Adéquation algorithme architecture pour flot optique sur GPU embarqué

Thomas Romera

Thèse présentée pour l'obtention du grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Spécialité

Ingénierie / Systèmes Informatiques

École doctorale

Informatique, Télécommunication et Électronique Paris (ED130)

Adéquation algorithme architecture pour flot optique sur GPU embarqué

Thomas Romera

Soutenue publiquement le : *13 octobre 2023*

Devant un jury composé de :

David Defour	Professeur, LAMPS, Université de Perpignan Via Domitia	*Rapporteur*
Claude Tadonki	Chargé de Recherche, CRI, Mines ParisTech	*Rapporteur*
Roselyne Chotin	Maître de Conférences, LIP6, Sorbonne Université	*Examinatrice*
Olivier Sentieys	Professeur, IRISA, INRIA, Université de Rennes	*Président*
Daniel Etiemble	Professeur Émérite, LRI, Université Paris-Saclay	*Invité*
Patrice Menard	Directeur Technique, LERITY-Alcen	*Invité*
Lionel Lacassagne	Professeur, LIP6, Sorbonne Université	*Directeur de thèse*
Quentin Meunier	Maître de Conférences, LIP6, Sorbonne Université	*Co-encadrant de thèse*

Remerciements

Je voudrais tout d'abord remercier les membres de mon jury pour l'intérêt qu'ils ont porté à ces travaux. J'adresse mes plus sincères remerciements à David Defour, Professeur des Universités à l'Université de Perpignan Via Domitia, et Claude Tadonki, Chargé de Recherche au CRI à Mines ParisTech, pour avoir accepté d'être les rapporteurs de cette thèse et d'avoir consacré de leur temps pour la lecture et l'évaluation de mon manuscrit. J'exprime également ma gratitude envers Roselyne Chotin, Maître de Conférences à Sorbonne Université, et Olivier Sentieys, Professeur des Universités à l'Unversité de Rennes, d'avoir accepté de faire partie de mon jury en tant qu'examinateurs. Je remercie enfin Daniel Etiemble, Professeur Émérite à l'Université Paris-Saclay d'avoir accepté mon invitation pour assister à ma soutenance de thèse.

Je veux ensuite remercier mon directeur de thèse Lionel Lacassagne. Certains évènements n'ont pas rendu le déroulement de cette thèse des plus simple. Lionel m'a épaulé et m'a motivé pour mener mes travaux jusqu'au bout. En plus de ses savoirs techniques, sa détermination me guide depuis le début de cette thèse, et depuis bien avant en master. Ses précieux conseils m'ont permis de construire la forme tant bien que la substance de ces travaux de thèse. Je le remercie pour sa patience et pour son soutient et pour les moments passés tant au laboratoire qu'en dehors. Je tiens également à remercier Quentin Meunier, mon co-encadrant de thèse. Il fut un point de repère qui m'a aidé à avancer au mieux durant les périodes d'instabilité de cette thèse. Ses qualités de relecteur et son aide précieuse durant la période de rédaction ont été cruciales pour l'achèvement du manuscrit. Enfin, je veux remercier LERITY, en particulier Patrice Menard, directeur technique et innovation et Pascal Dupuy, ancien président directeur général, pour m'avoir offert l'opportunité de réaliser ma thèse CIFRE au sein de leur entreprise. Ce fut pour moi une première expérience professionnelle que j'ai beaucoup appréciée, tant sur le plan humain que sur le plan technique.

Ensuite, je tiens à remercier ma famille qui ne m'a pas beaucoup vu en ces derniers temps, mais qui a toujours été présente. J'embrasse ma mère Danielle, mon père Manuel, ma sœur Apolline, mon frère Léo, ainsi que la famille plus éloignée en Alsace.

Un grand merci à tous mes amis que je n'ai malheureusement pas pu voir suffisamment pour les mêmes raisons, mais qui ont toujours su me remonter le moral. Un clin d'œil à Benjamin, Camille, Cédric, Charlotte, Katrin, Laura, Paul, Remi et Thomas. Merci aussi à tous mes collègues du LIP6 qui eux m'ont un peu trop vu, tant dans la fac que hors de la fac. Je pense à ceux de la barre 24-25 : Adrien, Alan, Ali, Arthur, Clara, Florian, Ilias, Jonathan, Mathuran, Maxime, Nathan, Ning, Rieul, Spyros et Theofilos. Je pense aussi à ceux de la barre 26-00 : Aymeric, Etienne, Hugo, Ilyas, Jonathan, Saalik, Vincent et sans l'oublier bien sûr, Baptiste. Sans vous tous, la fin de ma thèse aurait été moins agréable. Enfin, une pensée spéciale pour les collègues actuels ou anciens du bureau de Lerity : Abdelmounim, Andrea et Borris.

Abstract

Over the past two decades, commercial cameras have seen major advances in image and video quality, mainly thanks to technological progress in various components such as optics, digital storage, image stabilization, circuitry and digital sensors. The most notable advances have been in digital light sensors. To further improve the quality of camera images, innovations in image processing and computer vision are needed.

One of the major algorithmic blocks in this field is the estimation of pixel movement in a video, also known as optical flow. This block adds temporal information between frames of a video sequence that can be used to stabilize, denoise, unblur or increase the resolution. Most optical flow estimation algorithms are very effective in terms of quality, but their high processing time limits their real-time implementation on embedded platforms.

The work carried out in this thesis focuses on the optimization and efficient implementation of optical flow estimation algorithms on embedded graphics processors. Two iterative algorithms have been studied : the TV-L^1 estimation method and the Horn-Schunck estimation method. The primary objective of this work is to achieve real-time processing (less than 40 ms per frame) on low-power platforms, while maintaining acceptable image resolution and flow estimation quality for the intended applications.

Various optimization strategies have been explored. High-level algorithmic transformations, such as operator fusion and pipelining, have been implemented to maximize data reuse and enhance spatial and temporal locality. Additionally, GPU-specific low-level optimizations, including the utilization of vector instructions and numbers, as well as efficient memory access management, have been incorporated. The impact of floating-point number representation (single-precision versus half-precision) has also been investigated.

The implementations have been assessed in terms of execution time, power consumption, and optical flow accuracy. In addition to acceleration enabling real-time processing of near-4K resolution images on embedded platforms, the use of half-precision numbers delivers higher-quality results in the same amount of time compared to single-precision implementations.

This work has highlighted the importance of GPU-specific optimizations for computer vision algorithms, as well as the use of floating-point numbers with reduced precision. To the best of our knowledge, this work is a first concrete example of how reducing the precision of floating-point numbers can lead to higher-quality results.

Keywords : Embedded GPU, Iterative Stencils, Computer Vision, Optical Flow, Energy Consumption, Half-precision Computation

Résumé

Depuis deux décennies, les caméras commerciales ont connu d'importantes avancées en termes de qualité d'image et de vidéo, principalement grâce à des progrès technologiques dans divers composants tels que l'optique, le stockage numérique, la stabilisation de l'image, les circuits et les capteurs numériques. C'est le capteur numérique de lumière qui a connu les avancées les plus notables. Pour aller plus loin dans l'amélioration de la qualité des images des caméras, des innovations dans le domaine du traitement d'images et de la vision par ordinateur sont nécessaires.

L'un des blocs algorithmiques majeurs dans ce domaine est l'estimation du mouvement des pixels dans un flux vidéo, encore appelé flot optique. Ce bloc permet d'ajouter des informations temporelles entre les images d'une séquence vidéo et peuvent être notamment utilisées pour stabiliser, débruiter, déflouter ou encore augmenter la résolution. La plupart des algorithmiques d'estimation du flot optique sont très performants en termes de qualité mais leur temps de traitement trop élevé limite leur implémentation temps réel sur des plateformes embarquées.

Les travaux menés dans cette thèse portent sur l'optimisation et l'implémentation efficace d'algorithmes d'estimation du flot optique sur des processeurs graphiques embarqués. Deux algorithmes itératifs ont été étudiés : la méthode d'estimation TV-L[1] et la méthode d'estimation de Horn-Schunck. L'objectif est d'obtenir un traitement temps réel (moins de 40 ms par image) sur des plateformes embarquées à faible consommation énergétique, tout en gardant une résolution d'image et une qualité d'estimation du flot acceptable pour les applications visées.

Différentes stratégies d'optimisation ont été explorées. Des transformations algorithmiques de haut niveau, telles que la fusion et le pipeline d'opérateurs, ont été mises en œuvre pour maximiser la réutilisation des données et améliorer la localité spatiale et temporelle. De plus, des optimisations de bas niveau spécifiques aux GPU, notamment l'utilisation d'instructions et de nombres vectoriels, ainsi qu'une gestion efficace de l'accès à la mémoire, ont été intégrées. Enfin, l'impact de la représentation des nombres en virgule flottante (simple précision par rapport à demi-précision) a également été étudié.

Les implémentations ont été évaluées en termes de temps d'exécution, de consommation énergétique et de qualité du flot optique. En plus d'une accélération permettant le traitement temps réel d'images proches de la résolution 4K sur des plateformes embarquées, l'utilisation de nombres en demi-précision permet d'obtenir des résultats de meilleure qualité dans le même laps de temps par rapport à des implémentations en simple-précision.

Ces travaux ont souligné l'importance des optimisations spécifiques aux GPU pour les algorithmes de vision par ordinateur, ainsi que l'utilisation de nombres à virgule flottante de précision réduite. À notre connaissance, ces travaux constituent un premier exemple concret démontrant que la réduction de la précision des nombres flottants peut conduire à des résultats de meilleure qualité.

Mots-clés : GPU Embarqué, Stencils Itératifs, Vision par Ordinateur, Flot Optique, Consommation d'Énergie, Calcul en Demi-Précision

Table des matières

1 Introduction

Depuis 20 ans, les caméras commerciales ont connu des augmentations majeures en termes de qualité image et vidéo. Cela est dû à de multiples avancées technologiques des composants (optiques, stockage numérique, stabilisation de l'image, circuits et capteurs numériques...). Néanmoins, c'est le capteur numérique de lumière qui a connu les avancées les plus notables. Il existe aujourd'hui deux grandes familles de technologie de capteurs : la technologie *CCD* (« *Charge-Coupled Device* » ou dispositif à transfert de charges en français) et la technologie *CMOS* (« *Complementary Metal–Oxide–Semiconductor* » ou semiconducteur d'oxyde de métal complémentaire en français).

La technologie CCD est la plus vieille des deux technologies. Inventée en 1969 par George E. Smith et Willard Boyle, cette technologie se base sur une transformation des photons en paires électron-trou par effet photoélectrique. Ces charges sont ensuite transférées de photosite en photosite jusqu'à un registre de sortie où elles sont lues en tension de manière sérielle. Cette technologie présente plusieurs contraintes [Fos93] :

— elle nécessite un transfert de charges quasiment parfait pour obtenir une image la moins bruitée possible ;
— il est difficile d'utiliser cette technologie dans des conditions de faible luminosité ou température ;
— il est difficile de fabriquer des capteurs CCD de grande taille ;
— elle s'intègre mal avec le reste de la carte électronique ;
— il est difficile d'obtenir des cadences d'image élevées.

La technologie de capteurs CMOS s'est développée en parallèle de celle des CCD à mesure que la technologie des semi-conducteurs a évoluée. De par les limitations des technologies de microlithographie des années 70, les premiers capteurs CMOS étaient bien plus grands, avaient un temps de lecture long et produisaient une image plus bruitée que les capteurs CCD contemporains. Depuis les années 90 cependant, le processus de fabrication CMOS est devenu bien établi et stable et a ainsi permis l'utilisation de cette technologie pour la fabrication de capteurs concrets [Ren+90]. Un capteur CMOS est aujourd'hui plus rapide, moins bruité et plus efficace énergétiquement qu'un capteur CCD équivalent. Les améliorations

constantes de cette technologie ont ainsi permis aux CMOS de dépasser les CCD en termes de popularité depuis le début des années 2010. Trois grands domaines d'amélioration sont à noter.

La première et l'une des améliorations les plus notables concerne la diminution du bruit de lecture. Il s'agit du bruit introduit par la conversion entre les photons captés par le capteur et le signal électrique émis et ensuite traité par une carte électronique pour représenter une image. L'utilisation de technologies de capteurs comme *APS* (« *Active-Pixel Sensor* » en anglais) permet de diminuer ce bruit de lecture en diminuant fortement le bruit lié à la bande passante : plus de canaux sont disponibles, ce qui permet d'augmenter le temps de lecture d'un pixel individuel et en conséquence de diminuer le bruit de lecture [Koz+98].

La plus grande accessibilité à cette technologie permet ainsi aux capteurs commerciaux d'atteindre des niveaux de lecture de l'ordre du photon/électron [Big+06]. Grâce à leurs excellentes propriétés de lecture, ce type de capteurs est utilisé dans de nombreux contextes commerciaux et scientifiques, notamment en physique des particules.

La deuxième amélioration concerne l'augmentation du rendement quantique des capteurs. Il s'agit ici de capter le plus de photons possible arrivant en entrée du capteur. Les capteurs ont vu leur facteur de remplissage en éléments photosensibles (« *fill factor* ») augmenter considérablement et avoisiner aujourd'hui le 100% [Int+15]. Cela est dû à la diminution de la taille des électrodes des photodétecteurs et plus récemment à l'utilisation de capteurs CMOS illuminés par l'arrière – « *Backside Illuminated* » (BSI) – premièrement introduit par Sony en 2009 et présenté en figure 1.1. L'utilisation de microlentilles permet également l'augmentation du facteur de remplissage effectif en faisant converger la lumière vers le photodétecteur. Ces améliorations permettent ainsi une augmentation du rendement quantique jusqu'à 85% pour des capteurs commerciaux, comme le capteur Sony IMX183 [FLI19]. Des capteurs scientifiques atteignent des rendements encore supérieurs. Le capteur BSI DAVIS développé à l'ETH de Zürich atteint ainsi un rendement quantique de 93% [Tav+18] comparé à la précédente version du capteur en illumination frontale – « *Front-Side Illuminated* » (FSI) – ayant un rendement de 24%.

Enfin, nous pouvons également citer les avancées faites dans la diminution du courant d'obscurité. Ce courant parasite du photodétecteur en absence d'éclairement lumineux peut être diminué notamment grâce à l'utilisation de la technologie de photodiode *PPD* (« *pinned photodiode* ») ou encore l'utilisation du silicium noir fortement appauvri (« *deep depleted* »). Des capteurs à silicium complètement appauvri sont ainsi considérés dans des applications nécessitant des temps de lecture très court

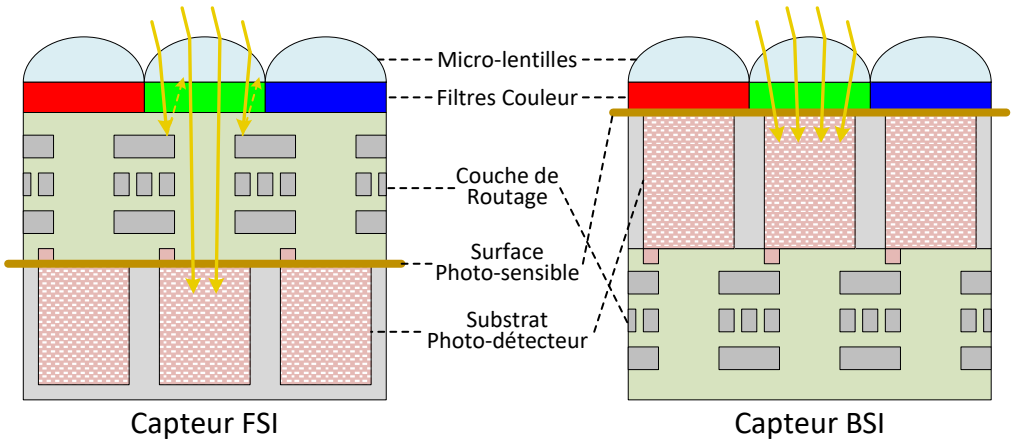


Fig. 1.1 : Différences des empilements de couches entre un capteur illuminé par l'avant (FSI) et un capteur illuminé par l'arrière (BSI).

comme pour le détecteur à haute luminosité du Grand collisionneur de hadrons (LHC) du CERN [Kug19].

Pour pouvoir encore accroître la qualité de restitution image des caméras, d'autres pistes sont à explorer. C'est dans ce contexte de besoin d'innovation dans le domaine du traitement d'images et de la vision par ordinateur que se positionne LERITY.

LERITY (anciennement LHERITIER) est un acteur du domaine de l'optronique, spécialisé dans les systèmes de vision haute définition dans le spectre visible, opérant notamment en condition de visibilité dégradée. Les produits conçus par LERITY incluent déjà certains traitements d'amélioration de la qualité vidéo. Ces améliorations se situent en général au plus proche du capteur via des modules électroniques et des modules de traitement FPGA. Inclure des traitements plus lourds au sein des caméras est complexe et onéreux en raison des restrictions imposées par le matériel embarqué, telles que la puissance de calcul, la consommation électrique, le coût, la dimension et la fragilité. Pour réaliser ces améliorations plus conséquentes, des modules déportés sont utilisés. Ils prennent en entrée le flux vidéo de la caméra, réalisent le traitement en temps réel et restituent le flux en sortie.

