développeurs une démarche de portage de ces modules sur une plateforme hétérogène CPU/GPU. Pour cela, nous la validons dans ce travail de thèse

sur le module de soustraction de fond CS-MoG qui fait partie du bloc de détection (voir annexe A.2).

D'autre part, cette thèse est financée par le projet HEAVEN (Heterogenous Architectures : Versatile Exploitation and programming [18]) financé par le LabEx PERSYVAL-Lab [19]. Ce projet a comme objectif de faciliter l'exploitation des architectures hétérogènes aux développeurs d'applications. Cela, en proposant un point d'entrée unique, tel qu'OpenMP, pour cibler des architectures hétérogènes CPU/GPU incluant également des FPGA. L'unicité du point d'entrée permet de répondre à des besoins de portabilité et d'accessibilité aux accélérateurs matériels. L'approche proposée sera validée sur des applications concrètes comme les applications de traitement d'image. Le GIPSA-Lab, dans lequel j'ai préparé cette thèse, a pour mission de proposer des applications de traitement d'image, plus particulièrement, l'implantation d'applications de vision embarquées, permettant de démontrer la faisabilité et la pertinence de l'approche (voir présentation détaillée dans annexe A.1 et A.2.4). Comme cas d'étude, nous ciblons l'algorithme de soustraction de fond avec acquisition comprimée CS-MoG.

## 1.3 Algorithmes de soustraction de fond : présentation et état de l'art

De nombreuses études sur la détection de véhicules ont été réalisées. Une partie se base sur des techniques avancées comme l'apprentissage profond (Deep Learning) [20]. Malgré la grande précision apportée par cette technique dans des scènes complexes, sa CC importante nécessite des systèmes HW très puissants. La méthode proposée dans [21], par exemple, nécessite une carte GPU très puissante (GeForce RTX 2080Ti) pour pouvoir tourner en temps réel. De plus, cette charge peut représenter un vrai obstacle quand on veut suivre les positions des véhicules dans une séquence d'images pour pouvoir estimer leurs vitesses ou leurs trajectoires. Il est alors nécessaire de viser des alternatives plus rapides surtout quand l'application en question est susceptible d'être implémentée sur un système embarqué.

### 1.3.1 Notion de soustraction de fond et contraintes de précision

La soustraction de l'arrière-plan (Background Subtraction [12]), appelée également la soustraction de fond ou la détection du premier plan, est l'une des techniques les plus utilisées dans le domaine de la vision par ordinateur pour détecter toutes les modifications significatives dans les séquences d'images. Pour les applications de détection des objets mobiles, cette technique intervient après l'étape de prétraitement de l'image (débruitage, conversion, etc) afin de permettre au premier plan d'une scène d'être extrait (Fig. 1.1) pour un traitement ultérieur (reconnaissance d'objet, etc). Cette technique est plus efficace quand les séquences d'images sont enregistrées avec une caméra fixe, qui permet d'avoir un arrière-plan statique. Elle repose sur la modélisation mathématique, statistique ou structurelle de l'arrière-plan. La structure de paramètres utilisée pour décrire l'évolution des segments pseudo-statiques dans la scène est appelée "modèle d'arrière-plan". Cette modélisation est triviale quand l'arrière-plan est complétement statique dans le temps, mais en réalité elle peut s'avérer très difficile pour les deux raisons suivantes :

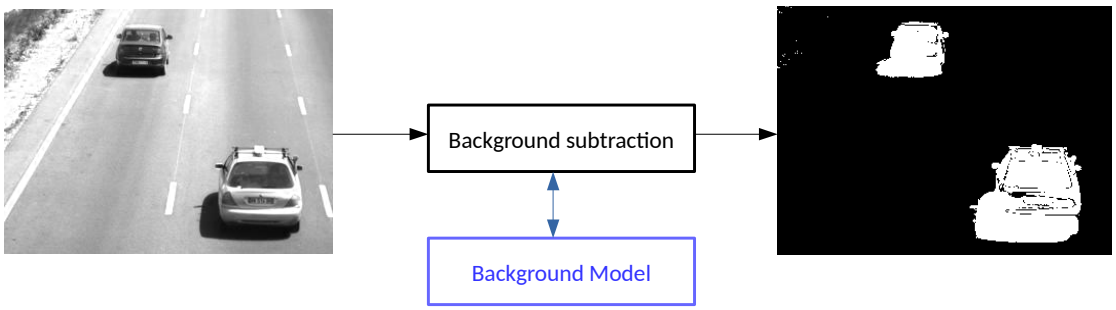


FIGURE 1.1 – Illustration de la détection des véhicules via la technique de soustraction de fond.

1. La présence de changements brusques ou de phénomènes vibratoires dans l'arrière-plan (feuilles d'arbre, bruits, vagues etc).
2. Les changements au long-terme qu'il faut prendre en compte comme, la variation de l'intensité de la lumière dans la journée et le mouvement d'ombre.

### 1.3.2 Algorithmes communs

Les techniques de soustraction de l'arrière-plan ont connu une évolution importante depuis une vingtaine d'années, et ainsi plusieurs méthodes ont été développées. Des revues sur les techniques les plus utilisées sont présentées dans [22], [12] et [23]. Selon [24], elles peuvent être classées en 8 catégories, en se basant sur la nature du modèle de de l'arrière-plan :

1. **Modélisations élémentaires** : il s'agit de techniques où le modèle de l'arrière-plan est simplement une image [25], une moyenne arithmétique [26], une médiane [27] [28], une analyse de l'évolution de l'histogramme dans le temps [29], ou une estimation de matrice à rang réduit [30].
2. **Estimation de fond** : le fond est estimé en utilisant un filtre. Un pixel de l'image capturée qui s'écarte considérablement de sa valeur prévue est détecté comme premier plan. Ça peut être un filtre de Wiener [31], un filtre de Kalman [32] ou un filtre de Tchebychev [33].
3. **Modélisations floues** : l'arrière-plan est modélisé à l'aide d'une moyenne mobile floue [34] ou un mélange flou de gaussiennes [35]. La détection de premier plan est faite en utilisant l'intégrale de Sugeno [36] ou l'intégrale de Choquet [37]. La détection de premier plan peut être effectuée aussi par des d'inférence floues [38].
4. **Apprentissage profond** : l'arrière-plan est représenté par un réseau de neurones qui apprend à classer chaque pixel en arrière-plan ou au premier plan [39].
5. **Modélisation par ondelettes** : le modèle de fond est défini dans le domaine temporel, en utilisant les coefficients de Transformée en Ondelettes Discrète (DWT) [40].
6. **Structures de paramètres** : des techniques utilisant des structures de paramètres, non statistiques, appelé clusters. Le modèle d'arrière-plan suppose que chaque pixel pourra être représenté temporellement par un ensemble de clusters avec lesquels il est comparé après chaque nouvelle image capturée. Pour cela, ces techniques utilisent des algorithmes comme Codebook [41] ou Vibe [42].

7. **Modélisation de signaux** : la soustraction d'arrière-plan peut être formulée en tant que problème d'estimation d'erreur dispersée [43].
8. **Modélisations statistiques** : l'arrière-plan est modélisé à l'aide d'une estimation de la densité du noyau (Kernel Density Estimation KDE [44]), des classes comme K-mean [45], d'une seule composante gaussienne [46], ou en utilisant **un mélange de gaussiennes (MoG)** [47] [13] [48].

Bien que ces méthodes de soustraction de l'arrière-plan récentes offrent une précision satisfaisante, l'utilisation de modèles complexes implique des CC élevées. Un algorithme efficace doit être précis et présenter une faible charge de calcul. Selon [22] et [24], la technique du Mélange de Gaussiennes (Mixture of Gaussians MoG) [13] [47] est un bon candidat pour faire face à ces contraintes, vu qu'il représente le meilleur compromis entre la précision et la performance. En effet, sa robustesse par rapport aux autres méthodes, ainsi que sa rapidité encore augmentée après l'intégration d'une phase préalable de Compressive Sensing [14]) a fait de MoG une bonne cible algorithmique pour cette thèse.

Il s'agit d'une approche probabiliste pour représenter des sous-populations réparties selon la loi normale [49], au sein d'une population globale. Tout comme la convolution et la transformée de Fourier rapide, MoG est aujourd'hui considéré comme l'un des outils les plus répandus dans le domaine du traitement du signal. Il a été introduit dans le traitement d'image par Friedman et Russell [47] et étendu par Stauffer et Grimson [13] pour la tâche de soustraction de l'arrière-plan. Ceci est réalisé en représentant chaque pixel de l'image, dans le modèle d'arrière-plan, par un mélange de composantes gaussiennes (appelées Fonction Densité de Probabilité Gaussienne FDPG). Dans [50], les auteurs ont décrit et comparé 29 méthodes de soustraction de fond différentes avec des vidéos synthétiques et réelles. L'analyse comprenait des méthodes de base allant de la différenciation d'images à des méthodes plus avancées utilisant plusieurs gaussiennes, des méthodes non paramétriques, des méthodes neuronales, etc. Les résultats ont montré que le mélange de gaussiennes figure parmi les cinq meilleures méthodes qui peuvent être utilisées pour la soustraction d'arrière-plan. Dans La Fig. 1.2, [6] positionne MoG dans l'espace performance-précision par rapport

aux algorithmes de soustraction d'arrière-plan les plus connus. Néanmoins, cette technique nécessite toujours des capacités de calcul élevées pour répondre à la contrainte du temps réel, en particulier lorsqu'il s'agit de systèmes embarqués ayant des capacités limitées.

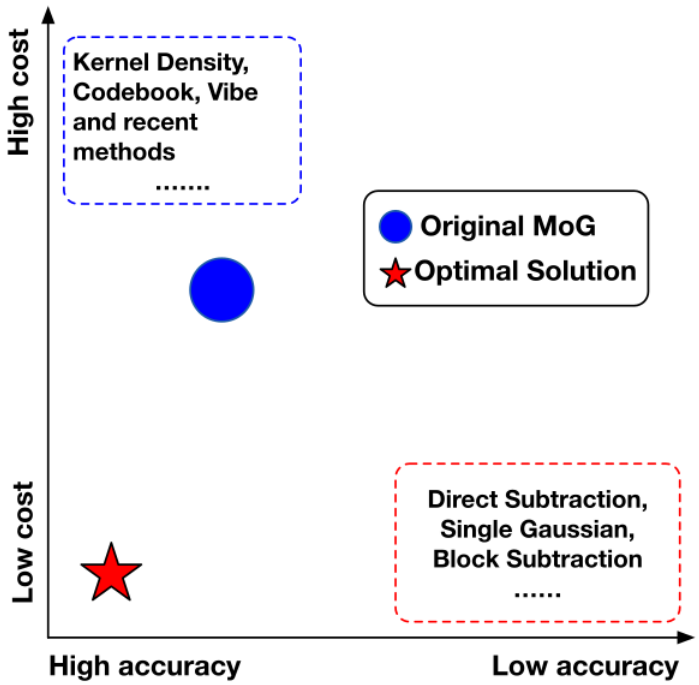


FIGURE 1.2 – Répartition des algorithmes de soustraction de fond dans l'espace précision-complexité algorithmique.

### 1.3.3 Mélange de gaussiennes MoG

Cette méthode statistique est basée sur un mécanisme simple qui détermine quels échantillons sont les plus présents dans l'arrière-plan pour un élément spatial donné. Il peut être appliqué directement aux pixels de l'image ou aux données réduites après une acquisition comprimée, comme dans le cas de nos travaux. Chaque échantillon du modèle d'arrière-plan est représenté par un mélange de $K$ FDPGs (Fig. 1.3). Selon [51] et [6], le choix de ce nombre $K$ entre 3 et 5 est suffisant pour représenter 99% des phénomènes de l'arrière-plan.

Chacune de ces fonctions est caractérisée par trois paramètres : une moyenne $\mu$, une variance $\sigma$ et un poids $\omega$ qui représente la persistance de cette fonction.

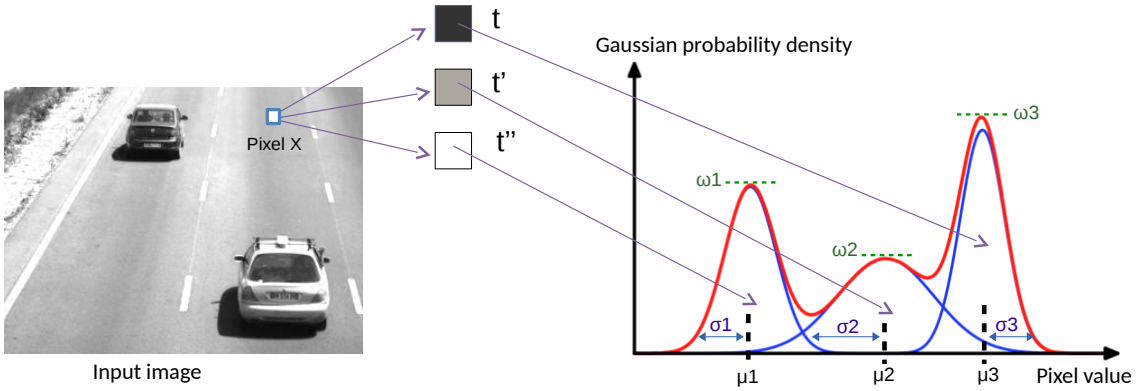


FIGURE 1.3 – Positionnement de chaque pixel de l'image par un mélange de FDPGs. Chaque fonction représente une gamme d'intensités de ce pixel dans des instants différents (t, t' etc.).

Pour un pixel $X$ donné, une FDPG $\eta(X, \mu, \sigma)$ est représentée par l'Eq. 1.1. suivante :

$$\eta(X, \mu, \sigma) = \frac{1}{\sigma\sqrt{(2\pi)}} e^{-0.5(\frac{X-\mu}{\sigma})^2} \tag{1.1}$$

Pour chaque image, MoG passe par trois étapes de calcul pour chaque pixel : (1) le calcul des correspondances, (2) la segmentation de l'arrière-plan et (3) la mise à jour du modèle.

#### 1.3.3.1 Calcul des correspondances

Cette étape consiste à calculer la correspondance entre chaque valeur du pixel et les FDPGs du modèle d'arrière-plan. Un pixel est considéré comme un candidat d'arrière-plan si sa valeur est comprise dans les 2,5 écart-type de l'une des $K$ FDPG constituant MoG [47]. Ceci est équivalent à la distance de Mahalanobis (MD) calculée selon l'Eq. 1.2, dans le cas des images en niveaux de gris :

$$MD(X_t, C_k) = |(X_t - \mu_k)| - 2.5\sigma_k, \tag{1.2}$$

où $X_t$ est la valeur du pixel actuel et $C_k$ la $k^{me}$ FDPG. Ce processus est répété jusqu'à trouver une correspondance, c'est à dire lorsque $MD <= 0$. Il est très rare qu'un pixel valide cette condition d'appartenance pour plus d'une FDPG, mais lorsque c'est le cas,

seule celle dont le poids est le plus élevé sera prise en compte. Pratiquement, cette grandeur est utilisée pour classer les FDPG d'arrière-plan dans le modèle de la plus persistante à la moins persistante. Cela permet de minimiser le nombre de comparaisons pour chaque nouveau pixel.

#### 1.3.3.2 Soustraction de l'arrière-plan

Il reste à vérifier si la FDPG correspondante au pixel fait partie de l'arrière-plan, ou si elle représente un objet temporaire. Le nombre de FDPGs représentant l'arrière-plan parmi les $K$ distributions utilisées est calculé selon l'Eq. 1.3 :

$$N_b = \underset{n\in\{1...K\}}{min} / \sum_{k=1}^{n} \omega_k > T_{hb}, \tag{1.3}$$

où $T_{hb} \in [0, 1]$ est la portion minimale de cohérence de l'arrière-plan. Lorsque nous diminuons cette valeur, nous imposons que l'arrière-plan soit représenté par moins de FDPGs. Ainsi, la condition représentée dans l'Eq. 1.3 ne peut être satisfaite que par la FDPG la plus pertinente. Par conséquent, tous les objets représentés par les autres FDPGs seront considérés comme des objets mobiles même s'ils ne le sont pas (comme les nuages, etc.), ce qui réduit la précision des résultats. Dans le cas contraire, si nous augmentons la valeur de $T_{hb}$, la majorité des FDPGs seront affectés à l'arrière-plan même s'ils représentent des objets mobiles. Nos tests montrent qu'une valeur de $T_{hb}$ comprise entre 0,5 et 0,7 donne la meilleure précision possible. Cette équation signifie que les $N_b$ premières distributions sont choisies comme modèle de l'arrière-plan. Par conséquent, une valeur de pixel située dans 2,5 $\sigma$ de ces $N_b$ distributions sera marquée comme arrière-plan.

#### 1.3.3.3 Mise à jour du modèle

Grâce au mélange de FDPGs, MoG peut efficacement gérer plusieurs scénarios d'arrière-plan. Dans le processus de mise à jour, les paramètres de la FDPG correspondant au pixel $X_t$ sont mis à jour selon les équations récursives suivantes :

$$\omega_{k,t} = (1 - \alpha).\omega_{k,t-1} + \alpha, \tag{1.4}$$

$$\mu_{k,t} = (1-\rho).\mu_{k,t-1} + \rho.X_t, \tag{1.5}$$

$$\sigma^2_{k,t} = (1-\rho).\sigma^2_{k,t-1} + \rho.(X_t - \mu_{k,t})^2, \tag{1.6}$$

où $\alpha$ et $\rho$ sont respectivement le facteur de mise à jour et le facteur d'apprentissage. D'après [51], les valeurs de ces deux paramètres sont fixées différemment pour chaque scène (ordre de grandeur de $10^{-4}$). La moyenne et la variance des FDPGs inactives restent intactes et leurs poids sont légèrement réduits selon l'Eq. 1.7 :

$$\omega_{k,t} = (1-\alpha).\omega_{k,t-1}. \tag{1.7}$$

Cette mise à jour permet de gérer plus efficacement les changements liés aux bruits et à l'éclairage. Lorsqu'aucune correspondance n'est trouvée, une nouvelle FDPG est initialisée avec la valeur la plus récente du pixel, et la FDPG de poids le plus faible est remplacée. Sa variance $InitVar$ et son poids $InitWeight$ sont fixés manuellement pour chaque séquence d'images, automatiquement grâce à l'algorithme appelé Expectation Maximization [52] :

$$\mu_{k,t} = X(t). \tag{1.8}$$

$$\sigma_{k,t} = InitVar. \tag{1.9}$$

$$\omega_{k,t} = InitWeight. \tag{1.10}$$

Un pseudo-code de cet algorithme est fourni dans la section 1.3.6

### 1.3.4 Principales améliorations de MoG : l'acquisition comprimée

Plusieurs propositions d'optimisations algorithmiques ont été apportées au MoG, soit pour augmenter sa précision, soit pour gagner en performance. Ces améliorations peuvent être classées en cinq catégories selon la stratégie adoptée :

— **Améliorations intrinsèques** : il s'agit des techniques qui consistent à être plus rigoureuses au sens statistique, ou à introduire des contraintes spatiales et/ou tem-

porelles dans les différentes étapes de modélisation. Par exemple, certains auteurs, [53][54], proposent de déterminer automatiquement et dynamiquement le nombre de FDPGs les plus robustes aux arrière-plans dynamiques. D'autres approches agissent au niveau de l'initialisation des paramètres MoG [55] [56]. Pour la maintenance du modèle, les vitesses d'apprentissage sont mieux définies [57] ou adaptées dans le temps [58]. Quant à la détection de premier plan, l'amélioration des résultats est obtenue en utilisant soit : une mesure différente pour le test de correspondance [59][60], une carte de persistance des pixels (PPM) [61], des probabilités [62], un modèle de premier plan [58][63] ou en utilisant le modèle le plus dominant en arrière-plan [64]. Pour la granularité, les approches par blocs [65] ou par groupes [66] sont plus robustes que celles par pixels. Pour le type d'entité, plusieurs entités sont utilisées à la place de l'espace RVB, comme des espaces de couleurs différents [67][68], des fonctions de bord [69], des fonctions de texture [70], des fonctions stéréo [71], des fonctions spatiales [72] et des fonctions de mouvement. Zheng et al. [73] combinent plusieurs caractéristiques telles que la luminosité, la chromaticité et les informations de voisinage. Les brevets récents concernent les approches par blocs [74], les caractéristiques de texture [75], les fonctions de mouvement [76] et les caractéristiques spatiales [77].

— **Améliorations extrinsèques** : une autre façon d'améliorer l'efficacité et la robustesse du MoG consiste à utiliser des stratégies externes. Certains auteurs ont utilisé des champs aléatoires de Markov [78], des approches hiérarchiques [79], des approches à plusieurs niveaux [80], des arrière-plans multiples [81], ou un post-traitement spécifique [82].

— **Amélioration Hybride** : toutes les améliorations précédentes concernent directement le MoG original. Une autre manière d'améliorer cette méthode consiste à améliorer les résultats de la détection de premier plan en utilisant une coopération avec une autre méthode de segmentation. Elle est obtenue par une coopération avec une segmentation de couleur [65] ou une détection de mouvement par région [83]. D'autres auteurs ont utilisé une coopération avec flux optique [84], couplage de blocs

[85], modèles prédictifs [86], modèles de texture [87], différence d'images consécutives [88]-[90]. Un brevet récent est attribué pour la coopération avec les statistiques d'histogramme [91].

— **Amélioration du temps de traitement** : toutes les améliorations intrinsèques et extrinsèques concernent la qualité de la détection du premier plan. Des améliorations cherchant à réduire le temps de calcul ont également été proposées. Pour le faire, on utilise des régions d'intérêt [92], un taux d'adaptation variable [93], un changement de modèle de fond [94] et des stratégies d'échantillonnage spatial [95], ou en utilisant une implémentation matérielle [96][97][94].

— **Compression des images** : Shen et al. [6] proposent une implémentation de MoG sur des réseaux de caméras embarquées utilisant le CS [14] (cf. section 1.3.4.1). C'est une technique qui permet à des périphériques disposant de peu de ressources de calcul d'effectuer des tâches complexes telles que l'acquisition et la reconstruction de signaux multimédias [98], [99]. Elle a ensuite été exploitée par [6] pour accélérer la version séquentielle de MoG. Cela se fait en réduisant la dimensionnalité des données via des projections aléatoires appliquées sur les images traitées. De cette manière, MoG n'est pas appliqué directement sur tous les pixels, ce qui a permis d'avoir une version séquentielle 6 fois plus rapide, pour un facteur de compression égale à 8, appelée CS-MoG, tout en préservant la même précision que la version originale. Cette implémentation est réalisée sur une plateforme embarquées de type PandaBoard (PandaBoard ES Rev B1) connectée à une caméra vidéo USB (webcam Logitech HD Pro C920).

Afin de comparer les versions améliorées de MoG, nous présentons dans la Table. 1.1, la précision et le temps de traitement des trois versions les plus communes dans la littérature. Ces mesures ont été tirées à partir des résultats donnés dans [6] et [53] en comparaison avec [13].

La table montre que la fusion de MoG et de CS donne le meilleur compromis précision/temps de calcul, cependant cette fusion n'a pas encore été exploitée pour tirer parti

| | MoG [13] | MoG amélioré[53] | CSMoG [6] |
|---|---|---|---|
| **Processing time (ms)** | 339 | 281 | 56.8 |
| **Detction/False-Alarm** | 1.22 | 1.26 | 1.18 |

TABLE 1.1 – Précision et temps du traitement de 100 images consécutives avec une résolution de 320x240 en utilisant différentes versions de MoG.

de l'évolution des systèmes parallèles hétérogènes CPU/GPU. Seul MoG sans CS a fait l'objet de certaines implémentations sur ces systèmes, (cf. chapitre 3 et 4).

#### 1.3.4.1 L'acquisition comprimée (Compressive Sensing)

La "détection par compression", également connue sous le nom de "l'échantillonnage compressé" ou "l'acquisition comprimée" est une technique utilisée pour trouver la solution la plus parcimonieuse pour représenter un système linéaire ou un ensemble de données. En traitement de l'image, elle est utilisée pour représenter les pixels par un plus petit nombre de valeurs contenant la majorité des informations. Ceci est réalisé directement par certains capteurs vidéo récents [100] (Fig. 1.4) ou par calcul après l'acquisition des images.

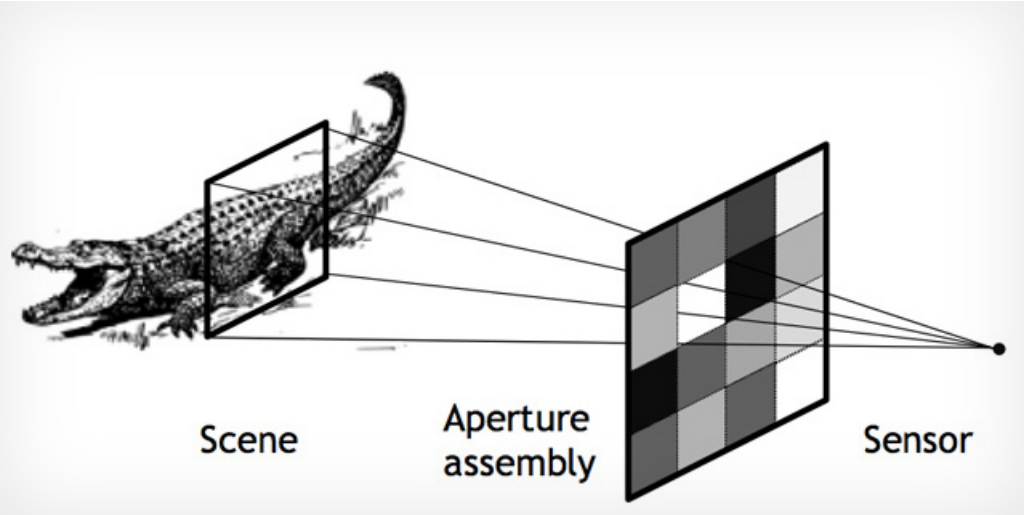


FIGURE 1.4 – Principe du CS par capteur vidéo : représentation de chaque région de la scène par un seul élément.

Il existe actuellement plusieurs méthodes pour réaliser l'acquisition comprimée. Parmi les plus courantes, on trouve l'Analyse en Composantes Principales (PCA) [101] qui s'appuie sur une transformation orthogonale pour convertir un ensemble d'observations en un

petit ensemble de valeurs. Une autre méthode largement utilisée est la Transformée en Cosinus Discrète (DCT) [102], qui exprime une séquence finie de points de données en termes de somme de fonctions cosinus oscillant à différentes fréquences. Il existe également des méthodes qui utilisent des mécanismes plus simples tels que la représentation de blocs de pixels (zones) par des sous-ensembles de valeurs correspondant à des pixels choisis au hasard ou à un vecteur de moyennes. Une méthode plus efficace consiste à générer des valeurs représentatives appelées projections, grâce à la multiplication de zones de pixels par une matrice de Bernoulli. Selon l'étude réalisée dans [6], cette approche surpasse toutes les méthodes de compression mentionnées ci-dessus lorsqu'elle est utilisée avec MoG. En effet, les projections générées nécessitent moins d'FDPGs dans le modèle d'arrière-plan pour obtenir des résultats précis (Fig. 1.5).

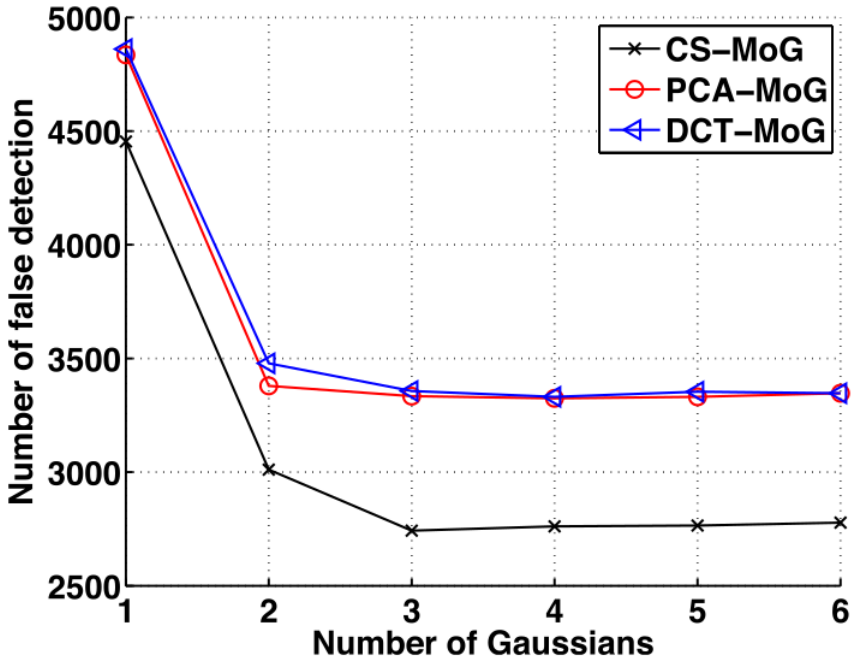


FIGURE 1.5 – Comparaison de la précision des techniques du CS quand ils sont utilisées avec MoG.

Le mécanisme est relativement simple. Nous divisons d'abord l'image en zones de 8x8 pixels. Ensuite, pour chaque zone, nous formons un vecteur de taille 1x64 $X_{(1,64)}$ des valeurs de pixels. Ensuite nous calculons des projections aléatoires $P_{(1,8)}$ de ce vecteur, comme

indiqué dans l'Eq. 1.11.

$$P_{(1,8)} = X_{(1,64)}.M_{(64,8)}, \tag{1.11}$$

où $M_{(64,8)}$ est la matrice de projection de Bernoulli utilisée, ayant une taille de 64x8. Cette matrice contient des valeurs $\pm 1$ avec une probabilité égale. De plus, chaque ligne doit contenir le même nombre de 1 et de -1 [6]. Il est également nécessaire de s'assurer que les 8 lignes sont toutes linéairement indépendantes afin d'assurer la non-corrélation des valeurs générées. Une fois que $M_{(64,8)}$ a été générée aléatoirement au début de la séquence vidéo, elle est utilisée pour toutes les images. De cette façon, la matrice $P_{(1,8)}$ obtenue pour chaque zone de l'image contient suffisamment d'informations représentant cette zone bien que le nombre d'éléments passe de 64 pixels à 8 projections uniquement. Le choix de zones de taille 8x8 pixels est le plus optimal d'après les tests réalisés par [6].

Après avoir calculé le vecteur de projections pour toutes les zones de l'image, nous appliquons la soustraction de l'arrière-plan (MoG) sur ces projections afin d'identifier celles appartenant au premier plan. Il reste à établir le lien entre la classification des projections et celle des pixels.

#### 1.3.4.2 Vote par zone

Après avoir terminé le traitement des $m = 8$ projections représentant chaque zone, nous obtenons $f$ éléments représentant le premier plan et $m - f$ appartenant à l'arrière-plan. Il est ensuite nécessaire de fusionner ces résultats afin de prendre une décision concernant l'ensemble de la zone. C'est pourquoi nous devons effectuer un vote. Les trois méthodes possibles pour ce vote sont :

— i) Le vote majoritaire : la zone est au premier plan si plus de la moitié de ses projections lui appartiennent : $f > m/2$.

— ii) Le vote maximal : la zone est au premier plan que si toutes ses projections lui appartiennent : $f = m$.

— iii) Le vote minimal : la zone est au premier plan si au moins une seule projection lui appartient : $f \geq 1$.

Ensuite, chaque zone du premier plan passera à un raffinement pixel par pixel, il est donc nécessaire d'être strict lors du vote afin d'obtenir des résultats plus précis. C'est pourquoi nous utilisons le vote minimal dans ce travail.

### 1.3.5 Raffinement du premier plan

La détection du premier plan est effectuée jusqu'à présent au niveau de chaque zone. Cette segmentation zonale peut être suffisante pour certaines applications, mais il est parfois nécessaire de travailler avec une granularité au niveau des pixels. C'est l'objet de cette étape appelée « raffinement du premier plan ». Le vote minimal adopté dans l'étape précédente permet d'avoir des zones d'arrière-plan qui ne contiennent aucune projection du premier plan : nous faisons l'hypothèse que tous les pixels de cette zone appartiennent également à l'arrière-plan. Cependant, ce n'est pas le cas pour les zones de premier plan. Elles peuvent contenir des pixels des deux catégories. C'est pourquoi nous n'effectuons ce raffinement que sur les pixels des zones du premier plan. Le nombre de ces zones est généralement réduit, car une séquence vidéo est souvent principalement composée de zones d'arrière-plan. Afin d'effectuer le raffinement sans introduire une CC importante, nous utilisons une stratégie d'apprentissage simple, décrite dans [6], pour chaque zone de l'arrière-plan. Le sous-modèle du pixel comprend une seule FDPG (une moyenne $M$, une variance $\delta$ et un poids égal à 1). Nous décidons si un pixel $X_t$ d'une image $t$ appartient à l'arrière-plan si la condition décrite dans l'Eq. 1.12 est remplie et nous sauvegardons le résultat dans une variable booléenne $\beta$.

$$\beta = [|X_t - M_t| \leq \sqrt{\delta}]. \tag{1.12}$$

Pour les pixels des autres zones, qui ne sont pas concernés par ce test, nous prenons $\beta = 1$. Quant à $\delta$, il est initialisé de la même façon que la variance initiale des FDPGs de MoG. Ensuite, la moyenne gaussienne est mise à jour, pour tous les pixels de l'arrière-plan, comme indiqué dans l'Eq. 1.13 :

$$M_{t+1} = \beta.(\alpha.X_t + (1-\alpha).M_t)) + (1-\beta).M_t. \tag{1.13}$$

La CC de ce processus est due principalement à ces deux dernières équations. La première concerne tous les pixels appartenant aux zones du premier, tandis que la deuxième est appliquée sur tous les pixels de l'arrière-plan. De ce fait, la charge globale du raffinement n'est pas directement proportionnelle à la dynamicité des images traitées. Par rapport au MoG, cette charge n'est pas très importante puisque l'équation de mise à jour ne concerne que la moyenne de l'FDPG, et non pas sa variance et son poids.

Le processus de raffinement est représenté sur la Fig. 1.6 pour un pixel dans une zone donnée. Les trois kernels de CS-MoG peuvent être résumées comme suivant :

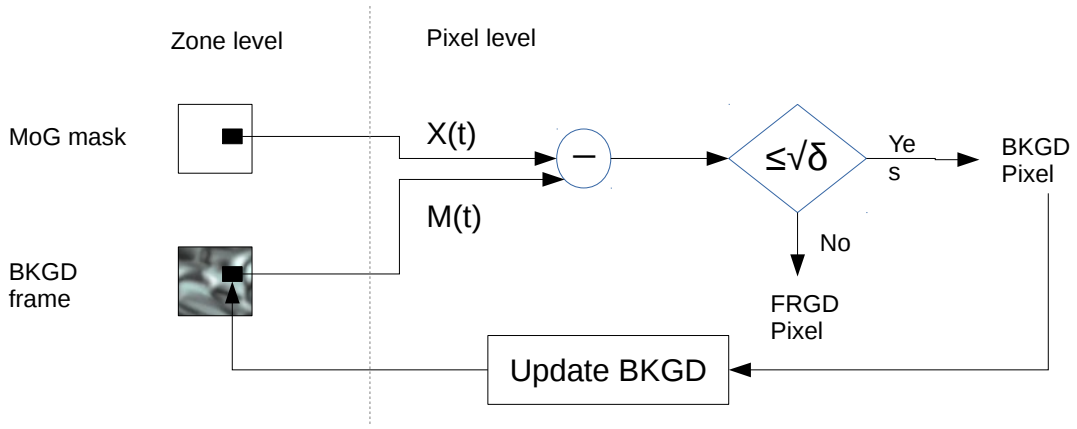


FIGURE 1.6 – Processus de raffinement appliqué sur un pixel dans une zone d'image donnée.

1. La projection zonale des images pour réduire la dimensionnalité des données
2. L'application MoG sur les données compressées pour détecter les zones appartenant à l'arrière-plan de l'image.
3. Et enfin le raffinement effectué au niveau des pixels pour détecter les objets en mouvement.

### 1.3.6 Pseudo-code de CS-MoG

Le pseudo-code qui suit reprend les différentes kernels de l'algorithme CS-MoG présentées précédemment.

**Algorithm 1** Projection

**Require:** $CompressedImage$, $BackgroundModel$

**for** every Zone in ImageZones **do**

**for** every Column in ProjectionMatrix **do**

$CompressedImage[Zone][Column] \leftarrow ScalarProduct(Zone, Column)$ ($Eq.$ 1.11)

**end for**

**end for**

**Algorithm 2** MoG

**Require:** $CompressedImage$, $BackgroundModel$

**for** every ProjectionIndex in CompressedImage **do**
  $ProjectionValue \leftarrow CompressedImage[ProjectionIndex]$
  $MD \leftarrow 0$
  $Match \leftarrow false$
  $MatchIndex \leftarrow -1$
  **for** every FDPG in Gaussians **do**
    $COMPUTE\ Mahalanobis$ ($Eq.$ 1.2)
    $COMPUTE\ VarianceThreshold$ ($Eq.$ 1.2)
    $Match \leftarrow (Mahalanobis < VarianceThreshold)$
    **if** $Match = 1$ **and** $MatchIndex = -1$ **then**
      $updateModelParameters(ProjectionIndex)$ ($Eqs.$ 1.4 − 1.7)
      $sortModelParameters(ProjectionIndex, weights)$
      $MatchIndex \leftarrow k$
      $Break$
    **end if**
  **end for**
  **if** $MatchIndex = -1$ **then**
    $initializeModelParameters(ProjectionIndex)$ ($Eqs.$ 1.8 − 1.10)
  **end if**
  $mask[ProjectionIndex] \leftarrow getBackgroundModel_(ProjectionIndex)$ ($Eq.$ 1.3)
**end for**
**for** every Zone in ImageZones **do**
  $vote \leftarrow 0$
  **for** every ProjectionIndex in Zone **do**
    $vote \leftarrow mask[projectionIndex] + vote$
  **end for**
  **if** $vote \geq 1$ **then**
    $zoneMask[Zone] \leftarrow 1$
  **else**
    $zoneMask[Zone] \leftarrow 0$
  **end if**
**end for**

**Algorithm 3** Refinement

**Require:** $CompressedImage$, $BackgroundModel$

**for** every Zone in ImageZones **do**
  **for** every Pixel in Zone **do**
    $Delta \leftarrow ABS(frame[Pixel] - BkgdFrame[Pixel])$
    **if** $(zoneMask[Zone] == 0)$ **or** $(zoneMask[Zone] == 1$ **and** $Delta \leq THRESHOLD\ )$ **then**
      $BkgdFrame[Pixel] \leftarrow UPDATE\_RATE * Delta + BkgdFrame[Pixel]$
      $Mask[Pixel] \leftarrow 0$
    **else**
      $Mask[Pixel] \leftarrow 1$
    **end if**
  **end for**
**end for**

### 1.3.7 Illustration des résultats

Comme présenté précédemment, après le vote effectué sur les projections, chaque zone de l'image correspondant à $8x8$ pixels va appartenir entièrement au premier plan (représenté en blanc) ou à l'arrière-plan (représenté en noir). Les résultats obtenus, en appliquant MoG sur les images projetées, sont représentés sur une image binaire où les unités de base sont les zones au lieu des pixels. Comme indiqué dans la Fig. 1.7.(b), ce processus permet de détecter les zones contenant les véhicules mais avec moins de précision. Ainsi, une précision au niveau des pixels ne pourra être obtenue qu'en faisant un raffinement sur les images binaires obtenues.

Même si le processus du raffinement est très simple, il permet en réalité d'obtenir des résultats précis. La Fig. 1.7.(c) montre le résultat final de la soustraction de l'arrière-plan obtenu après l'application de ce processus. Nous constatons qu'un ensemble de zones qui faisaient initialement partie du premier plan ont été classées, entièrement ou partiellement,

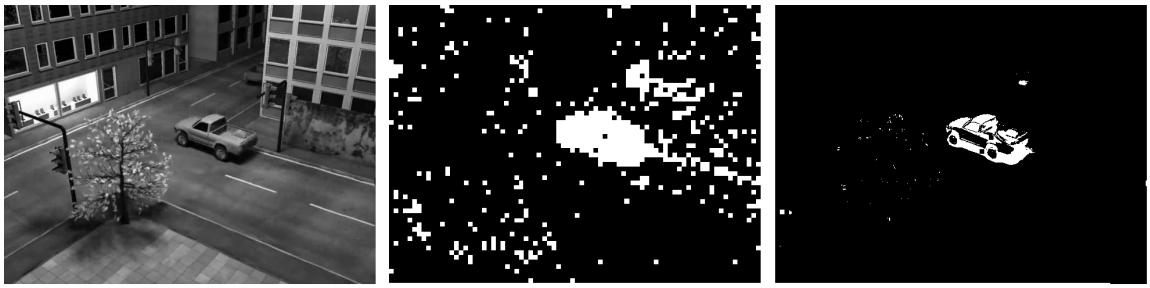


FIGURE 1.7 – Résultats obtenus en appliquant MoG et le raffinement sur des images compressées. A gauche l'image d'entrée, et à droite la segmentation obtenue (premier plan en blanc).

dans l'arrière-plan. Ce résultat est suffisamment précis pour les applications de détection de véhicules. Une confirmation de cette conclusion suite à un ensemble de tests sur 5 datasets différents est présentée dans les chapitres 3 et 4.

## 1.4 Conclusion

Dans ce chapitre, nous avons introduit l'ensemble des éléments représentant le cadre théorique ainsi que les applications visées par cette thèse. Les conclusions tirées de ce chapitre peuvent être résumées comme suit :

— Nos travaux de recherche visent deux projets : HEAVEN et MoViTS dont les objectifs sont complémentaires.

— Un état de l'art sur la détection des véhicules montre que l'algorithme de soustraction de fond via l'acquisition comprimée, CS-MoG, représente un meilleur compromis entre la précision des résultats et les performances temporelles.

— La nature de cet algorithme où le traitement des images peut être effectué avec une granularité importante nous a motivé à l'implémenter sur des architectures hétérogènes CPU/GPU. Cependant, la définition d'une méthode d'implémentation efficace nécessite également une connaissance approfondie de ces architectures

Le chapitre suivant est consacré à la présentation des caractéristiques techniques et fonctionnelles de ces architectures, ainsi que les outils informatiques permettant de les exploiter.

Chapitre 2

# État des lieux des architectures matérielles de calcul parallèle CPU, GPU, CPU-GPU.

## Sommaire

## 2.1 Introduction

Nous avons constaté précédemment que la soustraction de fond via un algorithme comme CS-MoG se fait en traitant chaque élément de l'image projetée indépendamment de son voisinage. Nous parlons d'un traitement local ou Pixel-Wise Processing [103]. Ainsi chaque élément (pixel ou projection) est comparé avec son équivalent dans le modèle d'arrière-plan afin de le classer et puis mettre à jour ce modèle.

L'exécution de tels calculs est plus efficace sur des processeurs graphiques GPU contenant un nombre élevé de cœurs de calcul identiques. Par contre, sur la plupart des plateformes de calcul, l'exploitation d'un GPU est effectuée à travers l'intermédiaire d'un CPU qui sert à lui envoyer les instructions ainsi que les données à traiter. Pour atteindre plus de performance, ce dernier peut être également exploité pour exécuter une partie des calculs. Nous parlons alors d'exécution hétérogène [104], offrant une grande flexibilité et une forte puissance de calcul. Cette intégration de différents types de processeurs connectés par un même bus, rend les plateformes hétérogènes les plus efficaces pour le traitement d'image. Il est alors nécessaire d'étudier profondément les architectures hétérogènes pour définir les défis de l'implémentation par rapport à chaque kernel de CS-MoG, ainsi que pour positionner ces défis du point de vue de la littérature. Ils peuvent être liés à la synchronisation des tâches, aux transferts de données, à l'ordonnancement des calculs, etc.

Nous commençons ce chapitre par la définition des architectures hétérogènes CPU/GPU, ainsi que la présentation de leurs caractéristiques fonctionnelles. Ensuite nous présentons des guides pratiques pour l'optimisation d'un algorithme donné sur les composantes de ces architectures. Un état de l'art détaillé sur les implémentations de CS-MoG sur CPU et GPU est présenté avant de conclure ce chapitre.

## 2.2 Architectures de calcul hétérogènes

### 2.2.1 Aperçu

Une architecture hétérogène est constituée de nœuds de calcul qui ont des capacités différentes. Ces différences concernent principalement la manières d'exécuter les instructions, le taux de parallélisme des calculs, les hiérarchies mémoire et les outils de programmation. Lorsque les systèmes multi-cœurs sont apparus, ils étaient homogènes : c'est-à-dire que tous les cœurs étaient similaires. Passer de la programmation séquentielle à la programmation parallèle, qui n'était auparavant qu'un domaine réservé aux programmeurs de haut niveau, a été considéré une révolution technologique. Les plateformes hétérogènes se retrouvent, aujourd'hui, dans tous les domaines de l'informatique, que ce soit des serveurs de hautes performances, ou des appareils embarqués à faible consommation d'énergie, y compris les téléphones portables et les tablettes. Dans la plupart des cas, les plateformes hétérogènes contiennent des CPU et des GPU en raison de leurs caractéristiques complémentaires.

Un microprocesseur multi-cœur CPU (multi-core CPU en anglais) est un processeur possédant plusieurs sous-processeurs appelés cœurs qui fonctionnent simultanément. Avoir ce dispositif matériel dans un ordinateur ne signifie pas seulement un fonctionnement plus rapide pour certains programmes, mais aussi plus de flexibilité en termes d'exploitation. En effet l'utilisateur à la possibilité d'utiliser les cœurs de calculs pour des tâches différentes. Un cœur contient généralement une unité arithmétique et logique (Arithmetic Logic Unit ALU) pour effectuer les calculs, ses propres registres, une unité de contrôle (Control Unit CU) permettant d'interpréter les instructions, et diriger l'exécution, et un espace mémoire cache. Ces deux dernières peuvent être partagées avec d'autre cœurs, comme présenté dans la Fig. 2.1. C'est le cas de certains processeurs ARM comme le Cortex-A9. Les CPU disposent d'une puissance de calcul importante lorsqu'il s'agit d'applications représentant un fort **parallélisme de tâches**. Autrement dit, nous parlons d'algorithmes dont plusieurs tâches indépendantes doivent être parallélisées, ou encore lorsque la même tâche représente

beaucoup de divergences entre les différents threads [15]. Les CPU sont également puissants quand les opérations à exécuter sont en double précision.

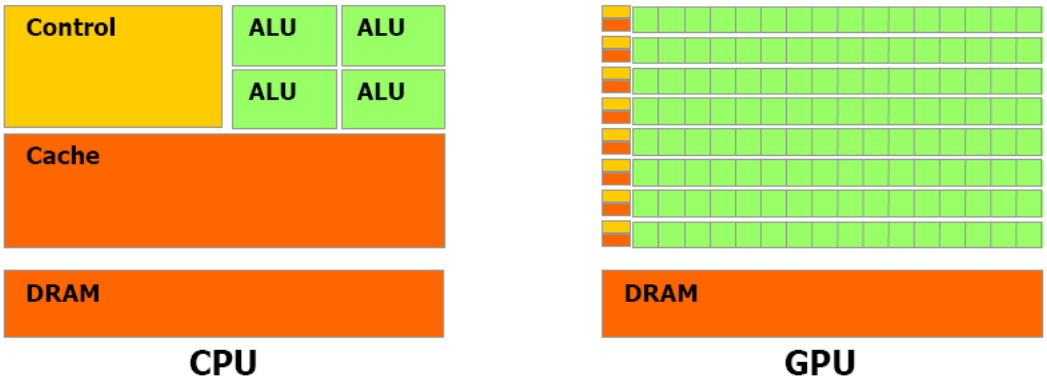


FIGURE 2.1 – Présentation des PUs contenues dans une architecture hétérogène.

Une unité de traitement graphique (GPU) est un circuit de calcul hautement parallèle, intégré dans la plupart des dispositifs qui incluent un afficheur : les systèmes embarqués, les téléphones mobiles, les ordinateurs personnels, les stations de travail et les consoles de jeux. Contrairement au CPU, les GPU contiennent des centaines voire des milliers d'ALUs partageant une mémoire globale. Ces ALUs représentent une force incomparable quand il s'agit des algorithmes où les mêmes instructions doivent être appliquées sur un grand nombre de données en parallèle, comme les pixels d'une image. On parle du **parallélisme de données**. Sur un ordinateur personnel, un processeur graphique peut être présent sur une carte vidéo d'extension connectée au bus PCie ou sur une puce intégrée à la carte mère. Ils sont aussi parfois directement intégrés au CPU [105][106].

### 2.2.2 Evolution et familles

L'industrie des systèmes à base de CPU a connu une grande évolution surtout après la commercialisation du premier processeur par Intel en 1971 (Intel 4004). De nos jours, il existe une concurrence intense entre plusieurs fabricants, vue l'importance de ces processeurs dans la plupart des dispositifs numériques. Parmi ces fabricants on trouve IBM (International Business Machines Corporation) qui est connu par ces supers ordinateurs

utilisés comme serveurs de Cloud Computing. Son Power System AC922 destiné à l'intelligence artificielle est classé à la tête des 500 superordinateurs les plus puissants au monde en 2019 (TOP500 [107]). Pour les microprocesseurs destinés à l'utilisation dans l'embarqué, ARM Holdings est connu par ces processeurs souvent intégrés dans les smartphones, les tablettes, les réseaux de capteurs et systèmes autonomes. Il suffit de citer par exemple sa microarchitecture ARMv8-A qui a été adoptée par Samsung dans son Galaxy Note 4 et par Apple dans l'iPhone 5S. Pour le traitement de signal, Texas Instrument est connu par ses DSPs (Digital Signal Processors) utilisés dans les radars et les systèmes d'aviation comme le cas des produits de Thales Air Systems.

Plus important encore, le marché des ordinateurs est dominé par les produits à base de microprocesseurs d'Intel et AMD. Un article publié sur Developpez.com en 2019 [108], annonce que "dans certains pays, en particulier sur les marchés allemands et asiatiques, l'arrivée des processeurs de la série Ryzen 3000 d'AMD, avec le Ryzen 7 3900X en tête de liste en attendant le Ryzen 9 3950X et ses 16 cœurs, semble être une étape cruciale qui marque un tournant décisif entre les deux sociétés américaines rivales AMD et Intel. Depuis le début de la commercialisation des CPU x86 Matisse d'AMD le 7 juillet 2019, les ventes de processeurs AMD Ryzen 3000 surclassent à elles seules celles de la gamme complète de puces x86 grand public d'Intel dans le circuit de distribution au détail - ou du DIY (Do It Yourself) - qui est plus proche du consommateur. Pour chaque processeur vendu par Intel en juillet dernier, AMD en a vendu quatre". Par contre, selon un sondage permanent sur Ranker.com, intitulé "The Best CPU Manufacturers", les consommateurs considèrent que les produits Intel sont toujours les plus dominants dans le marché mondial. Cela est dû au fait que ces derniers sont conçus pour répondre à une large gammfe de besoins et d'utilisations, partant des petits objets connectés jusqu'aux supercalculateurs (cf. section 2.2.3). Plus de détails sur toutes ces familles de CPU sont donnés dans l'annexe A.3.

En ce qui concerne l'évolution des GPU, les premières puces vidéo ont apparues à partir des années 70, mais le terme « GPU » a été inventé par Sony en référence au GPU Sony conçu par Toshiba utilisé dans la console PlayStation en 1994. Le terme a

été popularisé par NVIDIA en 1999, qui a commercialisé le circuit GeForce 256 en tant que "premier GPU au monde". Selon un sondage sur le même site Ranker.com, intitulé "The Best GPU Manufacturers", NVIDIA est considéré comme le premier fabricant de GPU, suivi par AMD. Certains autres fabricants tels que Asus, Intel, EVGA, Gigabyte et Saphire sont autorisés à fabriquer des cartes graphiques pour NVIDIA et AMD en tant que partenaires officiels, suivant des modèles de référence.

Dans cette thèse, nous ciblons des implémentations de CS-MoG sur les plateformes contenant des CPU Intel et des GPU NVIDIA. Dans un premier temps nous explorons chacun de ces composants grâce à des implémentations séparées. Par la suite, nous exploitons les résultats de ces implémentations afin d'effectuer un portage concurrent et simultané de CS-MoG sur CPU-GPU.

### 2.2.3 Multi-cœur CPU Intel

La Table. 2.1 représente une vue globale sur les familles des processeurs Intel les plus communs avec leurs utilisations et caractéristiques techniques [109]. Les valeurs (Cœurs, Fréquence et Cache) représentent le maximum de ce qui est atteint au niveau de chaque famille, mais ça ne veut pas dire forcément qu'elles sont regroupées dans un même produit de cette famille.

| Famille | Utilisation | Cœurs | Freq. | Cache |
|---|---|---|---|---|
| Quark | Internet des objets | 1 | 400 Mhz | 16KB |
| Pentium | Ordinateurs de bureau | 4 | 3.3 GHz | 4MB |
| Celeron | PC d'entrée de gamme à faible coût | 4 | 3.3 GHz | 4MB |
| Itanium | Calcul haute performance | 8 | 2.66 GHz | 32MB |
| Atom | Appareils mobiles | 16 | 2.4 GHz | 16MB |
| **Core** | **Multimédia et jeux** | **18** | **4.2 GHz** | **24.75MB** |
| Xeon | Cloud computing & AI | 56 | 4 GHz | 77MB |

TABLE 2.1 – Famille des processeurs Intel les plus communs avec leurs utilisations et caractéristiques techniques.

Les implémentations effectuées dans cette thèse ont ciblé des microprocesseurs de type Core qui sont les plus communs et les plus utilisés pour le traitement multimédia. Le

chip d'un processeur multi-cœur Intel Core contient généralement un ensemble de cœurs partageant un niveau de la mémoire cache. Ce chip est lié via un bus à une mémoire RAM externe. La Fig. 2.2 représente les composants d'un processeur Intel-i5 standard ainsi que sa structure en silicium.

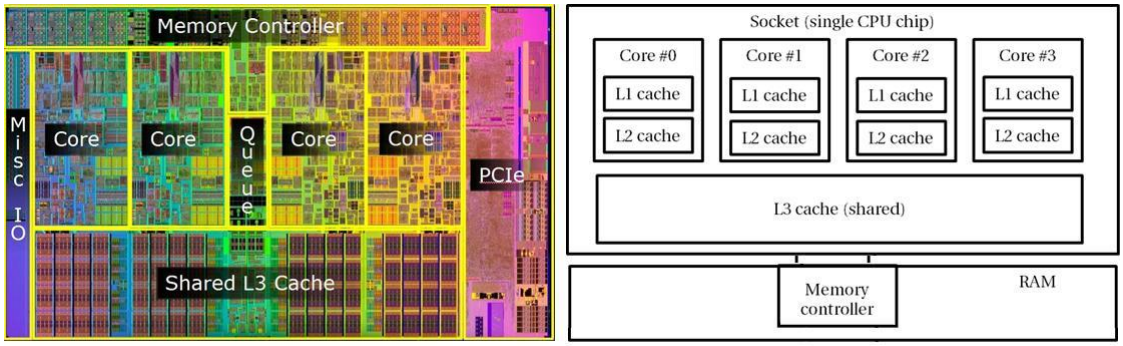


FIGURE 2.2 – Composants d'un processeur Intel Core i5 750 à droite, ainsi que sa structure en silicium à gauche.

### 2.2.4 GPU NVIDIA

NVIDIA arrive en tête de la liste des fabricants de GPU en raison de ces produits couvrant des domaines d'utilisations variés, et qui se déclinent en plusieurs gammes [110], [111] :

— Les GeForce : des produits de traitement graphique destinés au grand public, représentées en 2018 par les GeForce 20 Series.

— Les Quadro : des cartes pour professionnels destinées aux stations de travail de Conception Assistée par Ordinateur et de création de contenu numérique.

— Les NVS : des solutions graphiques professionnelles multi-écrans.

— Les Tesla : GPU polyvalents dédiés aux applications de génération d'images dans les domaines professionnels et scientifiques.

— Les Titan : cartes graphiques destinés également au grand public, dont les performances sont proches des GeForce GTX et Tesla en simple précision, qui sont destinées au montage vidéo et aux tâches qui demandent de grosses ressources vidéo.

— Les Tegra : des puces graphiques destinées aux appareils mobiles comme les smartphones, tablettes et consoles portables.

Un GPU NVIDIA contient généralement un grand nombre de processeurs (Streaming Multiprocessors SM) qui peut atteindre 80 pour le Tesla V100 [112]. Chacun de ces SM contient jusqu'à 192 cœurs selon l'architecture CUDA [113] (Table. 2.2).

| Tesla | Fermi | Pascal/Volta/Turing | Maxwell | Kepler |
|---|---|---|---|---|
| **8** | **32** | **64** | **128** | **192** |

TABLE 2.2 – Nombre de cœurs dans un Streaming Multiprocessor pour chaque architecture NVIDIA.

Durant les calculs, un ou plusieurs threads seront affectés à chaque cœur selon la dimension des données. Les threads sont regroupés en sous-groupes qui s'exécutent simultanément appelés Warps, dont la taille est généralement de 32. Pour une meilleure répartition des tâches et des données, les threads sont regroupés en blocs. Les threads d'un même bloc s'exécutent sur le même SM. Ils peuvent communiquer entre eux via une mémoire partagée, se synchroniser avec des barrières ou utiliser des opérations atomiques. La taille des blocs varie selon l'utilisation et la capacité du GPU. Plusieurs blocs sont combinés pour former une grille dont la dimension est choisie pour chaque application. La hiérarchie mémoire d'un GPU comprend (Fig. 2.3) :

1. une mémoire cache L1 locale et privée pour chaque thread.
2. une mémoire cache L2 partagée entre les threads d'un même bloc.
3. une mémoire globale de taille plus grande partagé entre tous les blocs, et accessible aussi par des dispositifs externes comme le CPU.
4. une mémoire constante utilisée pour les données qui ne changeront pas au cours de l'exécution du kernel.
5. une mémoire de texture conçue pour faciliter l'accès au voisinage 2D ou 3D d'un élément de donné dans la mémoire (grande localisation spatiale).

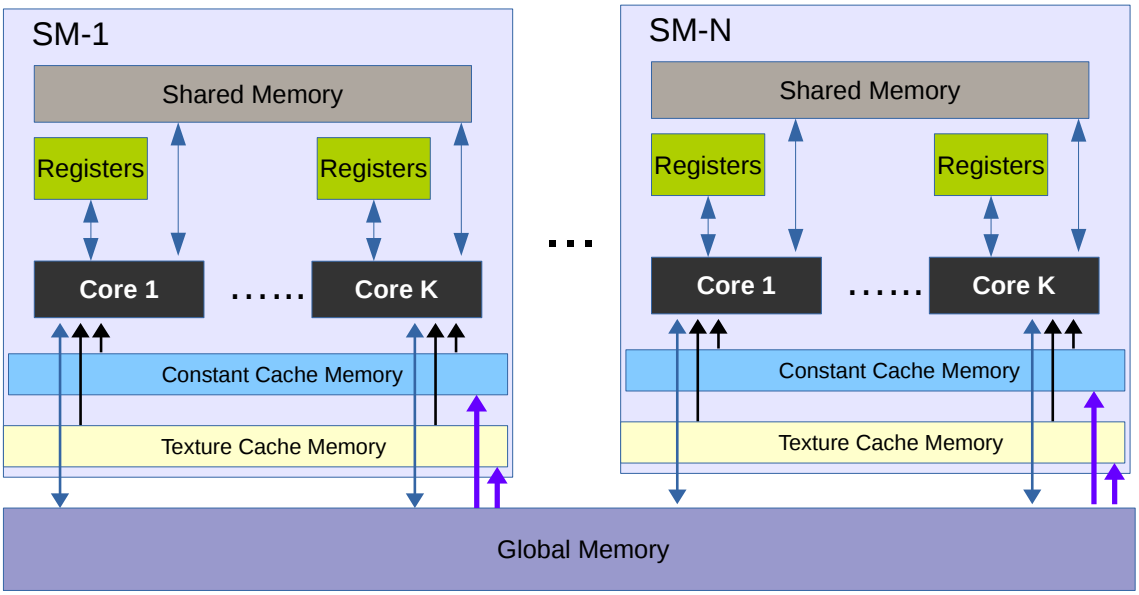


FIGURE 2.3 – Schéma de principe de l'architecture CUDA de NVIDIA.

Malgré tous les avantages qu'offre les CPU Intel et les GPU NVIDIA, la programmation et l'exploitation efficace des architectures hétérogènes basées sur ces technologies n'est pas triviale à cause de leurs hiérarchies mémoire et l'hétérogénéité des éléments de calcul. Par conséquent, l'implantation parallèle d'un algorithme séquentiel comme CS-MoG nécessite de prendre ces contraintes en considération.

### 2.2.5 Outils de programmation

La plupart des algorithmes de traitement d'image sont disponibles sous forme de codes séquentiels, en raison de la difficulté à créer de bons programmes parallèles. Il est alors nécessaire de chercher des stratégies efficaces pour permettre aux développeurs d'intégrer leurs codes d'une façon optimisée sur ces architectures parallèles. La démarche d'Adéquation-Algorithme-Architecture (AAA) [114] consiste à étudier simultanément les aspects algorithmiques et architecturaux en prenant en compte leurs relations dans le sens algorithme vers architecture, et vice versa.

Certaines interfaces (frameworks) et modèles de programmation facilitent le portage

de codes séquentiels sur les architectures CPU/GPU en faisant de simples annotations. OpenACC [115] initialement développé par PGI, Cray et NVIDIA, est une API décrivant une collection de directives de compilateur, qui permet de spécifier les boucles et les portions de code C, C++ ou FORTRAN à décharger d'un CPU vers un accélérateur GPU.

CUDA (Compute Unified Device Architecture) [113] est une autre API développée par NVIDIA pour effectuer des calculs généraux sur les GPU en utilisant le langage C. Cette API présente plusieurs particularités comme l'exposition de la hiérarchie mémoire (privée, locale, globale), et aussi le regroupement des threads en grilles de blocs, ce qui permet une bonne exploitation du matériel. Il existe d'autres frameworks plus généraux comme OpenCL [116], mais selon l'étude effectuée par [117], ce dernier génère moins de performances par rapport à CUDA sur les GPU NVIDIA.

Pour l'optimisation des calculs localement au niveau CPU, de nombreux modèles de programmation ont été proposés. La bibliothèque Boost Threads [118] est une solution couramment utilisée pour la gestion des threads contrôlée par le programmeur. Une alternative est la planification automatique des tâches, gérée par une bibliothèque externe. Les solutions les plus courantes de ce type sont la bibliothèque de blocs de threads TBB [119] et l'interface de programmation d'application Open Multi-Processing OpenMP [120], utilisées pour les systèmes à mémoire partagée, pour spécifier manuellement les régions devant être parallélisées à l'aide de simples directives.

Tous ces outils ont l'avantage de faciliter l'accélération d'une application en portant son implémentation séquentielle sur un accélérateur parallèle. Néanmoins, pour pouvoir tirer les meilleures performances du matériel, une démarche d'Adéquation Algorithme Architecture (AAA) plus profonde est nécessaire, ce qui exige l'intervention d'un expert. Dans cette thèse, nous exploitons les avantages de CUDA pour développer une méthode d'AAA efficace, permettant l'implémentation optimisée de CS-MoG sur des architectures hétérogènes CPU/GPU. OpenMP est adopté pour les optimisations de l'exécution sur CPU. Nos travaux concernent principalement des plateformes de type Desktop où Laptop.

## 2.3 Guides de portage

Une méthodologie bien définie pour d'optimisation d'algorithmes sur CPU et GPU doit être suivie pour obtenir des performances élevées. Pour définir cette méthodologie, il est nécessaire de prendre un ensemble d'aspects en considération selon un guide d'implémentation bien précis.

### 2.3.1 Optimisation d'algorithmes sur CPU

Malgré les avantages qu'offrent les APIs tels que OpenMP, l'optimisation d'une application sur une architecture multi-cœur CPU n'est pas évidente. Elle nécessite de prendre en compte plusieurs aspects associés à des niveaux d'optimisation différents [121]-[123]. Il est possible de distinguer 6 niveaux classés par leurs degrés d'impact sur les performances :

1. **Niveau de conception (design)** : cette étape vient juste après la définition du cahier des charges auquel doit répondre l'application visée. Son objectif est d'inspecter l'architecture de l'algorithme pour détecter les éléments limitant ses performances (charge de calcul, accès mémoire ou bande passante). Cela est effectué à l'aide d'une première version du code, on parle du profilage de code (profiling). Cette étape a un impact important. On cherche à réduire la complexité mathématique de l'algorithme et à optimiser la quantité des données à traiter.

2. **Niveau du code source (program)** : dans ce niveau, les optimisations sont liées à la capacité du programmeur à choisir l'environnement et le langage de programmation convenable, les structures de données, les bibliothèques adéquates, et les déclarations optimales (constantes, fonctions inline etc).

3. **Niveau de construction (build)** : les directives et les drapeaux (flags) de précompilation peuvent être utilisés pour améliorer les performances du code source et du compilateur. Grâce à ces directives (Ex. #pragma OpenMP) on peut exploiter des capacités matérielles spécifiques comme le multithreading, la prédiction des branchements ou même la désactivation de quelques fonctionnalités logicielles inutiles.

4. **Niveau de compilation (compile)** : certains compilateurs récents effectuent implicitement des optimisations du code comme l'utilisation de la mémoire cache etc. Cependant, parfois, il est nécessaire que le programmeur oriente le compilateur, comme le cas de GCC [124] et ICC [125], en choisissant le niveau d'optimisations convenable.

5. **Niveau d'assemblage (Assembly)** : dans certains cas, il est utile de faire appel localement à des langages de bas niveau (instructions intrinsèque ou Assembleur) pour réaliser certains calculs (ex. vectorisation). La connaissance de la partie matérielle est importante pour faire ce type d'optimisations. Cependant, le compilateur lui-même peut effectuer automatiquement certaines optimisations de ce type, telles que le Pipelining, la prédiction de branchements, l'Exécution spéculative etc. (Plus de détails sont donnés dans l'annexe A.4).

6. **Niveau d'exécution (Runtime)** : certaines optimisations peuvent être réalisées durant l'étape d'exécution comme l'ajustement des paramètres en fonction des données. Les processeurs récents effectuent automatiquement quelques optimisations de ce type durant le runtime comme l'exécution dans le désordre (Out-of-order execution), l'exécution spéculative (Speculative execution) et la prédiction des branches (Branch predictors).

### 2.3.2 Optimisation d'algorithmes sur GPU

Comme dans le cas du CPU, l'optimisation d'un algorithme sur GPU nécessite de respecter un certain nombre de règles liées à l'architecture de ce dernier et à la façon par laquelle ses différentes ressources sont utilisées. Cependant, avant de commencer le processus d'optimisation, il est nécessaire d'analyser les performances de base du code afin de détecter les éléments qui limitent ses performances, en faisant un profilage.

Il existe de nombreux outils pour profiler un code. L'exemple suivant est basé sur *gprof*, qui est un profileur de la collection GNU Binutils.

```
$gcc -O2 -g -pg myprog.c
```

```
$gprof ./a.out > profile.txt
```

Cela permet d'afficher le temps consommé par les différentes parties du code, et permet au programmeur d'effectuer des optimisations convenables sur les sections les plus lentes. Loin de ces outils génériques, certains concepteurs de GPU proposent des profileurs dédiés à leurs plateformes comme le Visual Profiler de NVIDIA (NVVP). C'est un outil de profilage qui fournit aux développeurs des informations essentielles pour l'optimisation des applications CUDA C/C ++. Introduit pour la première fois en 2008, Visual Profiler prend en charge tous les GPU NVIDIA compatibles avec CUDA sous Linux, Mac OS X et Windows. La Fig. 2.4 représente son interface graphique. Pour l'implémentation d'un algorithme donné

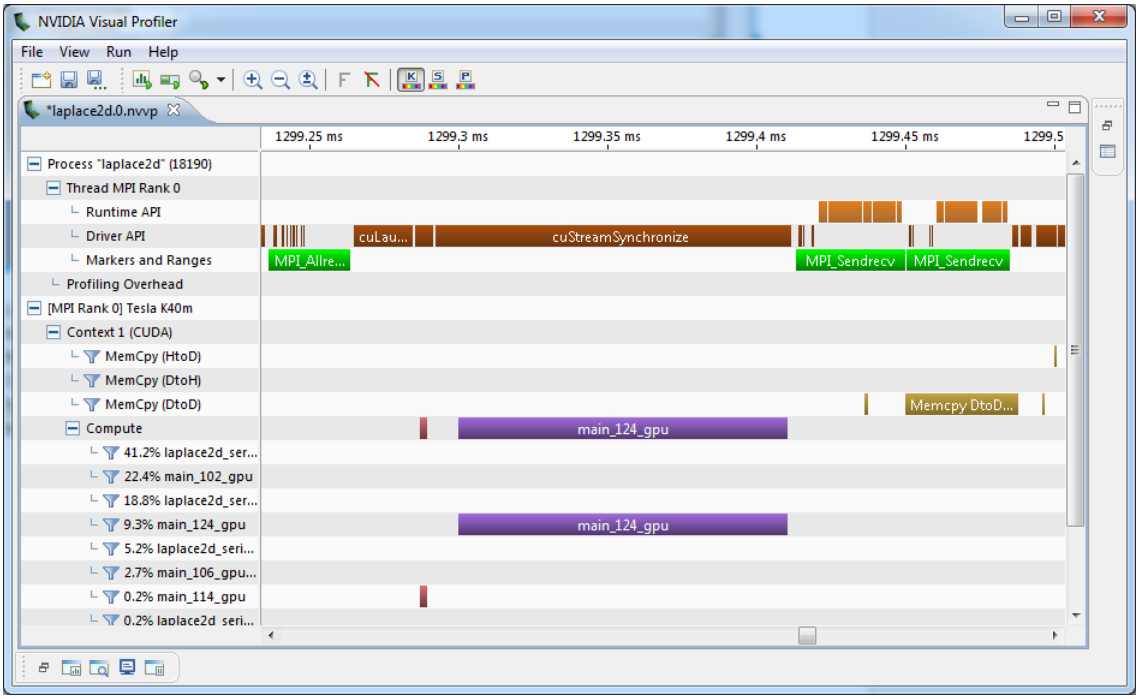


FIGURE 2.4 – Interface du profileur NVIDIA (NVVP).

(CS-MoG dans notre cas), il faut prendre en compte cinq niveaux d'optimisation pour chacun de ses kernels : l'optimisation lié aux accès mémoire, la communication, la configuration d'exécution (dimensionnement des blocs et de la grille), les instructions et le flux de contrôle. L'ordre est très important car les meilleures pratiques suggèrent qu'un niveau est ciblé uniquement une fois que toutes les optimisations des niveaux supérieurs sont

terminées.

— **Optimisation de la mémoire** : c'est l'aspect le plus important en termes de performances. L'objectif est de maximiser la bande passante, qui est mieux servie quand on utilise plus de mémoire rapide que de mémoire à accès lent. Par exemple, la mémoire cache partagée peut être utile dans plusieurs situations, par exemple pour éliminer des accès redondants à la mémoire globale. De plus, la distribution des données entre les threads successifs appartenant au même Warp, doit assurer un accès à des éléments mémoire contigus afin que le temps d'accès soit minimal.

— **Optimisation de la communication** : le processeur graphique disposent de moteurs indépendants pour le calcul et le transfert de données, il est vivement recommandé d'exécuter simultanément ces deux processus afin de masquer le temps de transfert des données entre les mémoires de CPU et GPU.

— **Optimisation de la configuration d'exécution** : l'une des clés de la performance consiste à garder les multiprocesseurs du périphérique aussi occupés que possible en optimisant l'exécution parallèle. Un périphérique dans lequel le travail est mal équilibré entre les multiprocesseurs donnera des performances sous-optimales. Par conséquent, il est important de concevoir l'application de manière à utiliser les threads et les blocs afin de maximiser l'utilisation du matériel. Un concept clé dans cet effort est l'occupation, qui est le pourcentage des Warps activement utilisés par rapport à la capacité maximale du matériel. La dimension de la grille d'exécution des threads, et la taille des blocs qui la constituent sont des facteurs importants. Le nombre de threads par bloc doit être un multiple de la dimension du Warp, car cela permet une efficacité informatique optimale et facilite la coalescence de la mémoire. Un minimum de 64 threads par bloc doit être utilisé.

— **Optimisation des instructions** : les opérations flottantes à simple précision offrent les meilleures performances et leur utilisation est vivement encouragée. Il est également important d'éviter la conversion automatique des doubles en simple précision et d'utiliser des opérations de décalage pour éviter les calculs coûteux de division

et de modulo. L'option de compilation *Fastmath* de CUDA peut également accélérer certaines fonctions avec un faible effet sur la précision. Toutes les techniques mentionnées dans ce point peuvent être appliquées à l'aide du compilateur NVIDIA *NVCC*.

— **Optimisation du flux de contrôle** : les instructions de contrôle de flux (if, switch, do, for, while) peuvent affecter de manière significative le débit d'instruction en faisant diverger les threads du même Warp (32 threads) qui vont suivre différents chemins d'exécution. Si cela se produit, les différents chemins d'exécution doivent être exécutés séparément. Cela augmente le nombre total d'instructions exécutées pour ce Warp.

Avant de commencer à appliquer ces niveaux d'optimisation sur CS-MoG, il est nécessaire d'étudier toutes les implémentations précédentes afin de connaitre les aspects qui ont été traités et ceux qui restent à explorer afin d'assurer des meilleures performances.

## 2.4 CS-MoG sur CPU-GPU : état de l'art

### 2.4.1 Implémentations sur CPU

La plupart des travaux trouvés dans la littérature ont ciblé le premier niveau d'optimisation sur CPU (design level), vu son importance. C'est le cas de [126] et [53] mentionnés dans le chapitre précédent qui ont apporté des modifications sur les équations mathématiques de MoG afin de réduire sa complexité informatique, tandis que [6] agit au niveau de la quantité de données traitées. Une première implémentation qui a ciblé les capacités des plateformes matérielles (build level) est celle effectuée par [51]. Il s'agit d'une implémentation parallèle de MoG sur des processeurs multi-cœurs CPU. Son objectif était de choisir une méthode de planification des tâches offrant une précision et une efficacité satisfaisantes. En testant plusieurs modes d'équilibrage des calculs entre les différents cœurs (statique et dynamique), les résultats de cette étude sont limités à une accélération de 3,5 alors que 12 cœurs physiques sont utilisés. Après examen du code, nous avons constaté que ce problème

de scalabilité est lié à la présence de dépendances entre les différents threads introduits par des variables partagées. Nous avons amélioré les performances de cette version après la suppression des de ces dépendances entre les threads [127] (cf. section 3.4).

En étudiant cet état de l'art, nous constatons trois conclusions importantes qui représentent des motivations pour notre travail. La première est que l'intégration du CS avec MoG n'a pas été prise en considération dans cette implémentation parallèle sur les multi-cœurs CPU. La deuxième est que les autres niveaux d'optimisation (code source, compilation, assemblage et runtime) n'ont pas été traités. La troisième est que les résultats obtenus dans la littérature concernant les deux niveaux (design et build) sont limités en termes de performance.

Nos contributions concernant ces différents points, et ainsi nos améliorations pour chaque couche d'optimisations sont présentées dans le chapitre III. Au niveau *design* nous avons préservé l'architecture mathématique de base de l'algorithme afin de ne pas influencer sa précision. Cependant nous avons, d'une part, agit au niveau du *codage* et *build* afin d'améliorer sa scalabilité en multi-cœur. D'autre part, nous avons effectué quelques optimisations au niveau des instructions en utilisant la technique de vectorisation. Les améliorations effectuées au dernier niveau (*runtime*) vont être traitées dans le chapitre 4.

### 2.4.2 Implémentations sur GPU

Certains travaux antérieurs ont proposé de mettre en œuvre une optimisation de MoG sur GPU. Ces travaux ont été axés sur l'optimisation de l'utilisation du matériel en analysant certains détails architecturaux. Toutefois, d'après notre étude de la littérature, l'intégration du CS avec MoG n'a jamais été abordée.

En 2006, Lee et Jeong [128] ont utilisé des GPU de NVIDIA pour implémenter la version de base de MoG [13]. Toutefois, il n'existe pas de description détaillée des techniques clés dans ce travail. [129] a mis en œuvre une soustraction de fond multimodale à l'aide de MoG sur un GPU NVIDIA 8600M GT qui a permis une accélération x5 par rapport à

la version séquentielle implémentée sur un CPU de type Core 2 Duo MacBook Pro. [130], [131], [132], [133] appliquent des optimisations générales liées au matériel, telles que les accès mémoire coalescents et le transfert de données en recouvrement avec le calcul, mais l'algorithme lui-même est en grande partie intact. Ces travaux améliorent également l'efficacité en utilisant la mémoire partagée pour conserver une partie des paramètres gaussiens sur le GPU, évitant ainsi les accès à la mémoire globale. Cependant, [130], [131], [132], [133] n'ont pas révélé les détails sur l'utilisation de la mémoire partagée (par exemple, la quantité utilisée) et n'ont pas montrer l'effet de cette optimisation indépendamment des autres pour connaitre son impact sur l'amélioration des performances.

Pour qu'une implémentation de GPU soit efficace, l'interaction avec CPU doit également être optimisée. Ainsi, d'autres travaux ont porté sur l'optimisation des transferts de données. Une méthode efficace pour l'apprentissage incrémental de GMM à l'aide de CUDA est implémentée dans [134]. Sa principale contribution consiste à masquer la latence du transfert de données en exploitant les fonctionnalités de CUDA appelées « exécution concurrente » et « masquage de transfert de données », de telle sorte que l'apprentissage incrémental de GMM puisse être implémenté de manière pipeline sur CUDA. [135] fournit des instructions aux développeurs pour accélérer les applications GPU : après avoir classifié l'algorithme ciblé en fonction du rapport communication/calcul, un modèle mathématique permet de choisir la stratégie de mise en œuvre optimale en estimant la partition, la granularité et la planification des transferts. Néanmoins, dans le cadre de nos travaux, même lorsque toutes ces conditions sont remplies, les meilleures performances ne sont pas pleinement atteintes tant que d'autres améliorations ne sont pas apportées, comme présenté dans les sections suivantes.

Des optimisations plus génériques ont été proposées dans d'autres études comme dans [136] qui propose un modèle de performances précis, pour une optimisation efficace de la divergence des flux de contrôle sur GPU. Cela en introduisant une métrique qui représente les performances des kernels ayant une divergence de flux de contrôle importante, il utilise ensuite cette métrique comme entrée aux algorithmes de dimensionnement des threads.

Toutefois, le regroupement de threads dans les Warps n'est pas toujours sûr car certains kernels reposent sur des dépendances entre les Warps eux même. Au-delà des implémentations CPU et GPU, une implémentation matérielle de MoG sur FPGA avec flux vidéo ultra-haute résolution et traitement temps réel a été réalisée dans [137].

Plus important encore, des travaux plus récents [138], [106], [139], [140], [141] ont optimisé l'implémentation de MoG selon plusieurs niveaux. Ces niveaux concernent, d'une part, les optimisations générales liés au GPU (telles que la coalescence de la mémoire, le recouvrement calcul/communication etc.), et d'autre part, les optimisations spécifiques à l'algorithme, notamment la réduction du flux de contrôle, l'optimisation de l'utilisation des registres ou l'optimisation de latence grâce à la mémoire partagée. Dans ce contexte, [138] cible une application où on a besoin de traiter des vidéos Full HD (1080p 60 Hz) sur les GPU de grande capacité, cependant [106] visent les GPU intégrés à faible puissance. L'approche décrite dans ce dernier a comme objectif de réduire la quantité des calculs en éliminant l'écart-type dans le modèle de l'arrière-plan, ainsi que les opérations coûteuses qui lui sont associées, en utilisant un nombre variable de FDPGs par pixel. Cela améliore les performances mais réduit la qualité des résultats. L'utilisation d'un nombre variable de FDPGs semble être une solution prometteuse sur CPU, toutefois, lorsque on cible le GPU, il peut générer des avantages limités. Cela est dû au fait que les threads parallèles sur GPU s'exécutent suivant le même chemin : tous les threads effectuent la même quantité de calcul, même avec un nombre variable de FDPGs. En conséquence, le thread avec le plus de FDPGs détermine la latence de tous les threads parallèles. [140] a pour objectif d'améliorer MoG avec un algorithme de remplissage de trous (Hole Filling) par le biais d'opérations morphologiques de post-traitement. Quant à la taille des blocs de threads, [141] affirme que des blocs plus petits constitués de 128 ou 256 threads permettent d'avoir une meilleure occupation du GPU.

### 2.4.3 Exploitation simultanée CPU-GPU

En ce qui concerne la répartition de la CC de CS-MoG sur PUs d'une plateforme hétérogène de manière automatique et équilibrée. Des travaux existants proposent des approches générales : Des études récentes comme [142] et [143] présentent des méthodes qui répondent à ce besoin. Commençons par les travaux qui remettent en question l'efficacité de l'exécution simultané sur CPU et GPU. Stafford et al. [144] analysent d'abord les gains potentiels de la répartition hétérogène de la CC d'un kernel, et fournit ensuite un modèle aidant les programmeurs à redistribuer cette CC. L'approche décrite n'est pertinente que lorsque l'on travaille sur des systèmes basés sur un seul kernel. Cependant, dans le cas de nombreux blocs fonctionnels nécessitant l'exécution de plusieurs kernels consécutifs comme CS-MoG, il est nécessaire de trouver un moyen optimal pour assurer une bonne distribution de leurs CC entre les PUs. Zhang et al. [145] prouvent à travers le portage de 42 programmes et l'analyse de leurs comportements de co-exécution sur des architectures hétérogènes intégrées, que seuls 8/42 atteignent les meilleures performances quand on utilise les deux PUs. Cependant 24/42 programmes sont plus rapides lorsqu'on utilise uniquement des GPU, et 7/42 quand on utilise des CPU. Les 3/42 programmes restants montrent peu de préférence de performances pour les différentes PUs. CS-MoG n'est pas inclus dans cette étude, mais nous avons remarqué que les algorithmes de segmentation d'images qui lui ressemblent, comme "k-means" et "Heart Wall", ont été classés dans la première catégorie. Nos résultats confirment que l'exécution hétérogène est avantageuse pour CS-MoG.

Certaines méthodes d'implémentation hétérogène se basent sur une répartition statique déduite à partir du taux du parallélisme dans l'application (faible, élevé), ou grâce à d'autres caractéristiques des tâches à exécuter. Grewe et al. [146], par exemple, exploite simultanément les cœurs CPU et GPU dans les plateformes de bureau (Desktop). Une telle allocation des tâches aux cœurs de calcul peut améliorer la vitesse de traitement et l'efficacité énergétique. Shen et al. [147] proposent un framework pour accélérer les applications déséquilibrées sur les plateformes hétérogènes. Il s'agit d'applications où chaque élément de données peut nécessiter une charge de calcul relativement différente. L'infrastructure de ce

framework est capable de détecter les caractéristiques de la CC de l'application, de faire des choix en fonction des solutions parallèles disponibles et de la configuration matérielle, et d'obtenir automatiquement la distribution optimale des tâches et des données sur les PUs. Plus importants encore, les travaux présentés dans [148] proposent une approche écoénergétique de répartition à l'exécution des threads pour exécuter des applications OpenCL simultanément sur les cœurs CPU et GPU tout en satisfaisant des exigences de performances. Cette étude cible l'exécution parallèle des applications qui sont indépendantes en matière de données et de calculs, ce qui n'est pas le cas de CS-MoG : ce dernier est décrit par une succession de trois kernels, qui sont exécutés de manière itérative. La complexité de ceci apparaît lors de la gestion des transferts de données entre les PUs ; d'autant plus que la quantité à transférer est lié au contenu des données, et change dynamiquement durant l'exécution. (Illustration sur la Fig. 5.3.c). Un autre problème est que cette approche ne prend pas en considération les dépendances entre les itérations, où un modèle persistant doit être maintenu en mémoire. Ce modèle, dans le cas d'applications comme CS-MoG, peut être partiellement déplacé d'une PUs vers une autre lorsque le partitionnement est mis à jour au moment de l'exécution.

Malgré les avantages que représentent le partitionnement statique présentés dans [147], [146] et [148], cette technique n'est pas applicable pour modifier la distribution des threads au moment de l'exécution. Ainsi, elle peut être moins efficace pour des applications telles que CS-MoG. En effet, pour avoir plus de performances, l'irrégularité des données doit être prise en compte pour mettre à jour à l'exécution les points d'équilibre, et ne pas utiliser uniquement des points estimés durant la phase d'entrainement.

D'autres méthodes se basent sur les performances relatives des PUs pour l'équilibrage des CC. Teodoro et al. [149] développent une stratégie d'implémentation des algorithmes qui sont représentés par le problème de propagation des fronts d'onde (Irregular Wavefront Propagation Algorithms) comme la reconstruction morphologique et la transformation de distance euclidienne. Cela est effectué à l'aide d'une structure de file d'attente à plusieurs niveaux sur des machines hybrides CPU-GPU. Des travaux comme [150]-[152] vont plus

bas, dans les détails des codes ciblés, pour répartir dynamiquement les boucles de traitement sur GPU et CPU. Cela peut être efficace, lorsqu'on travaille sur des codes simples contenant peu de boucles. Cependant, dans le cas de programmes plus sophistiqués comme CS-MoG, cette solution est difficilement applicable.

Li et al. [153] et Augonnet et al. [154] fournissent une approche, simple mais efficace, de partitionnement de graphes basé sur l'heuristique de séparation et de connexion pour optimiser le coût de transfert des données. Cette méthode est très puissante lorsqu'on travaille sur des graphes complexes contenant de nombreuses tâches (nœuds) avec des dépendances. Cependant, pour des petits graphes contenant moins de tâches (trois dans le cas de CS-MoG), les possibilités de distribution de ces tâches sont très limitées pour obtenir des performances suffisantes (illustration sur la Fig. 5.3.a). La solution à ce problème peut être de couper les données en petits grains, puis de générer un graphe plus grand contenant plusieurs sous-tâches. Par contre, sa gestion peut générer un temps supplémentaire important. De plus, la mise à jour du partitionnement durant l'exécution n'est pas triviale.