Le projet de module *VIRTANS* (pour « VIdeo Real-Time Algorithm Noise Suppression ») est une solution de ce type proposée par LERITY. Initié dans le cadre de la thèse d'Andrea Petreto intitulée « Débruitage Vidéo Temps Réel pour Systèmes Embarqués » [Pet20], ce module présenté en figure 1.2 contient une plateforme embarquée capable de réaliser un traitement de débruitage CPU sur un flot vidéo SDI entrant. Ce module est capable de débruiter un flux vidéo très basse lumière en temps réel sur des images 480×270 pixels tout en se limitant à une puissance

Fig. 1.2 : Prototype VIRTANS : VIdeo Real-Time Algorithm : Noise Suppression.

électrique d'une dizaine de watts. La chaîne de traitement RTE-VD développée pour ce projet présente un compromis « qualité/vitesse » intéressant pour les applications de vision et de débruitage en bas niveau de lumière. Alors que les algorithmes de débruitage de l'état de l'art VBM3D [DFE07] et VBM4D [Mag+12] produisent des résultats qualitativement meilleurs, RTE-VD est respectivement 200 et 4600 fois plus rapide pour un *PSNR* inférieur de seulement 2,5 décibels.

Par rapport un algorithme plus rapide comme STMKF [Pfl+19] qui est 2,2 fois plus rapide, RTE-VD présente un gain qualitatif de débruitage de 4 décibels. Un démonstrateur a été développé pour prouver la faisabilité d'une telle solution [Pet+20]. Les résultats du démonstrateur mettent en avant les bonnes performances et le bon compromis entre le temps d'exécution et la qualité de débruitage de RTE-VD. Cependant, même sur une plateforme de calcul plus puissante que le module VIRTANS (ici la plateforme embarquée Jetson Xavier de Nvidia), la résolution vidéo reste encore limitée à 960×540 pixels (un quart de la résolution full-HD) pour une cadence image « temps réel » de 25 images par secondes (soit 40 ms par image). En particulier, l'implémentation de cette chaîne n'utilise pas toutes les ressources de calculs à disposition du programmeur présentes sur la carte. Un gain de performance est possible en utilisant ces ressources supplémentaires. Mais, plus important, y a-t-il à faire les mêmes compromis entre le temps d'exécution et la qualité de traitement que sur le CPU ? L'utilisation de ces ressources supplémentaires peut aussi entraîner une consommation énergétique supplémentaire.

Ces questions nous ont poussés à explorer en détails les unités d'accélération des calculs présentes sur les cartes embarquées utilisées jusqu'à présent. Nous nous sommes ainsi concentrés sur le complexe de processeurs graphiques (GPU) embarqué.

Des travaux connexes ont déjà pu montrer l'intérêt des GPU par rapport aux CPU dans le cadre notamment de l'étiquetage en composantes connexes [Mau+22b ; Mau+22a ; LML22 ; HL19 ; HML19 ; CLE15 ; CL14 ; LZ09]. Par rapport au CPU, cet accélérateur de calcul contient plus de $100\times$ de cœurs de calcul et dépasse ainsi les performances en gigaflops du CPU. L'utilisation de ce complexe nous permet-il de gagner en performance temporelle et qualitative, mais aussi en efficacité de consommation électrique ? Notre but est ainsi de réaliser l'implémentation du principal goulot d'étranglement de la chaîne de traitement RTE-VD. Il s'agit de la partie réalisant une estimation du mouvement apparent des pixels, autrement appelé flot optique. Cette partie correspond à 97% du temps de traitement de la chaîne. Il est crucial de réaliser une implémentation efficace de cette dernière qui tire parti de toutes les ressources de la carte.

1.1 Plan de la thèse

Nous présentons dans le chapitre 2 le contexte ainsi que les notions fondamentales nécessaires à la compréhension des travaux de cette thèse. Nous exposons les algorithmes d'estimation du mouvement en vision par ordinateur, explorons les architectures parallèles (surtout GPU vs CPU), présentons les systèmes de représentation numérique avec leurs avantages et inconvénients, et concluons par une problématique détaillée pour guider les travaux de la thèse.

Nous présentons dans le chapitre 3 un état de l'art des algorithmes d'estimation du mouvement de pixels ainsi qu'un état de l'art des architectures parallèles embarquées et des optimisations GPU utilisées dans nos travaux.

Nous illustrons dans le chapitre 4 nos choix d'algorithmes d'estimation du mouvement ainsi que les contributions de cette thèse, à savoir les implémentations optimisées des algorithmes d'estimation du mouvement.

Nous présentons dans le chapitre 5 notre protocole expérimental, nos résultats et notre comparaison des différentes versions implémentées.

Enfin, nous concluons dans le chapitre 6 en répondant aux questions posées en chapitre 2 et en présentant les pistes d'améliorations et les travaux futurs à mener à la suite de cette thèse.

2 Problématique

Nous présentons dans ce chapitre les notions nécessaires à la définition de la problématique et des questions auxquelles cette thèse cherche à répondre. Dans le cadre de cette thèse, nous allons donc nous intéresser aux problèmes et obstacles liés à une implémentation efficace, sur plateforme embarquée, d'algorithmes du domaine de la vision par ordinateur. Ces algorithmes sont plus particulièrement des algorithmes d'estimation du mouvement individuel des pixels dans une vidéo, appelé flot optique. La première partie de ce chapitre consiste en une présentation de cette famille d'algorithmes. Elle constitue l'un des blocs principaux de nombreux algorithmes de traitement d'image et vidéo. Les algorithmes restituant un flot fidèle sont en général complexes et gourmands en temps et en énergie.

Dans le cadre de cette thèse, ces algorithmes sont confrontés à de fortes contraintes de temps d'exécution et de consommation électrique. Ces algorithmes ont pour but de fonctionner conjointement à des caméras portables sans alimentation en courant externe, c'est-à-dire en fonctionnant sur batterie. Dans ce contexte mêlant un besoin de puissance de calcul et une limitation en consommation énergétique, le choix a été fait d'utiliser des architectures parallèles se basant sur des processeurs graphiques embarqués. Nous allons donc présenter dans la deuxième partie de ce chapitre les principales notions de la programmation sur processeur graphique ou GPU.

Enfin, nous allons considérer les effets de la représentation des nombres sur les performances de nos implémentations. Les architectures considérées contiennent des unités de calcul spécialisées en fonction du format des nombres utilisés dans l'implémentation des algorithmes. Ces unités respectent la révision de 2008 du standard arithmétique IEEE 754 et utilisent des nombres de taille variable 16 bits, 32 bits et 64 bits. Les performances de ces unités diffèrent en termes de temps d'exécution et en précision des calculs. Il est nécessaire de qualifier cette précision de calcul sur la qualité du flot optique à estimer. Cette qualification nous permettra ainsi de choisir le type ayant le meilleur compromis entre le temps de calcul et la qualité du flot estimé. La troisième partie de ce chapitre va donc présenter les différentes façons de représenter les nombres dans nos architectures ainsi que leurs avantages et inconvénients.

2.1 Estimation du mouvement dans la vision par ordinateur

La vision par ordinateur est le domaine informatique qui traite de la manière dont les ordinateurs utilisent des images ou des vidéos numériques pour en extraire des informations haut-niveau. Ces informations haut-niveau sont utilisées pour décrire le monde réel représenté par les scènes capturées numériquement. En particulier, on cherche à décrire les propriétés des objets présents dans ces scènes comme leur forme, leur luminosité, leur couleur, leur déplacement, leur nombre, etc. Retrouver ces informations est un problème complexe : on cherche à résoudre des problèmes mal posés définis par des systèmes ayant un trop grand nombre d'inconnues. Il est alors nécessaire de recourir à des modèles physiques et probabilistes afin de réduire l'indétermination du problème initial. Ce domaine regroupe ainsi à la fois les méthodes d'acquisition, mais aussi les méthodes de modélisation, de traitement et d'analyse des images numériques.

De nos jours, la vision par ordinateur est omniprésente en raison de l'utilisation massive de caméras, téléphones, véhicules autonomes, drones, robots, missiles, etc. Les applications sont nombreuses et variées, à la fois dans les domaines scientifiques [JHG99] (vélocimétrie de particules, aide au diagnostic médical, thermographie...) et industriels [Low15] (reconnaissance optique des caractères, inspection automatisée des machines industrielles, voitures sans chauffeur...). Les algorithmes utilisés sont en général complexes et nécessitent de grands temps de traitement et une consommation énergétique conséquente. Ces mêmes algorithmes peuvent également être confrontés à des contraintes de traitement en temps réel et de consommation électrique restreinte, notamment dans le domaine des systèmes embarqués [Cio+23c ; Kan+23c]. Dans ce domaine embarqué, des compromis et des mises en œuvre efficaces sont cruciaux pour que les algorithmes qui sont initialement lents et consommateurs d'énergie puissent fonctionner sur des plateformes ayant de fortes restrictions. Ce sous-domaine de la vision par ordinateur constitue l'un des blocs de base de nombreuses applications industrielles. Les sous-parties suivantes ont pour but de présenter ce sous-domaine ainsi que les considérations à prendre dans le cadre d'une application industrielle embarquée et temps réel.

2.1.1 Introduction à l'estimation du mouvement

Un problème récurrent en vision par ordinateur est l'estimation du mouvement apparent des pixels entre deux images d'un flux vidéo, autrement appelé flot optique.

Ce concept a été introduit par James Gibson en 1950 [S51] dans le domaine de la psychologie et de la perception visuelle. Les premiers algorithmes pour l'estimation du flot optique furent introduits en 1981 par Berthold Horn et Brian Schunck [HS81] ainsi que par Bruce Lucas et Takeo Kanade [LK+81]. La première méthode se base sur la régularisation d'une fonction décrivant l'évolution de la luminosité des images au cours du temps. La deuxième utilise la méthode des moindres carrés pour estimer le flot optique dans des petites régions des images. Cette estimation du mouvement des pixels est utilisée dans de multiples applications scientifiques et industrielles. Nous pouvons citer en exemple :

— l'estimation de la vitesse d'objets dans une vidéo. Il s'agit d'une des applications les plus directes du flot optique : l'estimation du mouvement des pixels sert à estimer la vitesse réelle des objets dans la vidéo [KPM19 ; Cha+11] ;

— la segmentation d'image. Le flot optique est ici utilisé pour différencier les régions dans une vidéo : les groupes de pixels ayant des vecteurs vitesses similaires appartiennent à la même région [KAB15 ; Kla+09] ;

— la détection et le suivi d'objets dans une vidéo. Comme pour la segmentation, le flot optique est utilisé pour différencier des objets et des régions d'intérêt en mouvement par rapport au fond statique. Le flot optique peut ainsi être utilisé pour de la détection de météores dans le ciel [Ram+21] ;

— la reconnaissance d'actions et d'activités humaines. Il s'agit ici de différencier et d'identifier différentes activités et actions humaines grâce à des informations temporelles tirées des mouvements entre les images d'une vidéo [Sun+18b ; GIK10] ;

— la compression vidéo. Le flot optique permet de reconstruire des images interpolées dans une séquence vidéo [Rip+19 ; WKC94] ;

— l'amélioration de la qualité vidéo. Le flot optique permet d'apporter des informations temporelles qui peuvent être utilisées pour stabiliser [Den+20], débruiter [Pet+19], déflouter [HM15] ou encore améliorer la résolution [LFE23 ; Wro+19] d'une séquence vidéo.

Ces applications ont cependant des besoins différents en termes de qualité de flot optique et de temps de traitement. Des compromis entre le temps de calcul, la précision du flot optique et la consommation d'énergie pour les applications embarquées sont nécessaires. Il est ainsi crucial de choisir la technique d'estimation du flot optique la plus adaptée à la fois à la plateforme matérielle sous-jacente, mais aussi à l'application recherchée.

2.1.2 Différentes techniques, différentes performances

L'un des travaux de références concernant la nomenclature des algorithmes de flot optique est l'article de John Barron de 1994 [BFB94]. Dans ce papier, quatre grandes catégories d'algorithmes de flot optiques sont identifiées :

— les techniques « différentielles » ;
— les techniques « régionales » de corrélation de régions dans les images ;
— les techniques « fréquentielles » ;
— les techniques se basant sur la « corrélation de phases » des fréquences de l'image.

De nombreux algorithmes sont présents dans chaque catégorie. La base de données Middlebury [Bak+11] est l'une des bases de données de référence pour les algorithmes de flot optique. Les bases de données Sintel [But+12] et KITTI [GLU12 ; MG15] sont également des bases de données populaires. Ces bases comprennent chacune entre 200 et 300 publications et implémentations différentes. Les principaux critères de comparaison entre les algorithmes dans ces bases sont qualitatifs : on se base sur des métriques comme l'erreur quadratique moyenne point-à-point ou l'erreur angulaire moyenne entre le flot estimé par l'algorithme et une vérité terrain afin de classer les algorithmes. Il faut noter que la plupart des articles fournissent très peu d'informations sur le temps d'exécution ou encore la consommation électrique des plateformes sur lesquelles fonctionnent ces algorithmes. La qualité du flot optique est en effet considérée comme la principale – sinon la seule – métrique. Pourtant, la plupart de ces algorithmes sont itératifs et peuvent prendre beaucoup de temps pour converger. Les algorithmes d'apprentissage automatique représentent le deuxième grand type d'algorithmes dans ces bases et peuvent également nécessiter beaucoup de temps tant pour l'apprentissage que pour l'inférence. Un choix judicieux est donc à faire quant à la sélection d'un algorithme d'estimation du flot optique. Nous détaillerons ainsi dans le chapitre 4 les algorithmes ayant été retenus dans le cadre de nos travaux.

2.2 Architectures parallèles : le cas des GPU

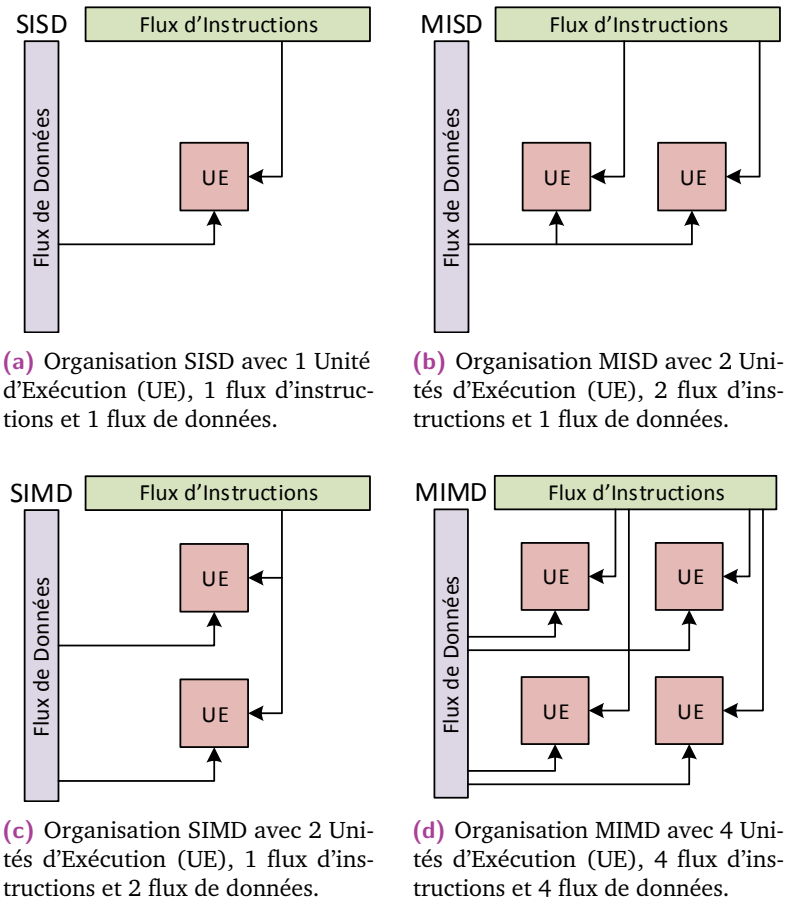
L'introduction du processeur POWER4 d'IBM en 2001 marqua un tournant dans le paradigme de conception des processeurs commerciaux. Des problèmes de consommation électrique, de dissipation thermique ainsi qu'une trop grande complexification des processeurs mono-cœur ont poussé les fabricants de processeurs à se tourner vers des architectures parallèles multi-coeurs. Le rapport technique de 2006 de Berkeley résume cela par la formule « Power Wall + Memory Wall + ILP Wall = Brick Wall » [Asa+06]. L'intérêt de telles architectures est d'augmenter le nombre d'instructions, de calculs ou de processus capables d'être exécutés simultanément. Les problèmes informatiques pouvant être divisés en sous-problèmes plus petits et résolus en même temps peuvent alors tirer parti de telles architectures.

Il existe plusieurs classifications et taxonomies des architectures parallèles. L'une des plus citées est la taxonomie de Flynn introduite en 1966 [Fly66] et complétée en 1972 [Fly72]. Cette classification se base sur l'existence de deux types de flots en entrée d'un processeur : un flot d'instructions correspondant à la suite d'instructions exécutées par le processeur et un flot de données correspondant à la suite de données sur lesquelles sont exécutées ces instructions. Le parallélisme opère sur ces deux flots séparément, selon que l'on est capable de traiter plusieurs instructions simultanément (parallélisme d'instructions ou *ILP – Instruction Level Parallelism* – en anglais) ou capable de traiter plusieurs données simultanément (parallélisme de données ou *DLP – Data Level Parallelism* – en anglais). Il en résulte ainsi quatre grandes catégories d'architectures parallèles présentées en figure 2.1 :

— La catégorie *SISD* (Single Instruction, Single Data), un seul flot d'instructions, un seul flot de données. Cela correspond aux architectures mono-processeur qui peuvent cependant exploiter le parallélisme d'instructions au sein du même flot d'instructions.