## 2.5 Conclusion

Nous avons présenté dans ce chapitre, les architectures matérielles hétérogène que nous ciblons dans ces travaux de thèse. Nous avons détaillé leurs caractéristiques fonctionnelles ainsi que les avantages qu'elles offrent pour le calcul parallèle de haute performance. Malgré ces avantages, nous avons constaté que les implémentations de MoG sur CPU et GPU, cités dans l'état de l'art, sont limités en terme de performance en raison des aspects suivants :

— CS n'est pas intégré dans l'implémentation parallèle de MoG ni sur les multi-cœurs CPU, ni sur GPU.

— la plupart des optimisations sur CPU ciblent l'algorithme lui-même sans exploiter au maximum de l'architecture.

— l'indépendance des threads n'est pas totalement assurée sur CPU dans la littérature, ce qui limite la scalabilité des performances à un facteur de 3,5 quand 12 cœurs

physiques sont utilisés.

— les méthodes proposées pour l'équilibrage de charges entre CPU et GPU se concentrent sur des cas extrêmes en terme du nombre et de la connectivité des kernels, qui ne représentent pas le cas de CS-MoG.

Notre objectif, dans les prochains chapitres, est d'interroger tous ces points pour améliorer les performances de CS-MoG sur CPU, GPU et CPU-GPU.

Chapitre 3

# Implémentation et optimisation de CS-MoG sur CPU

## Sommaire

## 3.1 Introduction

Une bonne exploitation matérielle des CPU-GPU signifie obligatoirement une mise en adéquation de chaque kernel avec chaque type de processeur. Ensuite, une exploitation simultanée et équilibrée des deux PUs est nécessaire pour minimiser le temps de traitement. Dans ce chapitre, nous présentons nos contributions concernant l'implémentation de CS-MoG sur CPU. Notre objectif est d'étudier et améliorer le comportement de chaque kernel de manière indépendante sur ce type de processeur. Nous commençons, dans la première partie de ce chapitre, par les optimisations de niveau cœur en utilisant la technique de vectorisation. Ensuite, nous passons à l'amélioration de la scalabilité des performances quand plusieurs cœurs de calcul sont utilisés grâce à la technique de multithreading. Ces contributions sont finalement comparées avec les versions les plus communes de la littérature.

## 3.2 Aperçu sur les niveaux de parallélisme

La parallélisation d'un programme séquentiel n'est pas évidente. Elle consiste à partager les calculs entre un grand nombre d'unités de traitement (cœurs), comme le cas de la projection d'une image (Eq. 1.11), ou le raffinement du premier plan (Eq. 1.12 et Eq. 1.13). Dans la version séquentielle, ces calculs sont effectués élément par élément via des boucles. Dans notre travail, toute boucle effectuant répétitivement une opération $Op$ est répartie équitablement entre les $C$ cœurs CPU (Fig. 3.1 .a). Avant cela, une vectorisation des données au niveau de chaque cœur (Fig. 3.1 .b) doit être réalisée afin de maximiser son exploitation (voir section 3.3).

## 3.3 Vectorisation

La plupart des architectures SIMD récentes possèdent de grands registres dans leurs processeurs. Par exemple, certaines architectures Intel ont jusqu'à 512 bits par registre [17]. Les performances peuvent être considérablement augmentées lorsque les mêmes opérations

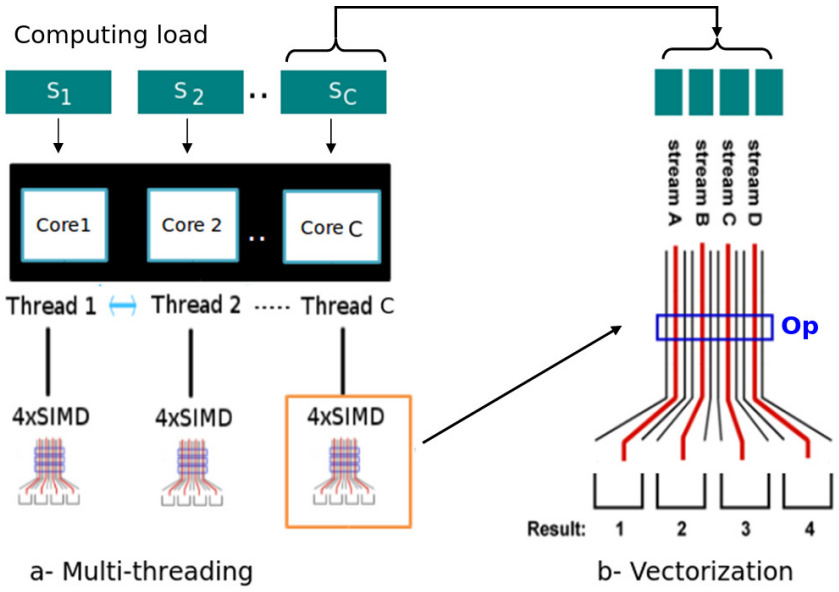


FIGURE 3.1 – Aperçu sur les niveaux de parallélisme pour une architecture multi-cœur CPU : a) multithreading et b) vectorisation.

sont effectuées sur plusieurs éléments de données, comme dans le cas du traitement de signal et d'image. Les langages vectoriels ou multidimensionnels sont utilisés pour généraliser les opérations scalaires afin de les appliquer de manière simultanée à des vecteurs et des tableaux.

Pratiquement, la vectorisation peut être réalisée de deux manières. Certains compilateurs récents essaient de détecter automatiquement les régions de calcul pouvant être vectorisées. Cependant, l'utilisation de cette approche nécessite que les portions à vectoriser soient regroupées dans des boucles simples. Par conséquent, cela n'est pas efficace dans le cas de MoG où la majorité des calculs sont conditionnés suivant un flux de contrôle (branches) [136]. Par exemple, dans la première étape où les correspondances entre les projections et les FDPGs sont calculées, la boucle de calcul est rompue dès qu'une première correspondance est trouvée. Une autre solution consiste à déterminer manuellement les régions du code pouvant être vectorisées par le programmeur, puis effectuer cette tâche manuellement à l'aide du langage assembleur ou des fonctions C/C ++ intégrées, appelées « intrinsèques ».

**Vectorisation**

Pour les architectures Intel et AMD, la famille de jeu d'instructions la plus courante est SSE (Streaming SIMD Extensions [155] [156] ) et AVX (Advanced Vector Extensions [17]). Initialement, nous avons choisi la version SSE2 pour valider notre approche, mais la même méthodologie peut être suivie en utilisant d'autres jeux d'instructions comme AVX. SSE2 est une version du jeu d'instructions SIMD conçu par Intel et contenant 70 nouvelles instructions, dont la plupart fonctionnent sur des données à virgule flottante en simple précision. Il comprend 144 instructions au total, et gère des registres de 128 bits pour les entiers ainsi que des données à virgule flottante en simple et double précision. OpenMP offre la possibilité de réaliser la vectorisation facilement à condition que le code soit écrit de manière à permettre cette opération. Il s'agit d'une simple directive à ajouter avant chaque section de code à vectoriser. Une comparaison de performance nous a montré que le temps gagné en utilisant OpenMP est le même que lors de l'écriture du code en assembleur qui demande un plus grand effort de programmation. Ainsi, après avoir amélioré notre code, nous avons adopté cette interface pour effectuer toutes les vectorisations possibles.

Dans le cas de CS-MoG, le défi consiste à détecter les portions du code qui prennent plus de temps de traitement, puis à déterminer le nombre d'opérations pouvant être parallélisées dans ces zones. Une première mesure du temps consommé par les différentes parties de CS-MoG montre que les portions pouvant être vectorisées représentent environ 42% du temps d'exécution. Après cette analyse, la question qui se pose est de savoir comment les données seront regroupées dans les registres d'entrée pour paralléliser ces régions. Pour des scènes dynamiques, la plupart des FDPGs sont initialisées et utilisées par la plupart des projections. Cela signifie que les calculs (distance MD) sont effectués pour la majorité de ces FDPGs. Notre approche consiste donc à fixer un nombre maximal de FDPGs égal à 4 pour chaque projection (valeur suffisante et satisfaisante même dans le cas de scènes très dynamiques [6]). Ensuite, au lieu de vectoriser le traitement de plusieurs projections, nous parallélisons les calculs effectués au niveau des FDPGs de la même projection. Ainsi, le calcul des distances MD est effectué selon le processus illustré sur la Fig. 3.2.

Etant donné que tous les calculs sont effectués en virgule flottante simple précision et

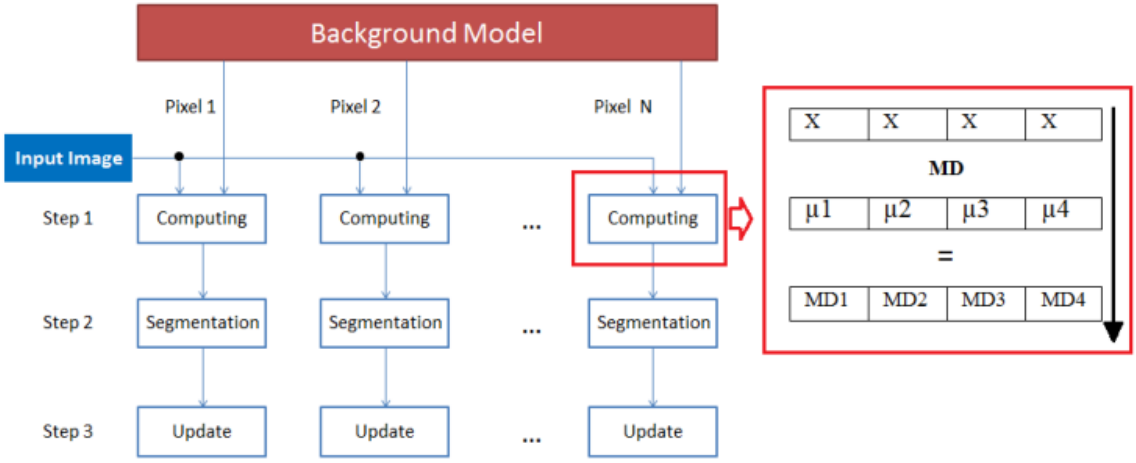


FIGURE 3.2 – Calcul des distances Mahalanobis (MD) en utilisant la vectorisation.

avec un jeu d'instructions dont la longueur des registres est de 128 bits, nous parvenons à exécuter 4 opérations simultanément. Lorsque la séquence d'images est plus statique (contient moins d'objets mobiles), certaines projections n'utilisent qu'une ou deux FDPGs au lieu de 4. Cela rend la vectorisation moins efficace ; toutefois, ces situations où la scène est vide ne sont pas très critiques à la base, vu qu'elles consomment moins de temps. D'une façon générale, en notant $p \in [0, 1]$ la portion de l'algorithme qui peut être vectorisée et $S \in \mathbb{N}$ le facteur d'accélération possible pour cette portion ; le temps de traitement $T_v$ pouvant être atteint après la vectorisation est exprimé comme suit (loi d'Amdahl [157]) :

$$T_v = (\frac{p}{S} + (1 - p)).T_i, \tag{3.1}$$

où $T_i$ est le temps initial avant la vectorisation. Le facteur d'accélération $S$ dans le cas de la vectorisation peut être exprimée comme étant le produit du nombre d'éléments de données $N$ pouvant être contenus dans les registres du processeur ($N = 4$ dans notre cas), et le pourcentage moyen réel des FDPGs utilisés par les projections $Kr \in [0, 1]$. Ainsi, l'équation précédente devient :

$$T_v = (\frac{p}{N.K_r} + (1 - p)).T_i \tag{3.2}$$

L'accélération théorique maximale que nous pouvons atteindre est alors :

$$A = \frac{T_i}{T_v} = \frac{1}{(\frac{p}{N.K_r} + (1-p))}. \tag{3.3}$$

Le gain en pourcentage peut être exprimé comme suit :

$$G = \frac{T_i - T_v}{T_i} = 1 - \frac{1}{A}. \tag{3.4}$$

Nos résultats présentés dans la section suivante montrent que nous avons atteint une vitesse d'accélération proche des valeurs attendues.

## 3.4 Amélioration de la scalabilité

Après la vectorisation des calculs sur un cœur (niveau des instructions), notre deuxième volet d'optimisation est la parallélisation multi-cœur utilisant le multithreading. Idéalement, l'accélération grâce au multithreading doit évoluer linéairement en fonction du nombre de cœurs, mais malheureusement, très peu de programmes peuvent atteindre cet état comme indiqué dans la loi d'Amdahl [157]. Cela peut être dû à plusieurs facteurs. Dans le cas de CS-MoG, nous avons constaté que la scalabilité est influencée par les éléments suivants :

— La répartition non-équitable des CC entre les threads : le terme équitable dans ce contexte ne signifie pas forcément que ces cœurs reçoivent la même quantité de données à traiter, mais plutôt la même quantité de calcul. La distinction entre ces deux aspects peut être constatée quand l'algorithme en question est très influencé par le contenu de l'image.

— Le temps (overhead) consommé par le support exécutif du multithreading (comme OpenMP) : lorsque la méthode choisie pour la gestion de threads, via un support exécutif donné, n'est pas optimale, ce support lui-même peut consommer un temps considérable qui peut, dans certains cas, masquer le temps gagné par le fait d'utiliser

plusieurs cœurs.

— La présence de sections critiques partagées entre les threads dans le code : l'indépendance totale des threads est une condition indispensable pour obtenir la meilleure scalabilité.

— Le mauvais choix de l'environnement d'expérimentation et des options de compilation.

— La présence de plusieurs boucles imbriquées ou dispersées contenant plusieurs branches divergentes.

— L'accès des threads à la mémoire peut engendrer une latence importante quand il n'est pas optimisé.

Un bon compromis entre les deux premiers points a été proposé dans [51]. En effet, contrairement aux applications de prétraitement comme le filtrage qui s'applique de la même façon sur toute l'image, la charge de certains algorithmes comme MoG dépend du contenu de l'image plus précisément des objets mobiles qui s'y trouvent. Ainsi le temps de traitement d'un même nombre de pixels dépond du contenu qu'ils représentent. Une répartition équilibrée des pixels sur les cœurs n'aboutit donc pas nécessairement à une charge de calcul équilibrée. Un tel déséquilibre se traduit par une perte de performance vu que certains cœurs vont finir leurs tâches avant les autres et vont rester inactifs. Szwoch et al. [51] ont choisi une méthode de planification optimale des tâches garantissant un équilibrage de CC entre les threads avec un coût d'exécution minimal. En effet, quand les données ne sont pas homogènes, ce qui est souvent le cas, [51] a procédé comme suit dans son implémentation parallèle : l'image est divisée en un grand nombre de très petites portions appelées tâches, puis ces tâches sont attribuées aux threads en alternance. Ainsi, le thread qui terminera sa portion recevra une nouvelle partie à traiter. Bien que cette méthode dynamique génère une surcharge liée au support exécutif qui gère les permutations entre les threads, elle permet de gagner un temps considérable grâce à cet équilibrage de charge.

Dans cette thèse nous agissons sur les autres éléments qui influencent la scalabilité. En ce qui concerne la présence de sections critiques entre les threads, nous avons constaté

dans le code fourni par [51] que certaines variables sont partagées entre les threads. En effet, lorsque plusieurs threads veulent accéder simultanément aux mêmes zones mémoire, le système d'exploitation ou le support exécutif interviennent pour organiser l'accès à ces zones critiques. Ainsi, cet accès est protégé par une exclusion mutuelle, en forçant les threads à y accéder exclusivement, ce qui influence négativement les performances globales. C'est pourquoi notre travail s'est concentré sur cet élément. En utilisant des copies locales indépendantes de toutes les variables, nous avons évité la concurrence entre les threads et avons obtenu une bonne scalabilité, comme indiqué dans la section (3.5.4).

Certaines précautions doivent être appliquées durant l'expérimentation pour ne pas influencer les performances. L'environnement utilisé, par exemple, est facteur très important. En effet, nous avons constaté que l'utilisation fréquente de certaines fonctions (comme celles de la bibliothèque OpenCV) pour le chargement des images influence la mesure du temps dans les portions du code qui suivent. Ce phénomène peut s'expliquer par le caractère non bloquant de certains appels, qui déclenchent les compteurs de temps avant même que leurs tâches ne soient terminées. Le fait que ces fonctions qui chargent les images ne sont pas parallélisables, et que leur temps consommé est compris involontairement dans la mesure du temps de traitement, peut engendrer des mesures incorrectes. Pour éviter ce genre d'effets dans la mesure des performances, nous chargeons toute une séquence d'images du disque dans la mémoire en un seul appel de lecture avant de lancer la boucle de traitement. Nous avons constaté que la différence entre les temps mesurés par ces deux méthodes est très importante.

L'environnement et les options de compilation influencent également les performances de l'algorithme. Dans notre cas, où nous utilisons l'architecture Intel, la génération du code exécutable avec le compilateur Intel correspondant [125] améliore le temps de traitement presque 2 fois par rapport à l'utilisation d'un compilateur standard tel que GCC [124]. Néanmoins, le fait que ce dernier est Open Source ainsi que sa grande adaptabilité à une large gamme d'architectures le rend plus commun. Les options d'optimisation utilisées pour générer le fichier exécutable pour l'architecture ciblée sont également importantes. Dans

nos tests, nous utilisons le niveau supérieur d'optimisation du compilateur (-O3) [127] ce qui permet d'atteindre le meilleur niveau de performances.

Les deux derniers éléments (structures de données et optimisation des boucles imbriquées) qui influencent la scalabilité sont traités en détails dans notre publication [158]. Nous avons constaté, dans le code fourni par [51], que MoG contient quatre boucles qui balayent respectivement les lignes, les colonnes, les FDPGs et les couleurs dans chaque image. Notre contribution consiste à fusionner ces boucles et à utiliser moins de variables d'incrémentation. Cet aspect permet un adressage simple avec un accès contigu à la mémoire. Ainsi, cela réduit le temps de lecture/écriture et permet également aux différents threads de profiter de la mémoire cache. De plus, nous avons amélioré la structure de MoG en réduisant le nombre d'instructions de contrôle, ce qui a permis de réduire la divergence entre les threads concurrents, et a amélioré l'équilibrage de la charge entre eux. Les mêmes optimisations ont été également effectuées sur les deux autres kernels de CS-MoG (Projection et Raffinement).

En combinant tous les éléments mentionnés ci-dessus, les résultats obtenus présentés dans la section suivante montrent que nous avons pu atteindre des performances et une scalabilité satisfaisantes. Cela permettra à cet algorithme d'être utilisé dans des applications temps réel pour traiter même des images de très haute résolution.

## 3.5 Résultats et synthèse

Cette section qui présente les résultats expérimentaux est divisée en cinq parties : la première est consacrée à la présentation de l'environnement de mesure : les plateformes utilisées, les méthodes auxquelles notre implémentation est comparée, et les datasets ciblés. La deuxième est dédiée à la présentation des résultats obtenus en termes de précision sur CPU. La troisième concerne l'illustration des performances obtenues grâce à notre première contribution sur cette plateforme qui est la vectorisation de CS-MoG. Dans la quatrième partie, nous présentons les résultats obtenus grâce à l'amélioration de la scalabilité de

cet algorithme. Finalement, la dernière partie est consacrée à la fusion de tous les tests précédents pour illustrer les résultats globaux.

### 3.5.1 Environnement de mesure

Afin de généraliser et tester la fiabilité de notre approche d'implémentation, Celle-ci a été testée sur quatre plateformes informatiques différentes en termes de capacités de calcul : deux stations de travail et deux ordinateurs portables. Le but de ce choix est de faire des évaluations avec plusieurs combinaisons de processeurs CPU-GPU. En effet, nous utilisons des CPU (contenant 2 à 6 cœurs), et des GPU ayant des performances distinctes. Les caractéristiques détaillées de ces plateformes sont présentées dans la Table. 3.1 avec l'environnement utilisé.

| | Laptop1 | Laptop2 | Workstation1 | Workstation2 |
|---|---|---|---|---|
| CPU | 2 cores i7-6500U | 4 cores i5-7300HQ | 4 cores i7-920 | 6 cores i7-5820k |
| GPU | GeForce 920MX | GeForce 1050 | GeForce 680 | GeForce 1080TI |
| Os | Linux Mint 18.1 | | | |
| Compilateur | GCC 5.4.0 (OpenMP 4.0) et ICC 15.0.3 | | | |

TABLE 3.1 – Caractéristiques des quatre plateformes matérielles utilisées pour tester les performances de la soustraction de fond.

Le portage de notre version améliorée de CS-MoG (Improved CS-MoG) sur CPU est comparé, d'une part, avec sa version de base (BasicMoG) qui est sans CS ; et d'autre part, à quatre autres implémentations, également sans CS, présentées dans l'état de l'art : deux versions implémentées dans OpenCV3.4.3 (que nous appelons CvMoG1 et CvMoG2), ainsi que celles présentées dans [141] et [51], que nous appelons successivement Original-Kovacev et Original-Szwoch) en référence à leurs auteurs. Ensuite, nous avons amélioré ces quatre versions de référence en ajoutant le CS et les appelons CS-CvMoG1, CS-CvMoG2 et CS-Kovacev et CS–Szwoch dans le reste de ce mémoire. Un récapitulatif sur toutes ces versions est présenté dans la Table. 3.2 suivante :

Comme la comparaison des performances ne peut avoir de sens sans prendre en compte la précision des résultats, cette dernière est testée à l'aide de plusieurs datasets téléchargés

| Version | Présentation |
|---|---|
| **Basic MoG** | Notre implémentation de base sans CS [127][16] |
| **Improved CS-MoG** | Notre implémentation améliorée avec le CS [159] |
| CvMoG1 | Version implémentée dans OpenCV3.4.3 basée sur [126] |
| CS-CvMoG1 | Notre amélioration de CvMoG1 grâce au CS |
| CvMoG2 | Version implémentée dans OpenCV3.4.3 basée sur [53] |
| CS-CvMoG2 | Notre amélioration de CvMoG2 grâce au CS |
| Original-Kovacev | Implémentation effectuée par Kovacev et al. dans [141] |
| CS-Kovacev | Notre amélioration de Original-Kovacev grâce au CS |
| Original-Szwoch | Implémentation effectué par Szwoch et al. dans [51] |
| CS-Szwoch | Notre amélioration de CS-Szwoch grâce au CS |

TABLE 3.2 – Méthodes comparées en termes de performances et de précision pour réaliser la soustraction de fond.

de [160]. Etant donné que la soustraction de fond s'effectue d'une façon ponctuelle et indépendante pour chaque pixel, l'utilisation de cette technique pour la détection des véhicules ou pour autre types d'objets mobiles peut être évaluée de la même façon. Ainsi nous avons profité de cette caractéristique pour évaluer et généraliser notre implémentation pour la détection de différents types de premiers plans dans des situations distinctes. De ce fait, nous avons choisi cinq datasets contenant plusieurs niveaux de luminosité (jour et nuit), plusieurs types d'objets (véhicules et personnes), environnements (intérieur, extérieur etc) et types de caméras (standard ou thermique). Les caractéristiques de chacun de ces datasets sont présentées dans la Table. 3.3. suivante :

| Datasets | Ref. in [160] | Scène |
|---|---|---|
| Dataset 1 | Library | Détection d'une personne dans une bibliothèque filmée par une caméra thermique |
| Dataset 2 | FluidHighway | Route bondée de véhicules roulant à haute vitesse pendant la nuit |
| Dataset 3 | Office | Une personne à intérieur de son bureau pendant le jour |
| Dataset 4 | Pedestrians | Détection de piétons dans une scène ensoleillée à l'extérieur |
| Dataset 5 | PETS2006 | Des personnes évoluant dans un lieu public avec un bon éclairage |

TABLE 3.3 – Caractéristiques des cinq datasets utilisés pour tester la précision de la soustraction de fond.

Pour mieux illustrer ces datasets, nous représentons dans la Fig. 3.3 une capture de chacun avec une mesure de sa dynamicité. Cette grandeur importante représente le nombre moyen de FDPGs utilisées par les projections de chaque dataset. La dynamicité d'un dataset est la caractéristique principale, liée à son contenu, qui peut influencer les performances.

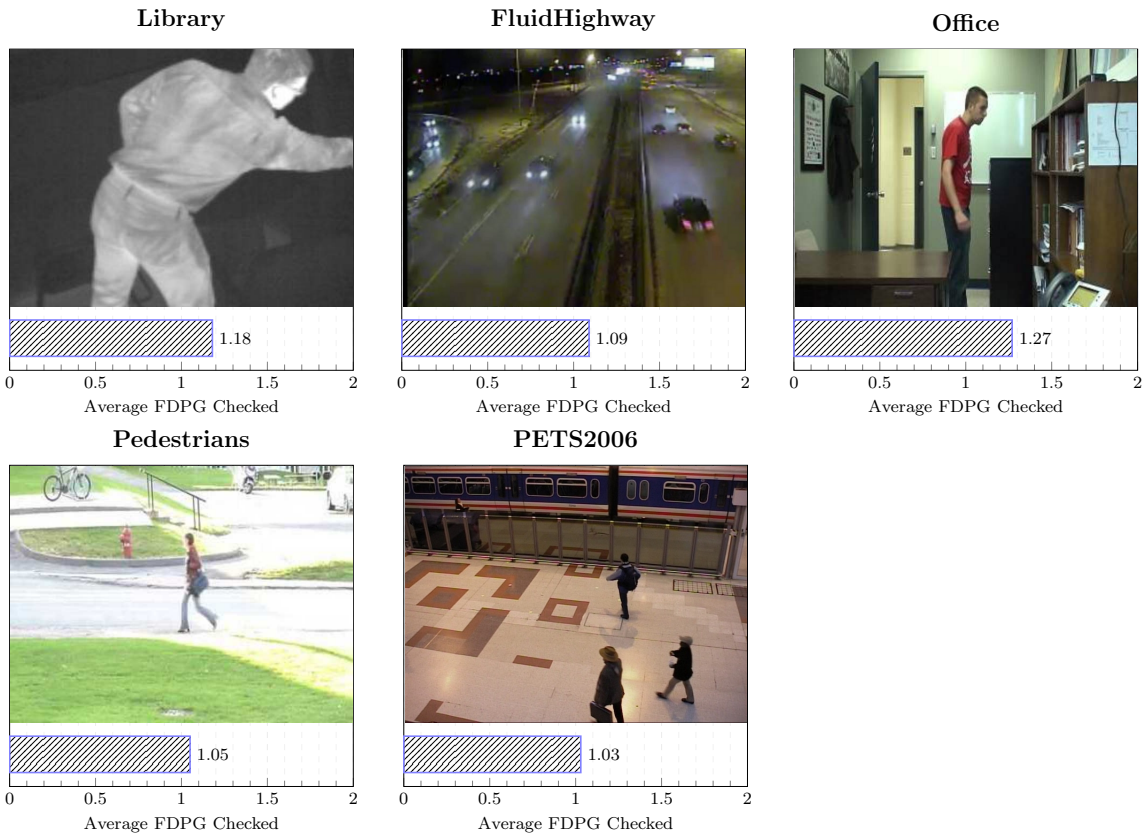


FIGURE 3.3 – Capture de chaque dataset avec une valeur représentant sa dynamicité.

Tous nos tests, y compris l'estimation de dynamicité, sont effectués sur les 300 images les plus pertinentes de chaque dataset. Il s'agit des images contenant le maximum d'événements. Ces dernières ont été redimensionnées pour avoir la même taille de 640x480, et elles sont toutes converties en niveaux de gris. Afin d'éviter les erreurs de mesure, chaque test est répété 10 fois et moyenné. Les valeurs de dynamicité représentées dans la Fig. 3.3 sont obtenues en faisant leurs moyennes lors du traitement des 300 images. La Fig. 3.4 nous donne une idée sur l'évolution de la dynamicité au niveau de chaque dataset avec le

temps.

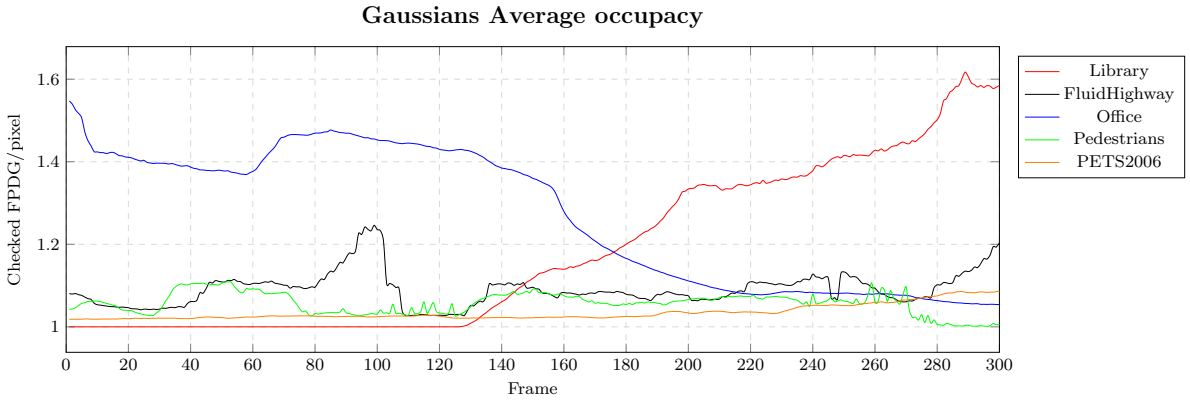


FIGURE 3.4 – Evolution de la dynamicité des 300 images les plus significatives de chaque dataset.

Avec cette variété de datasets couvrant une grande diversité de scénarios, nous serons capables de comparer la précision de notre implémentation avec celles des méthodes présentées précédemment.

### 3.5.2 Evaluation de la précision

La précision est évaluée en utilisant le F1-score [161] qui est une grandeur commune dans la littérature et qui représente à la fois la précision (ou la valeur prédictive positive) et le rappel (ou la sensibilité). Ces deux grandeurs sont suffisantes pour mesurer l'exactitude des résultats d'un algorithme de détection, vue que : la précision représente la proportion des éléments pertinents parmi l'ensemble des éléments détectés, tandis que le rappel représente la proportion des éléments pertinents détectés parmi l'ensemble des éléments pertinents. La valeur du F1-score est dans l'intervalle [0,1] (1 = cas idéal). Pour toutes les versions étudiées, les paramètres de MoG sont choisis de manière à obtenir les meilleurs résultats. Chaque graphique de la Fig. 3.5 représente une comparaison en termes de précision entres les 10 versions étudiées sur un des datasets. De plus, une courbe supplémentaire représente un résultat global (moyenne) sur tous les datasets avec un classement des méthodes selon le F1-score. Ces résultats ne changent pas selon la plateforme utilisée. Ces

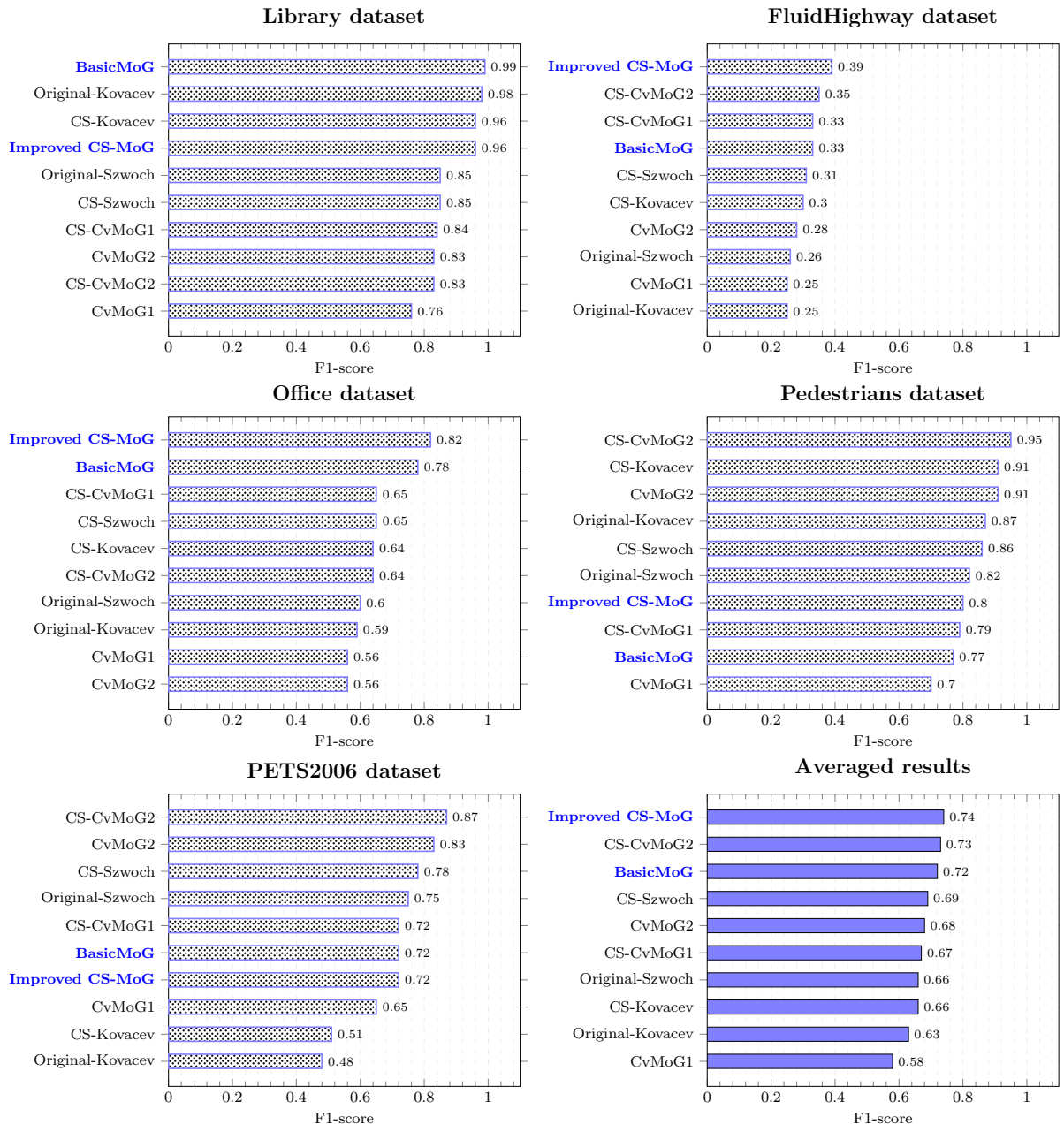


FIGURE 3.5 – Représentation des résultats obtenus sur la précision des méthodes étudiées pour chaque dataset sur CPU.

courbes peuvent être interprétées selon trois aspects : l'exactitude globale des méthodes de base, l'effet de l'intégration du CS, et l'effet de la dynamicité des datasets.

- En ce qui concerne la précision moyenne des cinq versions de base, nous consta-

tons que notre BasicMoG représente le meilleur F1-score moyen, suivi par CvMoG2, Original-Szwoch, puis Original-Kovacev tandis que CvMoG1 reste le moins précis. Bien qu'il s'agisse du même algorithme (MoG), ces différences en termes de précision sont dues aux modifications qu'applique chaque auteur afin de réduire le temps du traitement, ce qui influence la précision. Notre BasicMoG donne les meilleurs résultats vus que nos améliorations se concentre sur les aspects liés à l'exécution sur le matériel et non pas sur l'algorithme lui-même.

- Après avoir ajouté le CS, nous avons globalement amélioré la précision de toutes les versions avec des taux différents. Cela pourrait apparaitre anormal vu que la réduction de la dimensionnalité des données traitées doit forcément réduire la qualité de la détection, ce qui est le cas pour certains datasets, cependant nos mesures réfutent globalement cette intuition. En effet, la segmentation zonale 8x8 au lieu de la segmentation en pixels permet d'éliminer le bruit ponctuel présent dans nos masques d'objets mobiles. Concrètement, avant même l'étape de raffinement, une zone est classée soit entièrement comme premier plan ou l'inverse. Par conséquent, il est impossible d'obtenir des particules dans le premier plan qui ont une dimension moins de 8x8. Le taux d'amélioration au niveau de chaque méthode est plus important quand sa version de base génère beaucoup de bruit comme le CvMoG1 qui n'est plus dans la dernière position des cinq versions compressées. Néanmoins, le seul cas où le CS n'influence pas positivement la précision est quand le dataset lui-même ne contient pas du bruit. Pour analyser en détails l'effet du CS sur la précision des méthodes comparées, nous étudions chacun des graphiques indépendamment, et remarquons que les résultats obtenus sont différents. Pour la vision thermique (Library dataset), où nous avons un grand premier plan et un arrière-plan propre en termes de bruit, nous constatons que l'utilisation du CS n'est pas avantageuse. Toutefois, quand il s'agit d'une vision de nuit (FluidHighway dataset) contenant des bruits importants, les versions compressées sont ainsi plus précises. Ce même résultat est obtenu pour les vidéos contenant des bruits liés à l'éclairage, qu'elles soient filmées dans un envi-

ronnement fermé (Office dataset), sous les rayons du soleil (Pedestrians dataset), ou quand il s'agit d'un grand espace fermé éclairé artificiellement (PETS2006 dataset).

- En comparant les résultats des méthodes étudiées du point de vue de la dynamicité des images, nous constatons que nos deux implémentations donnent de meilleurs résultats quand les scènes sont plus dynamiques comme le cas des datasets Library, Office et FluidHighway. Cependant, la qualité de la détection est moins bonne quand il s'agit des scènes contenant de très petits objets, et celles qui sont moins dynamiques comme le cas des datasets Pedestrians et PETS2006. Cela ne représente pas un grand inconvénient vu que dans la majorité des cas réels et plus particulièrement la détection des véhicules, les scènes sont très dynamiques ce qui donne un avantage à notre implémentation.

Pour bien illustrer les conclusions de l'analyse ci-dessous, nous présentons dans la Fig. 3.6 les résultats visuels de la soustraction de fond obtenus grâce à notre Basic MoG et puis CS-MoG amélioré. Une capture de chaque dataset est présentée avec le masque correspondant aux objets mobiles détectés.

Après avoir étudié la précision des différentes implémentations de MoG sur CPU, nous consacrons les deux prochaines sections aux résultats en termes de performance, qui dépassent ceux présentés dans l'état de l'art.

### 3.5.3 Vectorisation

Comme présenté dans la section 3.3, la loi d'Amdahl nous permet de prévoir les gains de performance qui peuvent être atteint via la vectorisation. Pour appliquer cette loi dans notre cas, il nous fallait déterminer les paramètres $p$ et $Kr$ de l'Eq. 3.3. Une première analyse du temps consommé par les différentes parties de CS-MoG montre que les boucles pouvant être vectorisées, $p$, représentent environ 42% de ce temps. Ensuite le pourcentage de FDPGs $Kr$ utilisés par les projections dans chaque dataset peut être déduit en divisant la dynamicité présentée dans la Fig. 3.3 par le nombre maximal des FDPGs qui est 4. Dans la suite de ce mémoire, nous appelons cette grandeur le "Taux d'occupation". La Table. 3.4

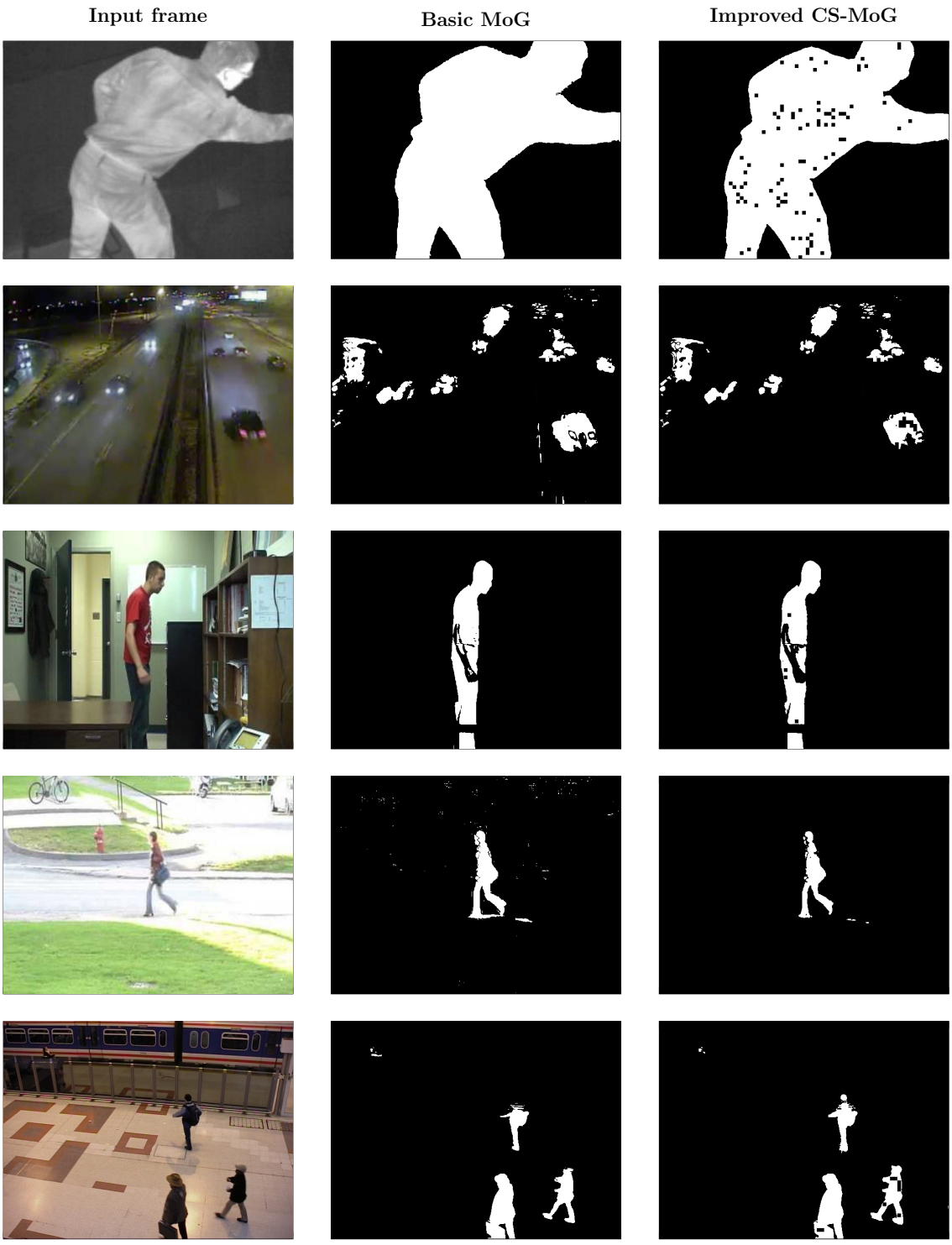


FIGURE 3.6 – Résultats visuels de la soustraction de fond obtenus grâce à notre Basic MoG et puis CS-MoG amélioré.

présente ces différentes grandeurs, avec les gains de performance prévus en appliquant la loi d'Amdahl, et ceux obtenus expérimentalement :

| | Dynamicité | Taux d'occupation Kr | Gain prévu | Gain obtenu |
|---|---|---|---|---|
| Library | 1.18 | 0.30 | 7% | **4.5%** |
| FluidHighway | 1.09 | 0.27 | 4% | **3%** |
| Office | 1.27 | 0.32 | 10% | **6%** |
| Pedestrians | 1.05 | 0.26 | 2% | **-1.2%** |
| PETS2006 | 1.03 | 0.26 | 1% | **-1.6%** |

TABLE 3.4 – Estimation des gains de performance en appliquant la loi d'Amdahl modifiée pour prendre en compte la dynamicité de chaque jeu de donnée (l'Eq. 3.3).

Cette évaluation montre que le gain potentiel apporté par la vectorisation peut atteindre $G = 10\%$, soit un facteur d'accélération $A = 1.1$. Cela est possible bien que nos datasets soient faiblement dynamiques, sachant que ce gain peut être encore plus important dans le cas des datasets très dynamiques. Expérimentalement, les accélérations obtenues sont un peu plus faibles que ce qui était prévu, mais elles restent très significatives et la tendance est respectée. Nous expliquons ces différences par le surcoût (overhead) généré par le processus de vectorisation, dans lequel des variables intermédiaires sont introduites et ainsi que des accès mémoire s'ajoutent. Il faut prendre en considération que ce coût peu masquer le gain obtenu par la vectorisation dans le cas des datasets ayant une très faible dynamicité comme le cas de PETS2006 et Pedestrians. Ainsi, pour ces deux derniers, nous n'arrivons pas à remarquer des gains mais plutôt quelques petites pertes, autour de 1%.

A partir de ce constat, et afin d'obtenir les meilleures performances possibles, nous recommandons d'utiliser la métrique de la dynamicité comme une base pour le choix entre l'utilisation de la version vectorisée ou non de CS-MoG. Cette métrique peut être estimée dans la phase d'entrainement sur un ensemble d'images représentatif, et elle est mise à jour à l'exécution. Ainsi, nous recommandons d'utiliser la version vectorisée du code quand les gains prévus par la loi d'Amdahl, dans le cas d'un jeu d'instructions SSE2, sont au moins supérieurs à 4%, qui est suffisant pour récompenser le coût de vectorisation. Ce seuil correspond à une valeur de dynamicité minimale de 9%.

Pour étudier l'influence de la plateforme matérielle sur la vectorisation, nous présentons dans la Fig. 3.7, les résultats détaillés de notre implémentation. Nous constatons que l'effet de cet aspect est négligeable lorsque nous utilisons le même jeu d'instructions.

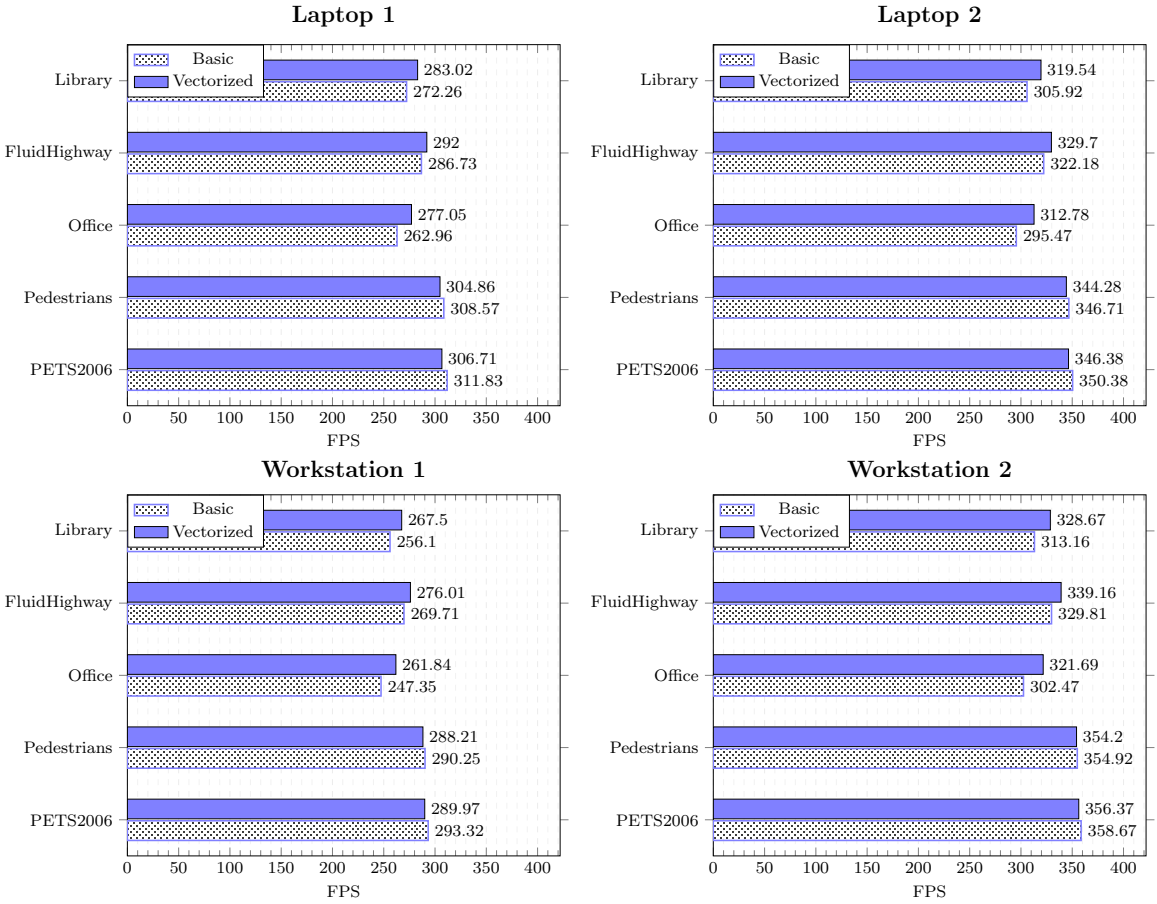


FIGURE 3.7 – Gains obtenus grâce à la vectorisation d'une partie des calculs de notre CS-MoG amélioré.

Afin de ne pas se limiter à un seul environnement de compilation, et voir si ce dernier a une influence sur les performances, nous avons recompilé notre code vectorisé avec le compilateur Intel ICC 15.0.3 [125] : tous les CPU que contiennent nos plateformes sont conçus par cette entreprise. La Fig. 3.8 représente les valeurs obtenues qui dépassement nos attentes. Le fait d'utiliser un compilateur dédié à l'architecture du CPU permet d'obtenir des gains de performance qui sont respectivement 33%, 28%, 34% et 30% pour les plateformes utilisées Laptop 1, Laptop 2, Workstation 1, et Workstation 2. Ces gains ne

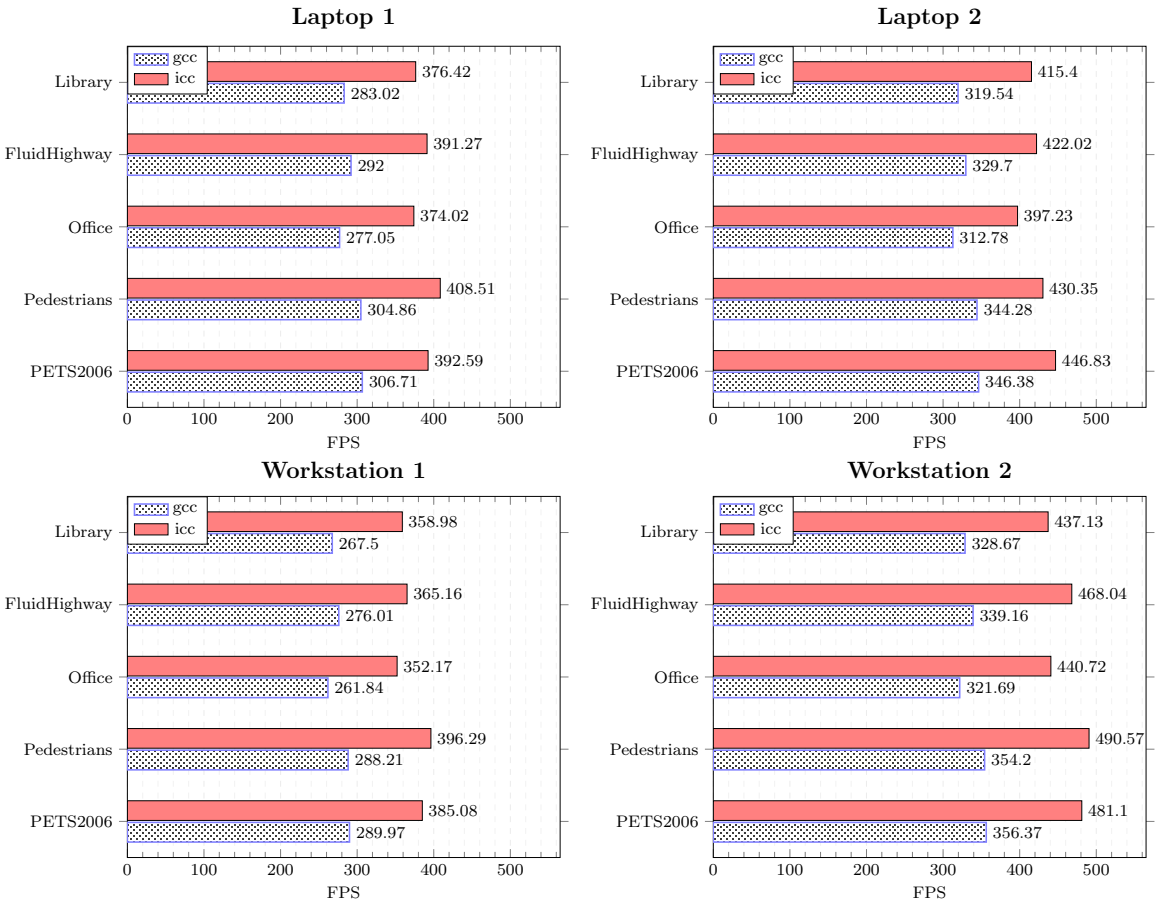


FIGURE 3.8 – Comparaison des performances obtenues sur CPU en utilisant une compilation avec GCC et avec ICC.

concernent pas uniquement les parties vectorisées mais la totalité du code. Il s'agit de toutes les optimisations implicites qui peuvent être effectuées par le compilateur sur la globalité des instructions et accès à la mémoire. Malgré les gains importants obtenus, l'inconvénient du compilateur ICC est qu'il n'est pas un logiciel libre, mais intégré dans une suite d'outils commerciale.

### 3.5.4 Scalabilité multi-cœur

Un autre point exploré au niveau du CPU est la scalabilité des performances lors de l'utilisation de plusieurs cœurs. Il s'agit de s'assurer que le temps de calcul est réduit d'une façon idéalement linéaire en fonction du nombre de cœurs de calcul que contient

la plateforme. En utilisant OpenMP comme interface de multithreading, nous comparons dans la Fig. 3.9 la scalabilité des versions compressées CS-CvMoG2, CS-Kovacev, CS-Szwoch et notre Improved CS-MoG sur la plateforme Workstation 2 contenant 6 cœurs de calcul. Les facteurs qui peuvent limiter la scalabilité ne sont pas liés au contenu des données

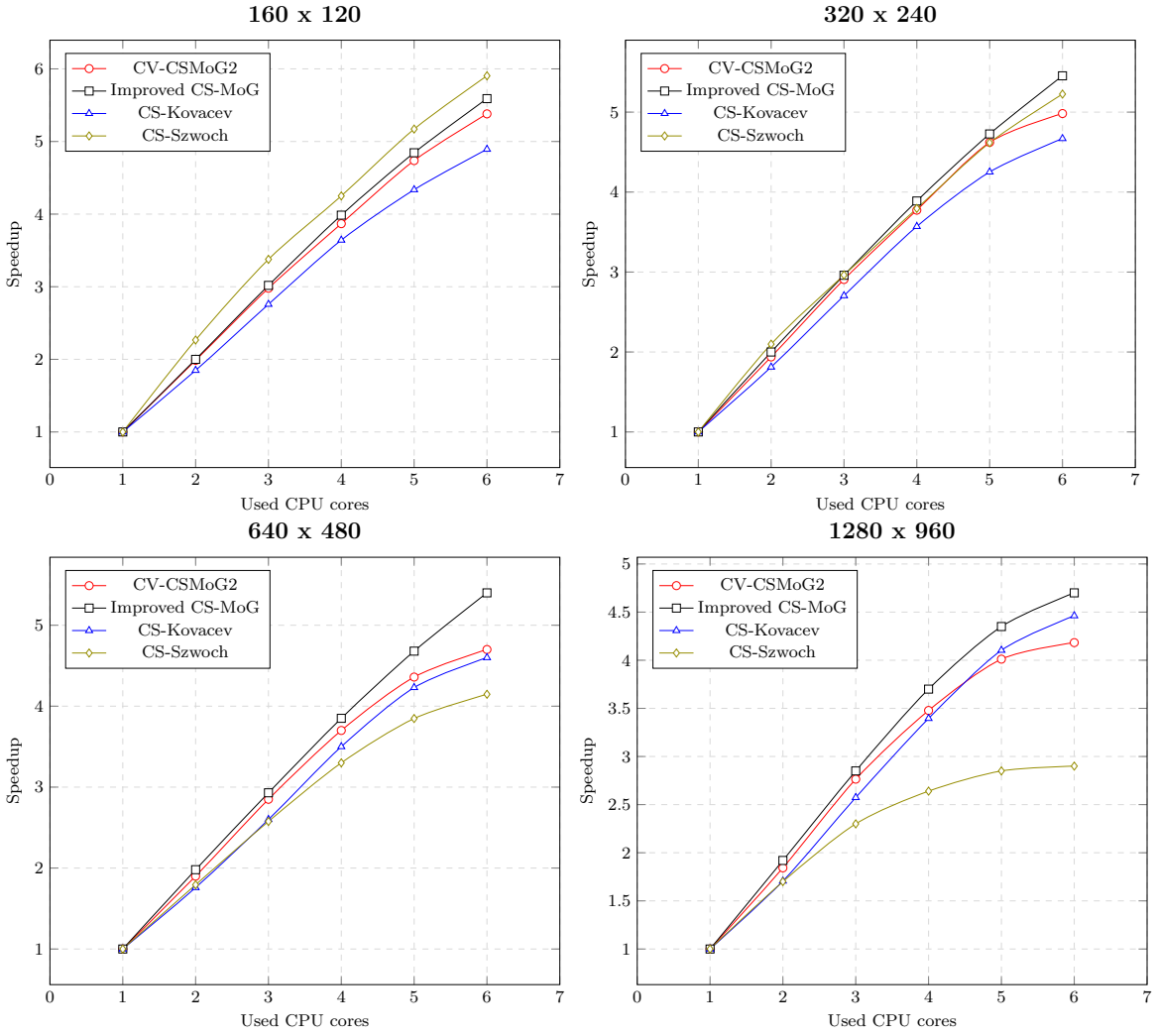


FIGURE 3.9 – Scalabilité des performances de CS-CvMoG2, CS-Szwoch, CS-Kovacev et de notre CS-MoG amélioré sur un processeur multi-cœur CPU. Le résultat obtenu est le même pour tous les datasets de la Table 3.3.

mais plutôt à leur taille : des écarts temporels entre les threads sont détectés lors de la répartition de longues boucles de calcul, même si cette répartition est équitable. Ces écarts sont négligeables quand on traite de petites quantités de données. Pour estimer cet effet

nous réalisons un test sur plusieurs tailles d'images redimensionnées à partir du datasets Library, vu que le résultat est le même pour tous les datasets. CS-CvMoG1 n'est pas inclus dans cette comparaison car son implémentation sur OpenCV ne prend pas en charge le multithreading.

La première observation est que, à part pour les très petites images de 160X120, notre Improved CS-MoG représente la meilleure scalabilité avec une accélération allant jusqu'à 5.45 quand 6 cœurs sont utilisés alors qu'il est limité à 4,95 avec CS-CvMoG2, 4,67 pour CS-Kovacev et 5.23 pour CS-Szwoch. La scalabilité obtenue est quasi linéaire, aussi le surcoût engendré par OpenMP est négligeable, surtout pour les images de petites résolutions. En comparaison avec les autres méthodes, CS-Szwoch représente les meilleures performances quand il s'agit des petites images, cependant, pour les grandes images, les accélérations obtenues ne dépassent pas 2.9. Cela est dû au fait que dans le code qui nous a été fourni par ses auteurs, nous avons remarqué l'existence de variables de comptages partagées entre les différents threads d'une façon atomique ce qui freine le calcul parallèle.

Afin de tester l'effet de l'environnement de compilation sur la scalabilité, nous recompilons notre code, de la même façon que la vectorisation en utilisant ICC, et nous mesurons le nombre d'images traitées dans les deux cas. Les valeurs obtenues sont représentées dans la Fig. 3.10. Nous constatons que la version compilée via ICC est plus rapide que celle avec GCC, grâce aux optimisations implicites dont nous avons parlé dans la partie de la vectorisation. Toutefois, au niveau de la scalabilité, la linéarité obtenue par GCC est meilleure alors que nous remarquons des atténuations importantes pour ICC quand nous augmentons le nombre de cœurs utilisés. Ces atténuations sont considérables également quand on augmente la taille des images, ce qui rend l'écart entre les deux compilateurs plus important pour les petites images.

### 3.5.5 Synthèse des résultats obtenus sur CPU

Nous présentons dans la Fig. 3.11 l'espace exactitude/temps de traitement pour cinq méthodes étudiées, ainsi que leurs dérivés que nous avons améliorés avec le CS, sur les

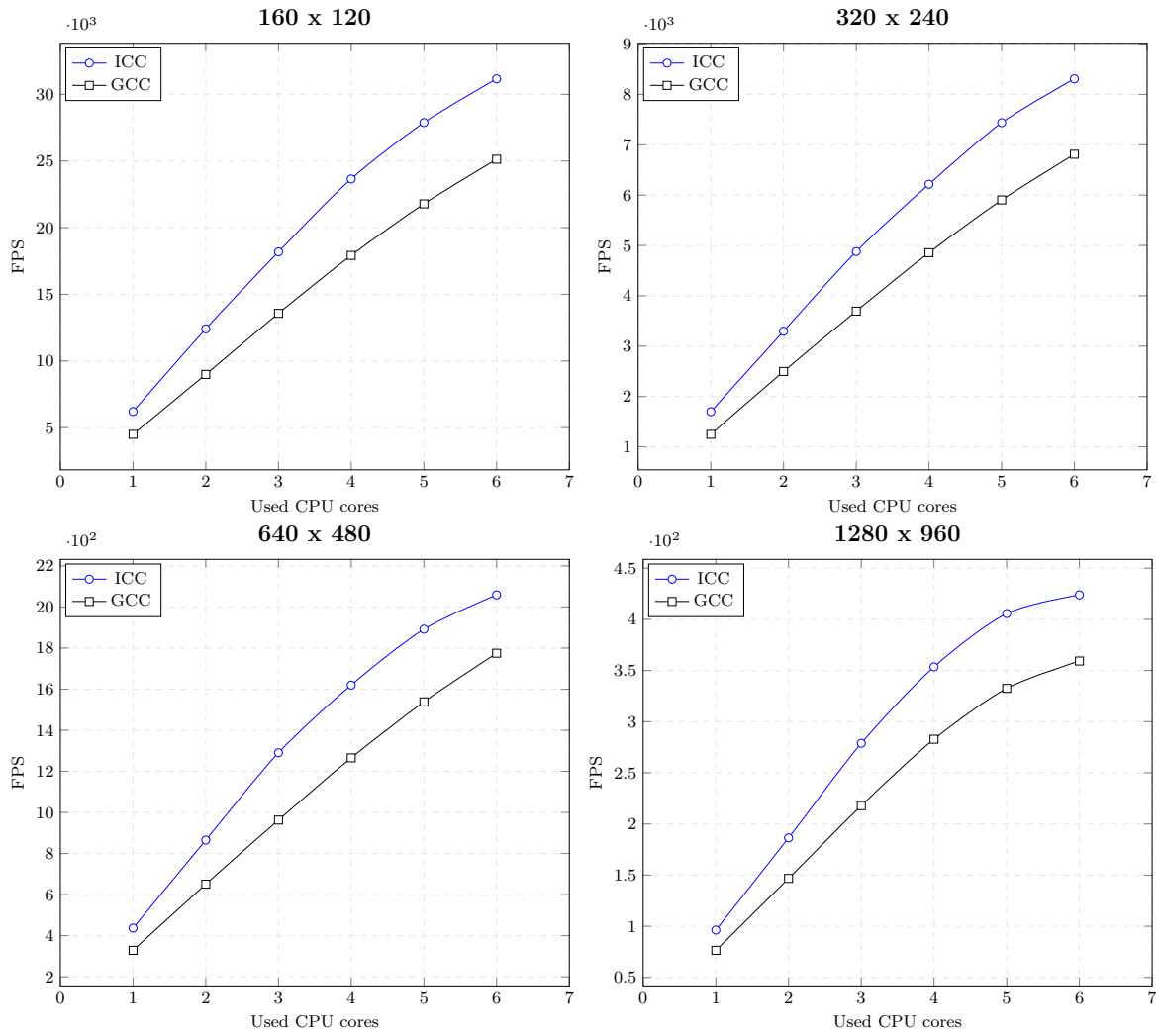


FIGURE 3.10 – Comparaison de la scalabilité des performances de notre CS-MoG amélioré sur un processeur multi-cœur CPU en utilisant les compilateurs GCC et ICC. La tendance est la même pour tous les datasets de la Table 3.3.

quatre plateformes. Pour simplifier la comparaison, nous ne présentons dans ces résultats que la valeur moyenne du temps de traitement obtenu sur les cinq datasets, et de la même manière, nous calculons la moyenne du F1-score obtenu. Le temps de traitement représenté dans cette figure est obtenu en utilisant tous les cœurs que possède chaque plateforme. La version non compressée de CvMoG1 n'apparait pas sur la majorité des graphiques puisqu'elle ne supporte pas le multithreading, ainsi le temps consommé par cette méthode est très important relativement aux autres.

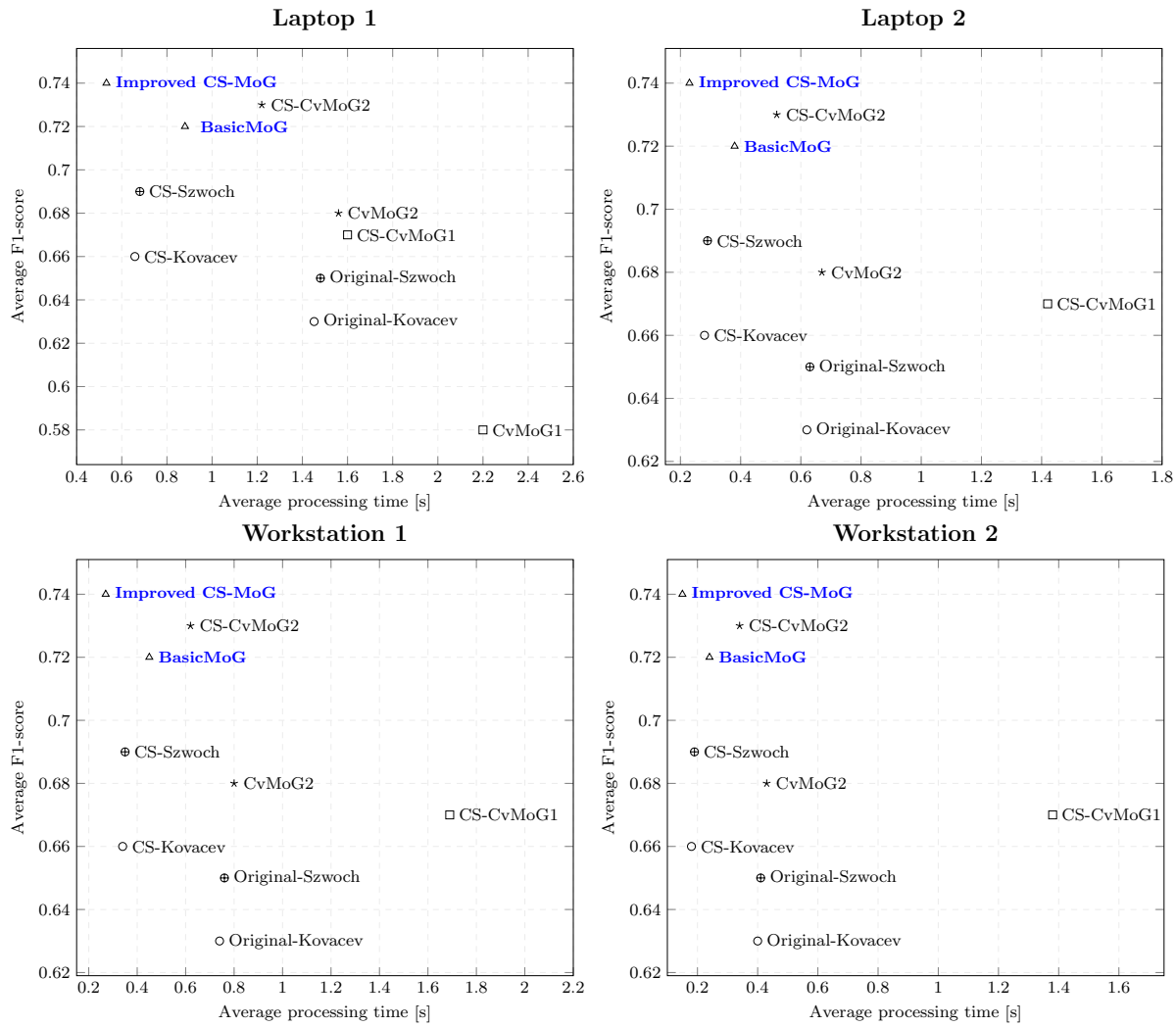


FIGURE 3.11 – Amélioration des performances et de la précision de quatre versions de MoG en utilisant le CS sur CPU : le F1-score et le temps consommé sont moyennés sur les cinq datasets de 300 images présentées dans la Table. 3.3.

La première remarque concernant cette figure est qu'en passant d'une plateforme à une autre, la distribution des méthodes étudiées change légèrement. Ce changement ne concerne pas le F1-score mais plutôt le temps de traitement vu que la puissance et le nombre de cœurs que contient chaque plateforme est différent. Sur les plateformes ayant un nombre important de cœurs de calcul (>=4) comme le cas du Laptop2, Workstation 1 et Workstation 2, l'écart, entre le temps consommé par les méthodes moins scalables et les autres, devient plus important. En comparant les cinq méthodes de base, notre BasicMoG

donne le meilleur compromis précision-performance, suivi par CvMoG2, Original-Szwoch, Original-Kovacev alors que CvMoG1 consomme plus de temps. L'ordre des versions compressées reste le même avec des écarts différents.

## 3.6 Conclusion

Les conclusions qui peuvent être tirées de chapitre sont les suivantes :

— le temps de traitement est réduit sur CPU pour toutes les versions étudiées après l'ajout du CS.

— la précision a été considérablement améliorée grâce à ce dernier.

— notre implémentation de CS-MoG donne le meilleur rapport précision/temps de traitement relativement aux autres méthodes.

Par conséquent, le gain sur trois niveaux (performances, précision et scalabilité multi-cœur) fait de notre CS-MoG amélioré une meilleure alternative pour les applications utilisant la soustraction d'arrière-plan. Il pourrait être intéressant de travailler à l'ajout de notre implémentation à une future version d'OpenCV qui, aujourd'hui, n'utilise pas le CS.

Chapitre 4

# Implémentation et optimisation de CS-MoG sur GPU

## Sommaire

## 4.1 Introduction

Après avoir étudié et analysé chacun des trois kernels (projection, MoG et Raffinement de l'avant-plan) constituant l'algorithme CS-MoG sur CPU, nous nous intéresserons dans ce chapitre à la présentation de nos contributions concernant l'implémentation de CS-MoG sur les processeurs graphiques GPU. De la même façon que le chapitre précédent, ces contributions sont comparées dans la section des résultats, avec les versions les plus communes de la littérature. Néanmoins, nous avons constaté que pour CS-MoG, il existe une influence entre l'optimisation des accès mémoire (étape 1 dans le guide de portage présenté dans la section 2.3.2) et l'optimisation de la configuration d'exécution (étape 3). Cela est dû au fait que, dans notre cas, le dimensionnement de la grille de threads ne doit pas uniquement garantir un taux d'occupation élevé mais aussi garantir des accès contigus à la mémoire. Pour cette raison, nous traitons ces deux aspects dans une même section que nous appelons "configuration d'exécution et optimisation des accès mémoire".

## 4.2 Configuration d'exécution et optimisation des accès mémoire

La question à laquelle il faut répondre dans cette partie est la suivante : pour un kernel donné, comment allons-nous regrouper les threads afin d'assurer à la fois des accès contigus à la mémoire*, un équilibrage de la charge de traitement entre les threads** et une occupation maximale au niveau des Warps*** ? la réponse à cette question diffère d'un kernel à un autre vue la nature des calculs réalisés. Contrairement au CPU où les approches d'optimisation sont appliquées de la même manière pour tous les kernels, l'optimisation sur GPU dépend de la structure de chacun. Pour cette raison, nous traitons les trois kernels séparément dans la partie suivante.

### 4.2.1 Projection

Le calcul des projections, cette tâche se présente comme un produit matriciel entre chaque zone de l'image (8x8 pixels) et la matrice de projection. Pour une image donnée, chaque zone de 64 pixels doit être multipliée par les 8 lignes de la matrice de projection. Ainsi, chaque pixel sera multiplié 8 fois avec les éléments qui lui correspondent dans la matrice de projection.

Cette tâche génère un vecteur de taille 8 pour chaque zone de taille 8x8 de l'image. La Fig. 4.1. présente différentes associations pixels/threads qui sont discutées dans la suite. Si nous assignons à chaque thread la tâche de projeter une zone entière (Fig. 4.1.A), les threads successifs d'un Warp traiteront des zones voisines et donc accéderont à des éléments qui sont séparés de 64 cellules dans la mémoire, générant ainsi une latence importante. Si nous affectons 64 threads à chaque zone (correspondant à sa taille, Fig. 4.1. B), chaque thread effectuera une seule multiplication, ce qui est très faible comme quantité de travail. De plus, nous devrons utiliser dans ce cas un autre thread pour additionner les 64 valeurs, ce qui n'est pas efficace. La Fig. 4.1.C. présente la solution que nous avons retenue : nous allouons 8 threads pour projeter chaque zone de l'image, et ainsi chaque thread calcule une projection. De cette manière, chacun aura assez de charge de calcul, et les huit threads voisins accèdent à des adresses de mémoire contiguës (dans la matrice de projection) ou identiques (dans les vecteurs de données). Cette solution est la plus optimale bien que chaque groupe de 8 threads dans un même Warp accèdent à des données distantes de 64 cases mémoire.

En adoptant cette configuration d'exécution des threads, le traitement d'une image de taille moyenne, (640x480=307200) par exemple, implique l'appel à 307200/8=38400 threads parallèles. Etant donné qu'un Warp contient 32 threads, nous exécutons ainsi 38400/32=1200 Warps en parallèle ce qui garantit un taux d'occupation suffisant pour un GPU récent. Concernant l'équilibrage des charges entres les threads, le calcul des projections est indépendant du contenu de l'image : il est effectué de la même manière pour

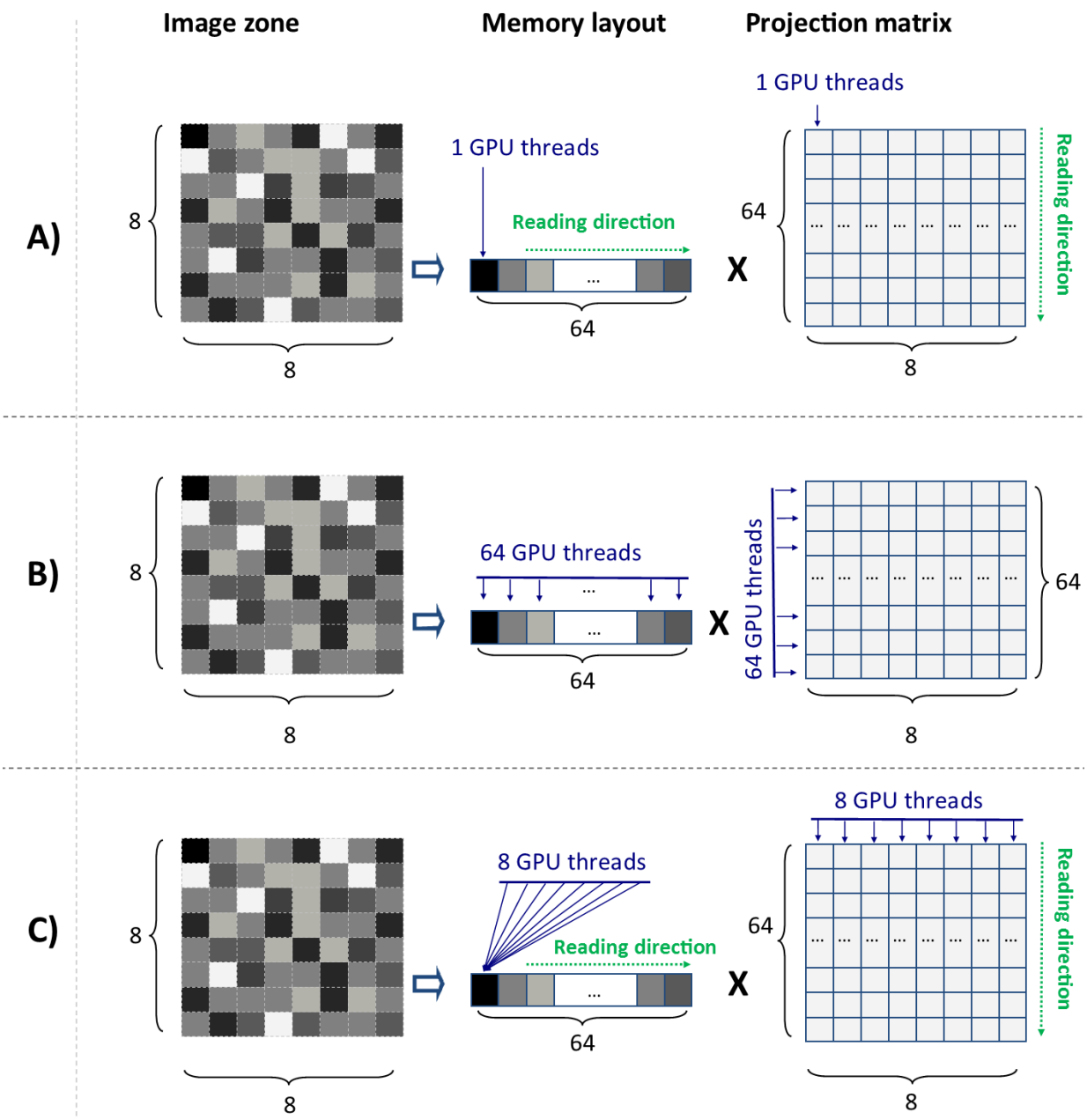


FIGURE 4.1 – Présentation des scénarios possibles pour l'exécution de la projection d'une zone 8x8 sur GPU.

toutes les zones. Par conséquent, le temps de calcul est constant et dépend uniquement de la taille des images traitées. Cela signifie un équilibrage de charge implicite entre les threads puisqu'ils reçoivent le même nombre de pixels à traiter.

Etant donné que la matrice de projection est utilisée d'une façon répétitive par tous les threads d'un même bloc, une autre amélioration que nous proposons consiste à charger la matrice de projection dans la mémoire cache partagée (Shared memory) au sein de ce bloc. Ceci permet d'éviter de faire des lectures répétitives à partir de la mémoire globale. La taille des blocs de threads utilisée, pour bien bénéficier de cet aspect, est égale à 256.

### 4.2.2 MoG

Pour MoG, les équations de Eq. 1.2. à l'Eq. 1.10. doivent être appliquées sur chaque projection. Ainsi, la complexité de ce kernel est suffisante pour bien exploiter un GPU. Notre première amélioration consiste à réduire la latence d'exécution liée à la structure des blocs de threads et augmenter l'occupation des ressources. Pour cela, nous utilisons des blocs de 256 threads au lieu de 32, 64 ou 128. Malgré ce choix, les performances ne sont pas forcément garanties vu que les threads peuvent traiter des contenus différents ce qui peut engendrer une divergence entre eux. Cette divergence est expliquée par le fait que les équations de MoG sont exécutées d'une façon conditionnée par le contenu des pixels. Pour réduire l'influence de ce contenu, nous avons testé différentes formes de blocs : 8x32, 32x8, 64x4, 4x64, 256x1 et 1x256. Ce test a pour objectif de choisir la forme via laquelle les threads d'un bloc traitent des données au maximum homogènes. Nos expériences montrent que le premier ainsi que le denier choix (8x32 et 1x256) offrent les meilleures performances pour tous les datasets utilisés. Pour le premier, cela est expliqué par le fait que des blocs de 8x32 garantissent une homogénéité vu leur forme rectangulaire qui permet aux threads de traiter des éléments de données proches dans une portion de l'image. Ces éléments représentent, dans la majorité des cas, le même objet ou la même section de l'arrière-plan. Néanmoins, les accès mémoire effectués en utilisant cette configuration ne sont pas optimaux vu que les lectures/écritures effectuées par les threads ne sont pas contigües. Par contre, le choix de la taille 1x256 permet de satisfaire cette dernière condition même si les données traitées sont moins homogènes.

Etant donné que chaque projection est représentée dans le modèle de l'arrière-plan par

un ensemble de FDPGs $(\mu, \sigma, \omega)$, la représentation du modèle par une structure de tableaux (Structure of Arrays), suggérée même dans l'état de l'art GPU, au lieu d'un tableau de structures (Array of structures) ne garantit pas nécessairement des accès contigus à la mémoire au niveau des threads voisins. Il est nécessaire de décider ce que va traiter chaque thread. En effet, nous supposons que chaque bloc traite $N$ projections $P_i$ ($N = 256$ dans notre cas) représentées avec $k = 4$ FDPGs. En assignant le traitement de chaque FDPG à un thread (Fig. 4.2.a), ces derniers vont accéder à des cases mémoire espacées de $k = 4$ éléments, ce qui n'est pas optimal en termes de performance. Par contre, le regroupement de toutes les FDPGs représentant un même niveau $k = i, i \in [1, 4]$ comme illustré dans la Fig. 4.2.b), permet à chacun des 8 Warps d'un même bloc de lire 32 éléments de manière contigüe, ce qui est équivalent à la lecture de 256 éléments par bloc.

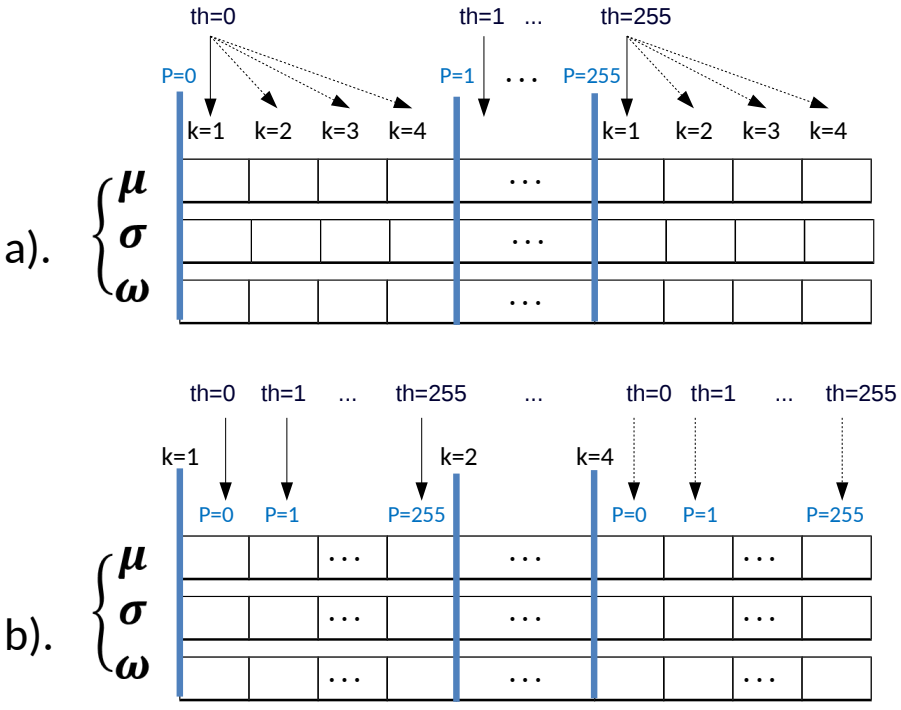


FIGURE 4.2 – Optimisation des accès à la mémoire dans le modèle d'arrière-plan MoG sur GPU.

Après avoir classé les projections comme éléments d'arrière-plan ou du premier plan,

un vote est effectué sur chacune des 8 projections qui représentent une zone d'image afin de prendre une décision globale. La charge de cette tâche est négligeable sur le processeur graphique, et n'influence pas le temps global de MoG même lors de l'affectation du traitement de chaque zone à un seul thread ($\leq 0.6\%$).

### 4.2.3 Raffinement

Après avoir masqué les zones de l'image représentant l'arrière-plan, un traitement au niveau des pixels est effectué sur les zones restantes conformément à l'approche décrite dans le chapitre I. La manière la plus optimale sur GPU est de traiter chaque pixel d'une zone d'image avec un seul thread, donc chaque zone est traitée par 64 threads (Fig. 4.3). Cette taille représente le maximum qu'on peut atteindre au niveau de l'occupation des

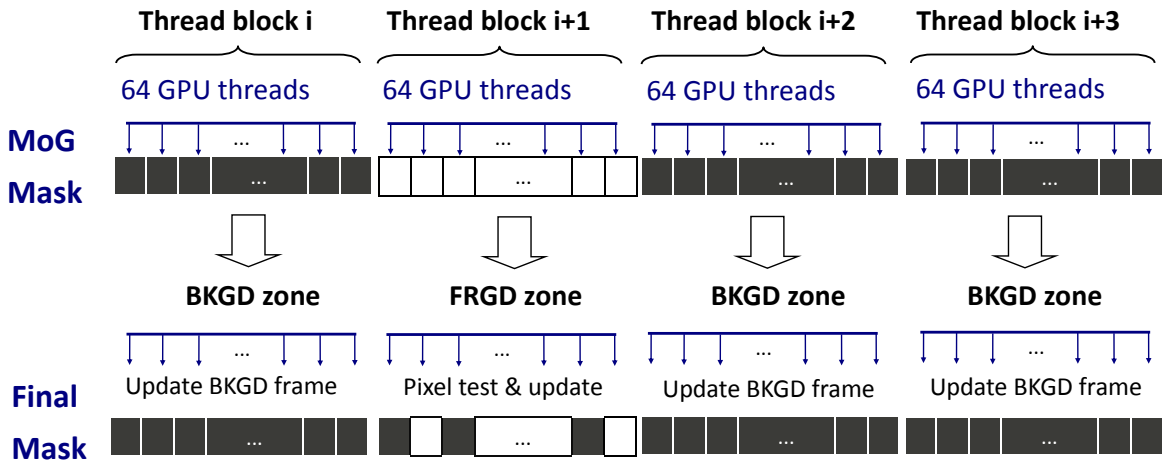


FIGURE 4.3 – Méthode d'exécution du processus de raffinement sur GPU : chaque zone de l'image est affectée à un bloc de threads.

ressources : contrairement au processus MoG, nous ne pouvons pas choisir des blocs de taille 256 qui permettent de traiter 4 zones, car le raffinement de ces dernières peut suivre des chemins d'exécution très divergents. En effet, une zone peut être segmentée comme arrière-plan alors que ses voisines le seront comme premier plan, or chacun des types de zones nécessite des traitements différents. Pour éviter cela, chaque bloc traitera une seule zone de l'image. Dans le cas où cette zone a été classée comme partie de l'arrière-plan, les

threads auront comme travail de mettre à jour l'image de l'arrière-plan en utilisant l'image en cours comme décrit dans l'Eq. 1.13. Ainsi, cette configuration nous a permis à la fois d'éviter les divergences entre les threads voisins, et aussi de garantir l'accès par ces derniers à des zones mémoire contiguës.