— La catégorie *SIMD* (Single Instruction, Multiple Data), un seul flot d'instructions, plusieurs flots de données. La même instruction est exécutée sur plusieurs flots de données en parallèle. De multiples unités d'exécutions sont chargées de traiter ces flots et possèdent chacune leur propre mémoire de données. Il n'y a cependant qu'une seule mémoire d'instructions et une seule unité de contrôle chargée de lire et de lancer les instructions. Ce modèle exploite le parallélisme de données.

— La catégorie *MISD* (Multiple Instruction, Single Data), plusieurs flots d'instructions, un seul flot de données. Il n'existe aujourd'hui pas de processeurs commerciaux de ce type. Certaines architectures spécialisées s'en rapprochent, notamment dans les domaines de tolérances aux pannes et de la redondance.



(a) Organisation SISD avec 1 Unité d'Exécution (UE), 1 flux d'instructions et 1 flux de données.

(b) Organisation MISD avec 2 Unités d'Exécution (UE), 2 flux d'instructions et 1 flux de données.

(c) Organisation SIMD avec 2 Unités d'Exécution (UE), 1 flux d'instructions et 2 flux de données.

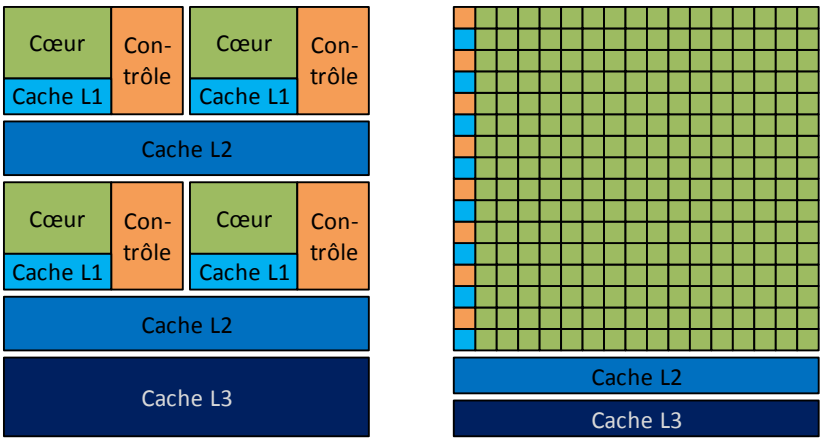
(d) Organisation MIMD avec 4 Unités d'Exécution (UE), 4 flux d'instructions et 4 flux de données.

Fig. 2.1 : Les 4 organisations architecturales de la taxonomie de Flynn.

— La catégorie *MIMD* (Multiple Instruction, Multiple Data), plusieurs flots d'instructions, plusieurs flots de données. Dans cette catégorie, chaque unité est chargée de lire et d'exécuter leurs propres instructions sur leurs propres données. Les unités peuvent être étroitement liées (cœurs sur une même puce, processeur dans un même système) ou plus librement liées (grappe de serveurs, centre de données).

Cette taxonomie exclut les architectures incluant uniquement des mécanismes bas-niveau de parallélisme comme le pipeline d'instructions, le super-scalaire, l'exécution out-of-order ou encore les processeurs VLIW (« *Very Long Instruction Word* ». Cette différentiation est nécessaire pour éviter de qualifier la quasi-totalité des architectures actuelles comme des architectures parallèles. Bien que les mécanismes cités précédemment permettent d'augmenter notablement les performances des processeurs, ils n'offrent pas à eux seuls la capacité de programmer explicitement de manière parallèle, que ce soit pour les instructions ou pour les données.

2.2.1 Architecture des GPU

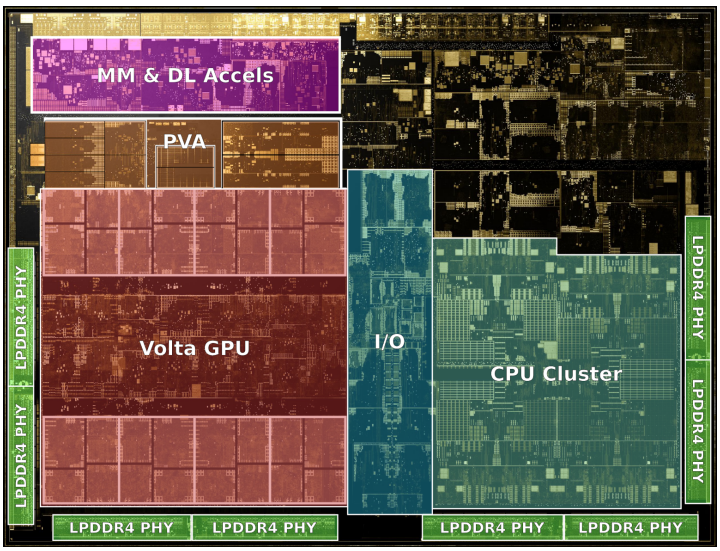


(a) Organisation d'un CPU type. **(b)** Organisation d'un GPU type.

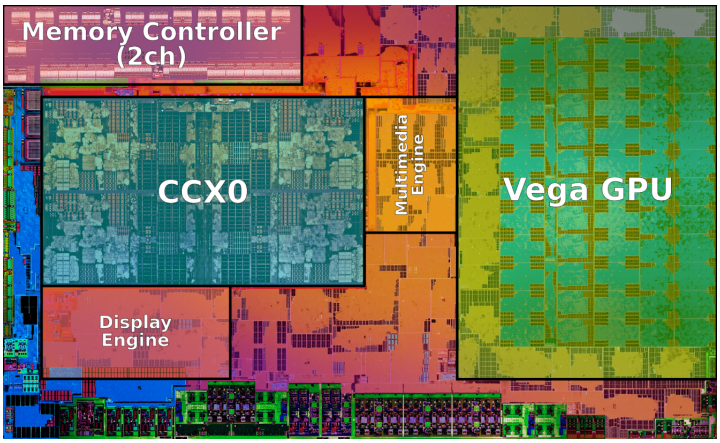
Fig. 2.2 : Schémas fonctionnels de l'organisation d'un CPU multi-cœur et d'un GPU moderne.

Dans le cadre de nos travaux, nous nous sommes intéressés à des architectures GPU embarquées modernes. Par rapport aux CPU, ce type d'architecture implémente les classes de parallélisme SIMD et MIMD de façon bien plus soutenue. Alors que les CPU aujourd'hui comportent typiquement entre 4 et 8 cœurs, voire jusqu'à 64 pour des processeurs haut de gamme, les GPU comportent plusieurs centaines de cœurs de calcul, voire plusieurs milliers et dizaines de milliers pour les GPU hautes performances. Les CPU sont des processeurs généralistes misant sur une réduction de la latence des instructions d'un programme pour gagner en performance. La majeure partie de l'espace des puces CPU n'est d'ailleurs pas dédié au calcul, mais à la mémoire cache. Les GPU eux sont des processeurs annexes spécialisés misant sur une optimisation du débit des données traitées en parallèle pour gagner en performance. La surface de puce dédiée aux calculs est ainsi largement supérieure à celle dédiée à la mémoire cache chez les GPU.

À titre d'exemple, nous avons en figure 2.3 deux microphotographies d'un « System-on-Chip » (SoC) NVIDIA Tegra Xavier et d'un SoC AMD Raven Ridge. Dans le cas du SoC Xavier, le complexe CPU comporte 8 cœurs de calculs et un total de $8 \times$ (64 Ko) $+ 4 \times$ (2048 Ko) $+$ 4096 Ko $=$ 12,5 Mo de cache de données L1, L2 et L3. La surface de ce complexe est d'environ 60 mm^2. Le complexe GPU comporte lui 512 cœurs et un total de $8 \times$ (128 Ko) $+$ 512 Ko $=$ 1,5 Mo de cache de données L1 et L2. La surface de ce complexe est d'environ 90 mm^2. Dans le cas du SoC Raven Ridge, le complexe CPU comporte 4 cœurs de calculs et un total de $4 \times$ (32 Ko) $+ 4 \times$ (512 Ko) $+$ 4096 Ko $=$ 6,13 Mo de cache de données L1, L2 et L3. La surface de ce complexe est d'environ 40 mm^2. Le complexe GPU comporte lui



(a) CPU (désigné par « CPU Cluster ») et GPU (désigné par « Volta GPU ») du SoC NVIDIA Xavier. Le complexe CPU fait environ 60 mm^2 pour 8 cœurs de calculs et le complexe GPU 90 mm^2 pour 512 cœurs de calcul. La surface totale de la puce nue est d'environ 350 mm^2



(b) CPU (désigné par « CCX0 ») et GPU (désigné par « Vega GPU ») du SoC AMD Raven Ridge. Le complexe CPU fait environ 40 mm^2 pour 4 cœurs de calculs et le complexe GPU 65 mm^2 pour 704 cœurs de calcul. La surface totale de la puce nue est d'environ 210 mm^2.

Fig. 2.3 : Microphotographies de deux SoC CPU-GPU NVIDIA Tegra Xavier et AMD Raven Ridge.

704 cœurs de calcul et un total de $11 \times (16\text{ Ko}) + 4096\text{ Ko} = 4{,}17$ Mo de cache de données L1 et L2. La surface de ce complexe est d'environ 65 mm^2. Dans ces deux exemples, le nombre de cœurs de calculs est environ multiplié par 100 en passant du CPU au GPU alors que la surface n'est elle que multipliée par 1,5. Cette plus petite surface CPU contient également plus de mémoire cache de données que la surface GPU.

Les quantités de mémoire disponibles ainsi que leur bande passante par cœur de calcul est également l'une des différences majeures entre les CPU et les GPU. Comme dit en début de section, les GPU sont conçus pour maximiser les débits, à la fois instruction et mémoire. Les cœurs de calculs et les mémoires sont plus petits et plus simples du côté GPU.

Dans la suite de cette section, nous allons présenter le paradigme de programmation associé aux GPU modernes. Nous allons prendre en exemple les GPU de NVIDIA. Ces GPU correspondent à plus de 78% des GPU dédiés non-intégrés aux CPU au premier trimestre 2022. Ils sont très répandus et utilisés tant par le grand public que par les milieux professionnels et industriels, et sont disponibles en tant que carte dédiée ou bien sont embarqués sur un SoC intégré avec un CPU. Il est important de comprendre à la fois l'organisation architecturale physique du GPU ainsi que l'organisation hiérarchique liée à la façon de les programmer. Dans le chapitre suivant, nous nous intéresserons plus précisément à l'organisation physique précise des GPU.

2.2.2 Paradigme de programmation

CUDA [NVI21] (à l'origine un acronyme pour « Compute Unified Device Architecture ») est à la fois le modèle de programmation et l'API de NVIDIA pour le domaine du GPGPU (« General-purpose Processing on Graphics Processing Units ») – le calcul générique sur processeur graphique. D'autres plateformes et modèles de programmation sont disponibles comme notamment OpenCL [Gro20] de Khronos Group, le standard OpenACC [Org22] ou encore l'API OpenMP [Boa21]. OpenACC et OpenMP se basent sur des directives de transformation de code qui sont appliquées par le compilateur. L'intérêt de ces deux modèles a d'ailleurs été démontré dans des implémentations efficaces d'algorithmes d'estimation du flot optique [Hag+19a ; Hag+19b]. Il existe cependant des différences en termes de performances. Les implémentations OpenCL peuvent être jusqu'à 60% plus lentes que les implémentations CUDA [KDH10] et peuvent nécessiter en moyenne $2\times$ plus de lignes codes que CUDA [Mem+17]. Ces considérations ainsi que l'existence d'un riche écosystème de développement à la fois local sur les cartes de développement, mais également en ligne nous ont finalement poussé à utiliser CUDA.

Le modèle de programmation CUDA se concentre sur l'utilisation de fils d'exécution légers (*threads*) qui sont automatiquement ordonnancés et répartis sur les cœurs physiques de calculs du GPU. Ces threads s'exécutent en parallèle sur des données différentes. Il s'agit d'une variante du modèle SIMD du traitement parallèle

de données. Le paradigme de base de programmation parallèle n'est plus l'utilisation d'instructions parallèles spécifiques capables de traiter plusieurs données en même temps, mais un paradigme « multithread » de plusieurs fils d'exécution parallèle. Ainsi, le parallélisme n'est plus visible directement dans le code. C'est le matériel qui assure le traitement parallèle des instructions sur des données différentes en provenances des threads du programme. Ce modèle d'exécution parallèle est désigné par le terme *SIMT* (« Single Instruction, Multiple Threads »). Le programmeur écrit alors les fonctions à destination du GPU, appelées *kernels* en utilisant des éléments scalaires. La vision parallèle de programmation est implicite à la différence de l'écriture de programmes SIMD pour CPU où le programmeur manipule des registres SIMD contenant plusieurs éléments scalaires.

Le plus bas niveau dans la hiérarchie CUDA est donc le thread. Ces threads sont regroupés en *thread blocks* ou blocs de threads. Ce sont des blocs logiques à différencier des blocs de traitement correspondant à un regroupement physique des cœurs de calcul CUDA. Ces blocs peuvent être 1D, 2D ou 3D et comportent au maximum jusqu'à 1024 threads. Ces blocs sont ensuite organisés en *grid* ou grille de blocs de threads, également 1D, 2D ou 3D. Les grilles peuvent contenir jusqu'à $2^{31} - 1$ blocs dans la direction x et jusqu'à 65535 dans les directions y et z. Un GPU peut également supporter l'exécution de plusieurs grilles en même temps.

Concernant l'exécution des threads sur le GPU, ceux-ci sont regroupés en groupes synchrones de 32 threads appelés *warps* exécutant la même instruction au même cycle. Les threads d'un même warps sont nécessairement synchronisés entre eux. La synchronisation au-delà de ce niveau nécessite l'emploi d'instructions de barrière spécifiques (comme pour la synchronisation de tous les threads d'un même bloc) ou d'une bibliothèque d'extension du modèle CUDA (pour la synchronisation des blocs au sein d'une grille, il faut employer l'extension *Cooperative Groups*).

D'un point de vue mémoire, il existe plusieurs espaces à la disposition du programmeur. Le tableau 2.1 liste les 6 différents types de mémoires du modèle CUDA. On notera que la mise en cache pour les mémoires *Local* et *Global* dépend de l'architecture GPU visée.

- Les *Registres* contiennent toutes les variables ainsi que les petits tableaux à adressage statique (taille et indice d'accès déterminé à la compilation).
- La mémoire *Local* contient les variables et les tableaux locaux aux threads qui ne peuvent être contenus par les registres. Cette mémoire étant située dans la mémoire du système à l'extérieur du GPU, elle est moins rapide que les registres.

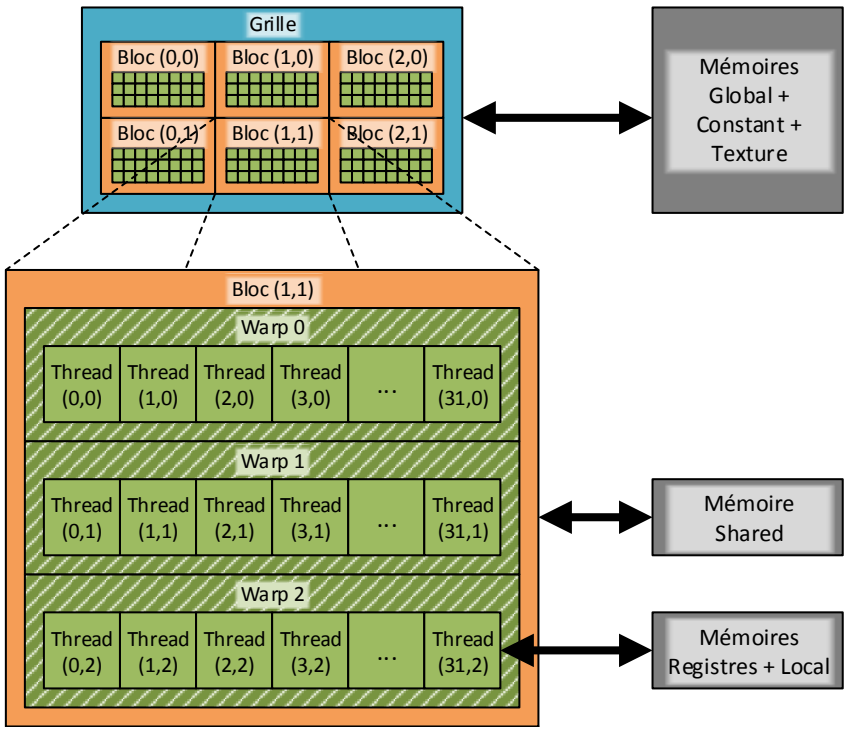


Fig. 2.4 : Organisation hiérarchique des threads, blocs et grille ainsi que des zones mémoire dans le modèle CUDA. Chacune des cases mémoire correspond à la localité d'accès de l'élément CUDA associé.

- La mémoire *Shared* est une mémoire allouée pour tous les threads d'un même bloc. Tous les threads du même bloc ont accès à cette mémoire rapide, situé au même niveau que le cache L1 proche des cœurs de calcul GPU. Cette mémoire permet aux threads du même thread bloc de partager leurs données et ainsi d'améliorer la réutilisation de ces dernières.
- La mémoire *Global* correspond à la mémoire système. Cette mémoire est la plus lente des mémoires disponibles dans CUDA. C'est elle qui est chargée de faire le pont entre les données du programme en entrée (du côté CPU) et le GPU. Sur les architectures les plus récentes, cette mémoire est mise en cache dans les niveaux L2 et L1. Pour accéder de manière efficace à cette mémoire, des accès contigus sont requis. Afin de réduire le nombre de transactions requises pour un warp, les threads du warp doivent réaliser des accès alignés capable d'être mutualisés en une seule transaction mémoire. Dans les cas d'accès non séquentiels, épars ou mal alignés, de multiples transactions séquentielles sont requises.
- La mémoire *Constant* correspond à une mémoire lecture seule de 64 Ko située dans la mémoire du système, mais mise en cache dans le GPU. Cette mémoire est conçue pour le partage de valeurs constantes : le contenu du cache lecture seule associé à cette mémoire peut être diffusé à plusieurs threads en même temps.

- La mémoire *Texture* correspond quant à elle à un espace mémoire côté système, mais mis en cache côté GPU, conçu pour la localité spatiale 2D : les threads d'un même warp réalisant des accès spatialement proche dans cette mémoire obtiennent les meilleures performances possibles pour cette mémoire. Dans certaines situations, cette mémoire permet d'obtenir de meilleures performances qu'avec la mémoire *Global*. Cette mémoire implémentée en matériel comprend également une gestion des bords et d'interpolation, utilisée notamment pour le cas d'un adressage fractionnaire en nombre flottant. Cette mémoire n'est cependant accessible qu'en lecture seule : il n'est pas possible de mettre à jour les valeurs du tableau sans ré-allocation.

Mémoire	**Emplacement**	**Mis en Cache**	**Accès**	**Portée**	**Durée de vie**
Registre	GPU	n/a	R/W	1 thread	Thread
Local	Système	Dépend GPU	R/W	1 thread	Thread
Shared	GPU	n/a	R/W	1 bloc	Bloc
Global	Système	Dépend GPU	R/W	GPU + système	Allocation système
Constant	Système	Oui	R	GPU + système	Allocation système
Texture	Système	Oui	R	GPU + système	Allocation système

Tab. 2.1 : Caractéristiques des différentes mémoires dans le modèle CUDA.

Ces multiples mémoires présentent des avantages et des inconvénients. Le programmeur doit être conscient de cela lors de l'utilisation de ces mémoires. Des détails propres aux architectures sont également à prendre en compte, notamment la taille et la bande passante des mémoires, ainsi que le nombre de cœurs de calculs et leur agencement dans le GPU. Ces détails seront abordés lors de la description des architectures GPU visées.

2.2.3 Limites de la parallélisation

Malgré les différentes techniques et architectures présentées précédemment, il existe plusieurs limites supérieures théoriques quant au gain que l'on peut obtenir grâce à la parallélisation. Il arrive un point où l'ajout de plus d'unités d'exécution parallèle apporte un gain négligeable quant au temps d'exécution du programme ou de l'algorithme parallèle.

Des modèles théoriques, tels que la loi d'Amdahl et la loi de Gustafson, sont utilisés pour prédire l'accélération potentielle maximale pouvant être obtenue en parallélisant un programme.

2.2.3.1 Loi d'Amdahl

La loi d'Amdahl est la plus ancienne de ces deux lois. Présenté en 1967 par Gene Amdahl [Amd67], cette loi donne l'accélération potentielle maximale d'un programme que l'on peut obtenir lorsque le nombre de processeurs augmente. On notera que la taille du problème que le programme doit traiter reste constante, peu importe le nombre d'unités parallèles. Globalement, le gain en performance est limité par les parties séquentielles du programme qui ne peuvent pas être parallélisées.

Un programme peut être divisé en une partie qui bénéficie de l'augmentation des ressources d'un système (la partie parallèle) et en une deuxième partie qui ne bénéficie pas de cette augmentation (la partie séquentielle). Le temps total de l'exécution du programme est dénoté par T et la fraction totale de la partie parallèle du programme par p. Le temps total est donc la somme des temps de la partie parallèle T_{par} et de la partie séquentielle T_{seq} :

$$\begin{aligned} T &= T_{par} + T_{seq} \\ T &= pT + (1-p)T. \end{aligned} \tag{2.1}$$

De plus, la partie parallèle est la seule partie qui bénéficie de l'augmentation des ressources du système. Si l'on suppose une parallélisation parfaite, le temps d'exécution de la partie parallèle est ainsi divisée par le nombre de processeurs parallèles n :

$$\begin{aligned} T_{par}(n) &= \frac{pT}{n} \\ &\Leftrightarrow \\ T(n) &= \frac{pT}{n} + (1-p)T \end{aligned} \tag{2.2}$$

On obtient ainsi la formulation d'Amdahl de l'accélération $S(n)$ obtenue par rapport au temps initial T :

$$S(n) = \frac{T}{T(n)} = \frac{1}{\frac{p}{n} + (1-p)}. \tag{2.3}$$

À titre d'exemple, si l'on suppose un programme parallélisable à 90%, l'accélération maximale que l'on peut obtenir grâce à l'augmentation du nombre d'unités d'exécution parallèle est :

$$\lim_{n \to +\infty} S(n) = \lim_{n \to +\infty} \frac{1}{\frac{0.9}{n} + 0.1} = 10. \tag{2.4}$$

On notera aussi que dans cet exemple, le gain d'accélération décroît à mesure que le nombre d'unités d'exécution augmente : on a $S(2) = 1,8$, $S(4) = 3,1$, $S(8) = 4,7$, $S(16) = 6,4$, $S(32) = 7,8$ ...

On tire ainsi de cette loi que c'est la partie séquentielle des programmes qui limite les gains pouvant être obtenu sur la partie parallélisable. Il est crucial de limiter cette proportion et de l'exécuter le plus rapidement possible. Cette loi repose sur des hypothèses fixes et ne s'applique ainsi que sur des programmes dont la charge reste constante, peu importe l'évolution des ressources mises à disposition.

2.2.3.2 Loi de Gustafson

La loi d'Amdahl nous indique une borne supérieure théorique des gains pouvant être obtenu grâce à l'amélioration du parallélisme dans un programme ayant une charge fixe. Passé un certain degré de parallélisme, les gains obtenus sont infinitésimaux. Cependant, il est possible d'argumenter que cette augmentation des ressources parallèles du système nous permet, en plus d'accélérer notre programme initial, d'augmenter la taille de la charge en entrée du programme pour une durée de traitement similaire. C'est cette idée d'augmentation de la taille de la charge à traiter en parallèle de l'augmentation du niveau de parallélisme du système qui est exprimé par la loi de Gustafson. Cette loi formulée en 1988 par John L. Gustafson [Gus88] nous indique les gains liés à une augmentation du parallélisme dans le cas où la taille du problème à traiter crois lui aussi avec la taille du système. Cette loi est ainsi moins stricte que la loi d'Amdahl et permet ainsi des gains théoriques plus importants. Sa formulation est la suivante. Comme pour la loi d'Amdahl, on suppose que le temps total d'exécution du programme T peut se séparer en un temps de la partie séquentielle T_{seq} et un temps de la partie parallèle T_{par} : $T = T_{par} + T_{seq}$. On suppose que le temps total obtenu sur un système parallèle à n processeurs est de 1 : $T = 1$. On a ainsi :

$$T = s + p, \tag{2.5}$$

avec s la proportion séquentielle du programme et p la proportion parallèle. Étant donné que la partie séquentielle du programme n'évolue pas avec une meilleure parallélisation, le temps obtenu sur un système n'ayant qu'un seul processeur est donc :

$$T' = s + pn. \tag{2.6}$$

Le gain en temps pour le programme entre le système mono-processeur le système multi-processeurs est donc :

$$\begin{aligned} S(n) &= \frac{T'}{T} = \frac{s+pn}{s+p} = s+pn, \\ &\Leftrightarrow \\ S(n) &= (1-p)+pn. \end{aligned} \tag{2.7}$$

À la différence de la loi d'Amdahl, cette loi admet une croissance linéaire et n'est plus bornée. Cette loi est moins pessimiste : si les données en entrées du programme croient en même temps que les ressources de calculs du système, il y aura toujours un gain lié à l'augmentation du parallélisme.

2.3 Représentation numérique des nombres

En fonction de l'algorithme à implémenter, un choix doit être fait quant à la représentation numérique de ses données. Le plus souvent, il s'agit de représenter des nombres. En fonction de l'ensemble numérique en entrée de l'algorithme et des opérations effectuées, plusieurs solutions s'offrent au programmeur.

2.3.1 La représentation entière

Le type *entier* permet la représentation d'un intervalle fini de nombres entiers. La représentation de ces nombres passe par une représentation binaire : bien que la valeur soit le plus souvent affichée en base 10, le stockage et fait en base 2 dans la mémoire de l'ordinateur.

En fonction de l'intervalle souhaité, ces entiers peuvent être plus ou moins grands (typiquement de 8 bits à 64 bits) et non-signés s'il s'agit de représenter des nombres positifs ou signés s'il s'agit de représenter des nombres relatifs. De par l'absence des signes plus et moins dans la représentation binaire des nombres, une représentation particulière est nécessaire pour coder les nombres négatifs. Les quatre méthodes les plus connues pour coder ces derniers sont : le bit de signe, le complément à 1, le complément à 2 et le biais.

- Un bit de signe consiste à réserver l'un des bits du nombre (en général le bit de poids fort) pour marquer le signe. Les autres bits encodent la valeur absolue comme dans le cas de la représentation d'entiers positifs.
  Bien que cette représentation soit conceptuellement simple, la mise en œuvre matérielle de l'arithmétique est relativement lourde. En effet, lors des opérations arithmétiques, une différenciation est à faire entre les opérandes positifs et négatifs. De plus, avec cette représentation, deux représentations sont possibles pour le nombre zéro.
- Dans la représentation utilisant le complément à 1, les nombres négatifs sont représentés par l'inverse bit à bit de leur opposé (positif). Cette opération d'inversion correspond au complément logique bit à bit d'un nombre : lorsque l'on ajoute un nombre et son complément à 1, on obtient un nombre ne contenant que des 1.
  Cette représentation est plus légère à implémenter matériellement que la représentation utilisant un bit de signe : les opérations arithmétiques peuvent ignorer le signe des opérandes et réaliser un report circulaire de la retenue en fin d'opérations et la comparaison entre deux nombres ne dépend pas du signe. Cette représentation comprend également deux zéros.
- Dans la représentation utilisant le complément à 2, les nombres négatifs sont représentés par l'inverse bit à bit de leurs nombres positifs correspondant auquel on ajoute 1. Cette opération permet de calculer le complément à 2 – plus précisément le complément à 2^n – d'un nombre de n bits. Un nombre est le complément à 2^n d'un autre nombre de n bits si et seulement si leur somme donne 2^n.
  Avec cette représentation, il n'y a plus qu'un seul zéro et il n'y a plus besoin de réaliser un report circulaire de la retenue en fin d'opération. L'implémentation matérielle est plus légère que pour la représentation en complément à 1 : le même circuit peut être utilisé pour l'addition entre les nombres non-signés et signés.
- Enfin, pour ce qui est de la représentation biaisée, un biais implicite est ajouté aux nombres, de sorte que la valeur non-signée correspondante est $n + K$ où n est le nombre relatif à représenter et K le biais. Dans la plupart des cas, pour des nombres de n bits, un biais correspondant à la moitié de la plage des valeurs possibles $K = 2^{n-1}$ est choisi. Pour des nombres de $n = 4$ bits et un biais de $K = 2^{4-1} = 8$, la valeur 0000 correspondra donc au nombre -8, la valeur 0001 à -7 et ainsi de suite jusqu'à la valeur 1111 qui correspondra à la valeur 7.
  Cette représentation est plus flexible que la méthode du complément à 2 : la valeur du biais peut être ajustée en fonction du type de calcul effectué. Il faut cependant soustraire le biais dans le cas où les opérations manipulent des nombres non-signés. Dans le cas où le biais correspond à $K = 2^{4-1} = 8$ il suffit seulement d'inverser le bit de poids fort.

Le tableau 2.2 illustre ces quatre systèmes de représentation d'entiers relatifs ainsi que la représentation non-signée associée pour des nombres sur 16 bits.

Séquence Binaire	**Valeur Base 16**	**Interprétation**				
		Non-Signée	**Bit de Signe**	**Complément à 1**	**Complément à 2**	**Biais 2^{16-1}**
0000000000000000	0000	0	0	0	0	-32768
0000000000000001	0001	1	1	1	1	-32767
...	...	...	...	...	...	...
0111111111111110	7FFE	32766	32766	32766	32766	-2
0111111111111111	7FFF	32767	32767	32767	32767	-1
1000000000000000	8000	32768	-0	-32767	-32768	0
1000000000000001	8001	32769	-1	-32766	-32767	1
...	...	...	...	...	...	...
1111111111111110	FFFE	65534	-32766	-1	-2	32766
1111111111111111	FFFF	65535	-32767	-0	-1	32767

Tab. 2.2 : Comparaison des quatre principales représentations d'entiers relatifs pour des nombres de taille 16 bits.

Dans la suite de cette thèse, nous ne considérons que des entiers relatifs utilisant la représentation en complément à 2. Il s'agit de la représentation la plus utilisée dans les processeurs généralistes modernes, notamment dans les processeurs x86, MIPS, ARM, Power ISA ou encore SPARC. Nous avons ainsi les intervalles de représentation suivants :

— en utilisant le type entier non-signé sur n bits, il est possible de représenter les entiers naturels de 0 à $2^n - 1$;

— en utilisant le type entier signé sur n bits, il est possible de représenter les entiers relatifs de -2^{n-1} à $2^{n-1} - 1$.

2.3.2 La représentation en virgule flottante

La méthode utilisant la *virgule flottante* permet la représentation de nombres décimaux. Cette méthode est analogue à la notation scientifique des nombres : on représente un nombre comme le produit entre un nombre n'ayant qu'un seul chiffre dans sa partie entière et un autre nombre étant une puissance de 10. Pour représenter un nombre n en utilisant la virgule flottante, on aura :

$$n = (-1)^s \times b^e \times m, \tag{2.8}$$

avec s le signe du nombre, m la mantisse, b la base utilisée et e l'exposant. Pour représenter le nombre $-123,456$ on aura donc la représentation en base 10 suivante :

$-123,456 = (-1)^1 \times 10^{-3} \times 123456$. Si un nombre nécessite plus de chiffres pour représenter sa partie entière ou fractionnaire, il n'est pas représentable exactement.

La norme IEEE 754 spécifie l'implémentation d'un tel format de nombre ainsi que les opérations arithmétiques pour l'addition, la soustraction, la multiplication, la division et la racine carrée. Cette norme ajoute à la représentation dite normalisée précédente certaines notions supplémentaires :

- Pour permettre des exposants négatifs, un biais implicite est ajouté à ce dernier. Plutôt que d'utiliser la technique du complément à deux pour la représentation des exposants négatifs, cette technique est plus rapide pour la comparaison de valeurs. La valeur de ce biais est de $2^{b_e-1} - 1$, avec b_e le nombre de bits utilisé pour la représentation de l'exposant. Pour récupérer la valeur de l'exposant réel, on soustrait le biais de l'exposant biaisé.
- Le premier bit de la mantisse est toujours égal à 1 ; il s'agit d'un bit implicite. Ce n'est cependant pas le cas si la valeur de l'exposant biaisé est de 0. Dans ce cas, le nombre est interprété en tant que nombre dénormalisé : le premier bit implicite de la mantisse est alors 0 et l'exposant interprété comme étant la plus petite valeur autorisée, c'est-à-dire la valeur du biais incrémenté de 1 : $2^{b_e-1} - 2$. Cette incrémentation de 1 garantit la continuité entre cette représentation et la représentation normalisée : il n'y a pas de changement d'exposant entre le plus petit nombre normalisé et le plus grand nombre dénormalisé. Cette gestion particulière permet une représentation plus fine et linéaire de valeurs entre les valeurs positives et négatives représentables. Elle permet également un soupassement arithmétique – l'obtention d'un résultat plus petit que la plus petite valeur représentable à la suite d'un calcul – plus graduel, car les calculs perdent plus lentement en précision lorsque le résultat est petit et proche de 0.
- Des valeurs spéciales permettent l'encodage des valeurs 0, des infinis et des NaN (« Not a Number »). Si l'exposant biaisé et la mantisse sont tous les deux nuls, alors la valeur représentée est ± 0 en fonction du bit de signe. Si l'exposant biaisé est égal à la valeur maximale représentable $2^{b_e} - 1$ et que la mantisse est nulle, la valeur représentée correspond à $\pm\infty$ en fonction du bit de signe. Si l'exposant biaisé est égale à la valeur maximale représentable et que la mantisse contient une autre valeur que 0, alors la valeur représentée correspond à un NaN.

Cette norme est ainsi résumée dans le tableau 2.3.

Pour ce qui est des représentation normalisées et dénormalisées, elles sont décrites en 2.9. Soient un exposant e de b_e bits et une mantisse m de b_m bits. On a le biais pour l'exposant défini par $e_{\text{biais}} = 2^{b_e-1} - 1$. On obtient ainsi :

Exposant biaisé	Mantisse	Représentation
$e = 0$	$m = 0$ $m \neq 0$	$\pm$ Zéros Nombres dénormalisés
$e \in [1, 2^{b_e} - 2]$	m quelconque	Nombres normalisés
$e = 2^{b_e} - 1$	$m = 0$ $m \neq 0$	$\pm$ Infinis NaNs

Tab. 2.3 : Nature des nombres représentés en fonction des valeurs de l'exposant et de la mantisse pour la norme IEEE 754 sur l'arithmétique à virgule flottante.

$$\begin{aligned} \text{Nombres normalisés} &: (-1)^s \times 2^{e - e_{\text{biais}}} \times 1, m \\ \text{Nombres dénormalisés} &: (-1)^s \times 2^{-e_{\text{biais}}+1} \times 0, m \end{aligned} \tag{2.9}$$

m contenant ainsi la partie fractionnaire des nombres décimaux $1, m$ et $0, m$; la partie entière du nombre étant respectivement 1 et 0.