## 4.3 Optimisation de la communication

La majorité des cartes graphiques disposent de deux moteurs indépendants pour réaliser les calculs et les transferts de données. Ces transferts concernent les images contenues initialement dans la mémoire du CPU qu'elles faut copier vers la mémoire globale du GPU, et les images traitées à copier dans le sens inverse. Cette fonctionnalité pourra être exploitée afin de lancer les deux tâches en parallèle et ainsi masquer le temps de transfert. Néanmoins, selon [135], ce masquage est plus utile quand ces deux temps sont de même ordre de grandeur, ce qui est le cas de CS-MoG. Cette opération est effectuée via une fonctionnalité de CUDA appelée les flux (Streams [162]). Contrairement à l'exécution en série, il s'agit d'une séquence d'opérations qui s'exécutent en parallèle dans l'ordre des tâches sur le GPU. Pour CS-MoG, nous faisons appel à deux streams en parallèle (Fig. 4.4). Le premier pour réaliser le traitement de l'image $N$ et le deuxième pour faire à la fois la copie de l'image $N+1$ de la mémoire CPU vers GPU pour qu'elle soit traitée dans le prochain cycle, et aussi pour la copie inverse du masque obtenu après le traitement de l'image $N-1$ du cycle précédent. Même si ces deux dernières tâches sont exécutées en série par le même moteur, leurs temps est complétement masqué par celui du traitement qui est plus important.

## 4.4 Optimisation des instructions et du flux de contrôle

Après l'optimisation de l'accès à la mémoire, du taux d'occupation des ressources et du temps de transfert des données, il est nécessaire de réduire le coût des instructions sans influencer la précision de l'algorithme. Pour cela, nous avons examiné le code pour détecter les opérations qui peuvent être éliminées ou exécutées autrement. Par exemple, nous classons les FDPGs dans le model selon le poids $\omega$ et non pas selon le rapport $\omega/\sigma$. Le

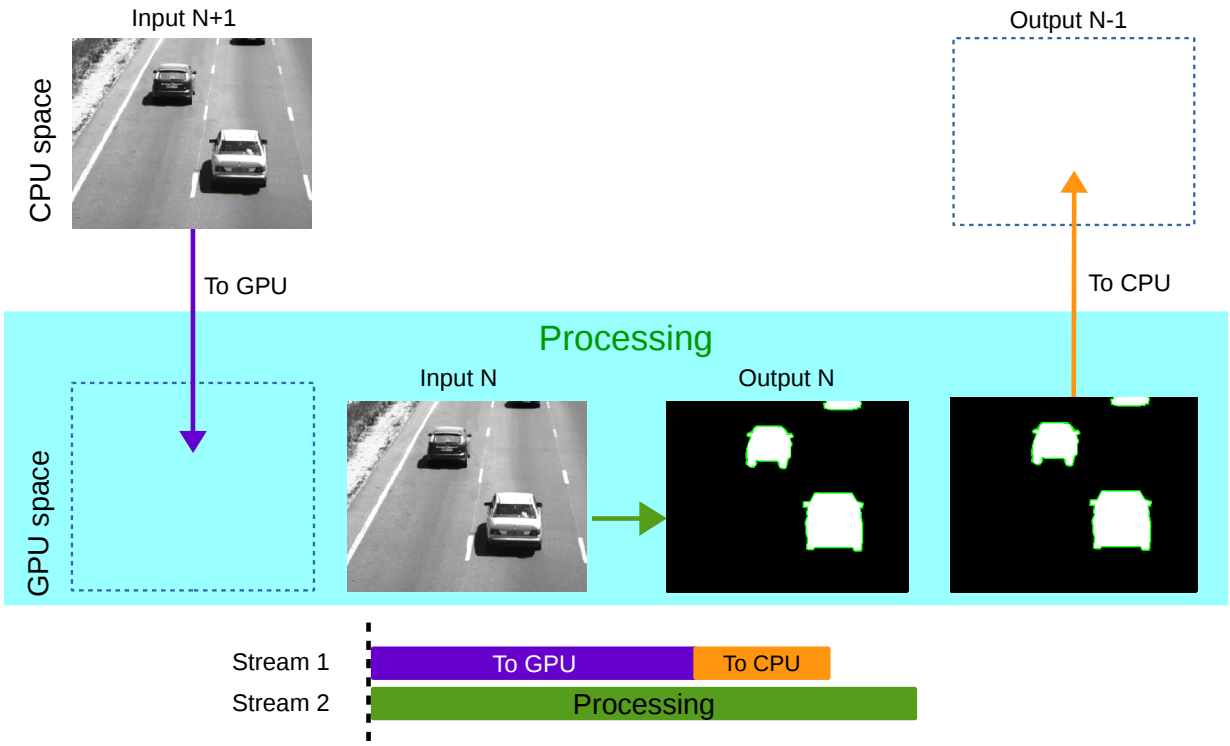


FIGURE 4.4 – Masquage du temps de transfert des images par le temps de traitement (CS-MoG) en utilisant le principe des streams CUDA.

fait d'éliminer l'effet de la variance sur le classement n'a aucune influence sur les résultats, mais permet d'éviter une division qui est coûteuse car elle est exécutée $k = 4$ fois pour chaque projection. Par ailleurs, nous avons remplacé toutes les fonctions mathématiques, comme la racine carrée, par leurs équivalents dans la bibliothèque Fast Math de CUDA. Cette bibliothèque permet à l'utilisateur d'exploiter des routines prédéfinies et hautement optimisées sur GPU, contrairement aux calculs standards effectués au niveau des ALUs. Ces routines permettent de réaliser des calculs mathématiques très rapidement avec des petites pertes au niveau de la précision. Pour certains algorithmes il est nécessaire d'assurer le maximum de précision, mais pour CS-MoG, tous les paramètres utilisés dans la littérature ne dépassent pas une précision de $10^{-4}$ qui est suffisante pour obtenir un F1-score satisfaisant.

Un dernier point aussi important est de ne pas utiliser des variables en double précision

qui peuvent réduire 30 fois la vitesse d'une opération sur certains GPU par rapport à l'utilisation de la simple précision. Cela doit être effectué généralement au niveau de la déclaration des constantes qui sont par défaut en double précision. L'extrait de code suivant illustre la déclaration de deux constantes VAR1 et VAR2 successivement en simple et double précision.

```
#define VAR1 3.1465f
#define VAR2 2.7182
```

Comme illustré précédemment dans la section 2.2.4, les GPU sont composées d'un ensemble de Warps à instruction unique et threads multiples (Single Instruction, Multiple Threads : SIMT). Ces Warps sont capables d'exécuter efficacement des tâches parallèles sur 32 entrées via des threads. Toutefois, si ces threads prennent des chemins de contrôle divergents, tous les chemins divergents sont exécutés en série (Exemple dans la Fig. 4.5). Dans le pire des cas, chaque thread utilise un chemin de contrôle différent et l'architecture hautement parallèle est utilisée en série par chaque thread.

Ce problème de divergence de flux de contrôle est bien connu dans le développement sur GPU ; la transformation de code et la réorganisation de la structure des données sont couramment utilisées pour réduire l'impact de la divergence. Ces techniques tentent d'éliminer ces divergences en regroupant des threads ayant des comportements identiques dans le même Warp (Fig. 4.6).

CS-MoG contient, dans les deux dernier kernels (MoG et raffinement), un ensemble de portions de code où la divergence des threads peut être importante. Pour remédier à ce problème, nous avons, d'une part, construit nos Warps par les threads traitant les données qui subissent les mêmes variations. Les tailles des blocs (rectangulaires) adoptées dans la section 4.2.2, nous permettent d'orienter les threads d'un même Warp pour traiter des projections ou pixels voisins. D'autre part, nous avons appliqué la technique de déroulage de boucles qui a pour objectif d'augmenter la vitesse de notre programme en réduisant ou en éliminant les instructions de contrôle telles que les tests de fin de boucle à chaque

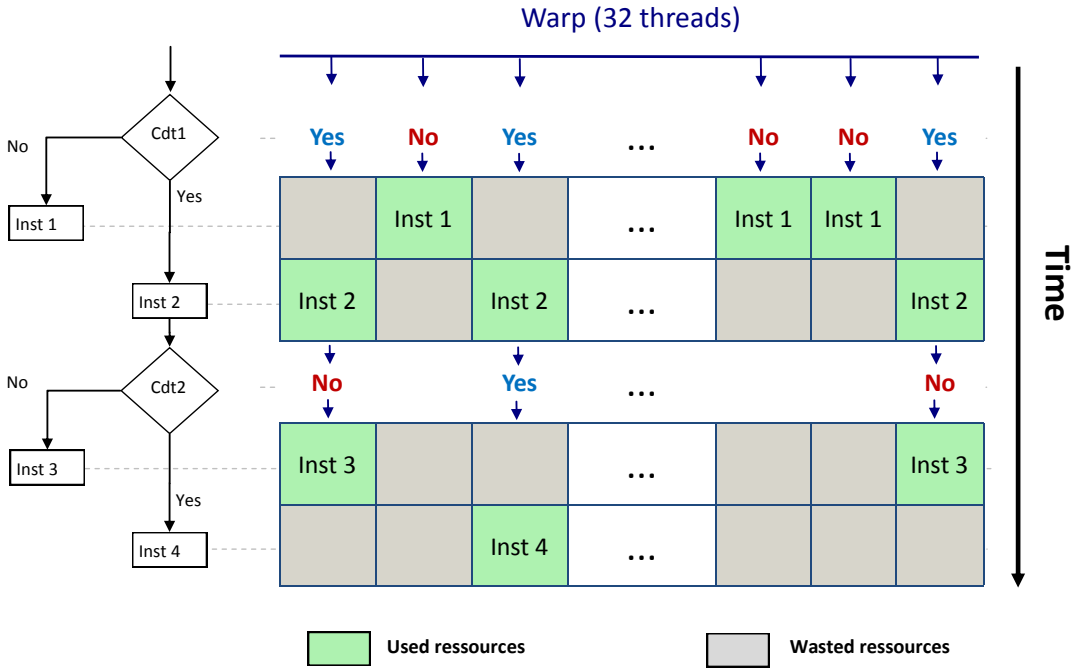


FIGURE 4.5 – Étapes et temps d'exécution d'un code simple lorsque les threads d'un Warp sont divergents.

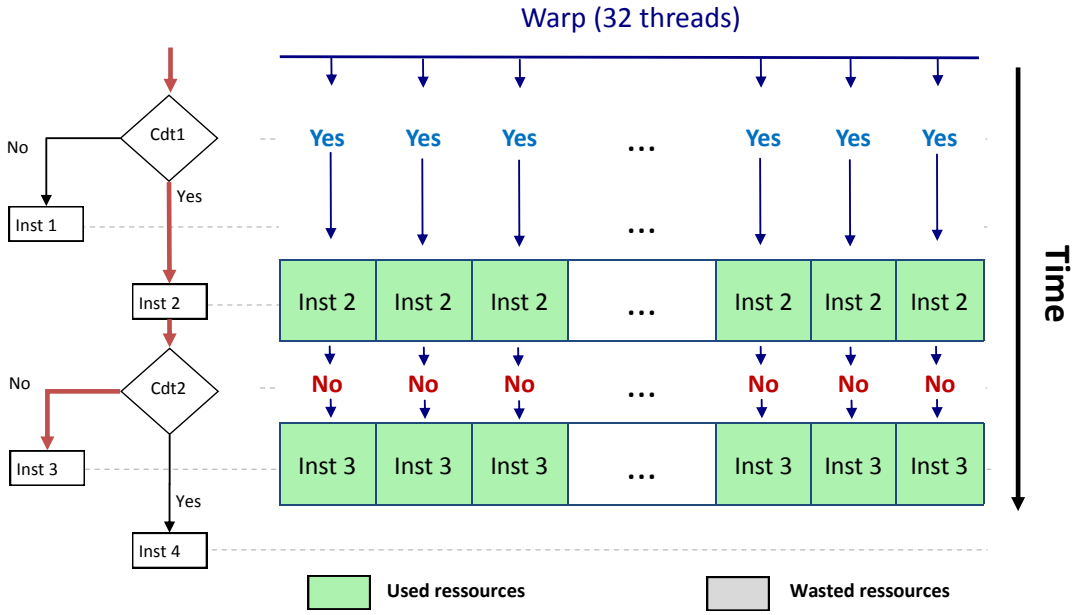


FIGURE 4.6 – Étapes et temps d'exécution d'un code simple lorsque les threads d'un Warp ont des chemins de contrôle identiques.

itération. Cela a permis également de réduire le délai de lecture des indices des boucles dans la mémoire. Les boucles sont ainsi réécrites sous la forme d'une séquence répétée d'instructions similaires. La directive #**pragma unroll** nous a permis de réaliser cette tâche facilement et directement par le compilateur. L'extrait du code de la Fig. 4.7 a comme objectif de calculer la distance de Mahalanobis entre les éléments d'une image (pixels ou projections) et les FDPGs correspondantes dans le modèle de l'arrière-plan MoG. Le code en haut représente les boucles originales, et en bas celui qui sera exécuté après le déroulage de la boucle interne.

```
for (pixel=0; pixel<IM_SIZE; pixel++) {

    #pragma unroll
    for (k=0; k<3; k++) {

            distance = pix_value- Model.mean[pixel][k];

            normD = d*d;

            match = normD<(VARIANCE_THRESHOLD * Model.var[pixel][k]);
    }
}
```

```
for (pixel=0; pixel<IM_SIZE; pixel++) {

            distance = pix_value- Model.mean[pixel][0];
            normD = d*d;
            match = normD<(VARIANCE_THRESHOLD * Model.var[pixel][0]);

            distance = pix_value- Model.mean[pixel][1];
            normD = d*d;
            match = normD<(VARIANCE_THRESHOLD * Model.var[pixel][1]);

            distance = pix_value- Model.mean[pixel][2];
            normD = d*d;
            match = normD<(VARIANCE_THRESHOLD * Model.var[pixel][2]);

}
```

FIGURE 4.7 – Calcul de la distance de Mahalanobis entre les pixels d'une image et les FDPGs correspondantes en utilisant le déroulage de boucles.

Le fait d'effectuer des optimisations sur plusieurs niveaux dans ce chapitre nous a permis d'avoir la version la plus optimale de chaque kernel de CS-MoG. Ainsi, nos résultats dépassent, en termes de performances et de précision, les deux implémentations OpenCV de MoG ainsi que [131] et [141].

## 4.5 Résultats et synthèse

De la même façon que le chapitre précédent, cette section qui présente les résultats expérimentaux est divisée en cinq parties : dans la première, nous présentons les petits changements au niveau de l'environnement de mesure tels que les compilateurs utilisés et les méthodes avec lesquelles nous nous comparons au niveau du GPU. La deuxième est consacrée à la présentation des résultats obtenus concernant la précision sur GPU. La troisième concerne l'illustration des résultats de notre première contribution sur ce processeur qui est l'optimisation des transferts de données. Dans la quatrième partie, nous présentons les gains de performance obtenus grâce à l'optimisation de l'exécution de CS-MoG. Finalement, la dernière partie est consacrée à la présentation des résultats globaux.

### 4.5.1 Environnement de mesure

Nos tests sont effectués en gardant les mêmes plateformes que le chapitre précédent, sauf que nous nous concentrons sur les GPU. Le seul changement effectué au niveau de l'environnement est que cette fois nous utilisons le compilateur NVCC (V8.0.61) de NVIDIA qui nous permettra de compiler des codes faisant appel à CUDA. Les implémentations avec lesquelles notre version améliorée de CS-MoG (Improved CS-MoG) est comparée dans ce chapitre sont les mêmes que le chapitre précédent sauf celle de Szwoch et al. [51] pour laquelle nous ne possédons pas d'implémentation sur GPU. Nous la remplaçons dans nos tests par l'implémentation GPU proposée par Pham et al. [131] dont nous avons pu récupérer le code, et que nous appelons Original-Pham. Nous avons également amélioré cette implémentation grâce au Compressive Sensing (CS-Pham).

Quant aux datasets, nous gardons exactement les mêmes que le chapitre précédent : Library, FluidHighway, Office, Pedestrians et PETS2006.

### 4.5.2 Evaluation de la précision

Afin de s'assurer que la précision des méthodes étudiées n'est pas influencée par le passage sur GPU, nous présentons dans la Fig. 4.8 les résultats du F1-score obtenu. En

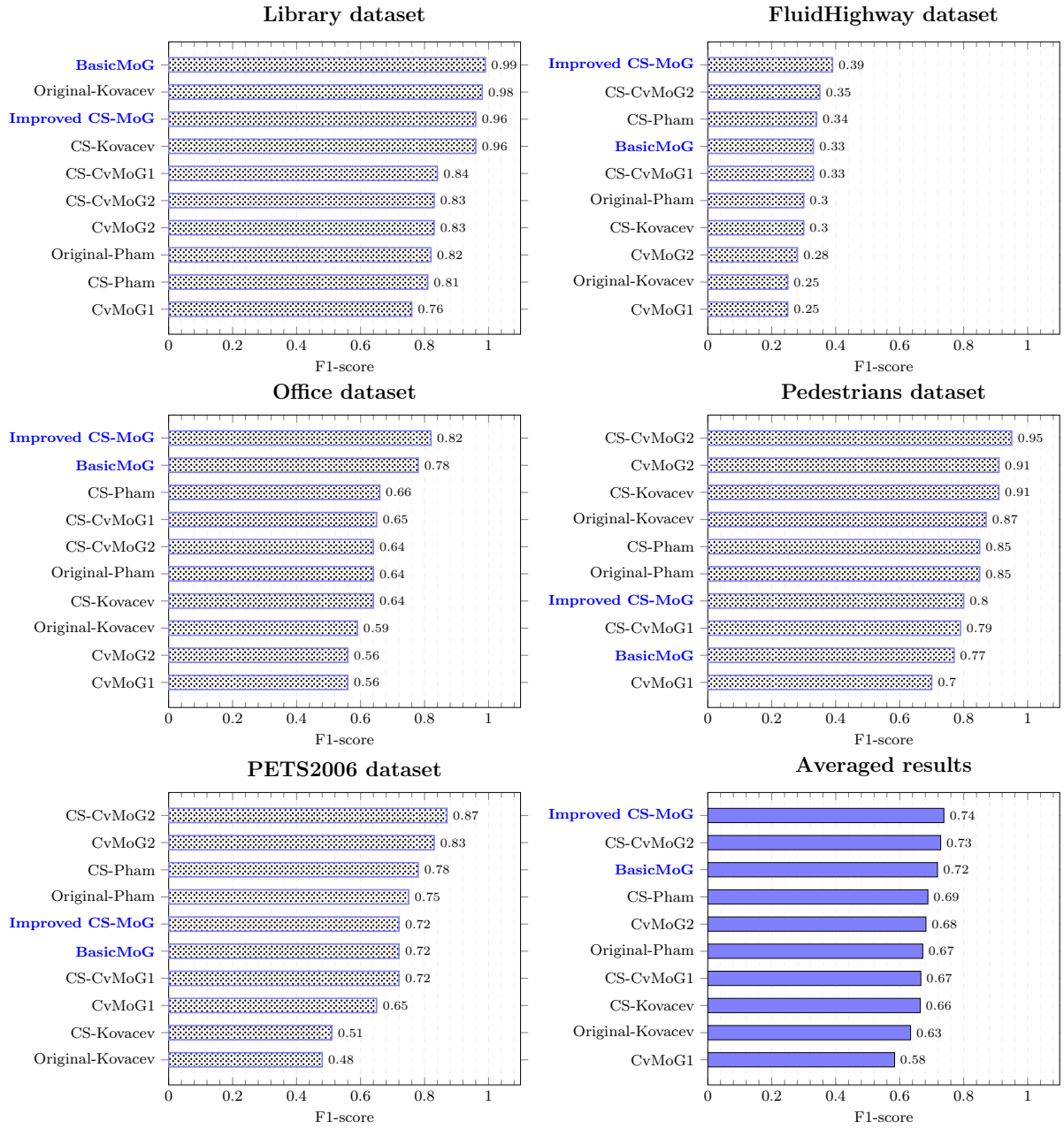


FIGURE 4.8 – Représentation des résultats obtenus sur la précision des méthodes étudiées pour chaque dataset sur GPU.

analysant ces graphiques nous constatons que la précision de ces méthodes n'a pas été influencée par le portage sur GPU. BasicMoG représente le meilleur F1-score moyen par rapport aux cinq versions de base, et notre Improved CS-MoG donne le meilleur résultat par rapport aux cinq versions compressées. Original-Pham, évalué pour la première fois, est classé globalement en 6ième position parmi 10, tandis que sa version que nous avons améliorée via le CS est classée en 4ième position. Cela nous nous permet de conclure que cette implémentation génère des résultats en moyenne satisfaisants.

Après cette étude de précision des différentes implémentations de MoG sur GPU, nous consacrons les deux prochaines sections aux résultats en termes de performance, via lesquels nous montrons que notre approche dépasse les autres méthodes étudiées.

### 4.5.3 Optimisation des transferts

Nous commençons la partie de l'évaluation des performances par les résultats des optimisations concernant le transfert des données présenté dans la section 4.3 de ce chapitre. Pour cela, il s'agit de s'assurer que le temps de transfert est masqué par le temps de traitement quand nous lançons les deux processus en parallèle. Nous comparons dans la Fig. 4.9 le nombre d'images traitées en utilisant notre implémentation améliorée (Improved CS-MoG) sur toutes les plateformes et tous les datasets, avec et sans recouvrement calcul/communication. L'objectif est de détecter les éléments qui influencent ou qui peuvent limiter les gains attendus de ce masquage du temps de transfert.

On remarque clairement que nous avons pu obtenir des gains considérables sur toutes les plateformes et datasets, mais avec des pourcentages différents. L'analyse de ces différents écarts passe par deux étapes :

— La première remarque concerne l'influence du matériel. Les accélérations moyennes obtenues sur les quatre plateformes utilisées sont successivement 1.52 sur Laptop1, 1.75 sur Laptop2, 1.70 sur Workstation1 et 1.73 sur Workstation2. L'accélération représente le pourcentage qu'occupait le temps de copie, dans la version séquentielle, par rapport au temps global. Il peut être exprimé théoriquement par le rapport

$copie/(copie + calcul)$.

— La deuxième remarque concerne l'effet de la dynamicité des datasets. Pour cela, nous avons calculé la valeur moyenne du gain obtenu pour chaque dataset sur les cinq plateformes, et nous avons obtenu les résultats présentés sur la Fig. 4.10. Nous remarquons une relation inverse entre la dynamicité et le gain obtenu. Cette relation s'explique par la même raison que celle donnée dans le paragraphe précédent : quand la dynamicité est élevée, le temps de calcul nécessaire pour comparer les projections avec leurs FDPGs est important ainsi, le gain relatif au temps de communication sur le temps de calcul est faible.

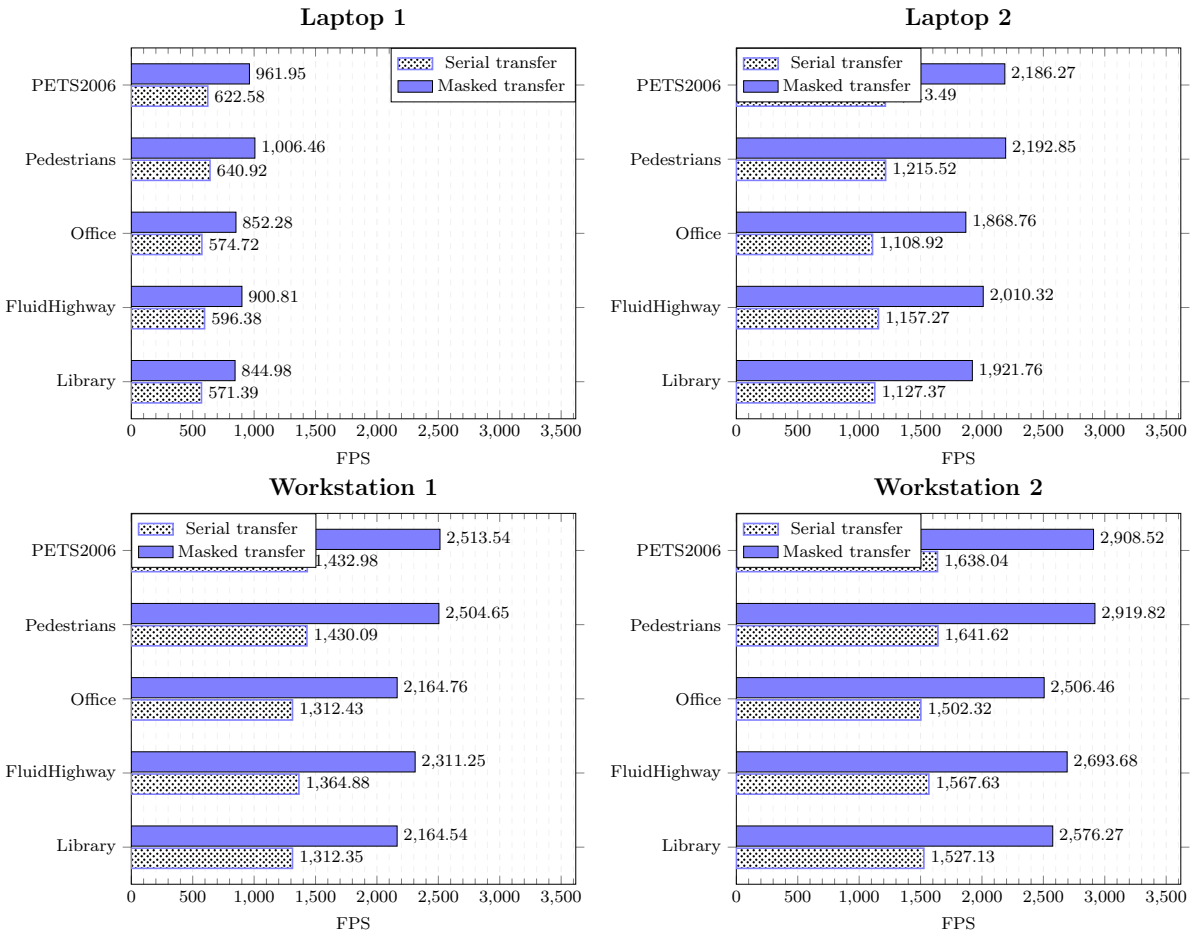


FIGURE 4.9 – Gains obtenus grâce au masquage du temps de transfert de CS-MoG sur GPU.

A partir de ces résultats, et afin d'obtenir les meilleures performances possibles, nous

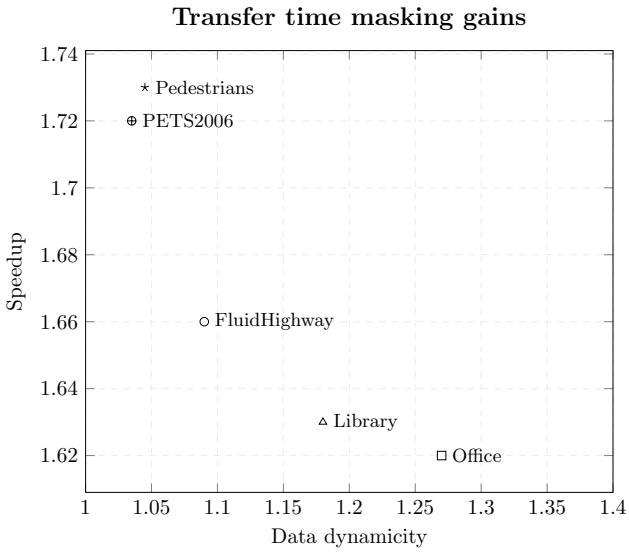


FIGURE 4.10 – Moyennes des gains obtenus grâce au masquage du temps de transfert de CS-MoG sur GPU en fonction de la dynamicité des données.

pourrions généraliser ces facteurs et considérer la dynamicité ainsi que la puissance relative des moteurs de calcul, de communication, comme des métriques qui permettraient d'estimer les gains qui peuvent être obtenu par le masquage du temps de transfert. Ces métriques pourraient être estimées durant la phase d'entrainement, et mises à jour durant l'exécution.

### 4.5.4 Optimisation de l'exécution

Dans cette partie des résultats, nous présentons les gains obtenus suite aux améliorations effectuées dans les sections 4.2 et 4.4 concernant l'optimisation de l'exécution des threads et des instructions. Vu que ces améliorations ne sont pas entièrement indépendantes l'une de l'autre, nous présentons le résultat global obtenu dans une seule figure (Fig. 4.11). Chaque graphique représente les valeurs obtenues sur chaque plateforme.

Pour pouvoir résumer ces résultats nous avons calculé, le gain moyen obtenu sur chaque dataset pour toutes les plateformes. De cette façon, les valeurs obtenues sont successivement : 49% pour Library, 48% pour FluidHighway, 46% pour Office, 46% pour Pedestrians et 49% pour PETS2006. Les différences très faibles entre ces gains nous permettent de conclure que l'effet de la dynamicité a été amoindrie par notre implémentation en ré-

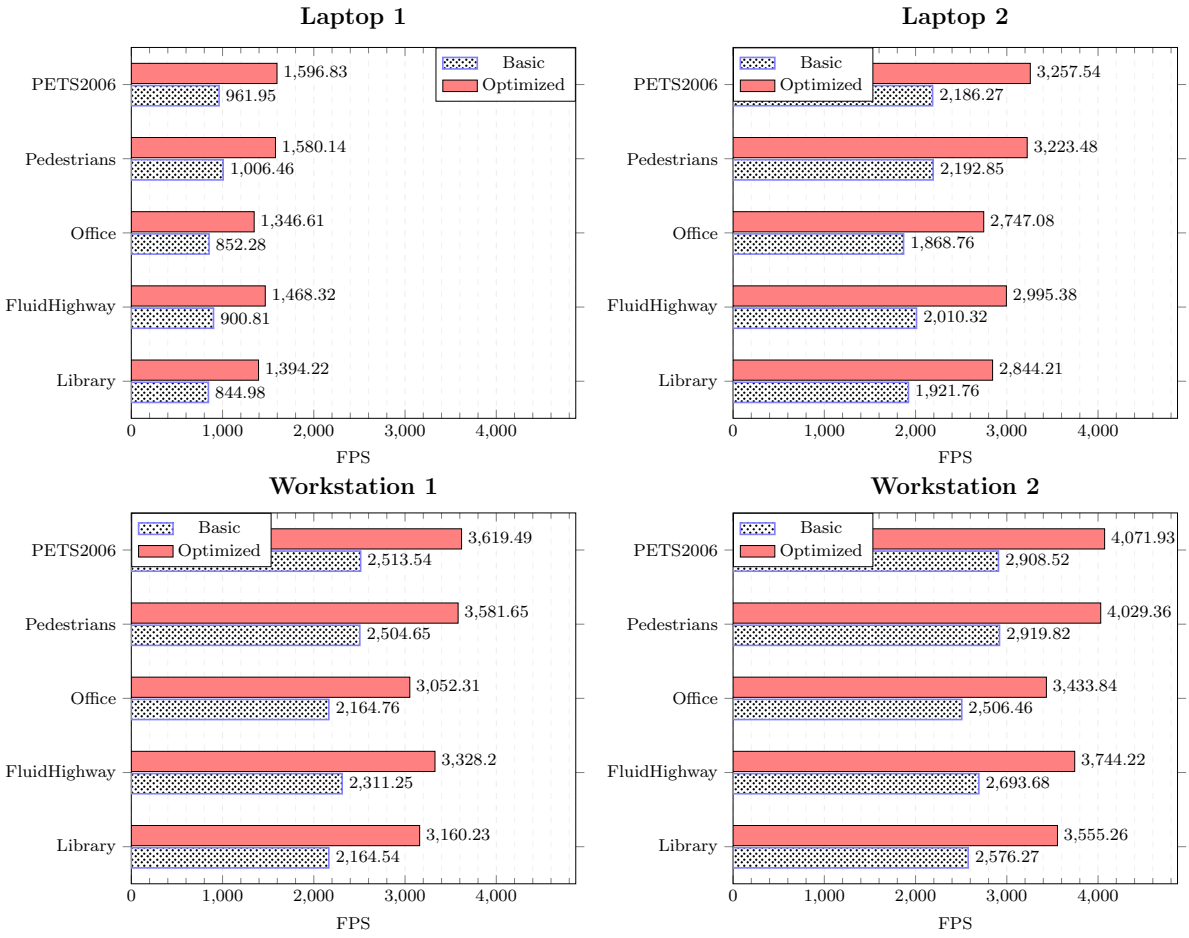


FIGURE 4.11 – Gains obtenus grâce à l'optimisation de l'exécution des threads et des instructions de CS-MoG sur GPU.

duisant au maximum les divergences entre les threads. Quant aux plateformes, les gains moyens obtenus sont 61% sur Laptop1, 48% sur Laptop2, 44% sur Workstation1 et 39% sur Workstation2. Nous remarquons que le gain obtenu est inversement proportionnel à la puissance des GPU. Cela est expliqué par le fait que sur ces plateformes, le temps de traitement avant les optimisations liées à l'exécution était faiblement supérieur à celui de la communication (qui est masqué). Par conséquent, après l'optimisation, ce dernier devient dominant et bloque les performances. Autrement dit, notre application a passé de la catégorie « bornée par le calcul (computing bounded)» au type « bornée par la communication (communication bounded)».

### 4.5.5 Synthèse des résultats obtenus sur GPU

Tous les résultats vus jusqu'à présent sont regroupés dans la la Fig. 4.12. L'objectif est

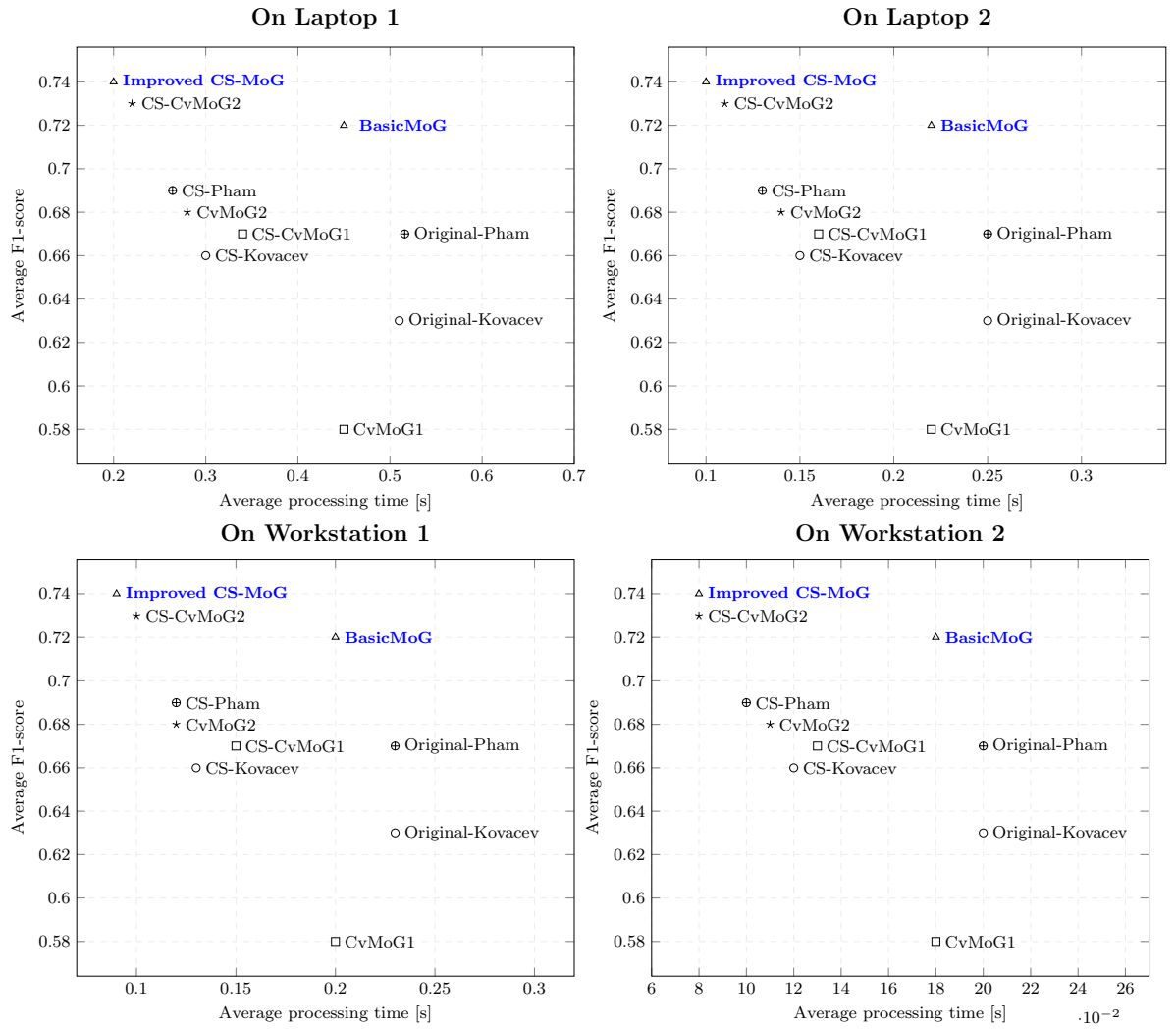


FIGURE 4.12 – Amélioration des performances et de la précision de quatre versions de MoG en utilisant le CS sur GPU : le F1-score et le temps consommé sont moyennés sur les cinq datasets présentés dans la Table. 3.3.

de comparer globalement les méthodes étudiées en présentant, de la même façon que pour CPU, l'espace "précision/temps de traitement" de ces méthodes, ainsi que leurs dérivés que nous avons améliorés. Le F1-score et le temps de traitement obtenus moyennés sur les cinq datasets pour chaque plateforme.

Nous constatons sur cette figure que le classement des méthodes étudiées selon le temps consommé et de F1-score ne change pas en passant d'une plateforme à une autre. Le temps de traitement est influencé, de la même façon pour toutes les méthodes, par la puissance de chaque plateforme. En comparant les cinq méthodes de base, notre BasicMoG donne le meilleur F1-score tandis que CvMoG2 représente le meilleur temps de calcul. Pour les versions améliorées, nous remarquons deux répartitions : Improved Cs-MoG et CvMoG2 dans la zone optimale, ensuite CS-Pham, CS-CvMoG1 et CS-Kovacev dans une zone moyennement optimale.

## 4.6 Conclusion

De la même façon que sur CPU, nous avons présenté dans ce chapitre nos contributions concernant l'implémentation de CS-MoG sur GPU. Ces contributions adaptées à chaque kernel ont été effectuées sur plusieurs niveaux : le dimensionnement des blocs de threads, l'optimisations des accès mémoire, le masquage du temps de transferts des données par les calculs et l'optimisation des instructions. De cette façon nous avons pu obtenir des gains allant jusqu'à 61% relativement à la version de base. Ainsi, notre implémentation génère les meilleurs résultats en termes de performances temporelles et de précision par rapport aux autres implémentations présentées dans l'état de l'art. Ces résultats vont être considérés comme un point d'entrée pour une exploitation simultanée CPU-GPU afin d'atteindre les performances maximales sur les plateformes hétérogènes de ce type.

Chapitre 5

# Implémentation et équilibrage de la charge de CS-MoG sur des plateformes hétérogènes CPU-GPU.

## Sommaire

## 5.1 Introduction

Le fait d'atteindre des performances élevées sur CPU et GPU ne garantit pas forcément une exploitation optimale d'une plateforme hétérogène. Cet objectif ne peut être réellement atteint que si toutes les PUs coopèrent simultanément d'une façon complémentaire et adaptée à la capacité de calcul de chacune. Dans la majorité des applications et des systèmes actuels, nous remarquons que seuls les GPU sont exploités pour effectuer des tâches de charge de calcul (CC) élevée, alors que les CPU restent souvent inactifs pendant ce temps. Une coopération de deux processeurs ayant des architectures et modes de fonctionnement différents, et qui sont liées par un bus de communication, n'est pas évidente. Cela nécessite à la fois un équilibrage de CC entre eux, et aussi le maintien de cet équilibre qui peut être influencé par le contenu des données durant l'exécution'.

Dans la première partie de ce chapitre, nous citons en détail les différentes contraintes et défis de l'exploitation simultanée des CPU et GPU pour exécuter une même tâche, tout en faisant le lien avec l'implémentation de l'algorithme CS-MoG. Ensuite, nous présentons l'approche ODDC, que nous proposons pour la répartition et l'équilibrage dynamique des données traitées par un algorithme, afin d'exploiter les PUs à leur maximum. Afin de rendre l'approche robuste aux variations brusques des temps de calcul sur les PUs (dues aux erreurs de mesure, interruptions par l'OS etc), un traitement particulier (filtrage et quantification) que nous présentons dans l'avant dernière partie doit être mis en place. Nous clôturons ce chapitre par les résultats obtenus.

## 5.2 Défis de l'équilibrage des charges de calcul

Les GPU sont généralement utilisés pour décharger le CPU du traitement des portions de code présentant un parallélisme massif de données (Data-intensive computing, Fig. 5.1). Pendant ce temps, le CPU reste inactif en attendant le retour des résultats afin de pouvoir continuer le traitement. Cette méthode classique représente une perte importante de performances, surtout si nous prenons en considération l'évolution des CPU ainsi que la

capacité de calcul qu'ils représentent de nos jours.

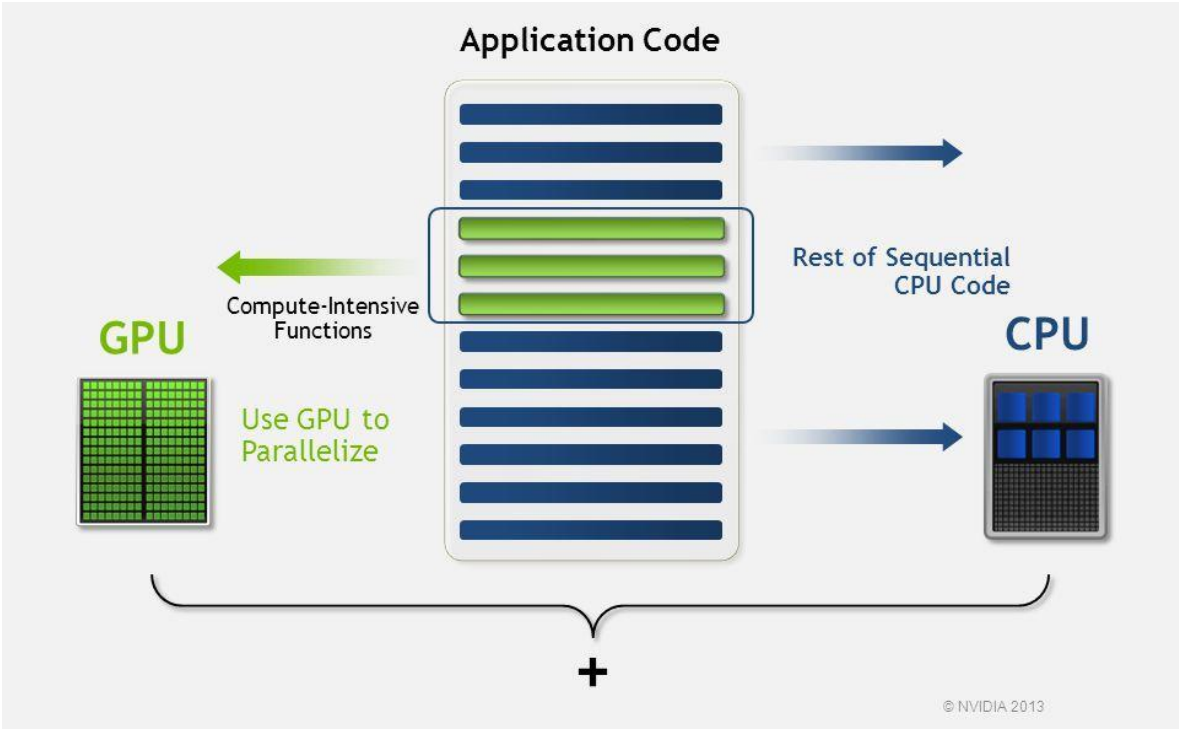


FIGURE 5.1 – Méthodologie classique d'exploitation des architectures hétérogènes : seuls les GPU exécutent les goulots d'étranglement.