Les formats flottants les plus utilisés aujourd'hui sont :

— le format *demi-précision* désigné par le nom *binary16* dans la norme IEEE 754-2008 et également appelé *half* ou F_{16}, codé sur 16 bits ;
— le format *simple-précision* désigné par le nom *binary32* dans la norme IEEE 754 et également appelé *single* ou F_{32}, codé sur 32 bits ;
— le format *double-précision* désigné par le nom *binary64* dans la norme IEEE 754 et également appelé *double* ou F_{64}, codé sur 64 bits ;
— le format *quadruple-précision* désigné par le nom *binary128* dans la norme IEEE 754-2008 et également appelé *quad* ou F_{128}, codé sur 128 bits.

Les formats *binary16* et *binary128* ont été ajoutés dans la révision de 2008 de la norme. L'emploi des nombres demi-précision est antérieure à cette dernière [EL04] et sont largement utilisés aujourd'hui dans le domaine du traitement image. On liste en figure 2.5 trois formats de nombres flottants demi-précision : le format $F16$ désormais utilisé dans la norme IEEE 754, le format $F13$ défini comme un format flottant léger pour les systèmes embarqués [Pis+06 ; EPL06 ; PLE06] et le format *bfloat16* ($BF16$) pour accélérer l'apprentissage automatique ainsi que pour faciliter les conversions avec les $F32$. Les formats $F13$ et $BF16$ correspondent d'ailleurs au format $F32$ tronqué :

— $F13$: exposant tronqué à 5 bits et mantisse tronquée à 7 bits,
— $BF16$: exposant non-tronqué et mantisse tronquée à 7 bits.

Pour conclure cette section de représentation numérique, nous résumons dans le tableau 2.4 l'ensemble des représentations entières et décimales utilisées dans nos travaux. D'autres représentations existent mais n'ont pas été utilisées de part des temps d'exécution trop long (notamment les nombres double-précision sur

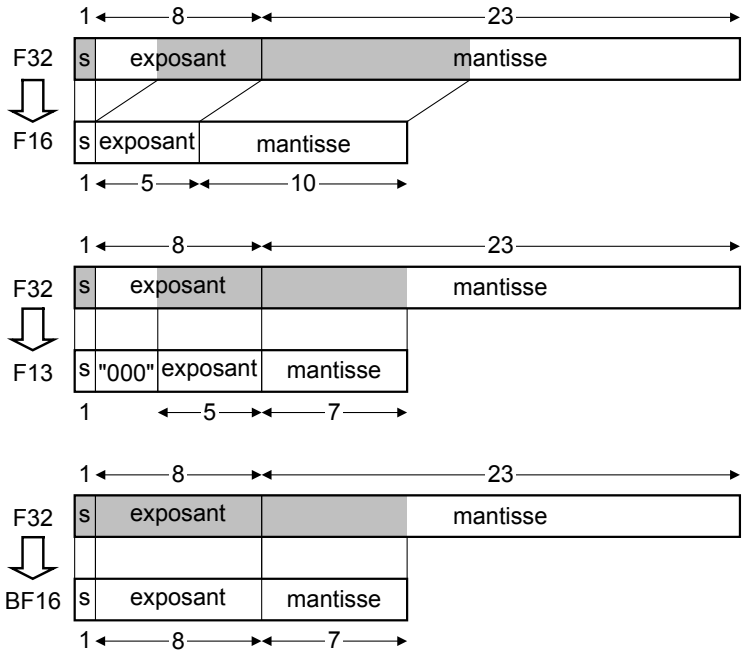


Fig. 2.5 : Formats flottants $F16$, $F13$ et $BF16$ comparés au format $F32$. On notera que le format $BF16$ est identique au format $F13$ mais avec 3 bits de poids fort supplémentaire dans l'exposant.

64 bits) ou une non-disponibilité sur les plateformes utilisées. Des outils existent aujourd'hui permettant à un programmeur de définir son propre type flottant. Cela permet une meilleure adéquation entre les performances numériques visées pour une application et les performances temporelles ou encore énergétiques effectivement obtenues [SM22 ; Def+20]. Il convient également de mentionner les FPGA qui offrent la possibilité d'une définition et d'une utilisation matérielle de nombres à virgule flottante de taille arbitrairement grande ou petite [BCL21]. Nous verrons dans le chapitre 3 les détails des performances de ces types sur les architectures utilisées dans nos travaux.

Type de Nombre	**Bits**	**Plage de Valeurs**	**Chiffres décimaux**
unsigned char	8	$[0; 255]$	0
char	8	$[-127; 128]$	0
unsigned int	32	$[0; 2^{32} - 1]$	0
int	32	$[-2^{31}; 2^{31} - 1]$	0
float16	16	$\pm[2^{-14}; (2 - 2^{-10}) \times 2^{15}] \approx [6,10 \times 10^{-5}; 6,55 \times 10^{4}]$	$\log(2^{11})$
float32	32	$\pm[2^{-126}; (2 - 2^{-23}) \times 2^{127}] \approx [1,18 \times 10^{-38}; 3,40 \times 10^{38}]$	$\log(2^{24})$

Tab. 2.4 : Résumé des représentations numériques utilisées dans nos travaux.

2.4 Conclusion

Nous avons abordé dans ce chapitre les points importants à prendre en compte pour notre contexte d'étude. Nous avons présenté les principes de base du domaine de l'estimation du mouvement des pixels dans une image ainsi que les points à prendre en compte lors du choix d'un algorithme spécifique. Nous avons ensuite présenté les avantages et les inconvénients liés l'utilisation d'architectures parallèles, en particulier les architectures GPU. Nous avons également abordé le paradigme de programmation des GPU. Enfin, nous avons présenté les systèmes de représentations des nombres dans les systèmes informatiques modernes, ainsi que leurs avantages et leurs limitations.

La présentation de ces sujets nous permet ainsi d'énumérer les questions auxquelles nous tenterons de répondre dans cette thèse :

Quel est le gain apporté par une utilisation efficace du GPU par rapport au CPU ? Ce gain est à étudier pour déterminer les conditions d'utilisations optimales des GPU et des CPU dans le domaine du traitement d'image embarqué. Il doit inclure le temps d'exécution de notre algorithme de flot optique ainsi que la consommation électrique dépensée. Les conditions expérimentales incluent les tailles d'images traitées, la qualité du traitement réalisé ou encore la consommation électrique des systèmes. Une question annexe concernant les techniques d'optimisation et d'implémentation se pose : certaines techniques sont-elles plus ou moins efficaces sur GPU que sur CPU ? Pour ce point, il sera nécessaire d'étudier et de comparer en détail les architectures GPU visées (mémoires disponibles, fréquence de fonctionnement, unités de traitement disponibles, bandes passantes, etc.).

Dans le cadre des algorithmes itératifs d'estimation du flot optique, une baisse de la dynamique des nombres utilisés permet-elle une accélération suffisante pour améliorer la convergence des algorithmes ? Le passage d'une quantification de nombres flottants sur 32 bits (c.-à-d. les nombres simple-précision) vers une quantification sur 16 bits (c.-à-d. des nombres demi-précision) entraîne nécessairement une perte de précision et dans certains cas une perte d'exactitude des résultats. Ce passage permet cependant une diminution notable du temps d'exécution des algorithmes (dans la plupart des cas, au moins une diminution d'un facteur deux) mais également une baisse de la consommation énergétique [Che+17]. Dans le même temps imparti, les algorithmes itératifs implémentés en demi-précision sur 16 bits peuvent ainsi réaliser plus d'itérations que leurs homologues simple-précision sur 32 bits. Cette

accélération est-elle suffisante pour compenser ou même dépasser la qualité des résultats obtenus en utilisant une quantification plus grande ?

Toutes ces optimisations et considérations sont-elles valables pour d'autres algorithmes itératifs de flot optique ? Il s'agit ici de voir la portée de nos techniques d'optimisations et d'implémentations. Nous nous proposons ainsi d'étudier deux algorithmes itératifs d'estimation du flot optique : la méthode de Horn-Schunck et la méthode TV-L^1. Bien que ces deux algorithmes réalisent un traitement comparable, ils présentent des différences notables à la fois en complexité algorithmique, mais aussi en quantité de données manipulées. Les mêmes optimisations seront réalisées sur les implémentations de ces deux algorithmes et comparées afin de déterminer si certains types d'algorithmes sont à exécuter de préférence sur GPU ou bien sur CPU.

3 Optimisations GPU pour l'estimation du mouvement

Dans ce chapitre, nous présentons les principales méthodes de l'état de l'art de l'estimation du flot optique sur GPU. Nous explorons également dans ce chapitre les détails architecturaux des GPU embarqués ciblés pour nos travaux, pour lesquels nous utilisons les cartes Jetson de NVIDIA. Enfin, nous exposons les concepts d'optimisation et de transformation algorithmique haut et bas niveaux utilisés dans nos travaux. Ces optimisations sont nécessaires afin de rendre temps réel l'exécution des algorithmes retenus sur nos plateformes embarquées.

Le but principal de ce chapitre est d'explorer les algorithmes et les plateformes utilisés dans nos travaux et de comprendre nos choix d'implémentations et d'optimisations. Les concepts et les exemples présentés dans ce chapitre se veulent les plus génériques possibles. C'est principalement le cas dans la section de présentation des optimisations algorithmiques. Cela nous permettra d'exposer plus simplement et de manière plus claire nos détails d'implémentation dans le chapitre suivant.

3.1 État de l'art des algorithmes d'estimation du mouvement

Les bases de référence des algorithmes d'estimation du mouvement Middlebury, MPI-Sintel et Kitti se focalisent principalement sur les aspects qualitatifs du flot optique produit en sortie des algorithmes. Pour les applications temps réel des travaux de cette thèse, nous devons nous concentrer principalement sur le temps d'exécution des algorithmes, tout en produisant un flot optique suffisamment juste et précis. Dans la plupart des publications, le temps d'exécution est fourni à titre indicatif et la consommation énergétique n'est généralement pas abordée. Il s'agit d'un problème de plus en plus central, à la fois d'un point vu écologique, mais aussi d'un point de vue de longévité des processeurs [DP13]. Certains de ces algorithmes font cependant l'objet d'optimisations sur les plateformes embarquées parallèles à la fois CPU et GPU et le temps est rapporté.

3.1.1 État de l'art CPU

Nous commençons par présenter l'état de l'art de l'estimation du flot optique sur CPU. Il est pertinent de commencer par cette présentation étant donné que ces implémentations précèdent celles des GPU de plusieurs décennies pour les plus vieilles. De plus, une majorité des implémentations optimisées sur GPU l'ont d'abord été sur CPU. Là encore, nous nous focalisons principalement sur l'aspect temps réel des algorithmes plus que sur l'aspect qualitatif du flot obtenu.

L'une des plus vieilles publications concernant le sujet remonte à 1984 [BSB84]. Dans cet article, les auteurs présentent des optimisations d'implémentations d'une méthode d'estimation se basant sur de la détection de contour [BB84]. Des implémentations SIMD sont présentées pour les processeurs spécialisés DAP (*Distributed Array Processor*) et GRID (*GEC Rectangular Image and Data*). La méthode décrite dans l'article [Cam97] présente une implémentation d'un algorithme se basant sur la corrélation par patchs d'images. Cet article présente des compromis entre le temps d'exécution et la magnitude maximale des mouvements des pixels, ainsi qu'une implémentation temps réel sur processeur HyperSPARC. L'article [Gar+14] présente une implémentation parallèle de l'algorithme de Lucas-Kanade [LK+81] en utilisant la norme *MPI* de parallélisation par passage de messages. Kroeger présente dans [Kro+16] une méthode par patch se basant sur les travaux de Baker [BM01] sur le recalage d'images. Cette méthode est parallélisée grâce à l'interface de programmation OpenMP. Plus récemment, Petreto présente dans [Pet+18b ; Pet+19] des implémentations SIMD efficaces de l'algorithme TV-L^1 [ZPB07] sur les systèmes embarqués NVIDIA Jetson. Ces implémentations sont également multi-cœurs et utilisent OpenMP.

Pour les raisons présentées dans le chapitre précédent et ayant trait au temps de traitement temps-réel, à la résolution image et à la limitation de la consommation énergétique, nous n'avons pas retenu une implémentation CPU dans le cadre de cette thèse. Nous utilisions cependant ces implémentations CPU comme une référence de comparaison afin de déterminer l'efficacité de notre travail d'implémentation et d'optimisation. En effet, certaines implémentations d'algorithmes sur CPU – grâce notamment au parallélisme multi-cœur et au parallélisme SIMD – peuvent être plus rapides que des implémentations GPU. C'est le cas sur des petites tailles de données ainsi que sur des algorithmes irréguliers. Nous pouvons citer en exemple l'algorithme TV-L^1 [Pet20] et les algorithmes de tri par base et par fusion [Sat+10].

3.1.2 État de l'art GPU

3.1.2.1 Algorithmes itératifs

L'efficacité des GPU a également été démontrée pour de nombreux algorithmes de flot optiques. La majorité des implémentations sur GPU correspondent à des algorithmes itératifs, de par leur régularité et leur ancienneté.

De par l'absence de codes sources, il ne nous est pas possible d'évaluer précisément les algorithmes tirés de l'état de l'art. Ainsi, le temps d'exécution indiqué pour chaque algorithme est normalisé en fonction du matériel GPU et de la configuration de l'algorithme avec la formule suivante :

$$C = \frac{T \times F}{N_{cores} \times N_{pix}} \tag{3.1}$$

où C est le nombre de cycles par cœur et par pixels, T est le temps d'un benchmark donné, F est la fréquence du GPU, N_{cores} le nombre de cœurs du GPU et N_{pix} le nombre de pixels dans l'image. Nous avons choisi de ne pas normaliser par le nombre d'itérations exécuté étant donné que la vitesse de convergence varie d'un algorithme à l'autre. Nous avons ainsi pu construire le tableau 3.1 contenant les résultats normalisés.

Dans certains cas, il nous a été possible de faire fonctionner une implémentation réelle sur des images suffisamment grandes (1024×1024 pixels) pour alimenter le GPU. C'est le cas pour l'algorithme « Flow-filter » [AM16] et pour les implémentations OpenCV sur GPU des algorithmes TV-L[1] [ZPB07], Brox [Bro+04], Lucas-Kanade [LK+81] avec une pyramide d'images et Farneback [Far03]. Nous avons testé ces versions sur le GPU de la carte embarquée NVIDIA Jetson AGX Xavier.

Grâce à ce tableau, il nous est possible d'identifier les implémentations les plus efficaces. L'algorithme « Flow-filter » semble être la plus rapide parmi les algorithmes du tableau. Cependant, un test d'exécution réel sur la carte Jetson AGX Xavier indique une dégradation des performances pour des images plus grande et pour un GPU embarqué. L'algorithme « eFOLKI » [PLC16] présente le deuxième temps d'exécution des algorithmes de l'état de l'art. Cependant, le code sources n'est plus disponible pour un lancement réel. De plus, là aussi l'algorithme semble être moins performant sur un GPU embarqué sur un ordinateur portable. Le passage d'un gros GPU NVIDIA GeForce GTX Titan vers le GPU d'ordinateur portable GeForce GTX 765M pour des images $4\times$ plus petites divise les performances par 10.

Algorithme	Temps (s)	Résolution (pix)	Carte GPU NVIDIA	Fréquence GPU (MHz)	Nombre de Cœurs	Cycles Normalisés
TV-L1 introduction [ZPB07]	0,029	256×256	GeForce Go 7900 GTX	500	24	9.322
P(GPU) [Wed+09]	0,039	640×480	GeForce GTX 285	1476	240	0.781
P-MF(GPU) [Wed+09]	0,055	640×480	GeForce GTX 285	1476	240	1.101
P-IT-MF(GPU) [Wed+09]	0,061	640×480	GeForce GTX 285	1476	240	1.221
A-Huber-L^1 [Wer+09]	1,130	640×480	GeForce GTX 280	1296	240	19.863
FED [Gwo+10]	0,734	640×480	GeForce GTX 480	1401	480	6.974
LDOF GPU [SBK10]	1,840	640×480	GeForce GTX 480	1401	480	17.482
TV-L1 for interactivity [dAn+11]	0,125	640×480	GeForce GT320M	1100	24	18.650
TV-L1 (DL) [Bao+14]	0,130	640×480	GeForce GTX 780 Ti	1019	2880	0.150
TV-L1 (DL2) [Bao+14]	0,140	640×480	GeForce GTX 780 Ti	1019	2880	0.161
TV-L1 (FP) [Bao+14]	0,540	640×480	GeForce GTX 780 Ti	1019	2880	0.622
TV-L1 (SB) [Bao+14]	0,301	640×480	GeForce GTX 780 Ti	1019	2880	0.357
eFOLKI [PLC16]	0,046	1920×1080	GeForce GTX 765M	850	768	**0,025**
eFOLKI [PLC16]	0,050	3840×2160	GeForce GTX Titan	876	2688	**0,002**
Flow-filter [AM16]	0,001	625×449 [1]	GeForce GTX 780	900	2304	**0,001**
Flow-filter [AM16]	0,013 [2]	1024×1024	Jetson AGX Xavier	1377	512	**0,034**
Horn-Schunck occlusion[LR19]	0,245	320 × 240	GeForce GTX 980 Ti	1075	2816	1.220
CLG GPU [Sez+22b]	0,017	640 × 515	Jetson AGX Xavier	1377	512	0.137
OpenCV TV-L1 GPU	0,020 [3]	1024×1024	Jetson AGX Xavier	1377	512	**0,053**
OpenCV Brox GPU	0,041 [3]	1024×1024	Jetson AGX Xavier	1377	512	0.105
OpenCV Lucas-Kanade GPU	0,006 [3]	1024×1024	Jetson AGX Xavier	1377	512	**0,016**
OpenCV Farneback GPU	0,046 [3]	1024×1024	Jetson AGX Xavier	1377	512	0.118

Tab. 3.1 : Temps d'exécution des algorithmes itératifs de l'état de l'art d'estimation du flot optique. Les cases grises correspondent à des vraies mesures sur la carte Jetson Xavier AGX.

Du côté des exécutions des implémentations GPU OpenCV, les deux algorithmes les plus rapides sont Lucas-Kanade et TV-L^1. Même si Lucas-Kanade est $3\times$ plus rapide que TV-L^1, il s'agit d'un algorithme plus simple, moins robuste au bruit dans l'image et qui n'est pas capable de gérer les discontinuités dans les images. TV-L^1 apporte lui un bon compromis entre qualité du flot optique, robustesse au bruit et complexité algorithmique. Dès son introduction en 2007, la méthode d'estimation TV-L^1 se basant sur une formulation duale [ZPB07] fut implémentée sur CPU, mais aussi sur GPU. Cette implémentation utilise le langage *Cg* (« *C for Graphics* ») de programmation de nuanceur graphique sans préciser les éventuelles optimisations en passant du CPU au GPU. Les auteurs indiquent cependant l'utilisation de nombres flottants demi-précision pour le stockage de certaines valeurs de l'algorithme. Une implémentation plus récente utilisant le langage de programmation CUDA [Poc+08] ainsi qu'une nouvelle version améliorée de cet algorithme contenant des étapes de filtrage supplémentaires a été présentée par la suite [Wed+09] et a également été implémentée sur GPU. D'autres méthodes similaires se basant sur la même formulation duale ont également été le sujet d'implémentation sur GPU, notamment la méthode

1. Il s'agit d'une résolution moyenne des images de la base de donnée Middlebury.
2. Temps obtenu sur Jetson AGX Xavier en lançant le programme de test du dépôt GitHub des auteurs disponible à l'adresse `https://github.com/jadarve/optical-flow-filter`.
3. Temps obtenu sur Jetson AGX Xavier pour OpenCV 4.5.5.

du point-fixe [Bro+04 ; SBK10 ; Bao+14], la méthode de split-Bregman [ARS12 ; Bao+14] ou encore une méthode [dAn+11] se basant sur l'algorithme itératif *FISTA* (« *Fast Iterative Shrinkage-Thresholding Algorithm* ») [BT09]. Pour ces raisons, mais également de par nos travaux d'optimisation sur CPU [Pet20 ; Pet+18b], TV-L[1] apparait comme un bon candidat d'optimisation sur GPU.

3.1.2.2 Apprentissage machine

De nombreuses nouvelles méthodes basées sur l'apprentissage machine ont montré des résultats impressionnants en termes de qualité de flot optique. Cependant, ces méthodes ne sont pas de bons candidats pour les systèmes embarqués. Leurs besoins en calculs sont tels que leur fonctionnement nécessite des temps d'exécution importants ainsi que de grosses plateformes. La plupart de ces méthodes font usage de GPU puissants et non-embarqués. En reprenant la méthode précédente de normalisation de cycles, on obtient le tableau 3.2 suivant :

Algorithme	Temps (s)	Résolution (pix)	Carte GPU NVIDIA	Fréquence GPU (MHz)	Nombre de Cœurs	Cycles Normalisés
FlowNet [Dos+15]	0,080	1024×436	GeForce GTX Titan	876	2688	0,058
FlowNet2 [Ilg+17]	0,123	1024×436	GeForce GTX 1080	1733	2560	0,186
LiteFlowNet [HTL18]	0,090	1024×436	GeForce GTX 1080	1733	2560	0,136
LiteFlowNet2 [HTL20]	0,040	1024×436	GeForce GTX 1080	1733	2560	0,061
LiteFlowNet3 [HL20]	0,590	1024×436	GeForce GTX 1080	1733	2560	0,895
PWC-Net [Sun+18a]	0,030	1024×436	Titan X	1531	3584	0,029
RAFT [TD20]	0,100	1024×436	GeForce GTX 1080 Ti[4]	1582	3584	0,099

Tab. 3.2 : Temps d'exécution des méthodes de l'état de l'art d'apprentissage automatique d'estimation du flot optique.

De plus, ces méthodes font usage de bibliothèques d'apprentissage machine accéléré par GPU (Pytorch, TensorFlow, cuDNN et Caffe nottament). Ces méthodes nécessitent des versions précises de ces bibliothèques qui ne sont pas toujours disponibles sur les plateformes embarquées ARM Jetson utilisées dans nos travaux. Des travaux récents ont cependant ciblé ce problème d'implémentations sur cartes embarquées [Sez+22a ; CZU22]. Ces travaux se basent sur les méthodes existantes sur plateformes non-embarqués PWC-Net et RAFT respectivement. Enfin, certaines de ces méthodes font l'utilisation d'environnement d'installation non disponible sur nos cartes (comme les environnements Conda ou Docker). Ces sévères restrictions ne nous permettent pas une comparaison précise avec l'état de l'art.

4. La méthode RAFT présentée utilise 2 GPUs GeForce RTX 2080 Ti pour l'entraînement et 1 GPU GeForce GTX 1080 Ti pour l'inférence.

Nous avons cependant pu tester la méthode FlowNet2 sur Jetson AGX. Sur le jeu de données MPI-Sintel de résolution 1024×436 pixels, il faut 660 ms par image avec le modèle pré-entraîné standard et 343 ms en utilisant le modèle pré-entraîné rapide. Un gain de vitesse de $\times 8.6$ est nécessaire pour obtenir un traitement en temps réel à 25 images par seconde. Cette résolution est environ $2,1\times$ inférieure en nombre de pixels à la résolution HD de 1280×720 pixels et $4,6\times$ inférieure à la résolution Full HD de 1920×1080 pixels. En comparaison, l'implémentation TV-L^1 OpenCV ne nécessite que 10 ms par image à la résolution de 1024×436 pixels [Rom+21 ; Bra00], sans pour autant fournir une implémentation entièrement optimisée. TV-L^1 est donc un bien meilleur candidat pour les applications embarquées que les méthodes par apprentissage automatique.

3.1.3 Synthèse

Nous avons pu explorer dans cette section les principaux algorithmes de l'état de l'art d'estimation du flot optique. Généralement, cette exploration met en avant les meilleurs algorithmes en termes de qualité et de fidélité du flot estimé. Ici, nous nous sommes principalement concentrés sur le temps d'exécution. Nos contraintes d'architecture visent des plateformes embarquées. Tenir compte de la complexité relative des algorithmes et donc du temps d'exécution est crucial. Il faut choisir les algorithmes qui bénéficient le plus d'optimisations sur plateformes embarquées, tout en produisant un flot optique le plus proche de la réalité avec la meilleure précision possible.

C'est dans cette optique de compromis que nous avons choisi de nous concentrer sur l'algorithme TV-L^1. Il s'agit d'un algorithme de référence utilisé dans de nombreuses applications telles que le débruitage vidéo [She+21 ; Pet+19], le block-matching [AM18 ; BLM16], la super-résolution [Bät+15 ; Mit+09] ou le défloutage [AM17 ; DS15]. Forts de notre expérience précédente sur CPU, nous avons fait le choix de cet algorithme tant par sa structure itérative et ses avenues d'optimisations que sa bonne qualité d'estimation de flot. De plus, il existe de multiples codes sources de référence tels que l'implémentation du journal IPOL [SMF13] ou les implémentations OpenCV en CPU et en GPU.

Du côté optimisation, la régularité de cet algorithme se porte bien sur les architectures GPU visées et l'aspect itératif de cet algorithme permet des optimisations visant à améliorer la localité spatiale et temporelle des données. Du côté qualité de flot, cette méthode possède plusieurs avantages, notamment en termes de robustesse

au bruit ainsi qu'aux variations de luminosité ainsi qu'en termes de gestion des discontinuités et des objets masqués. Nous avons également fait le choix d'implémenter et d'optimiser la méthode de Horn-Schunck à titre de comparaison.

3.2 Architectures GPU visées

Comme nous l'avons vu dans le chapitre précédent, les architectures GPU sont particulièrement adaptées aux algorithmes extrêmement parallèles (« embarrasingly parallel » en anglais). L'objectif du code à écrire est de tirer parti des centaines d'unités d'exécutions présentes dans les GPU pour exécuter un maximum d'instructions en parallèle. Une parallélisation grossière de l'algorithme à traiter n'est cependant pas suffisante pour atteindre les meilleures performances sur ces architectures. Il est nécessaire de prendre en compte les multiples spécificités des architectures GPU ainsi que d'être attentif aux limitations physiques présentes dans les plateformes visées.

Dans les travaux de cette thèse, nous avons utilisé les cartes embarquées Jetson de NVIDIA. Ces cartes introduites en 2014 intègrent des SoC NVIDIA Tegra contenant chacun un CPU et un GPU. Nous avons utilisé les cartes AGX Xavier, TX2 et Nano, listées en ordre décroissant de puissance de calcul dans le tableau 3.3. Pour compléter notre description, nous citerons également les anciennes et les nouvelles cartes non utilisées dans nos travaux. Du côté des anciennes cartes, les cartes TK1 et TX1, introduite en 2014 et 2015 respectivement, ne sont plus disponibles à la vente. Pour les cartes les plus récentes, les cartes TX2 NX et Xavier NX sont disponibles depuis 2022 et sont des variantes des cartes de même nom au format de connexion « edge » (et moins puissantes pour le cas de la Xavier NX). Une nouvelle génération de carte Jetson, la gamme de carte Orin, est également disponible depuis 2022. À chaque modèle de puce GPU est associé une version de « Compute Capability » ou Capacité de Calcul (CC). Ce numéro de capacité de calcul permet d'identifier les caractéristiques et les spécificités associées à ce modèle de puce.

Carte Embarquée	**Modèle de puce (Microarchitecture)**	**Compute Capability**	**Configuration #SM/#Cœurs**	**Plage de fréquences**
Jetson Nano	GM20B (Maxwell)	5.3	1/128	77–922 MHz
Jetson TX2	GP10B (Pascal)	6.2	2/256	115-1300 MHz
Jetson AGX Xavier	GV10B (Volta)	7.2	8/512	115-1377 MHz

Tab. 3.3 : Comparaison des architectures GPU des cartes Jetson Nano, TX2 et AGX Xavier.

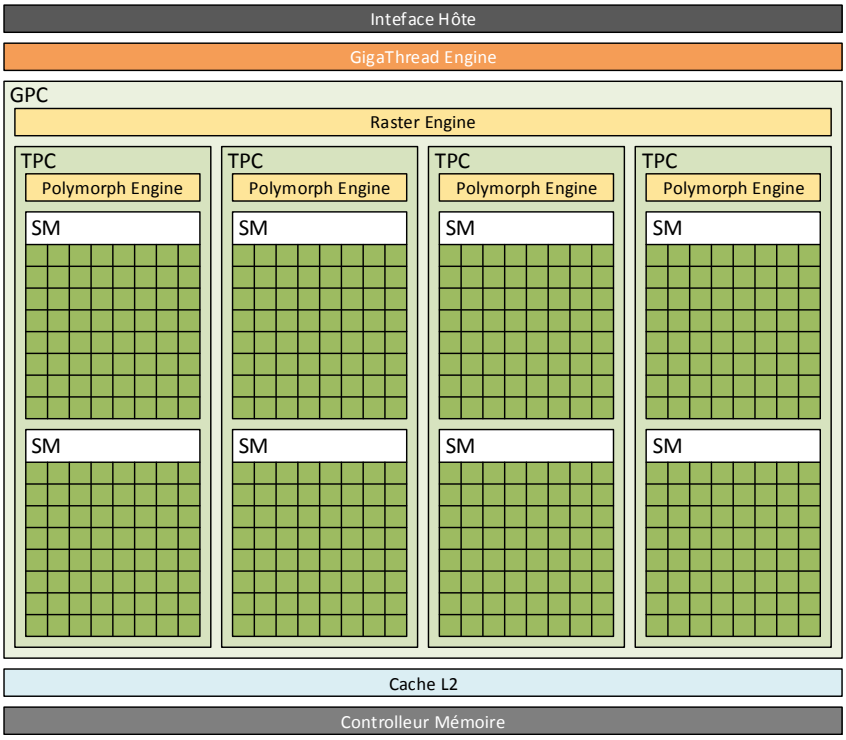


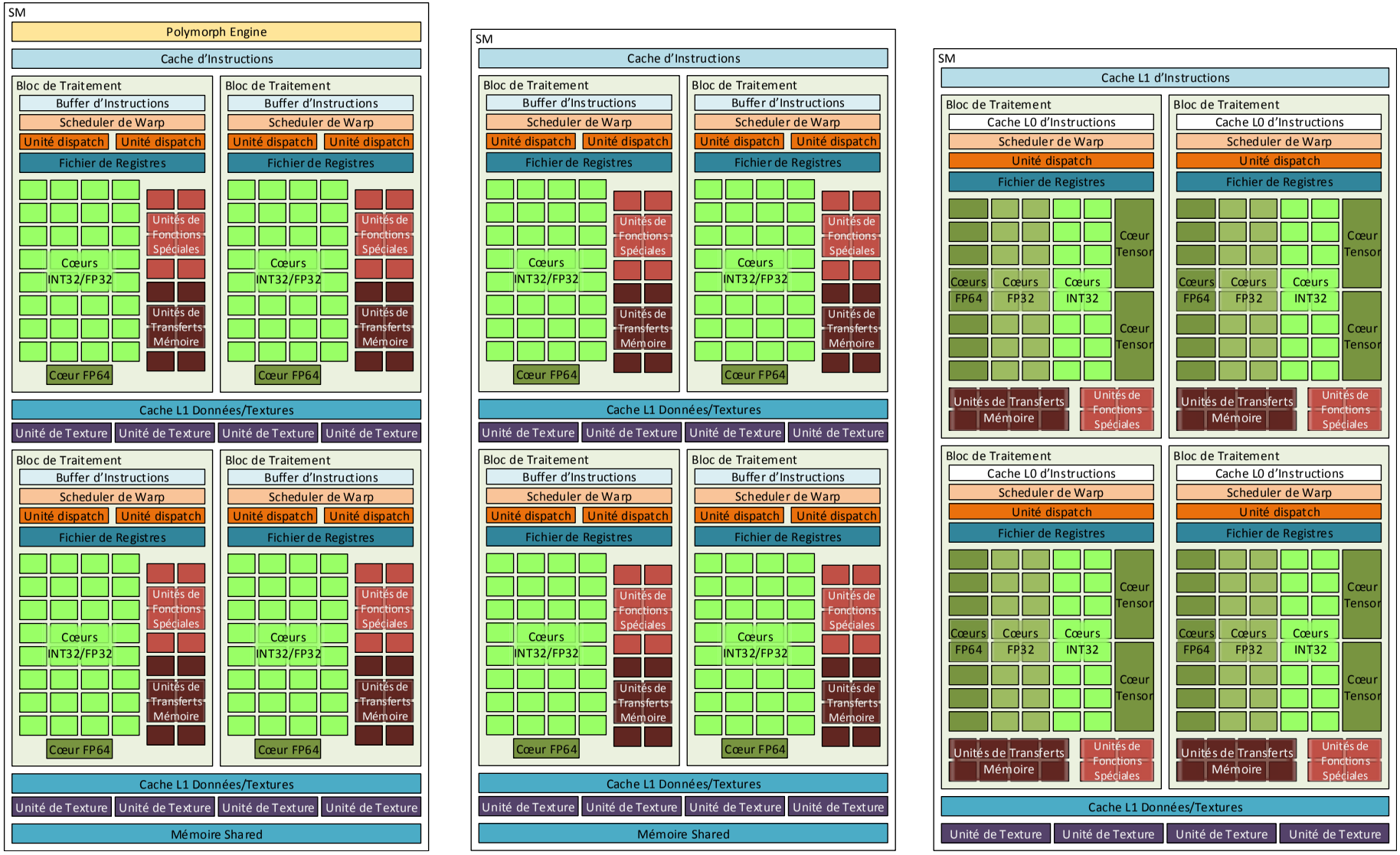
Fig. 3.1 : Schéma fonctionnel de l'organisation haut-niveau d'un GPU type de NVIDIA.

L'architecture de ces GPU est construite autour d'une hiérarchie contenant des composants en nombre variable selon le modèle de puce. Cette hiérarchie haut-niveau commune aux GPU NVIDIA est illustrée en figure 3.1. On remarque 4 niveaux hiérarchiques dans la figure 3.1 :

1. Le premier niveau le plus haut regroupe les éléments uniques à un GPU. Ce niveau comprend ainsi :
   - l'*interface hôte* qui est le lien de communication entre le CPU et le GPU : c'est par ce lien que le CPU envoie des instructions et contrôle le GPU ;
   - le *contrôleur mémoire* qui est lien entre le GPU et la mémoire principale, qui peut être propre au GPU ou partagée avec le CPU ;
   - le *cache de données de niveau L2* qui est unique et partagé avec tout le reste du GPU ;
   - le *GigaThread Engine* qui est chargé d'ordonnancer et de planifier le travail à faire sur l'ensemble du GPU.