L'objectif est alors d'exploiter simultanément toutes les unités que contiennent les architectures multi-processeurs/multi-cœurs pour exécuter les calculs intensifs. Dans le contexte de l'implémentation d'applications présentant un parallélisme massif de données comme CS-MoG, cela peut passer par la mise en place d'un support exécutif capable de calculer les partitions de données à faire traiter par chaque PU en vue d'aboutir à un équilibrage de charge optimal. Ce support doit prendre les lancements des traitements, les communications et synchronisations permettant de mettre en œuvre ces décisions. Dans le cas des applications de soustraction de fond comme CS-MoG, cela peut être effectué en affectant une partie de chaque image à traiter au CPU et le reste au GPU (Fig. 5.2).

L'estimation de la portion qu'il faut affecter à chaque processeur nécessite la prise en considération des trois contraintes suivantes :

1. La première est liée à la capacité de chaque PU. Sur un système donné, on peut

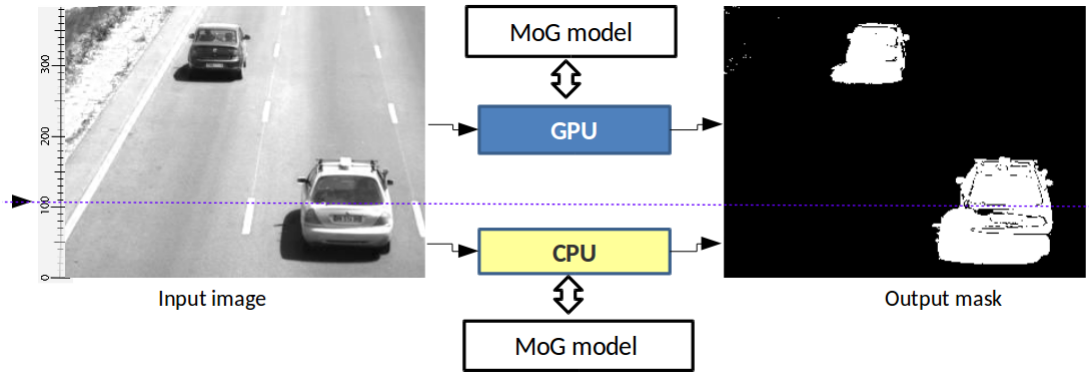


FIGURE 5.2 – Exemple d'une répartition de données entre CPU et GPU pour effectuer la soustraction de fond sur une scène routière.

trouver différentes combinaisons CPU/GPU, avec des puissances de calcul différentes. Par exemple, le GPU peut être 10 fois plus rapide que le CPU sur une plateforme donnée, alors que ce dernier peut être plus puissant sur une autre. Ainsi, l'équilibrage de CC doit pouvoir s'adapter automatiquement à chaque plate-forme.

2. L'autre défi concerne les données traitées. En effet, pour certains algorithmes, tels que CS-MoG, le contenu des images peut également influencer le temps de traitement ainsi que l'efficacité des PUs. Ce contenu diffère d'un dataset à un autre. De plus, pour un même dataset, ce contenu peut changer progressivement avec le temps.

3. La troisième concerne les caractéristiques de l'application ciblée telles que la quantité et la nature des calculs qu'elle contient ainsi que ses structures de mémorisation. Les GPU sont généralement plus puissants quand il s'agit de calculs où d'opérations identiques exécutées sur une grande quantité de données. Par contre, les CPU peuvent fournir de meilleures performances pour les applications où les transferts de données dominent le temps d'exécution, ou quand la divergence de branches ne permet pas l'exploitation simultanée de tous les cœurs du GPU. En outre, certains calculs sont moins efficaces sur GPU comme les divisions et les calculs en virgule flottante avec une double précision qui peuvent réduire ses performances 30 fois par rapport aux calculs en simple précision pour les gammes "grand public" de GPU.

Une bonne méthode d'équilibrage des CC doit alors pouvoir s'adapter à ces contraintes de

telle sorte à exploiter tous les processeurs le plus efficacement possible.

## 5.3 Curseur de distribution optimale de données

Les deux sections précédentes nous ont permis d'établir le cahier des charges auquel il faut répondre pour développer une méthode d'exécution hétérogène optimale. Cela peut être résumé au fait que cette méthode doit pouvoir estimer, pour chaque kernel, un point d'équilibre pour la répartition des charges (données ou tâches) entre CPU et GPU et le mettre à jour dynamiquement durant l'exécution. La définition et la mise à jour du point d'équilibre doivent prendre en considération la capacité de chaque PU sur la plateforme ciblée, la nature des données traitées et la caractéristique algorithmique du kernel.

La réponse à un tel cahier des charges peut être atteinte selon plusieurs approches. La première est un parallélisme de tâches (pipeline) qui consiste à répartir les kernels entre CPU et GPU et non pas les portions de données (exemple sur la Fig. 5.3.a). Cette méthode est efficace lorsqu'on travaille sur un système qui contient un très grand nombre de kernels. Cela donne l'opportunité de tester un grand nombre de scénarios, jusqu'à atteindre la distribution optimale. De plus, même quand on est devant un système qui satisfait cette condition, le test manuel de toutes les possibilités de répartition n'est pas évident. Pour répondre à cette contrainte, certaines runtimes comme StarPU [154] peuvent automatiser et optimiser cette répartition hétérogène, mais son efficacité est limitée par la surcharge générée par la gestion d'un grand nombre de kernels constituant le système. Un autre défi concernant le parallélisme de tâches peut être rencontré au niveau des transferts de données qu'il faut assurer entre les kernels s'exécutant sur des processeurs différents : le sens des transferts peut changer vu que la répartition des kernels est influencée durant l'exécution par le contenu des données. En revanche, l'adoption d'une répartition statique peut faciliter la gestion des transferts, mais avec des pertes de performance : pour le cas de CS-MoG, nous avons 3 kernels, et chacun d'eux a deux modes d'exécution possibles (sur CPU ou GPU) ; alors le nombre possible de distributions est $2^3 = 8$. Un test sur nos plateformes montre que dans aucun de ces 8 cas, nous obtenons un équilibrage de CC parfait.

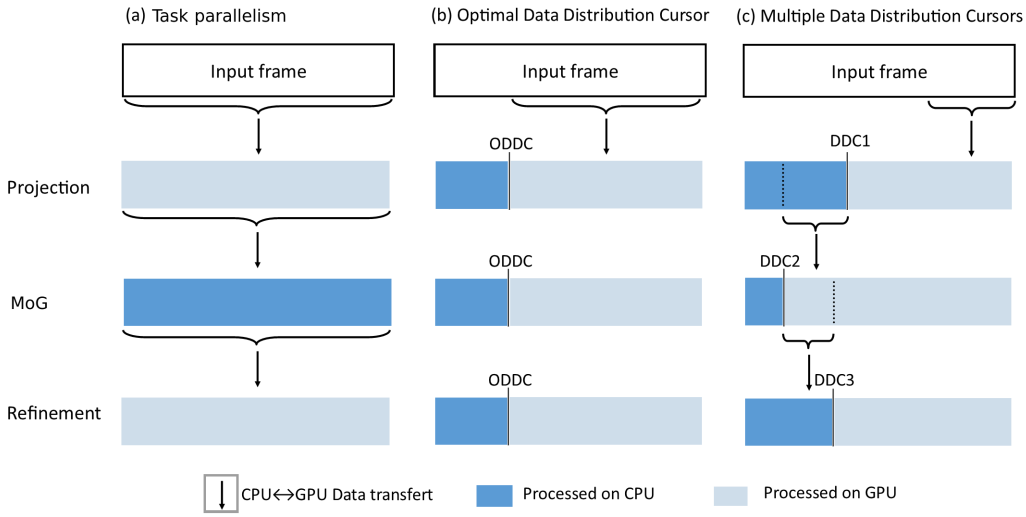


FIGURE 5.3 – Différentes possibilités de répartition de la charge de CS-MoG entre CPU et GPU d'un système hétérogène.

Pour obtenir plus de possibilités nous permettant d'obtenir des meilleures performances, nous décidons d'effectuer notre répartition au niveau des données. C'est ce qu'on appelle l'exécution concurrente, où un même kernel est exécuté simultanément sur toutes les PUs mais sur différentes portions de données. La seule limite de cette méthode est qu'elle ne peut être utilisée que dans le cas des traitements locaux, où des petits éléments de données (pixels ou zones) sont traités indépendamment comme dans le cas de CS-MoG. Le défi majeur de l'exécution concurrente est de trouver la meilleure position où il faut effectuer le découpage des données afin d'obtenir le minimum de temps d'exécution. Pratiquement, cette position sépare la partie traitée sur CPU de celle traitée sur GPU. Dans le reste de ce mémoire, nous appelons cette valeur normalisée le Curseur de Distribution de Données ou "Data Distribution Cursor" ($DDC \in [0,1]$). Donc pour des images de $S$ pixels, nous avons le nombre de pixels traités par CPU et GPU sont successivement :

$$S_{CPU} = DDC.S, \tag{5.1}$$

$$S_{GPU} = (1 - DDC).S, \tag{5.2}$$

Comme toute autre méthode de répartition des CC, la valeur du DDC peut changer selon les trois facteurs qui forment le contexte d'exécution (l'écart entre les capacités des PUs qu'on note $h$ (en référence au Hardware), la dynamicité du dataset $d$ et le kernel concerné $kl$). Chaque combinaison de ces trois métriques génère une réaction du système représentée par le temps que consomme CPU ($T_{cpu}$) et celui consommé par GPU ($T_{gpu}$). Pour simplifier le système, nous fusionnons ces deux dernières grandeurs en une seule que nous appelons l'accélération relative (Relative Speedup $RS$) comme indiqué dans l'Eq. 5.3.

$$RS(h, kl, d) = \frac{T_{cpu}(h, kl, d)}{T_{gpu}(h, kl, d)}, \tag{5.3}$$

L'estimation de la valeur de $RS$ n'a aucun sens si les quantités des données affectées aux CPU et GPU ne sont pas les mêmes. Autrement dit, $T_{cpu}$ et $T_{gpu}$ doivent représenter les temps consommés par ces deux PUs lorsqu'ils traitent des données ayant la même taille et la même dynamicité. Cette quantité peut être une image entière ou juste un ensemble de pixels. En effet, nos tests montrent qu'en modifiant le nombre de pixels traités via CS-MoG sur une image complètement homogène, $T_{cpu}$ et $T_{gpu}$ changent linéairement. Il est possible d'obtenir un même $RS$ pour deux contextes d'exécution différents. Cela est dû aux compensations entre les effets des trois métriques $h$, $kl$ et $d$. Cela signifie mathématiquement que le système est représenté par une application surjective $RS$. De ce fait, au lieu de concevoir un bloc de contrôle de notre système (estimant le DDC) en se basant sur les caractéristiques du contexte d'exécution, nous pouvons utiliser directement $RS$ qui représente l'effet global de toutes ces caractéristiques (Fig. 5.4).

La valeur du $RS$ est généralement supérieure à 1 car, sur une plateforme donnée, les GPU sont souvent plus puissants que les CPU, sauf dans le cas des kernels contenant des calculs qui favorisent le CPU : comme les applications où les transferts de données dominent le temps d'exécution, ou quand la divergence de branche ne permet pas l'exploitation

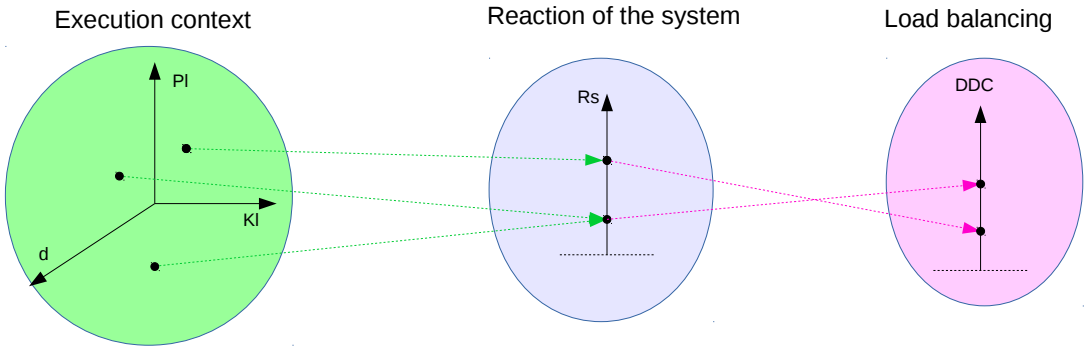


FIGURE 5.4 – Utilisation de l'application surjective $RS$ pour représenter l'effet global de toutes les caractéristiques du contexte d'exécution.

simultanée de tous les cœurs du GPU. Dans ce cas, le CPU est plus efficace, donc le $RS$ peut être inférieur à 1. Nous notons respectivement $RS_1$, $RS_2$ et $RS_3$ les RS des trois kernels (projection, MoG et raffinement) de CS-MoG. Afin d'estimer les répartitions de données les plus efficaces pour l'exécution de ces trois kernels, nous proposons deux méthodes. La première consiste à considérer chaque kernel comme un algorithme indépendant et à rechercher son point d'équilibrage de charge DDC (Exemple sur la Fig. 5.3.c). L'avantage de cette approche est qu'elle permet d'atteindre des performances maximales. Cela est dû au fait que la CC de chaque kernel est équilibrée sur CPU-GPU en fonction de son $RS$. Pour chaque kernel, le nombre de possibilités pour DDC est égal au nombre de grains de données qu'une image contient. Dans notre cas, nous travaillons, dans la version compressée de notre implémentation, avec des zones de taille 8x8, donc une image de 640x480 peut être répartie sur CPU-GPU suivant $(640x480)/(8*8) = 4800$ possibilités différentes. En conséquence, pour nos trois kernels, nous aurons $4800^3$ possibilités de distribution de données. Cela signifie que, grâce à trois DDCs indépendants, on peut viser théoriquement les meilleures performances pour nos kernels. l'Eq. 5.4. donne la valeur optimale de DDC.

$$DDC(h, kl, d) = \frac{1}{1 + RS_{kl}(h, kl, d)}, \tag{5.4}$$

par contre, le fait que ces curseurs sont indépendants, et qu'ils peuvent être déplacés

différemment durant les mises à jour effectuées (en exécution) à cause de la dynamicité des images, il y aura des transferts de données à gérer comme indiqué sur la Fig. 5.3.c). à l'intérieur d'une itération de l'algorithme : De plus, la taille de la mémoire allouée sur chaque PU changera d'une itération à l'autre. Il est alors difficile de masquer ces temps de transfert dynamiques en utilisant le mode de calcul asynchrone sur GPU : cette solution nécessite la mise en place de synchronisations à chaque étape de calcul.

La seconde solution que nous proposons pour remédier à ces problèmes consiste à utiliser un seul curseur représentant le cas optimal pour tous les kernels, et nous l'appelons le Curseur de Distribution Optimale des Données (Optimal Data Distribution Cursor ODDC Fig. 5.3.b). En adoptant cette approche, le nombre de possibilités de distribution est réduit par rapport à l'approche précédente, mais permet toujours un ajustement fin : pour une taille d'image de $640X480$, nous aurons 4800 possibilités de partitionnement (au lieu de $4800^3$ précédemment). Pratiquement, nos tests montrent que même la division de l'image en plusieurs grandes portions (100 grains seulement) est suffisante pour obtenir un bon ODDC. Le grand avantage de cette approche est qu'il n'y aura qu'un seul transfert de données à effectuer au début et à la fin de chaque itération de traitement ; ce qui nous permet de masquer son temps correspondant en utilisant le mode de transfert asynchrone.
Soient :
**ODDC** : la fraction des pixels de l'image traitée par le CPU,
$\mathbf{T_{cpu}}$ : le temps de traitement sur CPU d'une fraction ODDC des pixels de l'image,
$\mathbf{T_{gpu}}$ : le temps de traitement sur GPU d'une fraction (1-ODDC) des pixels de l'image,
**S** : la taille de l'image en pixels,
$\phi_{\mathbf{kl}}$ et $\theta_{\mathbf{kl}}$ : la vitesse de calcul en pixel/s du kernel $kl$ respectivement sur CPU et GPU,
$\mathbf{RS_{kl}}$ : le speedup GPU pour le kernel $kl$, i.e $\phi_{kl} = \theta_{kl}/RS_{kl}$,
alors :

$$T_{gpu} = S.(1 - ODDC).\sum_{kl} \frac{1}{\theta_{kl}}. \tag{5.5}$$

de la meme manière :

$$T_{cpu} = S.ODDC.\sum_{kl}\frac{1}{\phi_{kl}} = S.ODDC.\sum_{kl}\frac{RS_{kl}}{\theta_{kl}}, \tag{5.6}$$

Le défi est de trouver la valeur de l'ODDC pour qu'aucune des deux PUs ne soit inactive pendant le traitement des portions complémentaires dans une image donnée jusqu'à la fin de ce traitement. Cela signifie que les temps consommés par ces deux PUs doivent être égaux, soit l'Eq. 5.7.

$$ODDC.\sum_{kl}\frac{RS_{kl}}{\theta_{kl}} = (1 - ODDC).\sum_{kl}\frac{1}{\theta_{kl}}, \tag{5.7}$$

ce qui donne :

$$ODDC = \frac{\sum_{kl}\frac{1}{\theta_{kl}}}{\sum_{kl}\frac{1}{\theta_{kl}}+\sum_{kl}\frac{RS_{kl}}{\theta_{kl}}}. \tag{5.8}$$

Pour $kl \in [1, 3]$ on obtient l'Eq. 5.9. ci-dessous :

$$ODDC = \frac{\theta_2\theta_3+\theta_1\theta_3+\theta_1\theta_2}{(RS_1+1)\theta_2\theta_3+(RS_2+1)\theta_1\theta_3+(RS_3+1)\theta_1\theta_2}. \tag{5.9}$$

Le processus d'équilibrage de CC passe par trois phases, comme présenté sur la Fig. 5.5 : (i) après avoir pris $0,5$ par exemple comme valeur initiale du DDC, nous utilisons un ensemble suffisant d'images initiales pour réaliser un apprentissage qui nous permet de mesurer les $RS$ sur les différentes PUs pour chaque kernel de CS-MOG. (ii) Ensuite, en utilisant ces valeurs, nous estimons l'ODDC permettant d'avoir les meilleures performances à l'aide du modèle théorique. (iii) Et enfin, nous mettons continuellement à jour ce point lors de l'exécution pour prendre en compte la dynamicité des données traitées.

### 5.3.1 Cas critique : dominance du temps de transfert

L'approche que nous avons décrite précédemment est valable quand le temps de transfert de données vers le GPU est complètement masqué, et ainsi n'est pas pris en compte.

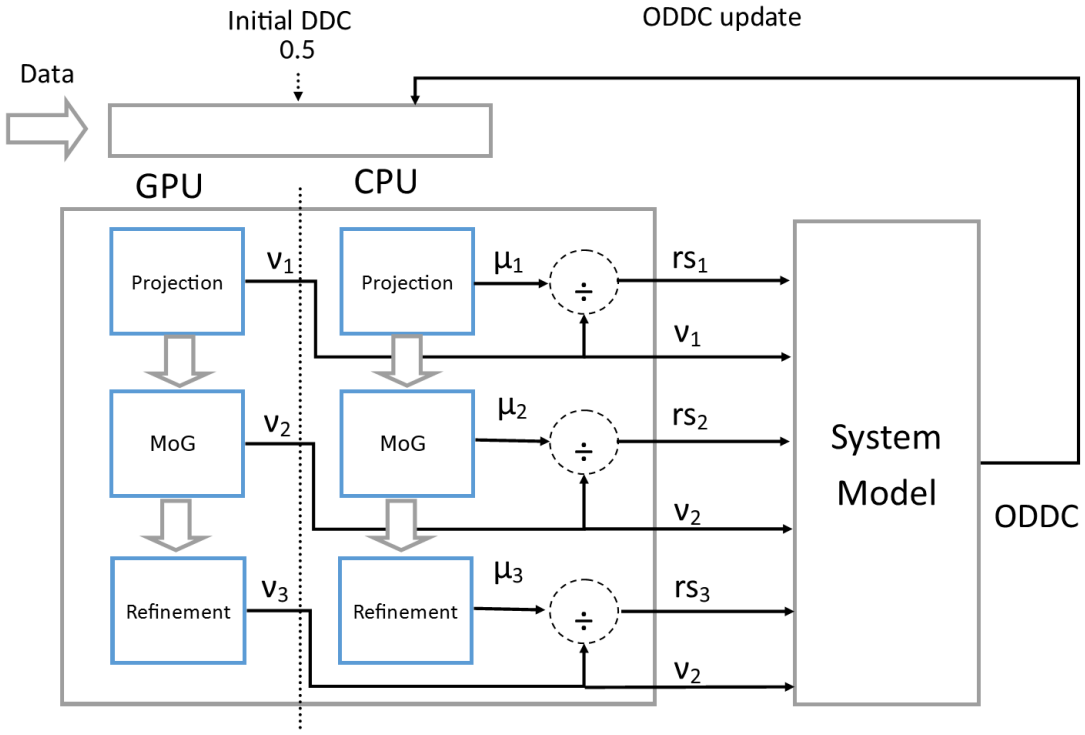


FIGURE 5.5 – Diagramme représentatif de l'initialisation, de l'estimation et de la mise à jour de l'ODDC pour une plateforme donnée.

Dans le cas où le temps de transfert est plus important que celui consommé par les calculs au niveau GPU, cette approche n'est plus valide. Pour prendre ce cas en considération nous avons procédé de la façon suivante : une comparaison entre $T_p$ (temps de calcul) et $T_t$ (temps de transfert) est effectuée régulièrement au niveau de chaque image. Si le cas critique est détecté, nous étendons le temps de calcul pour qu'il soit égal à celui du transfert. Autrement dit le temps de chaque kernel est étendu en se basant sur ce qu'il représente dans le temps initial (Fig. 5.6).

Ainsi le temps $T_{kl}$ consommé par le GPU pour exécuter un kernel donné devient $T_{kl}{}'$ selon l'Eq. 5.10.

$$T_{kl}{}' = \frac{T_{kl}}{T_p} * max(T_p, T_t). \tag{5.10}$$

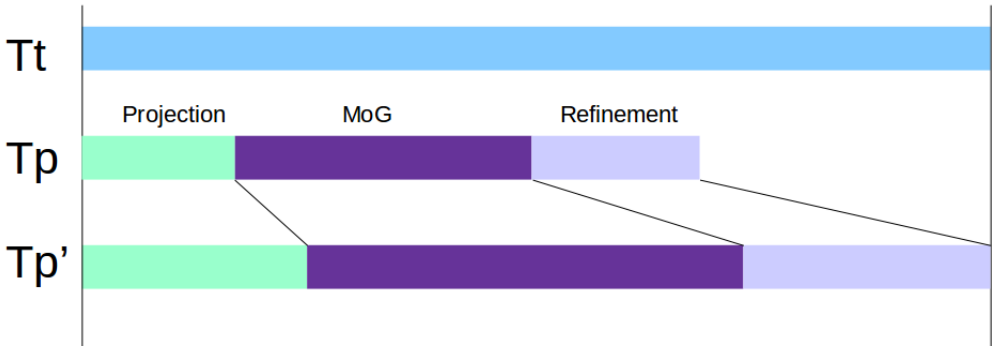


FIGURE 5.6 – Mise à jour du temps consommé par chaque kernel au GPU afin de prendre en compte les cas critiques où le temps de transfert est dominant.

De cette façon, le calcul des accélérations relatives prend en considération la latence due au transfert des images.

## 5.4 Stabilisation de répartition : filtrage et quantification

La précision de la mesure du temps consommé au niveau de chaque partie du programme peut influencer l'estimation de l'ODDC. En effet, quand le programme exécuté occupe entièrement les PUs d'une plateforme, le système d'exploitation intervient pour forcer l'exécution des processus système de haute priorité. Cela cause des erreurs de mesure qu'il faut prendre en compte durant la mise à jour de l'ODDC. Pour cela, nous appliquons un filtrage sur ce dernier en se basant sur son historique afin d'éviter de ne pas prendre en compte ces erreurs de mesure dans le calcul de sa mise à jour. La Fig. 5.7 représente le système amélioré : après la phase d'entrainement, ce système passe au traitement des données acquises en ligne, et génère pour chaque image les écarts de temps $\Delta_{(t,kl)}$ entre les PUs pour chaque kernel $kl$. Notre bloc ODDC utilise ces information ainsi que la valeur du curseur courant $ODDC_t$ pour générer le curseur suivant $ODDC_{t+1}$. Au lieu d'effectuer la répartition des données en se basant directement sur cette valeur, nous l'insérons dans une file de valeurs représentant son historique, et nous estimons la valeur médiane $\widehat{ODDC}_{t+1}$ de cette file. Nos tests montrent qu'une valeur médiane calculée sur 32 échantillons génère

des résultats satisfaisants. Ce filtre permet d'éviter les vibrations du curseur et réduire

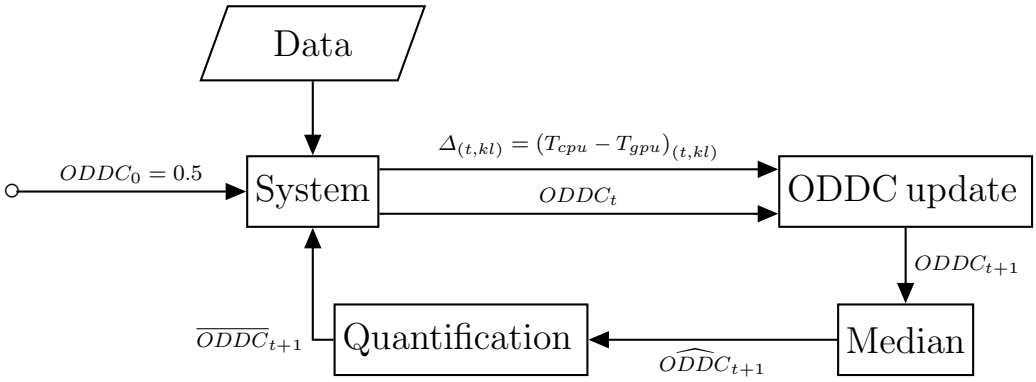


FIGURE 5.7 – Processus de stabilisation de l'ODDC en utilisant le filtrage médian et les valeurs approximatives quantifiées.

ainsi sa sensibilité, cependant, afin d'éviter des changements trop fréquents de l'ODDC, nous proposons de le mettre à jour que lorsqu'un écart suffisamment important entre les PUs est détecté. Dans le cas contraire, des transferts de très petites portions du modèle d'arrière-plan doivent être effectués entre ces PUs, ce qui est coûteux. Pour limiter ce phénomène nous autorisons la mise à jour de l'ODDC uniquement lorsque l'écart entre sa nouvelle valeur estimée et la précédente est supérieure à un seuil donné. De cette façon les mouvements du curseur sont quantifiés grâce à des valeurs approchées $\overline{ODDC}_{t+1}$. Cette étude est validée sur quatre architectures hétérogènes différentes et cinq datasets représentant quelques cas critiques. OpenMP est utilisé pour le parallélisme multi-cœur sur CPU, et CUDA pour l'exécution des calculs sur GPU. Les résultats obtenus montrent que grâce au masquage de tous les transferts, l'ODDC atteint une meilleure vitesse de traitement par rapport à l'utilisation de multiples DDC.

## 5.5 Résultats et synthèse

Dans cette section, nous testons notre approche (ODDC) uniquement sur la version CS-MoG que nous avons améliorée, vu qu'elle représente le meilleur candidat pour la soustraction de fond d'après les résultats obtenus dans les chapitres précédents. Cependant,

tout ce que nous appliquons à cette version peut également s'appliquer aux autres.

Afin de mesurer l'influence du contenu des images sur l'ODDC, nous utilisons d'autres datasets qui sont plus pertinents. En effet, ce qui nous intéresse dans ce chapitre est d'avoir des cas très différents en matière de dynamicité. Comme nous n'avons pas besoin de tester la précision des résultats, la disponibilité de la vérité du terrain (ground truth) n'est pas nécessaire. Ainsi, contrairement aux chapitres précédents, cela nous donne plus de choix vu que nous pouvons utiliser n'importe quelle vidéo qui satisfait la dynamicité recherchée.

Pour illustrer les différents cas, nous utilisons trois datasets (Fig. 5.8) : le premier est une scène capturée la nuit sur une autoroute bondée de véhicules. Le second est un cas intermédiaire qui représente une dynamicité moyenne sur une route urbaine. Le troisième est une scène artificielle représentant une route avec un minimum de véhicules en mouvement. Ces datasets sont nommés successivement MaxD, MedD et MinD dans le reste de ce manuscrit.

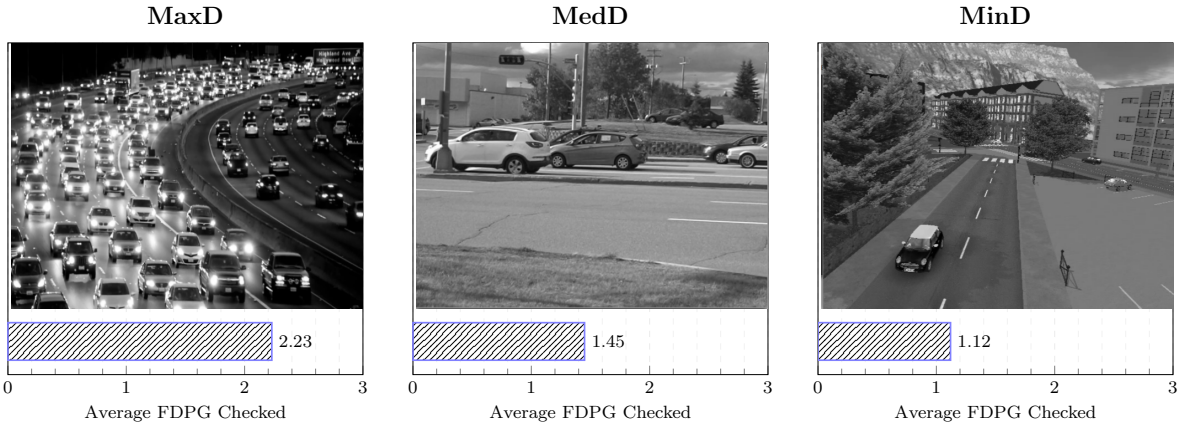


FIGURE 5.8 – Captures des datasets et taux moyens (%) des FDPGs (dynamicité) nécessaires dans le traitement de leurs images.

La dynamicité, dans notre travail précédent [159], est représentée par le taux de zones 8x8 qui sont segmentées comme faisant partie des objets de premier plan. Les éléments constituant ces zones sont supposés être décrits par un grand nombre de FDPGs dans le modèle d'arrière-plan correspondant. Le nombre moyen de FDPGs utilisé pour décrire les projections de chaque dataset est indiqué sur la Fig. 5.8. Ces valeurs sont obtenues pour

des séquences de 300 images de taille 640x480.

La première étape de notre méthode d'équilibrage des CC consiste à mesurer les *RS* des trois kernels qui constituent le CS-MoG amélioré. Ces valeurs seront le principal élément d'entrée pour l'estimation de l'ODDC. Comme indiqué dans la section 5.3.1, la mesure des *RS* doit prendre en considération le cas où le temps de transfert des données vers le GPU est supérieur à celui du calcul. La Table. 5.1 représente les valeurs obtenues de ces deux mesures pour chaque plateforme et chaque jeu de donnée.

| | | Laptop 1 | Laptop 2 | Wokstation 1 | Workstation 2 |
|---|---|---|---|---|---|
| MaxD | Transfer | 0.16 | 0.1 | 0.09 | 0.08 |
| | Process | 0.23 | 0.065 | 0.06 | 0.048 |
| MedD | Transfer | 0.16 | 0.1 | 0.09 | 0.08 |
| | Process | 0.22 | 0.058 | 0.055 | 0.044 |
| MinD | Transfer | 0.16 | 0.1 | 0.09 | 0.08 |
| | Process | 0.195 | 0.054 | 0.056 | 0.045 |

TABLE 5.1 – Comparaison des temps de transfert et du calcul obtenus sur les GPU de nos plateformes pour trois datasets, selon l'ODDC calculé par l'Eq. 5.9.

Nous constatons que pour le Laptop1, le temps de transfert est inférieur à celui du calcul, et l'opposé pour les trois autres plateformes (cas critiques). L'effet de ce phénomène est important surtout pour une exécution en ligne (sur flux d'images) où le transfert se fait en parallèle avec les calculs. Cependant, quand il s'agit d'un traitement hors ligne (sur lot d'image), il est possible, d'effectuer le transfert massif de plusieurs images vers la mémoire du GPU avant ou durant le traitement. Dans la suite de cette section, nous étudions ces deux cas d'une façon séparée.

### 5.5.1 Exécution sur flux d'images

Dans cette partie des résultats, le processus du transfert des images ainsi que le calcul sur GPU sont lancés simultanément. Ainsi, les *RS* sont estimés en utilisant l'Eq. 5.10. Sur la Fig. 5.9, chaque graphique montre les valeurs obtenues sur chaque plate-forme pour les trois kernels, et les barres représentent les datasets.

Cette figure peut être interprétée selon les trois aspects qui constituent le contexte d'exé-

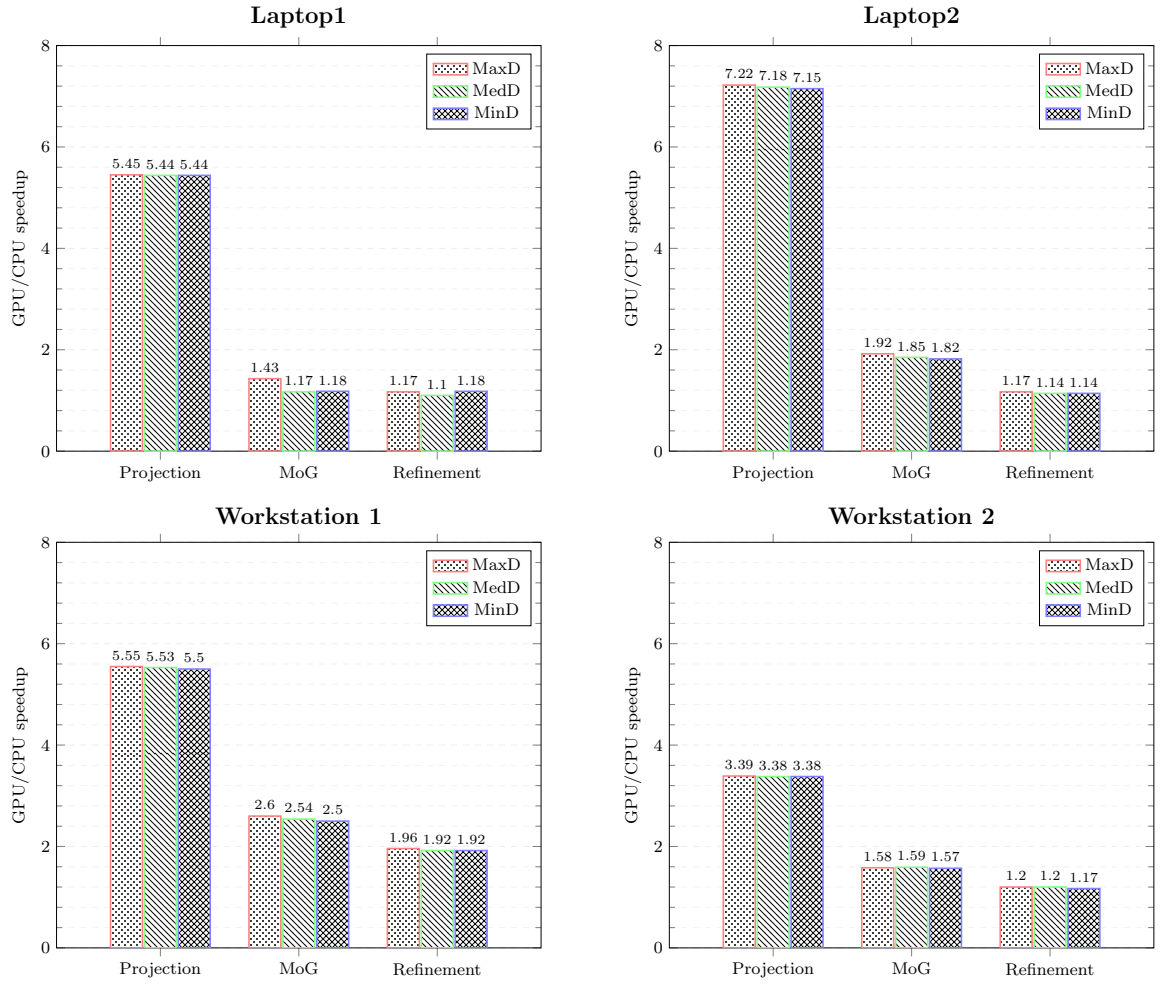


FIGURE 5.9 – L'accélération relative ($RS$) obtenue pour les kernels de CS-MoG amélioré avec la prise en compte du temps de transfert sur les plateformes présentées dans Table. 3.1 et les trois datasets présentés dans la Fig. 5.8.

cution : le kernel, la plateforme matérielle et le dataset. En ce qui concerne le premier, nous constatons que les valeurs obtenues au niveau de chaque kernel sont différentes des autres. Les $RS$ arrivent jusqu'à 7.22 pour la projection, et reste également important pour MoG (jusqu'à 2.6) et le raffinement (jusqu'à 1.96). Cela signifie que le calcul de projection est plus rapide sur GPU car il est basé sur des opérations matricielles simples et fortement parallélisables. Cependant, les processus de MoG et de raffinement ne sont pas aussi simples, car ces kernels contiennent des branches divergentes et un contrôle de flux

considérable. Dans la suite de l'interprétation des résultats nous allons se concentrer sur ces deux derniers kernels. Du point de vue plateforme, les ordres de grandeur des $RS$ sont similaires. L'effet des datasets est légèrement présent mais faible par rapport aux autres facteurs mentionnés ci-dessus. La relation entre le $RS$ et la dynamicité n'est pas linaire, et l'effet de cette dernière change d'une plateforme à une autre. Cela n'est pas considéré comme contrainte devant notre système d'équilibrage des CC vu qu'il n'utilise pas directement cette métrique : mais plutôt utilise le $RS$ qui est la combinaison des effets de tous les éléments du contexte d'exécution.

En utilisant ces mesures comme entrées du modèle qui décrit notre approche, nous pouvons déduire l'ODDC qui donne les meilleures performances. La Table. 5.2 représente les valeurs attendues de l'ODDC, pour chaque plate-forme et chaque dataset, en utilisant les équations 5.9. et 5.10.

| **Platform** | **MaxD** | **MedD** | **MinD** |
|---|---|---|---|
| Laptop1 | 0.305 | 0.316 | 0.297 |
| Laptop2 | 0.306 | 0.328 | 0.323 |
| Workstation 1 | 0.249 | 0.225 | 0.237 |
| Workstation 2 | 0.352 | 0.379 | 0.395 |

TABLE 5.2 – ODDC attendu entre CPU et GPU pour trois plateformes et datasets, selon l'Eq. 5.9. et 5.10.

Pour vérifier l'efficacité de ces attentes, nous avons effectué un ensemble de mesures où nous avons testé toutes les possibilités d'ODDC entre 0 et 1, en utilisant des sauts de 0.01 pour chaque plateforme et chaque dataset. Les graphiques sur la Fig. 5.10 représentent le temps de traitement obtenu ainsi que les attentes d'ODDC présentées sur la Table. 5.2. Pour tous les cas présentés, le temps de traitement global commence par une grande valeur lorsque le curseur est à 0 (c'est-à-dire lorsque toutes les données sont traitées sur GPU et que le CPU reste inactif). Ensuite, le temps de traitement commence à diminuer puisque nous commençons à déplacer certaines parties des données à traiter vers CPU, jusqu'à atteindre un minimum. Ce minimum peut être considéré comme un point d'équilibrage de

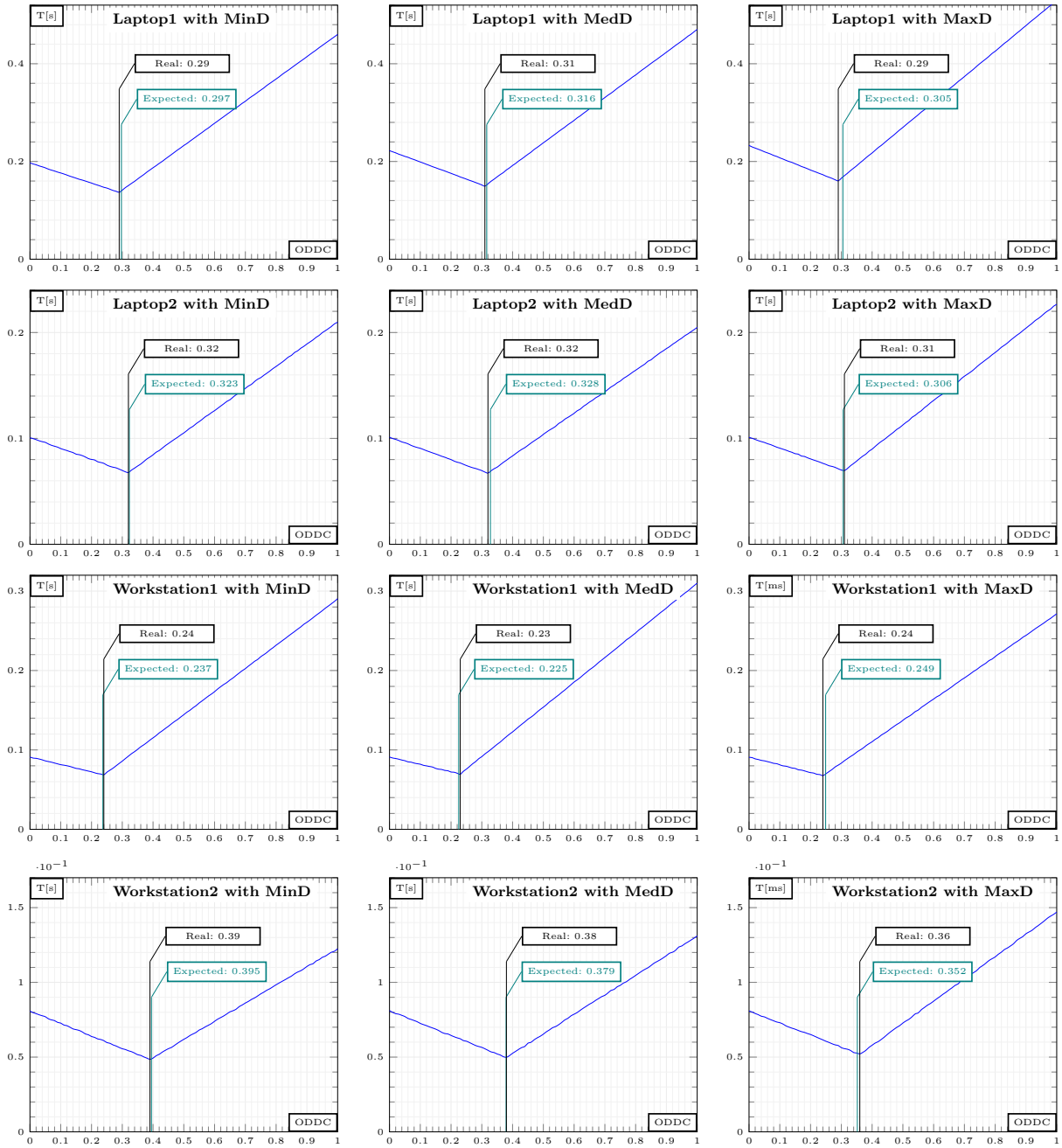


FIGURE 5.10 – Comparaison de l'ODDC estimé avec le temps minimal obtenu sur en utilisant 300 images de chaque dataset présenté sur la Fig. 5.8.