2. Le niveau suivant est celui du ou des *Graphics Processing Clusters* (GPC). Le GPU est ainsi organisé en un tableau de GPC, reliés ensemble par un réseau de communication de type « crossbar ». Ce niveau contient également des composants se chargeant de la matricialisation des éléments graphiques pour les afficher sur un écran (on parle de « rasterization » en anglais). Ces composants forment ainsi le *Raster Engine*.
3. Le niveau suivant est celui du *Texture Processing Cluster* (TPC). Le GPC précédent regroupe 1 ou plusieurs TPC. Ce niveau contient également le *PolyMorph Engine* chargé de diviser les objets graphiques polygonaux en structures capables d'être rendues visuellement et affichées sur un écran (on parle de tesselation en anglais).
4. Enfin, le plus petit niveau de cette hiérarchie correspond au *Streaming Multiprocessor* (SM). Chaque TPC contient en général 2 SMs. C'est dans cette unité hiérarchique que sont contenus les cœurs CUDA de calcul. Le terme de cœur CUDA est légèrement ambigu. Les cœurs physiques de calcul dans le GPU sont hétérogènes et opèrent sur différents types de nombres (demi-précision, simple-précision, double-précision, entiers...). Le terme de cœur CUDA se réfère généralement seulement aux cœurs de calcul simple-précision. Dans la suite de la thèse, nous utiliserons ce terme pour désigner les cœurs de calculs présents dans le GPU, indépendamment du type de nombre utilisé. Ce sont les SM qui varient le plus de générations en générations de GPU. Dans les sous-sections suivantes, nous présenterons plus en détails ce composant.

Les GPU sont ainsi des ensembles extensibles de SM reliés entre eux et avec la mémoire principale du système, externe au GPU. Nous présentons maintenant les spécificités des SM des 3 GPU des cartes Jetson utilisées.



(a) Jetson Nano (génération Maxwell).

(b) Jetson TX2 (génération Pascal)

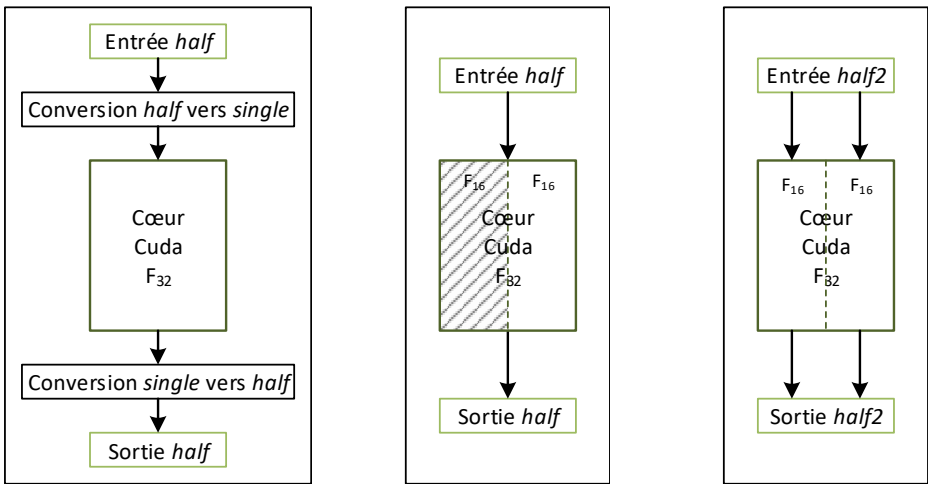
(c) Jetson AGX Xavier (génération Volta)

Fig. 3.2 : Schéma fonctionnel de l'organisation des SM des cartes Jetsons.

3.2.1 GPU de la Jetson Nano

Le GPU de la Jetson Nano est un GPU de micro-architecture « Maxwell 2.0 » GM20B, similaire à celui de la carte obsolète Jetson TX1, mais avec $2\times$ moins de cœurs de calculs et une légère réduction de fréquence horloge. Il s'agit de la génération de GPU la plus vieille parmi les cartes NVIDIA Jetson à la vente aujourd'hui et utilise le procédé de fabrication « 20 nm TSMC ». L'organisation du GPU est représentée en figure 3.2a. Par rapport à la génération précédente « Kepler », cette micro-architecture comporte un nombre de cœurs de calculs par SM divisible par la taille d'un warp de 32 threads, un groupe de threads de taille minimal exécutant la même instruction. Alors que dans la génération Kepler certains cœurs ne permettaient que d'exécuter des warps partiels, ce nouvel alignement du nombre de cœurs sur la largeur d'un warp améliore le partitionnement de la puce. Ce meilleur partitionnement a pour effet de faire des économies en surface de puce ainsi qu'en consommation énergétique. Les cœurs CUDA de calculs sont capables de réaliser nativement des opérations sur des nombres entiers sur 32 bits, mais aussi sur des nombres flottants simple-précision sur 32 bits. Ces unités sont également utilisées pour le calcul de nombres flottant demi-précision. Il n'existe cependant que 4 unités de calculs double précision par SM.

La capacité de calcul 5.3 est la première à implémenter matériellement le calcul demi-précision. Auparavant, ce type n'était utilisé que pour le stockage. Une conversion était requise pour réaliser des calculs, comme indiqué dans le schéma 3.3a. Depuis cette génération de carte, il est possible de réaliser matériellement les opérations de calculs sur un ou deux nombres de type *half* en même temps, comme on peut le voir sur les schémas 3.3b et 3.3c. On parle alors de parallélisme de sous-mots : les unités de calculs voient en entrée un type *half2* de 32 bits dans lequel se trouvent deux nombres *half*, l'un occupant la moitié haute des bits et l'autre occupant la moitié basse. Elles produisent ainsi deux résultats dans chaque partie du type *half2* en sortie. L'utilisation de ce type vectoriel est nécessaire pour tirer de la pleine puissance des unités d'exécution 32 bits. Dans le meilleur des cas, la conversion d'un programme simple précision *float* vers la demi-précision *half2* permet des gains de $\times 3$ en temps d'exécution, sans tenir compte des éventuels gains en termes de bande passante mémoire [HW17]. Dans le cas où des nombres non-vectoriels *half* sont utilisés, peu ou pas de gains en temps sont mesurés dans la plupart des cas [HW17]. L'utilisation optimale des nombres demi-précision nécessite donc une réécriture fine du code afin de tirer parti du parallélisme et du débit qu'offre le type *half2*.



(a) Cas sur les anciennes architectures NVidia (avant CC 5.3).

(b) Cas non-vectoriel : les nombres *half* ont le même débit que le nombre *float*.

(c) Cas vectoriel : les nombres *half* ont $2\times$ plus de débit que le nombre *float*.

Fig. 3.3 : Opérations sur des nombres demi-précisions dans les cœurs CUDA. Ces schémas sont tirés de la publication [HW17].

Les principales spécifications architecturales ayant un impact direct sur le lancement des blocs de threads sont indiquées en tableau 3.4. Ce sont ces valeurs qui déterminent si un kernel peut être lancé sur le GPU. Si trop de ressources sont demandées, le kernel n'est pas lancé.

#SM	#Cœurs par SM	Max #Registres 32 bits			Max Mémoire *Shared*	
		par SM	par Bloc	par Thread	par SM	par Bloc
1	128	65536	32768	255	64 Ko	48 Ko

Tab. 3.4 : Principales spécifications techniques du GPU de la Jetson Nano influant sur le lancement ou non des kernels CUDA.

Enfin, on peut également lister les quantités de caches de données disponibles sur le GPU. Nous listons les quantités totales de mémoire cache selon les différents niveaux en tableau 3.5 ainsi que la quantité disponible par cœurs CUDA. À titre de comparaison, nous ajoutons également ces mêmes quantités pour le CPU présent sur la carte. Le CPU de cette carte est composé de 4 cœurs ARM Cortex-A57. On remarque une nette différence en termes de quantité mémoire disponible par cœur entre le CPU et le GPU. Il y a $256\times$ plus de mémoire cache L2 et $43\times$ plus de mémoire cache L1 par cœurs du côté CPU que du côté GPU.

Architecture	Cores	Cache L2		Cache L1	
		Total	Par cœur	Total	Par cœur
GPU	128	256 Ko	2 Ko	24 Ko	192 o
CPU	4	2048 Ko	512 Ko	32 Ko	8192 o

Tab. 3.5 : Quantités de cache disponible pour le CPU et pour le GPU de la Jetson Nano.

3.2.2 GPU de la Jetson TX2

La carte Jetson TX2 est dotée d'un GPU de micro-architecture « Pascal » GP10B. L'organisation du GPU est représentée sur la figure 3.2b. Concernant l'organisation des SM, cette génération est identique à la génération précédente. Cette génération garde un nombre de cœurs puissance de 2 pour coïncider avec des warps de 32 threads : il n'y a pas de cœurs de calcul réalisant l'exécution d'un « demi-warps » de 16 threads. Les évolutions par rapport à la génération précédente concernent principalement le procédé de fabrication qui est maintenant le procédé « 16 nm TSMC ». Cette réduction de taille – ou encore « die shrink » en anglais – permet d'intégrer davantage de SM sur une puce de même surface.

On notera que cette génération de GPU est scindée en 3 familles d'organisation de SM, chacune correspondant à une version de Capacité de Calcul différente.

La première sous-famille correspond au CC 6.0 et est composée des cartes professionnelles GPGPU Quadro GP100 et Tesla P100. Les SM de ces cartes comprennent 64 cœurs de calculs simple-précision F_{32} et 32 cœurs double-précision FP64 organisés en 2 deux sous-blocs de traitement de 32 cœurs F_{32} et 16 cœurs FP64 chacun. Il y a au total 64 cœurs simple précision par SM. Ces cœurs simple précision sont également capables de traiter deux nombres demi-précisions F_{16} en même temps, comme les cœurs de la génération Maxwell précédente.

La deuxième sous-famille correspond au CC 6.1 et est composée du reste des cartes non-embarquées de la génération Pascal. Les SM sont organisés différemment. Ils contiennent 4 sous-blocs de traitement pour un total de 64 cœurs F_{32} et donc un débit de calcul simple précision $2\times$ plus élevé par SM. De plus, contrairement au CC 6.0, il n'y a qu'un seul cœur de calcul FP64 par bloc pour un total de 4 par SM, soit le même nombre que pour la génération précédente. Le débit de calcul double précision par SM est divisé par 8. La plus grosse différence concerne la capacité de calcul demi-précision F_{16}. Dans cette sous-famille, les cœurs de calculs ne sont pas capables de réaliser des opérations sur des nombres demi-précision. Afin de garantir une compatibilité en termes de code, il existe une seule unité de calcul F_{16} par SM. Le débit demi-précision par SM est divisé par 64.

Enfin, la troisième sous-famille correspond au CC 6.2 des cartes embarquées Jetson TX2. Cette famille reprend l'organisation en 4 sous-blocs de traitement des GPU de CC 6.1 pour un total de 128 cœurs simple précision et 4 cœurs double-précision par SM. Elle reprend également le comportement demi-précision des cœurs des GPU du CC 6.0 : les cœurs simple précision sont capables de traiter deux nombres

demi-précision en même temps. C'est cette organisation qui est représentée sur la figure 3.2b.

Tout comme la carte précédente, nous listons dans le tableau 3.6 les principales spécifications architecturales ayant un impact direct sur le lancement des blocs de threads. On remarque que ces contraintes sont identiques à la génération précédente.

#SM	#Cœurs par SM	Max #Registres 32 bits			Max Mémoire *Shared*	
		par SM	par Bloc	par Thread	par SM	par Bloc
2	128	65536	32768	255	64 Ko	48 Ko

Tab. 3.6 : Principales spécifications techniques du GPU de la Jetson TX2 influant sur le lancement ou non des kernels CUDA.

Enfin, tout comme la carte précédente, nous listons les quantités de caches de données disponibles sur GPU et sur CPU en tableau 3.7 ainsi que la quantité disponible par cœurs CUDA. Les CPU de la Jetson TX2 sont hétérogènes : il y a 2 cœurs « Denver2 » – des processeurs développés par NVIDIA implémentant le jeu d'instruction ARMv8-A 64/32-bit – et 4 cœurs ARM A57. Il y a également ici plus de mémoires par cœur disponible du côté CPU. Il y a entre $512\times$ et $256\times$ plus de mémoire cache L2 et entre $85\times$ et $64\times$ plus de mémoire cache L1 du côté CPU que du côté GPU – selon que l'on utilise les cœurs Denver2 ou les cœurs ARM A57.

Architecture	Cœurs	Cache L2		Cache L1	
		Total	Par cœur	Total	Par cœur
GPU	256	512 Ko	2 Ko	48 Ko	192 o
CPU	2× Denver2	2048 Ko	1024 Ko	32 Ko	16384 o
	4× ARM A57	2048 Ko	512 Ko	48 Ko	12288 o

Tab. 3.7 : Quantités de cache disponibles pour le CPU et pour le GPU de la Jetson TX2.

3.2.3 GPU de la Jetson AGX Xavier

La carte Jetson Xavier AGX est quant à elle dotée d'un GPU de micro-architecture « Volta » GV10B (certaines documentations [NVI20] NVIDIA font référence à un modèle GV11B pour certaines nouvelles cartes sans mentionner de différences). Le CC de ce GPU est 7.2. Les modèles non-embarqués de cette génération de carte ont un CC de 7.0. Ces modèles non-embarqués sont identiques en termes d'organisation que les GPU embarqués, hormis des caches plus grands et plus de mémoire *Shared* possible par bloc de threads. L'organisation des SM diffère sensiblement de celle des générations précédentes.

Comme on peut le voir en figure 3.2c, l'architecture Volta conserve l'organisation en 4 sous-blocs de traitement, mais avec une seule unité de dispatch. Volta perd ainsi la capacité d'émettre deux instructions indépendantes au même coup d'horloge. Le GPU est cependant toujours capable d'émettre une instruction indépendante pour 1 thread à chaque cycle, ce qui permet de cacher la latence d'instruction. Concernant les cœurs de calcul, ils sont scindés non plus en 2, mais 3 sortes : un type de cœur spécifique pour les nombres entiers sur 32 bits (INT32), un pour les nombres flottants sur 32 bits (F_{32}) et un pour les nombres flottants sur 64 bits (FP64). Les cœurs F_{32} sont également toujours capables de traiter 2 nombres flottants demi-précision F_{16} en même temps. Par ailleurs, il y a des différences en nombre de cœurs d'un seul type par SM. Ainsi, il y a une réduction des cœurs F_{32} et des cœurs INT32 à 64 pour chaque type, mais une augmentation du nombre de cœurs FP64 à 32. Cette réduction du nombre d'unités simple précision est cependant accompagnée d'une augmentation du nombre de SM par GPU afin d'obtenir une augmentation globale du nombre de cœurs par GPU.

Enfin, on notera l'ajout d'un nouveau type d'unité de calculs appelé « *Cœur Tensor* ». Ces unités de calculs ont été conçues pour réaliser des opérations multiplication-et-accumulation $A \cdot B + C$ sur des matrices 4×4 F_{16}. L'accumulation est faite avec une matrice F_{32} et le résultat peut être ensuite abaissé vers une matrice F_{16}. Le but de ces unités est ainsi d'accélérer les calculs utilisés dans les réseaux de neurones. L'accès et la programmation à ces unités se fait via le biais des bibliothèques NVIDIA cuBLAS et cuDNN et plus récemment via CUDA directement (depuis la version 9.0).

D'un point de vue de la hiérarchie mémoire, il existe également des différences :

— Le cache L1 de données et la mémoire *Shared* ont été unifiés et se partagent le même emplacement mémoire physique,

— Il existe désormais un vrai cache L0 d'instructions pour chaque sous-bloc de traitement ainsi qu'un cache L1 par SM.

Nous listons dans le tableau 3.8 les principales spécifications architecturales ayant un impact direct sur le lancement des blocs de threads. Ce tableau permet également d'illustrer les différences matérielles par rapport à la génération précédente de GPU.

Enfin, comme pour la carte précédente, nous listons les tailles des caches de données sur GPU et sur CPU dans le tableau 3.9, ainsi que la quantité disponible par cœur CUDA. Le CPU de la carte est composé de 8 cœurs homogène « Carmel », organisés en 4 paires. Ces cœurs sont issus du projet Denver de NVIDIA et utilisent

#SM	#Cœurs par SM	Max #Registres 32 bits			Max Mémoire *Shared*	
		par SM	par Bloc	par Thread	par SM	par Bloc
8	64	65536	65536	255	128 Ko [5]	48 Ko [5]

Tab. 3.8 : Principales spécifications techniques du GPU de la Jetson AGX Xavier influant sur le lancement ou non des kernels CUDA.

le jeu d'instructions ARMv8.2-A. Par rapport à la carte précédente, on remarque ici une diminution de la quantité de cache L2 par cœur GPU : il y en a moitié moins. Au total, il y a $512\times$ plus de cache L2 disponible par cœur entre le CPU et le GPU. Concernant le cache L1, bien qu'il semble y en avoir plus de $10\times$ plus que sur les cartes précédentes, il ne faut pas oublier que ce cache L1 de données se partage le même espace de 128 Ko pour chaque SM. Si un kernel requiert trop de mémoire *Shared*, il ne restera plus d'espace pour le cache L1 de données. Néanmoins, dans le cas où un kernel ne nécessite pas de mémoire *Shared*, il y a $4\times$ plus de cache L1 disponible par cœur entre le CPU et le GPU.