CC. Lorsque nous dépassons ce point en déplaçant plus de données vers CPU, le temps de traitement global commence à augmenter de manière significative en raison de l'inactivité croissante du GPU.

Comme le montre cette figure, les performances optimales obtenues expérimentalement en utilisant ODDC atteignent, pour toutes les plateformes et datasets, environ 98% des performances maximales possibles alors que cette valeur ne dépasse pas 87,7% dans les résultats présentés dans [145]. Cela montre que notre approche permet de définir une répartition optimale des données après la phase d'entrainement. De plus, afin de prendre en compte les éventuels changements de scène dans le temps, une mise à jour est effectuée en ajustant cette répartition dans des intervalles réguliers. Cette mise à jour est effectuée lors de l'exécution une fois que nous commençons à avoir un décalage qui dépasse 2.5% entre le temps de traitement mesuré sur le CPU et celui obtenu sur GPU. De cette façon, une nouvelle estimation permet de déplacer l'ODDC dans la bonne direction et aide à rééquilibrer le système. Globalement, cette distribution adaptative de la CC a amélioré les performances de l'algorithme étudié. Les gains obtenus par rapport à l'implémentation uniquement sur CPU ou GPU sont présentés dans la Table 5.3. En utilisant l'exécution en

| Gains | Laptop 1 | Laptop 2 | Wokstation 1 | Workstation 2 |
|---|---|---|---|---|
| % CPU Only | 70% | 69% | 78% | 65% |
| % GPU Only | 32% | 33% | 25% | 40% |

TABLE 5.3 – Gains obtenus grâce à l'exploitation simultanée en ligne du CPU et du GPU par rapport à l'exécution en utilisant qu'une de ces PUs.

ligne, les gains obtenus par rapport à l'exécution en exploitant uniquement CPU arrivent jusqu'à 78%. Ce chiffre pourrait être plus important, mais à cause du temps de transfert des données, l'apport de l'implication du GPU dans les calculs n'est pas maximal. En revanche, l'implication du CPU réduit le temps de calcul jusqu'à 40% par rapport à l'exécution uniquement sur GPU, ce qui est important comme gain malgré l'écart théorique en termes de capacités entre ces deux PUs.

### 5.5.2 Exécution hors ligne

Afin d'estimer les capacités de calcul relatives entre CPU et GPU sans prendre en compte la latence due à la communication, nous lançons le transfert de toutes les images vers GPU avant les calculs. En réalité, ce processus peut être appliqué aux applications où

les données à traiter sont entièrement disponibles. Dans ce cas, le fait d'appeler les fonctions de transfert pour des grandes quantités de ces données (toutes les images) est plus efficace que des appels transférant de petites quantités. Sur la fig. 5.11, chaque graphique montre les valeurs obtenues sur nos plateformes de la même façon que la section précédente.

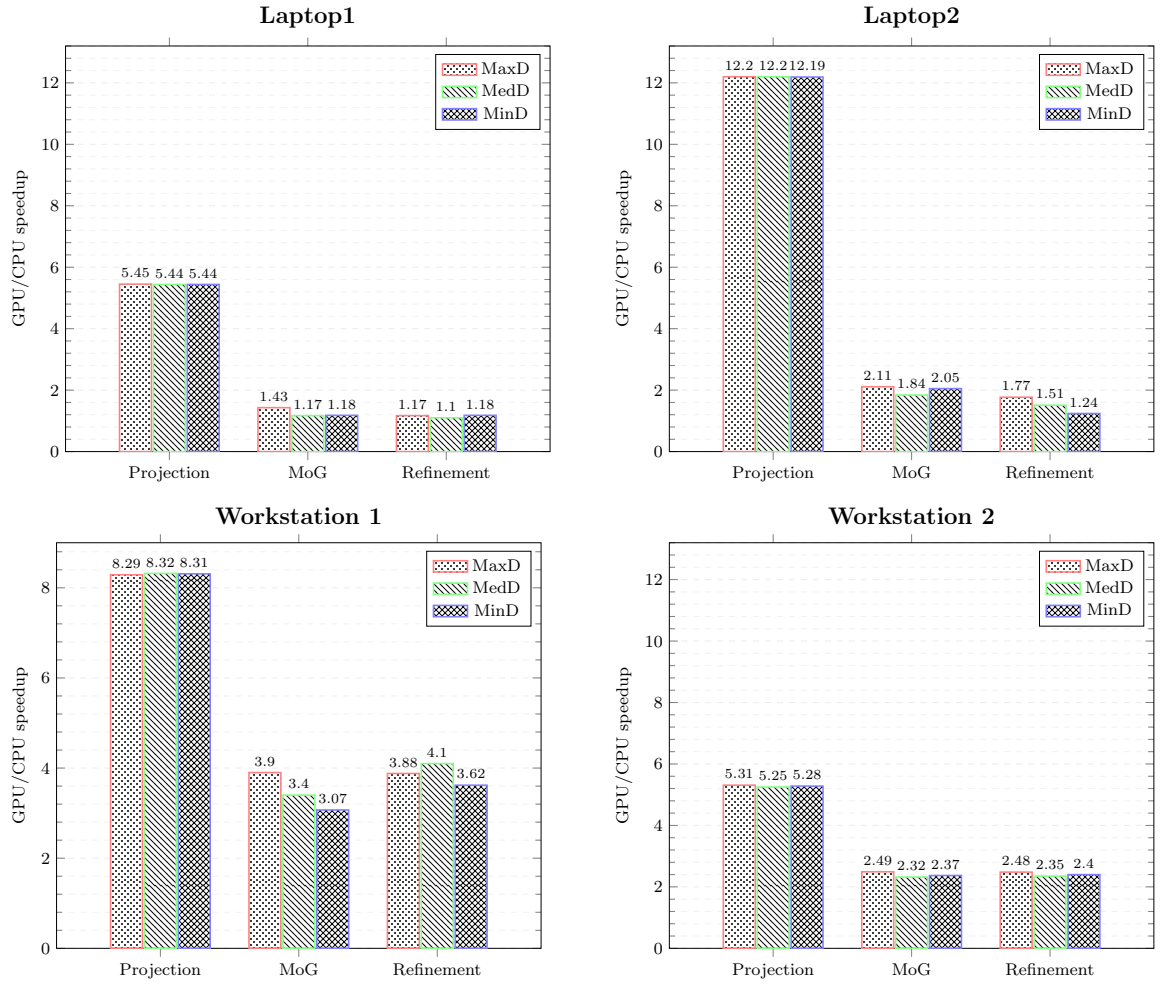


FIGURE 5.11 – L'accélération relative ($RS$) sur CPU-GPU pour les kernels de CS-MoG amélioré, sans prise en compte du temps de transfert sur les plateformes présentées dans la Table. 3.1 et les trois datasets présentés dans la Fig. 5.8.

Tout ce que nous avons dit durant l'analyse de la Fig. 5.9 dans la section précédente reste valable pour celle-ci. La seule différence constatée est que l'ordre de grandeur des $RS$ a augmenté. Cela est dû au fait que les capacités des GPU ne sont plus limitées par les transferts des données. Le Laptop1 n'est pas concerné par ce changement vu qu'il n'était

pas impacté par les transferts de données. En appliquant notre modèle cette fois sur ces nouvelles mesures, nous estimons les valeurs de l'ODDC correspondant à cette exécution hors ligne. Les valeurs obtenues sont présentées dans la Table 5.4.

| **Platform** | **MaxD** | **MedD** | **MinD** |
|---|---|---|---|
| Laptop1 | 0.305 | 0.316 | 0.297 |
| Laptop2 | 0.215 | 0.218 | 0.208 |
| Workstation 1 | 0.141 | 0.146 | 0.154 |
| Workstation 2 | 0.242 | 0.251 | 0.268 |

TABLE 5.4 – ODDC attendu entre CPU et GPU pour trois plateformes et datasets lors de l'exécution hors ligne, selon l'Eq. 5.9.

En faisant le test de toutes les possibilités d'ODDC entre 0 et 1 par pas de 0.01, nous comparons la Fig. 5.12 les valeurs permettant d'avoir le minimum de temps de traitement et celles attendu grâce à notre modèle théorique. De la même façon que l'exécution en ligne, nous constatons que la partition des données prévue par notre approche correspond à la valeur minimale du temps de traitement dans tous les cas étudiés. Par rapport à l'implémentation homogène en utilisant uniquement CPU ou GPU, des gains de performances importants sont obtenus et présentés dans la Table. 5.5.

| Gains | Laptop 1 | Laptop 2 | Wokstation 1 | Workstation 2 |
|---|---|---|---|---|
| % CPU Only | 70% | 79% | 86% | 76% |
| % GPU Only | 32% | 23% | 20% | 27% |

TABLE 5.5 – Gains obtenus grâce à l'exploitation simultanée hors ligne du CPU et du GPU par rapport à l'exécution en utilisant qu'une de ces PUs.

En utilisant l'exécution hors ligne, les gains obtenus par rapport à l'exécution uniquement sur CPU sont plus importants (arrive jusqu'à 86%). Cela signifie que l'implication du GPU dans les calculs dans ce cas a un grand impact sur les performances, vu que la capacité de ce dernier n'est pas limitée par les transferts de données.

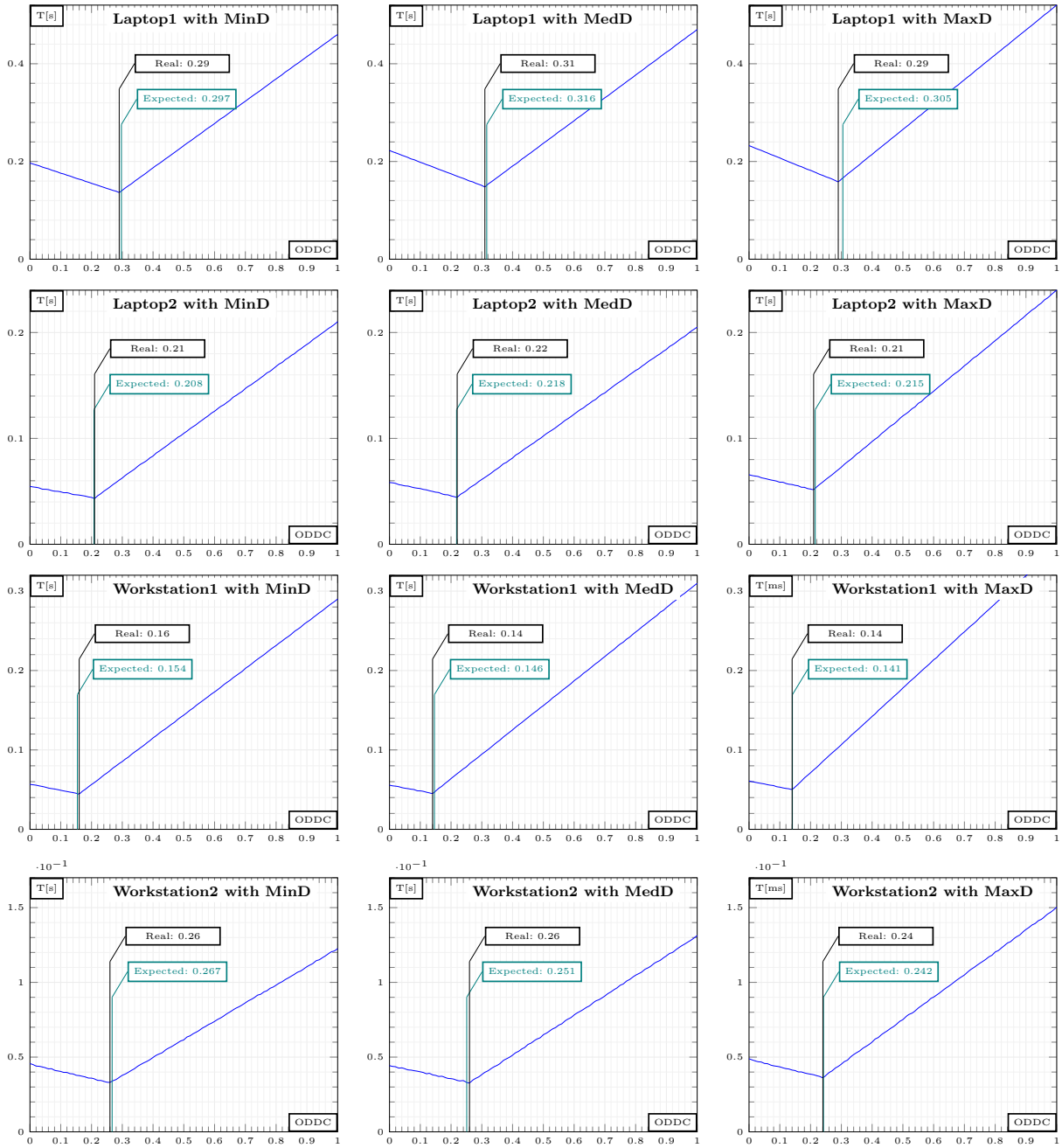


FIGURE 5.12 – Comparaison de l'ODDC estimé avec le temps minimal obtenu sur trois plateformes en utilisant 300 images de chaque dataset présenté sur la Fig. 5.8.

### 5.5.3 Stabilisation de l'ODDC

Pour illustrer les résultats de stabilisation de l'ODDC mentionnée dans la section 5.4, la Fig. 5.13. représente l'effet de chaque bloc de filtrage (Médian et quantification) sur la stabilité de notre curseur. Cette figure est générée grâce au dataset MaxD, sur le Laptop1, avec une valeur initiale de l'ODDC égale à 0.5.

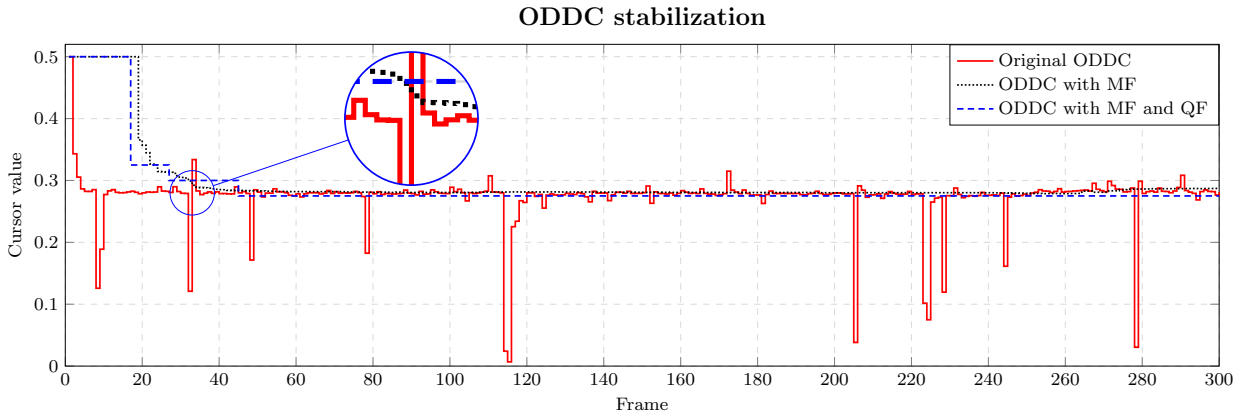


FIGURE 5.13 – Effet des améliorations, filtrage médian (MF) et quantification (QF) sur la stabilité de l'ODDC.

La courbe rouge représentant les valeurs de l'ODDC brutes estimées. Ces valeurs sont parfois erronées à cause des arrêts de CS-MoG forcé par l'OS afin d'exécuter d'autres processus plus prioritaires, ou à cause des erreurs de mesure. Cela ralentit brusquement le CPU, ainsi l'ODDC non stabilisé affecte plus de données au GPU, ce qui explique les pics orientés vers le bas. Après avoir appliqué le filtre médian, en utilisant 32 échantillons, ce phénomène est éliminé, et nous obtenons des valeurs plus stables. Ce nombre d'échantillons est choisi de telle sorte à assurer une meilleure stabilisation sans générer un surcoût de calcul. L'application de la quantification permet de conditionner la mise à jour de l'ODDC pour ne prendre en compte que les changements significatifs. Cette stabilisation permet d'optimiser le temps du traitement global grâce à la réduction des transferts dus aux très petits mouvements de l'ODDC. Une comparaison des temps consommés est présentée dans Table. 5.6.

| | Original ODDC | + MF | + MF and QF |
|---|---|---|---|
| $T_{cpu}$ | 0.154 | 0.162 | 0.16 |
| $T_{gpu}$ | 0.162 | 0.157 | 0.159 |
| Update surcoût | 0.032 | 0.018 | 0.005 |
| Global time | 0.195 | 0.18 | 0.166 |

TABLE 5.6 – Comparaison des temps de traitement de 300 images du dataset MaxD, obtenus en appliquant les différentes phases de stabilisation, filtrage médian (MF) et quantification (QF), à l'ODDC sur le Laptop1.

Sans stabilisation, nous constatons un surcoût de 0.032s causé par les transferts inutiles de petits grains du modèle de l'arrière-plan. Ce surcoût représente environ 5.4% du temps de traitement total. Nous remarquons également un déséquilibre entre les PUs D'environ 0.008s. Avec le filtrage médian, le surcoût diminue vers 3% ainsi que le décalage qui devient 0.005s. Les meilleures performances sont obtenues après la quantification (QF) avec seulement un surcoût de 0.8% et un décalage de 0.001 s. Concernant la comparaison des trois méthodes de répartition des CC présentées dans la Fig. 5.3 (Répartition des tâches, multiple DDC, et ODDC), nous avons mesuré le temps minimal pouvant être obtenu par chaque méthode sur nos plateformes. Le temps de traitement de 300 images obtenu est présenté dans la Fig. 5.14. Comme prévu, nous pouvons voir que l'équilibrage des CC via le parallélisme de tâches représente le maximum de temps, ainsi le minimum de performances dans la plupart des cas. L'utilisation de plusieurs DDC devrait normalement être la meilleure approche, mais les multiples transferts de données entre les mémoires CPU et GPU limitent les performances. Cela rend notre solution, en utilisant un ODDC unique, un meilleur choix. En outre, il peut également être appliqué sur de nombreux algorithmes autres que CS-MoG qui contiennent des kernels successifs.

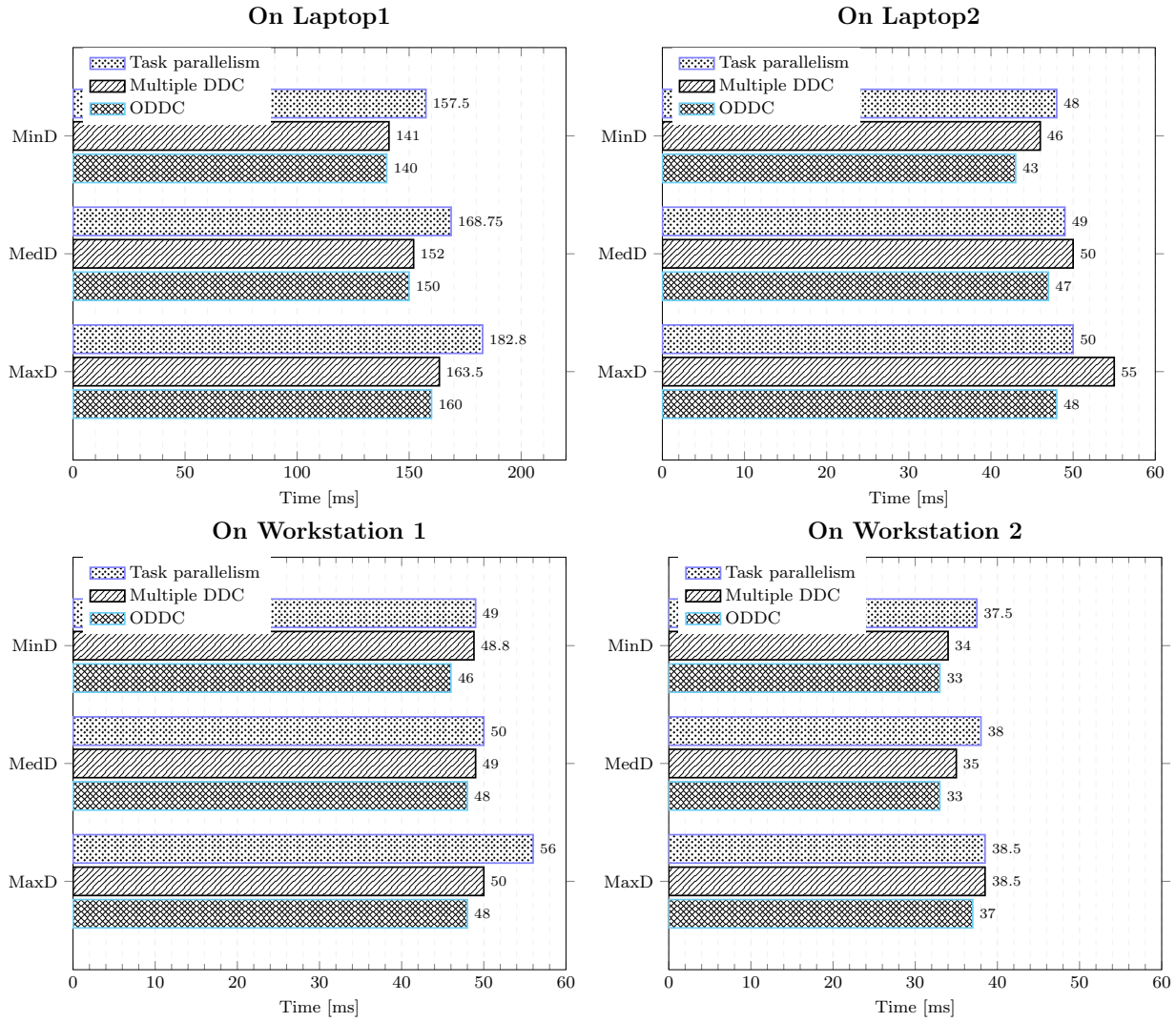


FIGURE 5.14 – Temps de traitement de 300 images de 640 x 480 en utilisant les trois approches possibles pour l'équilibrage de la charge de travail.

## 5.6 Conclusion

Nous avons introduit dans ce chapitre les différentes contraintes et défis de l'exploitation simultanée des CPU et GPU pour exécuter CS-MoG. Ensuite, nous avons présenté l'approche ODDC, que nous proposons pour la répartition et l'équilibrage dynamique des données traitées par cet algorithme, afin de mieux exploiter les PUs. Un bloc de stabilisation a été mis en place afin de rendre notre approche robuste aux erreurs de mesure. Les

résultats obtenus dans ce chapitre nous permettent de tirer les conclusions suivantes :

— L'exploitation simultanée de CPU et de GPU pour effectuer la soustraction de fond via CS-MoG permet d'optimiser significativement les performances.

— notre approche d'équilibrage de CC, ODDC, permet de prévoir la répartition de données permettant d'atteindre les performances maximales sur une plateforme donnée.

— le processus de mise à jour et de stabilisation de l'ODDC permet de prendre en considération les différents changements dans l'environnement d'exécution pour maintenir un équilibre entre les PUs tout en optimisant les transferts de données.

# Conclusions et perspectives

Nous avons présenté dans cette thèse l'algorithme CS-MoG qui est une combinaison de deux techniques largement adoptées en traitement d'image : le mélange de composantes gaussiennes (MoG) qui est utilisée pour la soustraction de fond, et l'acquisition comprimée (CS) qui sert à réduire la dimensionnalité des données. Malgré les avantages que représente CS-MoG en matière de rapidité et de précision, nous avons remarqué que cette méthode n'a pas encore été implémentée de sorte à exploiter l'évolution des architectures de calcul CPU et GPU et leur combinaison. Les principales contributions et conclusions de cette thèse seront résumées dans cette dernière partie du mémoire, avant de finir par la présentation des prochains travaux et perspectives.

Les contributions de cette thèse peuvent être répertoriées en deux parties principales :

— La première consiste en la mise en adéquation séparée des kernels constituant CS-MoG sur les architectures multi-cœur CPU et GPU. Cette adéquation a permis d'obtenir une version plus rapide que d'autres implémentations publiées précédemment, y compris celles de la bibliothèque OpenCV.

— La deuxième concerne la proposition d'une approche qui exploite simultanément toutes les PUs d'une plateforme hétérogène tout en assurant un équilibrage de charge entre eux.

Pour la première contribution, nous avons détaillé les caractéristiques de chaque type de processeur, et nous avons défini un guide pratique permettant de porter un algorithme donné comme CS-MoG sur ce processeur. Sur CPU, d'une part, nous avons utilisé la technique de vectorisation pour maximiser l'exploitation interne de chaque cœur. En appliquant les instructions de calcul sur des vecteurs de données au lieu d'un seul élément, nous avons constaté que les gains qui peuvent être obtenus dépendent du contenu des images. Ce contenu est représenté par une métrique que nous avons appelée "dynamicité", et qui permet au programmeur de prévoir l'efficacité de la vectorisation. La dynamicité

des datasets utilisés pour les tests a permis d'obtenir des gains allant jusqu'à 6%. D'autre part, nous avons ciblé l'amélioration de la scalabilité de l'algorithme en question lorsque plusieurs cœurs de calcul sont utilisés. Nous avons ainsi obtenu une accélération qui atteint un facteur de 5.45 quand 6 cœurs sont utilisés. Cela signifie que nous avons obtenu une scalabilité quasi linéaire malgré le coût que génère le framework OpenMP utilisé pour le multithreading.

Sur GPU, nous avons mis en œuvre le masquage du temps de transfert par celui du traitement en utilisant les *streams* CUDA. L'amélioration des performances grâce à cet aspect atteint jusqu'à 75%. Egalement, nous avons effectué des optimisations concernant l'exécution des threads et des instructions. Cela a permis d'obtenir un gain allant jusqu'à 61% grâce à l'équilibrage de la charge de traitement entre les threads, la mise en place d'accès contigus à la mémoire, l'occupation maximale des Warps, l'optimisation des opérations et ainsi que le flux de contrôle.

Dans la deuxième contribution, nous avons estimé l'accélération relative entre CPU et GPU pour chaque kernel de CS-MoG. Nous avons ensuite étudié une série de possibilités pour équilibrer la charge de travail entre ces PUs et nous avons présenté une solution optimale qui prend cette accélération relative comme entrée. La méthode proposée est efficace, simple à implémenter et s'adapte automatiquement à toute plateforme hétérogène ciblée. De plus, elle prend en considération la variabilité de la charge de travail en fonction de l'irrégularité des données. Pour n'importe quelle plateforme, notre modèle peut en outre estimer la répartition optimale de cette charge entre GPU et CPU, ce qui permet d'obtenir jusqu'à 98% des performances maximales possibles. Cette exploitation simultanée de toutes les PUs a amélioré les performances de l'algorithme étudié jusqu'à 78% par rapport à une implémentation sur CPU uniquement, et jusqu'à 40% par rapport à une implémentation uniquement sur GPU.

Les perspectives envisageables après cette thèse concernent 3 objectifs importants :

— **À court terme,** nous allons répondre au manque de datasets constaté dans la littérature. Nos contributions consisteront à proposer de nouvelles séquences vidéo de haute résolution représentant différentes situations. Ces séquences vont être fournies avec leur vérité de terrain (ground truth), ainsi que des métriques qui les caractérisent comme la dynamicité. Ensuite, des méthodes proposées dans l'automatique pour la modélisation et la régularisation formelle des systèmes informatiques peuvent être utilisées dans notre approche. Ces méthodes vont nous permettre, de bien représenter les éléments constituant le contexte d'exécution (l'algorithme, la plateforme et les données), et aussi, de reformuler les différents problèmes et aspects de parallélisme. Ainsi, la version finale de notre implémentation pourra être intégrée dans les futures versions d'OpenCV.

— **À moyen terme,** en se basant sur notre modélisation formelle, nous allons pouvoir généraliser l'ODDC sur d'autres algorithmes largement utilisés dans le domaine du traitement de signal et de l'image. Nous allons cibler deux familles d'algorithmes : celle où la quantité des calculs dépend du contenu des données traitées, comme le cas de l'étiquetage des composantes connexes (CCL), et celle où les calculs ne sont pas sensibles à cet aspect comme la Transformée de Fourier Rapide (FFT).

D'autres environnements doivent être explorés en termes de plateformes. Des architectures embarquées avec des ressources plus limitées peuvent être ciblées, ainsi que d'autres composants de traitement tels que les FPGAs et les DSPs. Cela nous permettra d'élargir notre modèle pour prendre d'autres aspects en considération comme la consommation d'énergie qui est un élément critique dans le domaine de l'embarqué. Le fait d'expérimenter d'autres composants matériels va nous permettre ainsi de concevoir une stratégie de portage plus flexible et plus globale.

— **À long terme,** notre approche ODDC pourra être combinée avec d'autres frameworks d'équilibrage de charge comme StarPU, cité dans la littérature, qui se basent sur une répartition de tâches. Notre objectif est d'introduire le parallélisme de don-

nées, et d'améliorer la granularité de StarPU, surtout quand l'algorithme ciblé est représenté par un graphe contenant peu de tâches. L'idée de base est d'appliquer la répartition des tâches que propose ce framework d'une façon itérative sur des petits grains de données au lieu de cibler des images entières. Cette alternative va permettre de chercher un point d'équilibrage de charges optimal en évitant au maximum le coût généré par le framework.

# Bibliographie


[1] L. A. Manwaring, *The Observer's Book of Automobiles*, 13th edition. FREDERICK WARNE, 1966.

[2] worldmeters, *Cars produced in the world - Worldometers*, août 2019. adresse : https://www.worldometers.info/cars/ (visité le 01/09/2019).

[3] P. R. Center, *Methodology – Spring 2018 Global Attitudes Survey: U.S.-German relations*, mars 2019. adresse : https://www.pewresearch.org/global/2019/03/04/u-s-german-relations-methodology-spring-2018-global-attitudes-survey/ (visité le 02/09/2019).

[4] R. Saussard, *TEL - Thèses en ligne - Méthodologies et outils de portage d'algorithmes de traitement d'images sur cibles hardware mixte*, 2017. adresse : https://tel.archives-ouvertes.fr/tel-01587727 (visité le 31/08/2019).

[5] C. Wong, E. Yang, X. Yan et D. Gu, « Adaptive and intelligent navigation of autonomous planetary rovers — A survey », *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, p. 237-244, juill. 2017.

[6] Y. Shen, W. Hu, M. Yang, J. Liu, B. Wei, S. Lucey et C. T. Chou, « Real-Time and Robust Compressive Background Subtraction for Embedded Camera Networks », *IEEE Transactions on Mobile Computing*, t. 15, n° 2, p. 406-418, fév. 2016.

[7] H. Liu, T.-H. Hong, M. Herman, T. Camus et R. Chellappa, « Accuracy vs Efficiency Trade-offs in Optical Flow Algorithms », *Computer Vision and Image Understanding*, t. 72, n° 3, p. 271-286, déc. 1998.

[8] S. Saha, S. Basu, M. Nasipuri et D. K. Basu, « An Offline Technique for Localization of License Plates for Indian Commercial Vehicles », *arXiv:1003.1072 [cs]*, mars 2010.

[9] C. Creusot et A. Munawar, « Real-time small obstacle detection on highways using compressive RBM road reconstruction », *2015 IEEE Intelligent Vehicles Symposium (IV)*, p. 162-167, juin 2015.

[10] O. Bourja, K. Kabbaj, H. Derrouz, A. E. Bouziady, R. O. H. Thami, Y. Zennayi et F. Bourzeix, « MoVITS: Moroccan Video Intelligent Transport System », *2018 IEEE 5th International Congress on Information Science and Technology (CiSt)*, p. 502-507, oct. 2018.

[11] S. Sivaraman et M. M. Trivedi, « Looking at Vehicles on the Road: A Survey of Vision-Based Vehicle Detection, Tracking, and Behavior Analysis », *IEEE Transactions on Intelligent Transportation Systems*, t. 14, n° 4, p. 1773-1795, déc. 2013.

[12] M. Piccardi, « Background subtraction techniques: a review », *2004 IEEE International Conference on Systems, Man and Cybernetics (IEEE Cat. No.04CH37583)*, t. 4, 3099-3104 vol.4, oct. 2004.

[13] C. Stauffer et W. E. L. Grimson, « Adaptive background mixture models for real-time tracking », *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, t. 2, 246-252 Vol. 2, juin 1999.

[14] M. Heredia Conde, « Fundamentals of Compressive Sensing », *Compressive Sensing for the Photonic Mixer Device: Fundamentals, Methods and Results*, M. Heredia Conde, éd., p. 89-205, 2017.

[15] A. Jason Shameem; Roberts, *Multi-Core Programming Increasing Performance through Software Multithreading*. Hillsboro, Or : Intel Corporation, 2006.

[16] L. Mabrouk, D. Houzet, S. Huet, S. Belkouch, A. Hamzaoui et Y. Zennayi, « Single Core SIMD Parallelization of GMM Background Subtraction Algorithm for Vehicles Detection », *2018 IEEE 5th International Congress on Information Science and Technology (CiSt)*, p. 308-312, oct. 2018.

[17] J. Reinders, *Intel® AVX-512 Instructions*, 2013. adresse : `https://software.intel.com/en-us/articles/intel-avx-512-instructions` (visité le 31/08/2019).

[18] Persyval-Team, *Heterogenous Architectures: Versatile Exploitation and programing | Persyval-Lab*, 2019. adresse : `https://persyval-lab.org/en/sites/heaven` (visité le 31/08/2019).

[19] Persyval, *LabEx PERSYVAL-Lab | Persyval-Lab*, 2019. adresse : `https://persyval-lab.org/` (visité le 31/08/2019).

[20] P. Druzhkov et K. Valentina, « A survey of deep learning methods and software tools for image classification and object detection », *Pattern Recognition and Image Analysis*, t. 26, p. 9-15, jan. 2016.

[21] V. Mondéjar-Guerra, J. Rouco, J. Novo et M. Ortega, « An end-to-end deep learning approach for simultaneous background modeling and subtraction », *British Machine Vision Conference (BMVC), Cardiff*, 2019.

[22] Y. Benezeth, P.-M. Jodoin, B. Emile, H. Laurent et C. Rosenberger, « Review and evaluation of commonly-implemented background subtraction algorithms », *2008 19th International Conference on Pattern Recognition*, p. 1-4, 2008.

[23] T. Bouwmans, « Subspace Learning for Background Modeling: A Survey », *Recent Patent On Computer Science*, t. 2, n° 3, p. 223-234, nov. 2009.

[24] T. Bouwmans, « Recent Advanced Statistical Background Modeling for Foreground Detection - A Systematic Survey », *Recent Patents on Computer Science*, t. 4, n° 3, p. 147-176, sept. 2011.

[25] D. Stalin Alex et A. Wahi, « BSFD: Background subtraction frame difference algorithm for moving object detection and extraction », *Journal of Theoretical and Applied Information Technology*, t. 60, p. 623-628, fév. 2014.

[26] B. Lee et M. Hedley, « Background estimation for video surveillance », *Image and amp; Vision Computing New Zealand (IVCNZ '02)*, 2002.

[27] B. Laugraud, S. Piérard, M. Braham et M. Van Droogenbroeck, « Simple Median-Based Method for Stationary Background Generation Using Background Subtraction Algorithms », *New Trends in Image Analysis and Processing – ICIAP 2015 Workshops*, Lecture Notes in Computer Science, V. Murino, E. Puppo, D. Sona, M. Cristani et C. Sansone, éd., p. 477-484, 2015.

[28] N. J. B. McFarlane et C. P. Schofield, « Segmentation and tracking of piglets in images », *Machine Vision and Applications*, t. 8, n° 3, p. 187-193, mai 1995.

[29] J. Zheng, Y. Wang, N. L. Nihan et M. E. Hallenbeck, « Extracting roadway background image: Mode-based approach », *Transportation research record*, t. 1944, n° 1, p. 82-88, 2006.

[30] E. J. Candès, X. Li, Y. Ma et J. Wright, « Robust Principal Component Analysis? », *J. ACM*, t. 58, n° 3, 11:1-11:37, juin 2011.

[31] K. Toyama, J. Krumm, B. Brumitt et B. Meyers, « Wallflower: Principles and practice of background maintenance », *Proceedings of the seventh IEEE international conference on computer vision*, t. 1, p. 255-261, 1999.

[32] S. Messelodi, C. M. Modena, N. Segata et M. Zanin, « A kalman filter based background updating algorithm robust to sharp illumination changes », *International Conference on Image Analysis and Processing*, p. 163-170, 2005.

[33] R. Chang, T. Gandhi et M. M. Trivedi, « Vision modules for a multi-sensory bridge monitoring approach », *Proceedings. The 7th International IEEE Conference on Intelligent Transportation Systems (IEEE Cat. No. 04TH8749)*, p. 971-976, 2004.

[34] M. H. Sigari, N. Mozayani et H. Pourreza, « Fuzzy running average and fuzzy background subtraction: concepts and application », *International Journal of Computer Science and Network Security*, t. 8, n° 2, p. 138-143, 2008.

[35] F. El Baf, T. Bouwmans et B. Vachon, « Type-2 fuzzy mixture of Gaussians model: application to background modeling », *International Symposium on Visual Computing*, p. 772-781, 2008.

[36] H. Zhang et D. Xu, « Fusing color and texture features for background model », *Fuzzy Systems and Knowledge Discovery: Third International Conference, FSKD 2006, Xi'an, China, September 24-28, 2006. Proceedings 3*, p. 887-893, 2006.

[37] F. El Baf, T. Bouwmans et B. Vachon, « Fuzzy integral for moving object detection », *2008 IEEE International Conference on Fuzzy Systems (IEEE World Congress on Computational Intelligence)*, p. 1729-1736, 2008.

[38] D. Manjula et M. Sivabalakrishnan, « Adaptive background subtraction in dynamic environments using fuzzy logic », *Int. J. Video Image Process. Netw. Secur*, t. 10, p. 13-16, 2010.

[39] D. Culibrk, O. Marques, D. Socek, H. Kalva et B. Furht, « Neural Network Approach to Background Modeling for Video Object Segmentation », *IEEE Transactions on Neural Networks*, t. 18, n° 6, p. 1614-1627, nov. 2007.

[40] S. Biswas, J. Sil et N. Sengupta, « Background modeling and implementation using discrete wavelet transform: a review », *Journal ICGST-GVIP*, t. 11, n° 1, p. 29-42, 2011.

[41] Kyungnam Kim, T. H. Chalidabhongse, D. Harwood et L. Davis, « Background modeling and subtraction by codebook construction », *2004 International Conference on Image Processing, 2004. ICIP '04.*, t. 5, 3061-3064 Vol. 5, oct. 2004.

[42] O. Barnich et M. Droogenbroeck, « ViBe: A Universal Background Subtraction Algorithm for Video Sequences », *Image Processing, IEEE Transactions on*, t. 20, p. 1709-1724, juill. 2011.

[43] M. Dikmen et T. S. Huang, « Robust estimation of foreground in surveillance videos by sparse error estimation », *2008 19th International Conference on Pattern Recognition*, p. 1-4, 2008.

[44] J. Lee et M. Park, « An Adaptive Background Subtraction Method Based on Kernel Density Estimation », *Sensors (Basel, Switzerland)*, t. 12, p. 12 279-300, déc. 2012.

[45] D. Butler, S. Sridharan et V. J. Bove, « Real-time adaptive background segmentation », *2003 IEEE International Conference on Acoustics, Speech, and Signal Processing, 2003. Proceedings.(ICASSP'03).*, t. 3, p. III-349, 2003.

[46] C. R. Wren, A. Azarbayejani, T. Darrell et A. P. Pentland, « Pfinder: real-time tracking of the human body », *IEEE Transactions on Pattern Analysis and Machine Intelligence*, t. 19, n° 7, p. 780-785, juill. 1997.

[47] N. Friedman et S. Russell, « Image Segmentation in Video Sequences: A Probabilistic Approach », *Proc. 13th Conf. on Uncertainty in Artificial Intelligence*, fév. 2013.

[48] Hayman et Eklundh, « Statistical background subtraction for a mobile observer », *Proceedings Ninth IEEE International Conference on Computer Vision*, 67-74 vol.1, oct. 2003.

[49] H. Cramér, *Random variables and probability distributions*. Cambridge University Press, 2004, t. 36.

[50] A. Sobral et A. Vacavant, « A comprehensive review of background subtraction algorithms evaluated with synthetic and real videos », *Computer Vision and Image Understanding*, p. 4-21, mai 2014.

[51] G. Szwoch, D. Ellwart et A. Czyżewski, « Parallel implementation of background subtraction algorithms for real-time video processing on a supercomputer platform », *Journal of Real-Time Image Processing*, t. 11, n° 1, p. 111-125, jan. 2016.

[52] T. K. Moon, « The expectation-maximization algorithm », *IEEE Signal processing magazine*, t. 13, n° 6, p. 47-60, 1996.

[53] Z. Zivkovic et F. van der Heijden, « Efficient adaptive density estimation per image pixel for the task of background subtraction », *Pattern Recognition Letters*, t. 27, n° 7, p. 773-780, mai 2006.

[54] J. Cheng, J. Yang, Y. Zhou et Y. Cui, « Flexible background mixture models for foreground segmentation », *Image and Vision Computing*, t. 24, n° 5, p. 473-482, 2006.

[55] D.-S. Lee, « Online adaptive Gaussian mixture learning for video applications », *International Workshop on Statistical Methods in Video Processing*, p. 105-116, 2004.

[56] M. Amintoosi, F. Farbiz, M. Fathy, M. Analoui et N. Mozayani, « QR decomposition-based algorithm for background subtraction », *2007 IEEE International Conference on Acoustics, Speech and Signal Processing-ICASSP'07*, t. 1, p. I-1093, 2007.

[57] Q. Zang et R. Klette, « Evaluation of an adaptive composite Gaussian model in video surveillance », *International Conference on Computer Analysis of Images and Patterns*, p. 165-172, 2003.

[58] J. Lindstrom, F. Lindgren, K. Ltrstrom, J. Holst et U. Holst, « Background and foreground modeling using an online EM algorithm », *The sixth IEEE international workshop on visual surveillance (VS 2006). IEEE*, p. 9-16, 2006.

[59] Y. Ren, C.-S. Chua et Y.-K. Ho, « Motion detection with nonstationary background », *Machine Vision and Applications*, t. 13, n° 5-6, p. 332-343, 2003.

[60] D.-S. Lee, « Improved Adaptive Mixture Learning for Robust Video Background Modeling. », *MVA-Workshop on Machine Vision Applications*, p. 443-446, 2002.

[61] A. Pnevmatikakis et L. Polymenakos, « 2D person tracking using Kalman filtering and adaptive background learning in a feedback loop », *International Evaluation Workshop on Classification of Events, Activities and Relationships*, p. 151-160, 2006.

[62] S.-Y. Yang et C.-T. Hsu, « Background modeling from GMM likelihood combined with spatial and color coherency », *2006 International Conference on Image Processing*, p. 2801-2804, 2006.

[63] J. L. Landabaso et M. Pardas, « Cooperative background modelling using multiple cameras towards human detection in smart-rooms », *2006 14th European Signal Processing Conference*, p. 1-5, 2006.

[64] M. Haque, M. Murshed et M. Paul, « A hybrid object detection technique from dynamic background using Gaussian mixture models », *2008 IEEE 10th Workshop on Multimedia Signal Processing*, p. 915-920, 2008.

[65] X. Fang, W. Xiong, B. Hu et L. Wang, « A moving object detection algorithm based on color information », *Journal of Physics: Conference Series*, t. 48, p. 384, 2006.

[66] H. Bhaskar, L. Mihaylova et S. Maskell, « Automatic target detection based on background modeling using adaptive cluster density estimation », *Gesellschaft für Informatik*, 2007.

[67] M. Xu et T. Ellis, « Illumination-Invariant Motion Detection Using Colour Mixture Models. », *British Machine Vision Conference (BMVC)*, p. 1-10, 2001.

[68] W.-H. Wang et R.-C. Wu, « Fusion of luma and chroma GMMs for HMM-based object detection », *Pacific-Rim Symposium on Image and Video Technology*, p. 573-581, 2006.

[69] O. Javed, K. Shafique et M. Shah, « A hierarchical approach to robust background subtraction using color and gradient information », *Workshop on Motion and Video Computing, 2002. Proceedings.*, p. 22-27, 2002.

[70] Y.-L. Tian et A. Hampapur, « Robust salient motion detection with complex background for real-time video surveillance », *2005 Seventh IEEE Workshops on Applications of Computer Vision (WACV/MOTION'05)-Volume 1*, t. 2, p. 30-35, 2005.

[71] G. Gordon, T. Darrell, M. Harville et J. Woodfill, « Background estimation and removal based on range and color », *Proceedings. 1999 IEEE Computer So-*

*ciety Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*, t. 2, p. 459-464, 1999.

[72] P. Dickinson et A. Hunter, « Scene modelling using an adaptive mixture of Gaussians in colour and space », *IEEE Conference on Advanced Video and Signal Based Surveillance, 2005.*, p. 64-69, 2005.

[73] H. Zheng, Z. Liu et X. Wang, « Research on the video segmentation method with integrated multi-features based on GMM », *2008 International Conference on Computational Intelligence for Modelling Control and Automation*, p. 260-264, 2008.

[74] Z. Zhang, A. J. Lipton, P. L. Venetianer et W. Yin, *Background modeling with feature blocks*. US Patent US8150103B2, avr. 2009.

[75] Y.-L. Tian, M. Lu et A. Hampapur, « Robust and efficient foreground analysis for real-time video surveillance », *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, t. 1, p. 1182-1187, 2005.

[76] A. Mittal, N. Paragios, V. Ramesh et A. Monnet, *Method for scene modeling and change detection*. US Patent US7336803B2, fév. 2008.

[77] L. Meng, W. Wu et Y. Li, *System and method for segmenting foreground and background in a video*. US Patent US8280165B2, oct. 2012.

[78] P. Kumar et K. Sengupta, « Foreground background segmentation using temporal and spatial markov processes », *Department of Electrical and Computer Engineering, National University of Singapore, http://citeseerx.ist.psu.edu*, 2000.

[79] Y. Sun et B. Yuan, « Hierarchical GMM to handle sharp changes in moving object detection », *Electronics Letters*, t. 40, n° 13, p. 801-802, 2004.

[80] Q. Zang et R. Klette, « Robust background subtraction and maintenance », *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, t. 2, p. 90-93, 2004.

[81] T.-M. Su et J.-S. Hu, « Background removal in vision servo system using Gaussian mixture model framework », *IEEE International Conference on Networking, Sensing and Control, 2004*, t. 1, p. 70-75, 2004.

[82] D. H. Parks et S. S. Fels, « Evaluation of background subtraction algorithms with post-processing », *2008 IEEE Fifth International Conference on Advanced Video and Signal Based Surveillance*, p. 192-199, 2008.

[83] J. Zhang et C. H. Chen, « Moving objects detection and segmentation in dynamic video backgrounds », *2007 IEEE Conference on Technologies for Homeland Security*, p. 64-69, 2007.

[84] D. Zhou et H. Zhang, « Modified GMM background modeling and optical flow for detection of moving objects », *2005 IEEE international conference on systems, man and cybernetics*, t. 3, p. 2224-2229, 2005.

[85] P. Jaikumar, A. Singh et S. K. Mitra, « Background Subtraction in Videos using Bayesian Learning with Motion Information. », *British Machine Vision Conference (BMVC)*, t. 2008, p. 615-24, 2008.

[86] A. Shimada et R. Taniguchi, « Object detection based on Gaussian mixture predictive background model under varying illumination », *International Workshop on Computer Vision, MIRU*, 2008.

[87] C.-C. Lien, Y.-M. Jiang et L.-G. Jang, « Large Area Video Surveillance System with Handoff Scheme among Multiple Cameras. », *MVA-Workshop on Machine Vision Applications*, p. 463-466, 2009.

[88] M. Al Najjar, S. Ghosh et M. Bayoumi, « A hybrid adaptive scheme based on selective Gaussian modeling for real-time object detection », *2009 IEEE International Symposium on Circuits and Systems*, p. 936-939, 2009.

[89] S. Xuehua, C. Yu, G. Jianfeng et C. Jingzhu, « A robust moving objects detection algorithm based on Gaussian mixture model », *2009 International Conference on Information Technology and Computer Science*, t. 1, p. 566-569, 2009.

[90] R. YAN, X. SONG et S. YAN, « Moving object detection based on an improved Gaussian mixture background model », *2009 ISECS International Colloquium on Computing, Communication, Control, and Management*, t. 1, p. 12-15, 2009.

[91] D. XIE, « Background model initializing and updating method based on video monitoring », *China Patent CN101489121B*, 2009.

[92] P. K. ATREY, V. KUMAR, A. KUMAR et M. S. KANKANHALLI, « Experiential sampling based foreground/background segmentation for video surveillance », *2006 IEEE International Conference on Multimedia and Expo*, p. 1809-1812, 2006.

[93] D. R. MAGEE, « Tracking multiple vehicles using foreground, background and motion models », *Image and vision Computing*, t. 22, n° 2, p. 143-155, 2004.

[94] R. KRISHNA, K. MCCUSKER et N. E. O'CONNOR, « Optimising resource allocation for background modeling using algorithm switching », *2008 Second ACM/IEEE International Conference on Distributed Smart Cameras*, p. 1-7, 2008.

[95] Y.-H. LIANG, Z.-Y. WANG, X.-W. XU et X.-Y. CAO, « Background Pixel Clissification for Motion Segmentation Using Mean Shift Algorithm », *2007 International Conference on Machine Learning and Cybernetics*, t. 3, p. 1693-1698, 2007.

[96] H. JIANG, H. ARDO et V. OWALL, « Hardware accelerator design for video segmentation with multi-modal background modelling », *2005 IEEE International Symposium on Circuits and Systems*, p. 1142-1145, 2005.

[97] K. APPIAH et A. HUNTER, « A single-chip FPGA implementation of real-time adaptive background model », *Proceedings. 2005 IEEE International Conference on Field-Programmable Technology, 2005.*, p. 95-102, 2005.

[98] J. YANG, X. YUAN, X. LIAO, P. LLULL, D. J. BRADY, G. SAPIRO et L. CARIN, « Video Compressive Sensing Using Gaussian Mixture Models », *IEEE Transactions on Image Processing*, t. 23, n° 11, p. 4863-4878, nov. 2014.

[99] A. Kulkarni et T. Mohsenin, « Accelerating compressive sensing reconstruction OMP algorithm with CPU, GPU, FPGA and domain specific many-core », *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, p. 970-973, mai 2015.

[100] M.-A. Petrovici, C. Damian, C. Udrea, F. Garoi et D. Coltuc, « Single Pixel Camera with Compressive Sensing by non-uniform sampling », *2016 International Conference on Communications (COMM)*, p. 443-448, juin 2016.

[101] M. Liang, Y. Li, M. A. Neifeld et H. Xin, « Principal Component Analysis (PCA) based compressive sensing millimeter wave imaging system », *2015 USNC-URSI Radio Science Meeting*, p. 341-341, juill. 2015.

[102] N. Ponomarenko, V. Lukin, K. Egiazarian et J. Astola, « DCT based high quality image compression », *Scandinavian Conference on Image Analysis*, p. 1177-1185, 2005.

[103] P. G. H. Center, « Pixel-wise Image Processing », *PCI Geomatics Help Center*, 2016. adresse : `http://support.pcigeomatics.com/hc/en-us/articles/207373336-Pixel-wise-Image-Processing-` (visité le 31/08/2019).

[104] P. Rogers, « Heterogeneous system architecture overview », *2013 IEEE Hot Chips 25 Symposium (HCS)*, p. 1-41, août 2013.

[105] Y. Yang, P. Xiang, M. Mantor et H. Zhou, « CPU-assisted GPGPU on fused CPU-GPU architectures », *IEEE International Symposium on High-Performance Comp Architecture*, p. 1-12, fév. 2012.

[106] S. Azmat, L. Wills et S. Wills, « Accelerating Adaptive Background Modeling on Low-Power Integrated GPUs », *2012 41st International Conference on Parallel Processing Workshops*, p. 568-573, sept. 2012.

[107] TOP500, *TOP 10 Sites for November 2019*. adresse : `https://www.top500.org/lists/2019/11/` (visité le 26/11/2019).

[108] MINDFACTORY, *Mindfactory Report December 2019*, 2020. adresse : https://imgur.com/a/Xk2E763#RqoPsmR.

[109] INTEL, *Processeurs Intel® pour serveurs, PC, IoT et appareils mobiles*. adresse : https://www.intel.com/content/www/fr/fr/products/processors.html (visité le 26/11/2019).

[110] WIKIPEDIA, *List of Nvidia graphics processing units*. adresse : https://en.wikipedia.org/w/index.php?title=List_of_Nvidia_graphics_processing_units&oldid=940977129 (visité le 16/02/2020).

[111] MICROWAY, *Comparison Between NVIDIA GeForce and Tesla GPUs*. adresse : https://www.microway.com/knowledge-center-articles/comparison-of-nvidia-geforce-gpus-and-nvidia-tesla-gpus/ (visité le 16/02/2020).

[112] S. MARKIDIS, S. W. DER CHIEN, E. LAURE, I. B. PENG et J. S. VETTER, « Nvidia tensor core programmability, performance & precision », *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, p. 522-531, 2018.

[113] J. SANDERS et E. KANDROT, *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*, 1 edition. Addison-Wesley Professional, juill. 2010.

[114] V. BOULOS, « Adéquation Algorithme Architecture et modèle de programmation pour l'implémentation d'algorithmes de traitement du signal et de l'image sur cluster multi-GPU », thèse de doct., déc. 2012. adresse : https://tel.archives-ouvertes.fr/tel-00876668.

[115] S. WIENKE, P. SPRINGER, C. TERBOVEN et D. a. MEY, « OpenACC — First Experiences with Real-World Applications », *Euro-Par 2012 Parallel Processing*, p. 859-870, août 2012.

[116] J. E. Stone, D. Gohara et G. Shi, « OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems », *Computing in Science Engineering*, t. 12, n° 3, p. 66-73, mai 2010.

[117] R. Membarth, F. Hannig, J. Teich, M. Körner et W. Eckert, « Frameworks for GPU accelerators: A comprehensive evaluation using 2D/3D image registration », *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*, p. 78-81, 2011.

[118] A. Williams et V. B. Escriba, « The Boost Thread library », 2011. adresse : `https://www.boost.org/doc/libs/1_64_0/doc/html/thread.html` (visité le 20/02/2020).

[119] A. Kukanov et M. J. Voss, « The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks. », *Intel Technology Journal*, t. 11, n° 4, 2007.

[120] B. Chapman, G. Jost et R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. The MIT Press, oct. 2007.

[121] S. Chellappa, F. Franchetti et M. Püschel, « How to write fast numerical code: A small introduction », *International Summer School on Generative and Transformational Techniques in Software Engineering*, p. 196-259, 2007.

[122] J. L. Bentley, *Writing efficient programs*. Prentice-Hall, Inc., 1982.

[123] P. Hsieh, « Programming optimization », *Retrieved from Azillion Monkeys*, 2016. adresse : `http://www.azillionmonkeys.com/qed/optimize.html` (visité le 18/02/2020).

[124] GNU-Project, *GCC 5 Release Series — Changes, New Features, and Fixes*. adresse : `https://gcc.gnu.org/gcc-5/changes.html` (visité le 26/11/2019).

[125] X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito et E. Su, « Intel® OpenMP C++/Fortran Compiler for Hyper-Threading Technology: Implementation and Performance. », *Intel Technology Journal*, t. 6, n° 1, 2002.

[126] P. Kaewtrakulpong et R. Bowden, « An Improved Adaptive Background Mixture Model for Realtime Tracking with Shadow Detection », *Proceedings of 2nd European Workshop on Advanced Video-Based Surveillance Systems; September 4, 2001; London, U.K*, mai 2002.

[127] L. Mabrouk, S. Huet, D. Houzet, S. Belkouch, A. Hamzaoui et Y. Zennayi, « Efficient parallelization of GMM background subtraction algorithm on a multicore platform for moving objects detection », *2018 4th International Conference on Advanced Technologies for Signal and Image Processing (ATSIP)*, p. 1-5, mars 2018.

[128] S.-j. Lee et C.-s. Jeong, « Real-time Object Segmentation based on GPU », *2006 International Conference on Computational Intelligence and Security*, t. 1, p. 739-742, nov. 2006.

[129] P. Carr, « GPU Accelerated Multimodal Background Subtraction », *2008 Digital Image Computing: Techniques and Applications*, p. 279-286, déc. 2008.

[130] P. Kumar, A. Singhal, S. Mehta et A. Mittal, « Real-time moving object detection algorithm on high-resolution videos using GPUs », *Journal of Real-Time Image Processing*, t. 11, n° 1, p. 93-109, jan. 2016.

[131] V. Pham, P. Vo, V. T. Hung et al., « GPU implementation of extended gaussian mixture model for background subtraction », *2010 IEEE RIVF International Conference on Computing and Communication Technologies, Research, Innovation, and Vision for the Future (RIVF)*, p. 1-4, 2010.

[132] Y. Li, G. Wang et X. Lin, « Three-level GPU accelerated Gaussian mixture model for background subtraction », *Image Processing: Algorithms and Systems X; and Parallel Processing for Imaging Applications II*, t. 8295, p. 829 514, fév. 2012.

[133] P. Guler, D. Emeksiz, A. Temizel, M. Teke et T. Taskaya Temizel, « Real-time multi-camera video analytics system on GPU », *Journal of Real-Time Image Processing*, t. 11, mars 2013.

[134] C. Chen, N. Zhang, S. Shi et D. Mu, « An efficient method for incremental learning of GMM using CUDA », *2012 International Conference on Computer Science and Service System*, p. 2141-2144, 2012.

[135] B. Liu, W. Qiu, L. Jiang et Z. Gong, « Software pipelining for graphic processing unit acceleration: Partition, scheduling and granularity », *The International Journal of High Performance Computing Applications*, t. 30, n° 2, p. 169-185, mai 2016.

[136] Y. Liang, M. T. Satria, K. Rupnow et D. Chen, « An Accurate GPU Performance Model for Effective Control Flow Divergence Optimization », *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, t. 35, n° 7, p. 1165-1178, juill. 2016.

[137] P. Janus et T. Kryjak, « Hardware implementation of the Gaussian Mixture Model foreground object segmentation algorithm working with ultra-high resolution video stream in real-time », *Signal Processing - Algorithms, Architectures, Arrangements, and Applications (SPA)*, p. 140-145, 2018.

[138] C. Zhang, H. Tabkhi et G. Schirner, « A GPU-Based Algorithm-Specific Optimization for High-Performance Background Subtraction », *2014 43rd International Conference on Parallel Processing*, p. 182-191, sept. 2014.

[139] X. Ye et W. Wan, « Fast background modeling using GMM on GPU », *2014 International Conference on Audio, Language and Image Processing*, p. 937-941, juill. 2014.

[140] A. Nurhadiyatna, R. Wijayanti et D. Fryantoni, « Extended Gaussian Mixture Model Enhanced by Hole Filling Algorithm (GMMHF) Utilize GPU Acceleration », *Information Science and Applications (ICISA) 2016*, Lecture Notes in Electrical Engineering, K. J. Kim et N. Joukov, éd., p. 459-469, 2016.

[141] P. Kovačev, M. Mišić et M. Tomašević, « Parallelization of the Mixture of Gaussians Model for Motion Detection on the GPU », *2018 Zooming Innovation in Consumer Technologies Conference (ZINC)*, p. 58-61, mai 2018.

[142] R. K et N. N. Chiplunkar, « A survey on techniques for cooperative CPU-GPU computing », *Sustainable Computing: Informatics and Systems*, t. 19, p. 72-85, sept. 2018.

[143] S. Mittal et J. S. Vetter, « A survey of CPU-GPU heterogeneous computing techniques », *ACM Computing Surveys (CSUR)*, t. 47, n° 4, p. 69, 2015.

[144] E. Stafford, B. Pérez, J. L. Bosque, R. Beivide et M. Valero, « To Distribute or Not to Distribute: The Question of Load Balancing for Performance or Energy », *Euro-Par 2017: Parallel Processing*, Lecture Notes in Computer Science, F. F. Rivera, T. F. Pena et J. C. Cabaleiro, éd., p. 710-722, 2017.

[145] F. Zhang, J. Zhai, B. He, S. Zhang et W. Chen, « Understanding Co-Running Behaviors on Integrated CPU/GPU Architectures », *IEEE Transactions on Parallel and Distributed Systems*, t. 28, n° 3, p. 905-918, mars 2017.

[146] D. Grewe, Z. Wang et M. F. P. O'Boyle, « OpenCL Task Partitioning in the Presence of GPU Contention », *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, C. Caşcaval et P. Montesinos, éd., p. 87-101, 2014.

[147] J. Shen, M. Arntzen, A. Varbanescu, H. Sips et D. Simons, « A framework for accelerating imbalanced applications on heterogeneous platforms », *Computing Frontiers*, p. 14-16, 2013.

[148] A. K. Singh, A. Prakash, K. R. Basireddy, G. V. Merrett et B. M. Al-Hashimi, « Energy-Efficient Run-Time Mapping and Thread Partitioning of Concurrent OpenCL Applications on CPU-GPU MPSoCs », *ACM Trans. Embed. Comput. Syst.*, t. 16, n° 5s, 147:1-147:22, sept. 2017.

[149] G. Teodoro, T. Pan, T. M. Kurc, J. Kong, L. A. Cooper et J. H. Saltz, « Efficient irregular wavefront propagation algorithms on hybrid CPU–GPU machines », *Parallel computing*, t. 39, n° 4-5, p. 189-211, 2013.

[150] A. Navarro, F. Corbera, A. Rodriguez, A. Vilches et R. Asenjo, « Heterogeneous parallel_for Template for CPU–GPU Chips », *International Journal of Parallel Programming*, jan. 2018.

[151] A. Vilches, R. Asenjo, A. Navarro, F. Corbera, R. Gran et M. Garzaran, « Adaptive Partitioning for Irregular Applications on Heterogeneous CPU-GPU Chips », *Procedia Computer Science*, International Conference On Computational Science, ICCS 2015, t. 51, p. 140-149, jan. 2015.

[152] H.-F. Li, T.-Y. Liang et Y.-J. Lin, « An OpenMP programming toolkit for hybrid CPU/GPU clusters based on software unified memory », *Journal of Information Science and Engineering*, t. 32, p. 517-539, mai 2016.

[153] L. Li, R. Geda, A. B. Hayes, Y. Chen, P. Chaudhari, E. Z. Zhang et M. Szegedy, « A Simple Yet Effective Balanced Edge Partition Model for Parallel Computing », *Proceedings of the 2017 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '17 Abstracts, p. 6-6, 2017.

[154] C. Augonnet, S. Thibault, R. Namyst et P.-A. Wacrenier, « StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures », *Euro-Par 2009 Parallel Processing*, Lecture Notes in Computer Science, H. Sips, D. Epema et H.-X. Lin, éd., p. 863-874, 2009.

[155] Intel, *Intel Intrinsics Guide*. adresse : `https://software.intel.com/sites/landingpage/IntrinsicsGuide/` (visité le 26/11/2019).

[156] S. Blair-Chappell, *Intel Compiler Labs: The significance of SIMD, SSE and AVX*. adresse : `https://www.scribd.com/document/321776325/3a-SIMD` (visité le 26/11/2019).

[157] G. M. Amdahl, « Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, NJ, Apr. 18–20), AFIPS Press, Reston, Va., 1967, pp.

483–485, when Dr. Amdahl was at International Business Machines Corporation, Sunnyvale, California », *IEEE Solid-State Circuits Society Newsletter*, t. 12, n° 3, p. 19-20, 2007.

[158] L. Mabrouk, S. Huet, S. Belkouch, D. Houzet, Y. Zennayi et A. Hamzaoui, « Performance and Scalability Improvement of GMM Background Segmentation Algorithm on Multi-core Parallel Platforms », *Proceedings of the 1st International Conference on Electronic Engineering and Renewable Energy*, p. 120-127, avr. 2018.

[159] L. Mabrouk, S. Huet, D. Houzet, S. Belkouch, A. Hamzaoui et Y. Zennayi, « Efficient adaptive load balancing approach for compressive background subtraction algorithm on heterogeneous CPU–GPU platforms », *Journal of Real-Time Image Processing*, sept. 2019.

[160] Y. Wang, P.-M. Jodoin, F. Porikli, J. Konrad, Y. Benezeth et P. Ishwar, « CDnet 2014: an expanded change detection benchmark dataset », *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, p. 387-394, 2014. adresse : `www.changedetection.net`.

[161] D. M. Powers, « Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation », *J. Mach. Learn. Technol*, p. 2229-3981, 2011.

[162] N. D. Blog, *GPU Pro Tip: CUDA 7 Streams Simplify Concurrency*, jan. 2015. adresse : `https://devblogs.nvidia.com/gpu-pro-tip-cuda-7-streams-simplify-concurrency/` (visité le 31/08/2019).

# Annexes

## A.1 Projet HEAVEN

Les architectures informatiques deviennent de plus en plus complexes, exposant un parallélisme massif, des mémoires organisées hiérarchiquement et des unités de traitement hétérogènes. De telles architectures sont extrêmement difficiles à programmer car, la plupart du temps, les programmeurs d'applications choisissent entre la portabilité et les performances. Alors que les environnements de programmation standard comme OpenMP évoluent actuellement pour prendre en charge l'exécution d'applications sur différents types d'unités de traitement, ces approches souffrent de deux problèmes principaux. Premièrement, pour exploiter des unités de traitement hétérogènes au niveau de l'application, les programmeurs doivent traiter explicitement des mécanismes de bas niveau spécifiques au matériel, tels que les transferts de mémoire entre la mémoire hôte et les mémoires privées d'un coprocesseur par exemple. Deuxièmement, comme l'évolution des environnements de programmation vers une programmation hétérogène se concentre principalement sur les plateformes CPU / GPU, certains accélérateurs matériels sont encore difficiles à exploiter à partir d'une application parallèle à usage général. L'FPGA en fait partie. Contrairement aux CPUs et aux GPU, cet accélérateur matériel peut être configuré pour répondre aux besoins de l'application. Il contient des tableaux de blocs logiques programmables qui peuvent être câblés ensemble pour construire un circuit spécialisé pour l'application ciblée. Par exemple, les FPGAs peuvent être configurés pour accélérer des portions de code qui sont connues pour mal fonctionner sur les CPU ou les GPU. L'efficacité énergétique des FPGA est également l'un des principaux atouts de ce type d'accélérateurs par rapport aux GPU, ce qui encourage la communauté scientifique à considérer les FPGA comme l'un des éléments constitutifs des plateformes multi cœurs hétérogènes à faible puissance et à grande échelle. Cependant, seule une fraction de la communauté envisage de programmer des FPGA pour l'instant, car les configurations doivent être conçues à l'aide de langages

de description de bas niveau tels que le VHDL que les programmeurs d'applications ne connaissent pas. Des membres des laboratoires de recherche LIG, TIMA et GIPSA-Lab sont impliqués dans ce projet. Son objectif principal de ce projet est d'améliorer l'accessibilité des architectures hétérogènes contenant des accélérateurs FPGA aux programmeurs d'applications parallèles. Le projet proposé se concentre sur trois aspects principaux :

— Portabilité : nous ne voulons pas que les programmeurs d'applications repensent complètement leurs applications pour tirer parti des périphériques FPGA. Cela signifie d'étendre les environnements de programmation parallèle standard comme OpenMP pour prendre en charge FPGA. Améliorer la portabilité des applications signifie également tirer parti de la plupart des mécanismes de bas niveau spécifiques au matériel au niveau du système d'exécution.

— Performance : nous voulons que notre solution soit suffisamment flexible pour tirer le meilleur parti de toutes les plateformes hétérogènes contenant des appareils FPGA en fonction des besoins de performance spécifiques, comme le débit de calcul ou la consommation d'énergie par exemple ;

— Expériences : Expérimenter avec des accélérateurs FPGA sur des applications scientifiques réelles est également un élément clé de notre proposition de projet. En particulier, les solutions développées dans ce projet permettront des comparaisons entre architectures sur des applications réelles de différents domaines comme le traitement du signal.

Une programmation efficace et l'exploitation d'architectures hétérogènes impliquent le développement de méthodes et d'outils de conception de systèmes, embarqués ou non. La proposition de projet HEAVEN s'inscrit dans l'action de recherche PCS du PERSYVAL-lab.

## A.2 Projet MoVITS

Au Maroc, les routes sont meurtrières et la tendance des accidents de la route est à la hausse. Ceci s'est confirmé par le bilan du CNPAC affirmant que le nombre des tués s'est élevé à 3593 en 2016, contre 3565 en 2015. La première cause des accidents dans le pays est le non-respect des règles du code de la route par ses usagers.

Dans le but de réduire le nombre d'accidents annuelle, un grand nombre de différents systèmes ont été étudiés. Il y a des systèmes qui font partie de l'infrastructure routière (signalisation horizontale et verticale, panneaux à messages variables, capteurs à boucle magnétique, radars, etc.) ou de véhicules routiers (divers systèmes de soutien à la conduite : les équipementiers automobiles ont mis en place des systèmes tels que la détection de la voie et la ligne de départ, les assistants d'aide au stationnement, évitement de collision, régulateur de vitesse adaptatif, etc). Malgré l'avancé technologique des véhicules et le développement des infrastructures routières, la situation reste alarmante puisque le besoin principal est de trouver une solution efficace permettant de changer le comportement des conducteurs et de les rendre beaucoup plus prudent sur la route.

Aujourd'hui, la mesure et la gestion des systèmes de circulation dans le domaine des ITS, utilisent le plus souvent des algorithmes de vision par ordinateur avec des caméras vidéo. Les caméras sont utilisées comme une partie de l'infrastructure de la route dans le but d'extraire à partir de la séquence vidéo obtenue, des informations sur le trafic routier de haut niveau en temps réel avec une haute précision. Ces informations doivent être analysée, en mesurant les propriétés de l'image afin de comprendre le flux vidéo à savoir la détection des incidents, la classification des véhicules, la détection des infractions et des anomalies, etc. L'avantage de la vision par ordinateur est qu'elle est précise, et permet de détecter n'importe quel type d'objet (voiture, piéton, cycliste, etc.), le système est facilement installé sans interruption du trafic et le coût de maintenance est faible, ainsi que le coût d'achat dépend de la qualité des caméras à installer.

Le but de ce projet nommé MoVITS (Moroccan Video Intelligent Transport System), est de développer un système intégré de gestion de trafic et de détection d'infractions de circulations routière. Son architecture est divisée en 3 couches (Fig. A.15.), chacune présente un ensemble des fonctionnalités différentes :

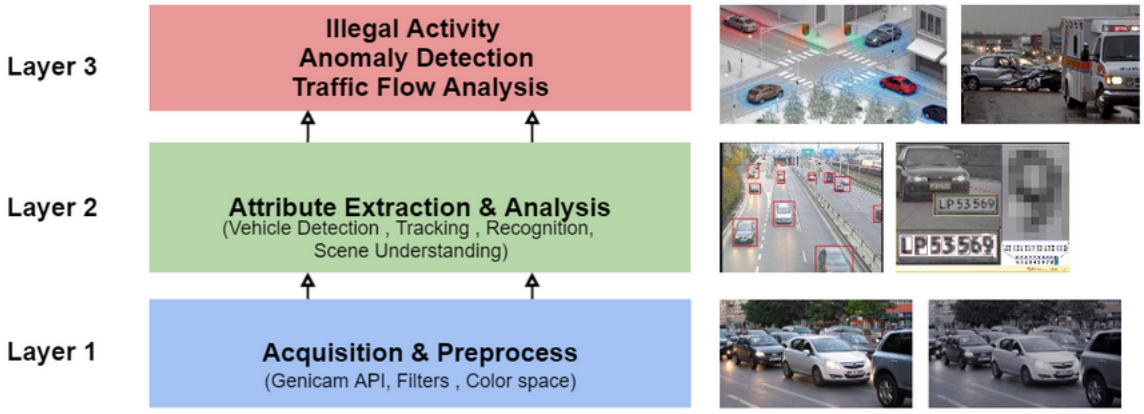


FIGURE A.15 – Couches fonctionnelles principales du projet MOVITS

### A.2.1 Acquisition et prétraitement (couche 1)

La fonction de cette couche est d'assurer l'acquisition des images à partir d'un système stéréoscopique calibré et synchronisé à l'aide d'une API Genicam. Ces images passent par une phase de prétraitement qui permet de changer l'espace de couleur, d'appliquer des filtres pour éliminer les bruits etc.

### A.2.2 Extraction et analyse d'attributs (couche 2)

Sur la base des images obtenues, cette couche est utilisée pour extraire les attributs statiques et dynamiques des véhicules nécessaires pour gérer le trafic : la détection des véhicules, extraction des trajectoires, les identifiés (plaque d'immatriculation, logo, couleur etc.), extraction des éléments des signalisations verticales et horizontales dans la scène. Dans cette couche s'effectue l'analyse des attributs, comprendre les comportements et enfin percevoir l'état du trafic.

### A.2.3 Activité illégale, analyse des flux de trafic et détection d'anomalies (couche 3)

Sur la base de sorties des deux couches précédentes, cette couche fournit des services ITS pour la gestion efficace et le contrôle de la route. Elle a une fonctionnalité décisionnelle. Elle permet de détecter les infractions routières (tel que non-respect du stop/ feu rouge, excès de vitesse, fausse plaque d'immatriculation, changement de direction non autorisé...) et les anomalies sur la route (accident, feu rouge en panne etc.). Ensuite une phase d'analyse du flux trafic est mise en place pour gérer le trafic localement et globalement.

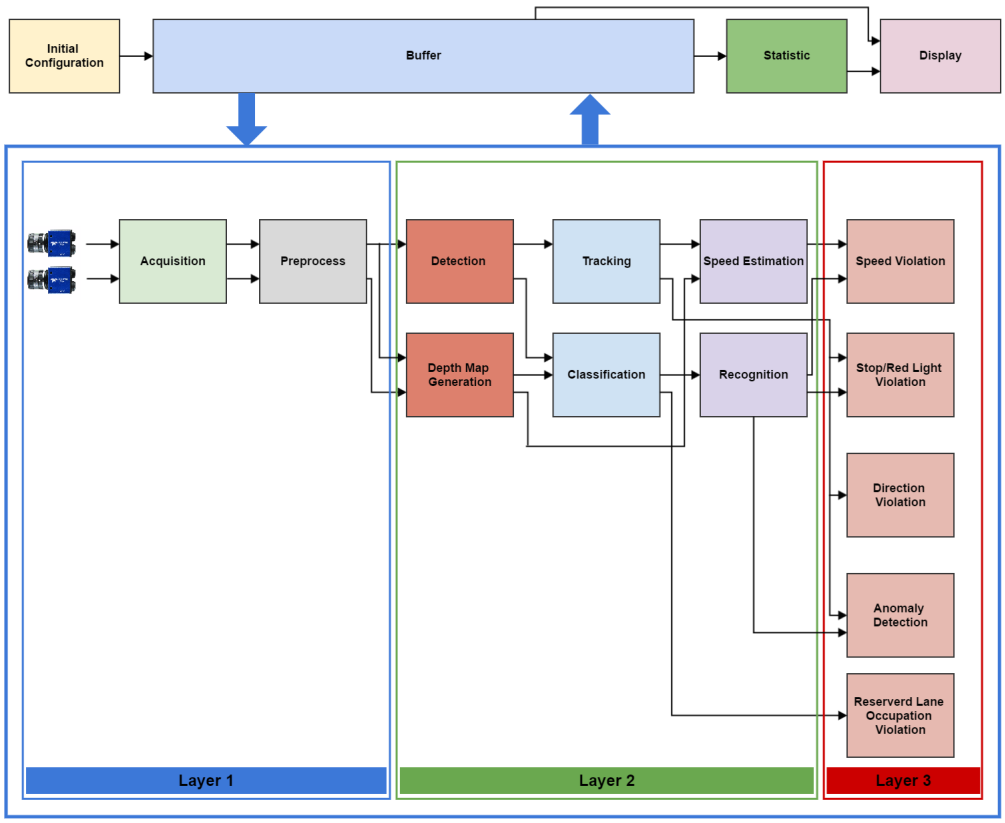


FIGURE A.16 – Architecture modulaire des trois couches algorithmiques du projet Movits [10].

Chacune de ces trois couches est composée de plusieurs modules. La Fig. A.16 représente un schéma qui résume cette architecture modulaire. Plus de détails sur ce système

peuvent être trouvés dans [10].

### A.2.4 Organigramme des partenaires

L'organigramme global des partenaires des deux projets HEAVEN et MoViTS est présenté sur la Fig. A.17. Plus de détails sur ces deux projets sont présentés dans l'annexe A.1 et A.2.


FIGURE A.17 – Présentation des différents partenaires et intervenants dans les projets MoViTS et HEAVEN.

## A.3 Familles de microprocesseurs

| **Famille** | **Fabriquant** | **Jeu d'inst.** | **Arch. (bits)** | **Sous-familles** |
|---|---|---|---|---|
| 4000 | Intel | CISC | 4 | 40**xx** |
| <8086 | Intel | CISC | 8 | 80**xx** |
| Zilog | Zilog | CISC | 8, 16, 32 | Z80, Z8000, Z80000 |
| 6800 | Motorola | CISC | 8 | 68**xx** |
| 65xx | MOS Technology | CISC | 8, 16 | 65**xx** |
| 68000 | Motorola | CISC | 16, 32 | 68**xxx**, ColdFire, DragonBall |
| 88000 | Motorola | RISC | 32 | 88100 |
| I | Intel | RISC | 32 | i860, i960 |
| x86 | IBM, Intel, AMD, cYRIX | CISC | 8, 16, 32, 64 | >8085, Pentium, Celeron, Xeon, Core, Core 2, Core i3, Core i5, Core i7, AMD K5, AMD K6, Duron,Sempron, AthlonOpteron, Turion, Haipad |
| Itanium ( IA-64) | Intel | EPIC | 64 | Itanium, Itanium 2 |
| Crusoe | transmetta | VLIW | 128, 256 | Crusoe, Efficeon |
| POWER | IBM | RISC | 32, 64 | POWER**x** |
| PowerPC | IBM et Motorola | RISC | 32, 64 | PowerPC **xxx** |
| SPARC | Sun | RISC | 32, 64 | Sun Sparc, SuperSparc, MicroSparc, HyperSPARC, UltraSPARC, LEON |
| ARM | ARM, Intel et TI | RISC | 32, 64 | ARM**x**, ARMv**x**-Cortex-**xx**, StrongARM, XScale |
| Mips | MIPS | RISC | 32, 64 | R**x**000 |
| DEC | DEC, Compaq, HP | RISC | 64 | Alpha 21**xxx** |
| PA | HP | RISC | 32, 64 | PA 8**xxx** |
| SuperH | Hitashi | RISC | 32 | Sh**x** |
| MCore | Freescale | RISC | 32 | MMC2**xxx** |

Table A.7 – Familles de CPU les plus connus.

# A.4 Techniques d'optimisation implicite d'instructions

## A.4.1 Pipelining

En informatique, le pipelining d'instruction est une technique pour implémenter le parallélisme au niveau de l'instruction dans un seul processeur. Le pipeline tente de garder chaque partie du processeur occupée par certaines instructions en divisant les instructions entrantes en une série d'étapes séquentielles (le "pipeline" éponyme) exécutées par différentes unités de processeur avec différentes parties d'instructions traitées en parallèle. Le pipeline RISC classique comprend 5 phases (Fig. A.18.) :

| Clock cycle / Instr. No. | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | IF | ID | EX | MEM | WB | | |
| 2 | | IF | ID | EX | MEM | WB | |
| 3 | | | IF | ID | EX | MEM | WB |
| 4 | | | | IF | ID | EX | MEM |
| 5 | | | | | IF | ID | EX |

FIGURE A.18 – Pipeline de base en cinq étapes

1. IF (Instruction Fetch) : Récupération de l'instruction
2. ID (Instruction decode) : Décodage de l'instruction
3. EX (Execution) : Exécution
4. MEM (Memory access) : Accès mémoire
5. WB (Register write back) : Réécrire le registre

## A.4.2 Prédiction de branchements

La prédiction de branche est une caractéristique d'un processeur qui lui permet de prédire le résultat d'une branche. Cette technique permet à un processeur de rendre l'uti-

lisation de son pipeline plus efficace. Avec cette technique, le processeur effectuera une exécution spéculative : il pariera sur le résultat d'un branchement, et poursuivra l'exécution du programme avec le résultat du pari. Si le pari échoue, les instructions chargées par erreur dans le pipeline sont annulées.

### A.4.3 Exécution spéculative

L'exécution spéculative est une technique d'optimisation dans laquelle un système informatique effectue une tâche qui peut ne pas être nécessaire. Le travail est effectué avant que l'on sache s'il est réellement nécessaire, de manière à éviter un retard qui devrait être encouru en effectuant le travail après que l'on sait qu'il est nécessaire. S'il s'avère que le travail n'était pas nécessaire après tout, la plupart des modifications apportées par le travail sont annulées et les résultats sont ignorés.

L'objectif est de fournir plus de simultanéité si des ressources supplémentaires sont disponibles. Cette approche est utilisée dans une variété de domaines, y compris la prédiction de branche dans les processeurs pipelinés. En effet, les microprocesseurs modernes utilisant un pipeline se servent de l'exécution spéculative pour réduire le coût des instructions de branchement conditionnel. Quand une instruction de branchement conditionnel est rencontrée, le processeur devine quelle voie du branchement le programme est susceptible de suivre (c'est ce que l'on appelle la prédiction de branchement), et commence à exécuter les instructions correspondantes. Si la supposition s'avère être incorrecte, tous les calculs qui ont eu lieu après le branchement sont rejetés. L'exécution prématurée est relativement peu coûteuse du fait que, sans celle-ci, les étages du pipeline impliqués dans le calcul seraient restés inactifs jusqu'à ce que le résultat du branchement soit connu. Toutefois, les instructions gaspillées utilisent des cycles d'horloge, et sur un ordinateur portable par exemple, ces cycles consomment de l'énergie.

### A.4.4 Renommage de registres

En architecture des ordinateurs, on appelle renommage de registres le fait qu'une microarchitecture alloue dynamiquement les registres architecturaux à un ensemble plus vaste de registres physiques au cours de l'exécution d'un programme. Dans une microarchitecture superscalaire, le processeur essaie d'exécuter plusieurs instructions en parallèle. Il analyse donc localement le programme afin de réduire les dépendances entre instructions, et de les réorganiser en conséquence, afin de profiter du parallélisme sans introduire d'erreur. Puisque les dépendances entre instructions machine limitent les performances de l'exécution dans le désordre, car il arrive souvent que plusieurs instructions essayent d'utiliser le même registre en raison du parallélisme qui a été introduit. Une des instructions doit alors être bloquée en attendant que la ressource soit disponible. Cependant, dans de nombreux cas, ces dépendances n'apparaissent qu'au niveau du registre, mais ne nécessitent pas de réelles dépendances dans le flux de données traité par le programme. Ce problème est d'ailleurs d'autant plus prégnant que le compilateur a effectué des optimisations basées sur l'utilisation des registres.

Une solution consiste donc à dupliquer les ressources : les registres architecturaux ne correspondent plus à des registres physiques dans la microarchitecture, mais sont alloués dynamiquement à un ensemble plus grand de registres physiques, ce qui permet d'éliminer une partie des dépendances introduites artificiellement par le nombre restreint de registres. Par exemple, l'architecture IA-32 définit 16 registres architecturaux. Le Pentium dispose de 128 registres physiques et effectue du renommage de registres. la Fig. A.19. illustre un exemple de renommage de registre pour une séquence d'instructions.

### A.4.5 Exécution dans le désordre

L'exécution dans le désordre (ou l'exécution dynamique) est un paradigme utilisé dans la plupart des unités centrales de traitement hautes performances pour exploiter des cycles d'instruction qui seraient autrement gaspillés. Dans ce paradigme, un processeur exécute des instructions dans un ordre régi par la disponibilité des données d'entrée et

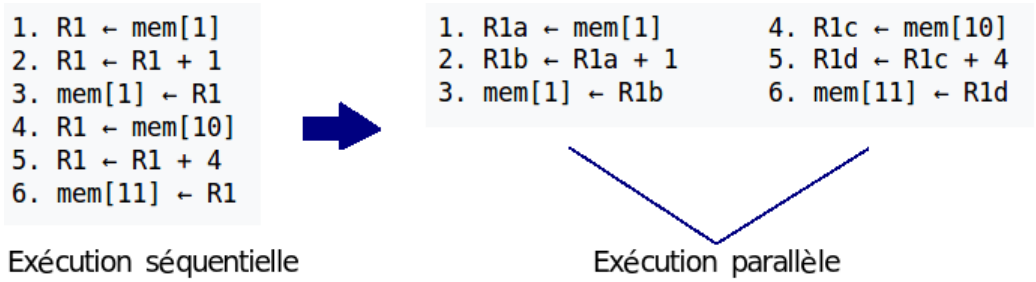


FIGURE A.19 – Renommage de registre pour une séquence d'instructions

des unités d'exécution plutôt que par leur ordre d'origine dans un programme. Ainsi, le processeur peut éviter d'être inactif en attendant la fin de l'instruction précédente et peut, dans l'intervalle, traiter les instructions suivantes qui peuvent s'exécuter immédiatement et indépendamment.