Architecture	Cores	Cache L2		Cache L1	
		Total	Par cœur	Total	Par cœur
GPU	512	512 Ko	1 Ko	1024 Ko [5]	2048 o [5]
CPU	4	2048 Ko	512 Ko	32 Ko	8192 o

Tab. 3.9 : Quantités de cache disponible pour le CPU et pour le GPU de la Jetson Xavier.

3.2.4 Synthèse

Comme nous l'avons vu dans cette section, bien que nous nous concentrions sur 3 plateformes embarquées de la même famille et de la même marque, il peut y avoir des variations importantes d'une carte à l'autre. C'est particulièrement vrai sur la carte de calcul la plus récente – la Jetson AGX Xavier – qui, en plus de comporter des évolutions architecturales majeures, est la carte qui comporte le GPU le plus grand en nombre de cœurs de calculs et le plus rapide en fréquence d'horloge. Ces particularités architecturales sont ainsi à prendre en compte dans notre travail d'optimisation. En effet, il est possible qu'un schéma d'optimisation se comportant bien sur une carte ne soit pas optimale sur une autre. Il sera important de diversifier nos optimisations et surtout de les tester sur les cartes afin de déterminer les meilleures sur chacune des cartes.

Les deux principales contraintes physiques limitant les performances des programmes sont :

5. Le cache L1 et la mémoire *Shared* se partage le même espace de 128 KB par SM.

— la vitesse et la quantité d'instructions traitées par le processeur par unité de temps. Si un programme est subordonné au temps de calcul (« compute bound » en anglais), une augmentation des performances du processeur ou une réduction du nombre de calculs suffisent à réduire le temps de traitement.
— la vitesse et la quantité de mémoire à disposition du programme. Si les performances d'un programme sont limitées par la mémoire (« memory bound » en anglais), la diminution des accès à la mémoire ou l'utilisation de mémoires plus rapides ou ayant une plus grande bande passante suffisent à accélérer le programme.

Les optimisations faites par le programmeur vont donc jouer sur ces deux facteurs afin d'obtenir les meilleures performances à la fois en puissance de calcul mais aussi en bande passante mémoire. Une métrique à utiliser est *l'intensité arithmétique* (IA) correspondant au rapport entre le nombre d'opérations flottantes et le nombre d'accès en mémoire :

$$\text{IA} = \frac{\#\ \text{FLOP}}{\#\ \text{Accès}}. \tag{3.2}$$

Le but est ainsi d'obtenir le code avec la plus haute intensité arithmétique possible en limitant le nombre d'opérations et en diminuant les accès mémoires dans le code.

Pour ce faire, plusieurs techniques de plus ou moins haut niveau sont à la disposition du programmeur. La section suivante a pour but de présenter ces techniques afin de mieux comprendre les versions optimisées de nos algorithmes d'estimation du flot optiques présentées dans le chapitre suivant. Nous suivons ici un ordre de présentation des optimisations des plus généralistes possibles aux optimisations nécessitant une exploration sur les plateformes visées.

3.3 Transformations algorithmiques (HLT)

Les techniques de transformations dites de haut niveau – « High-Level Transforms » (HLT) en anglais – regroupent l'ensemble des optimisations impliquant un changement de code profond difficilement réalisable par un compilateur optimisant. Certaines de ces optimisations peuvent casser l'arbre des dépendances des instructions du code d'origine ainsi que certaines barrières de synchronisation initiales. La tâche du programmeur est donc de s'assurer que ce relâchement de contraintes produit un code valide. Ces transformations sont indépendantes du langage de programmation, elles interviennent la plupart du temps sur la sémantique du code.

Dans notre travail d'optimisation, nous avons principalement utilisé deux transformations :

— la *fusion d'opérateurs* ou de fonctions qui consistent à utiliser directement des résultats intermédiaires avant leur écriture en mémoire,

— le *pipeline d'opérateurs* qui consiste à réutiliser les résultats intermédiaire le plus tôt possible avant le traitement complet des données en utilisant un chaînage particulier d'opérations dans le code.

Ces optimisations ont pour but de maximiser la réutilisation des données en cache (voire en registre), mais peuvent nécessiter une augmentation du nombre d'opérations à faire.

En général, on retrouve ce type d'optimisations sur CPU, notamment VLIW [Lam88] ainsi que SIMD [Lac+14 ; Lac+09]. On les retrouve également dans le domaine des FPGAs, là où le programmeur a un contrôle fin des unités de calculs, des éléments mémoires et de leurs interconnexions [BCL22 ; BCL21 ; Ye+13 ; Ye+12]. Ce genre d'optimisations se retrouvent désormais sur GPU, notamment la fusion d'opérateurs [Fil+15] ou encore plus généralement la bonne maîtrise de la parallélisation des boucles [Tad20].

3.3.1 Fusion d'opérateurs

La plupart des programmes sont constitués d'une suite d'opérations plus ou moins complexes sur des données communes. Le but de cette transformation est de factoriser au maximum le nombre d'opérations dans le code afin de réutiliser un maximum de données dans les registres du processeur. La diminution du nombre d'opérations fait également baisser le coût des appels de fonctions (diminution de la taille de la pile d'appel des fonctions, diminution du temps d'échauffement GPU notamment). Pour ce faire, il est nécessaire d'identifier le schéma d'accès aux données en entrée et en sortie de chaque opérateur. Cela est nécessaire pour déterminer la bonne composition des opérateurs qui respecte les dépendances de données.

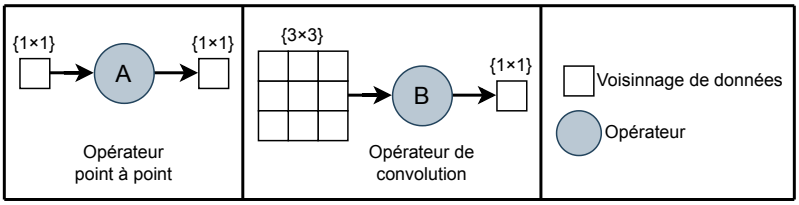


Fig. 3.4 : Représentation schématique des opérateurs et de leurs dépendances de données.

Afin d'illustrer les règles de fusion des opérateurs, nous utilisons la représentation schématique présentée en figure 3.4. Afin de simplifier les notations, nous utilisons la notation de la composition de fonctions $\circ$. Pour rappel, cet opérateur prend deux fonctions en entrées f et g et produit la fonction $g \circ f(x) = g(f(x))$ en sortie. Nous nous concentrons sur des opérateurs 2D de par leur grande utilisation dans le domaine du traitement image. Un opérateur prend ainsi un voisinage de points de tableaux 2D en entrée et produit un autre voisinage de points en sortie. Dans le cas d'un voisinage correspondant à un seul point, on parle d'un opérateur ponctuel et dans le cas d'un voisinage plus grand, on parle d'un opérateur à voisinage ou encore « stencil » en anglais.

On peut différencier 4 cas de bases illustrés en figure 3.5.

1. Le premier cas correspond à la fusion de 2 opérateurs ponctuels. C'est le cas le plus simple : on peut enchaîner directement un appel de la fonction F suivi d'un appel de la fonction G.
2. Le deuxième cas correspond à la fusion d'un stencil avec un opérateur ponctuel. La fusion se fait également directement en enchaînant la sortie ponctuelle de la fonction F avec l'entrée de la fonction G.
3. Le troisième cas correspond à la fusion d'un opérateur ponctuel suivi d'un stencil. Dans ce cas, il faut appeler la fonction ponctuelle F sur chaque point d'entrée du stencil G. Pour un voisinage 3×3 il faudra donc appeler la fonction F 9 fois.
4. Le quatrième cas correspond à la fusion de 2 stencils F et G. C'est le cas le plus complexe étant donné que comme précédemment, il faut appeler la fonction F sur tous les points en entrée de la fonction G mais il faut également prendre en compte le voisinage de la fonction F. Dans le cas de 2 stencils 3×3, il faut appeler la fonction F 9 fois. L'union des voisinages en entrée des stencils aura ainsi une taille de 5×5.

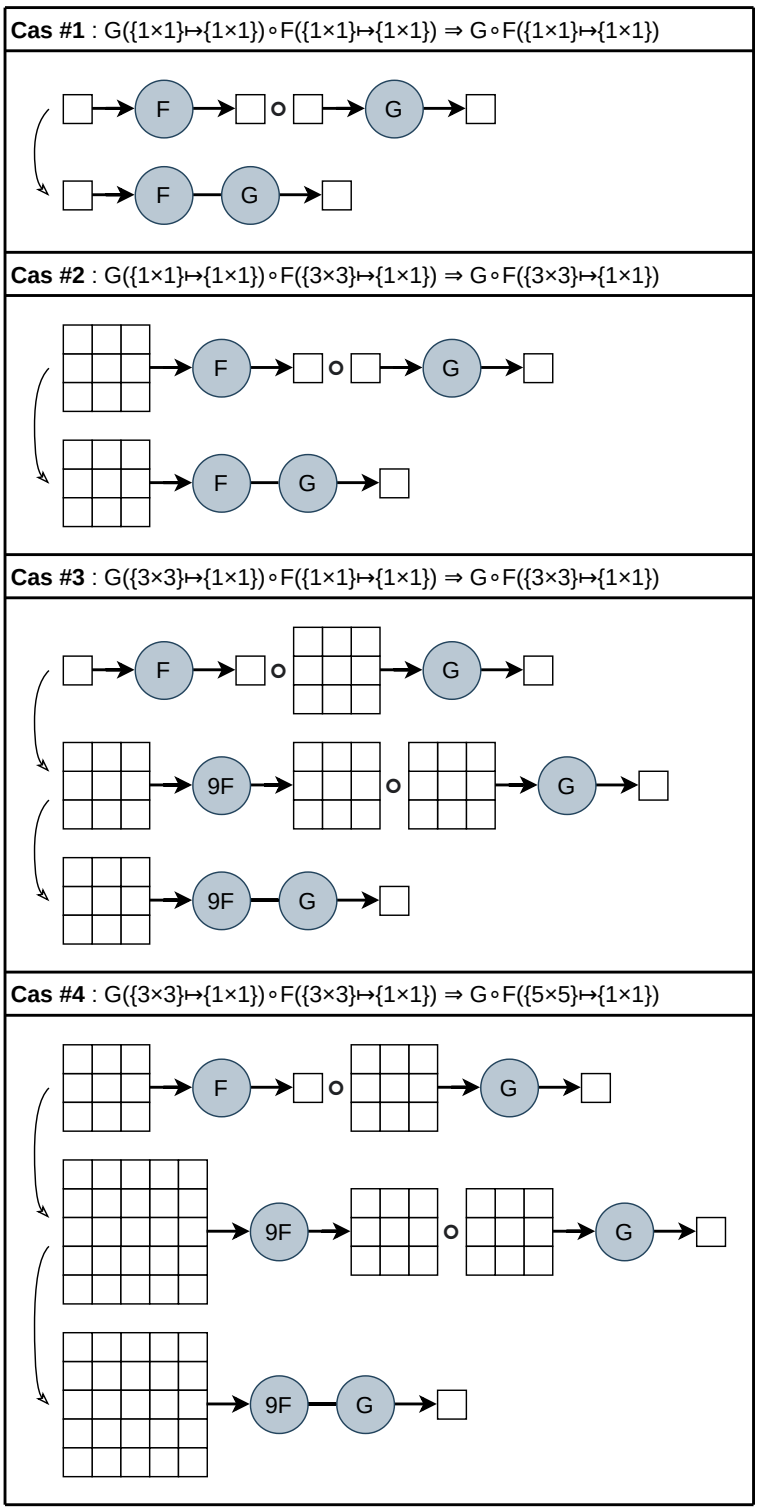


Fig. 3.5 : Cas de base de la fusion d'opérateurs. Des opérateurs ponctuels et des convolutions 3×3 sont utilisés en exemple. Dans la représentation schématique des dépendances spatiales, on notera une inversion de l'ordre des entrées de l'opérateur de composition afin de le faire coïncider avec l'enchainement visuel des fonctions.

Les impacts des transformations pour les 4 cas de base sont présentés dans le tableau 3.10. On notera que pour les deux premiers cas, ces transformations améliorent toujours les performances étant donné qu'il y a suppression des accès mémoires intermédiaires et un meilleur enchainement des opérateurs réalisant le traitement souhaité. Les deux derniers cas n'entrainent pas toujours une amélioration des performances. En effet, ces cas génèrent plus de calculs afin d'avoir suffisamment de points prêts pour l'exécution du deuxième opérateur de même taille ou plus grand que le premier. Ces calculs supplémentaires peuvent également entrainer le chargement de plus de points en mémoire comme dans le quatrième cas qui nécessite à la fois plus de calculs mais aussi plus d'accès mémoires qu'avant la fusion. Cependant, il peut rester intéressant de réaliser cette transformation pour des stencils de grande taille, notamment dans le cas où les données calculées peuvent être réutilisées dans d'autres calculs (notamment pour le calcul de données voisines) et dans le cas où les opérations et les accès mémoires se font sur des unités et des mémoires rapides (comme les caches).

Cas	F	G	Avant fusion		Après fusion	
			Mem	OP	Mem	OP
#1	$\{1\times1\} \mapsto \{1\times1\}$	$\{1\times1\} \mapsto \{1\times1\}$	4	2	2	2
#2	$\{3\times3\} \mapsto \{1\times1\}$	$\{1\times1\} \mapsto \{1\times1\}$	12	2	10	2
#3	$\{1\times1\} \mapsto \{1\times1\}$	$\{3\times3\} \mapsto \{1\times1\}$	12	2	10	10
#4	$\{3\times3\} \mapsto \{1\times1\}$	$\{3\times3\} \mapsto \{1\times1\}$	20	2	26	10

Tab. 3.10 : Impact de la fusion d'opérateurs sur les accès mémoires et sur le nombre d'opérations.

3.3.2 Pipeline d'opérateurs

Comme nous l'avons vu précédemment, la fusion d'opérateurs peut augmenter le nombre d'accès ainsi que le nombre d'appels aux fonctions réalisant le traitement. Cela peut impliquer une augmentation du temps d'exécution du programme et ainsi une réduction des performances. De plus, dans le cas d'opérateurs non-réguliers (c.-à-d. qui ne sont pas assimilables à des convolutions comme par exemple les filtres récursifs), la fusion d'opérateurs peut ne pas être possible. Pour pouvoir améliorer la localité mémoire malgré tout, il est cependant possible de réaliser un pipeline d'opérateurs. Cette opération s'effectue sur les boucles non-fusionnables du programme. Elle consiste en une ré-écriture du code afin qu'une partie des données produites par un premier opérateur soit utilisée au plus tôt par un deuxième sans que le premier ait terminé son traitement sur l'ensemble de ses données d'entrée. À la différence du pipeline logiciel [All+95], le but n'est pas ici de réaliser un maximum

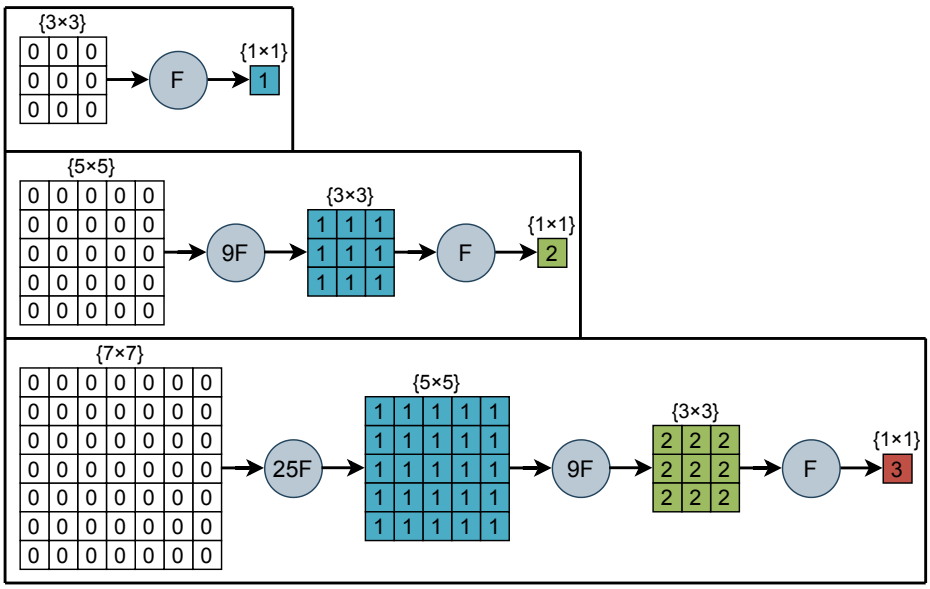


Fig. 3.6 : Croissance des dépendances des données pour le calcul d'un pixel pour un stencil 3×3 itéré 1, 2 et $3\times$. Les chiffres indiquent les itérations dans lesquelles se trouvent les données avant et après applications des stencils.

d'opérations en parallèle mais d'améliorer la rapidité des accès mémoire. Dans le cadre du traitement image, les données sont le plus souvent organisées en tableaux 2D. Pour améliorer la localité mémoire dans les caches, le pipeline est alors le plus souvent réalisé sur les lignes du tableau à traiter : dès qu'une ligne est produite par un opérateur, elle est consommée par l'opérateur suivant.

Pour illustrer cette transformation, nous reprenons l'exemple d'un stencil 3×3 itéré plusieurs fois sur un tableau 2D. Le pipeline va servir ici à superposer les appels à l'opérateur au lieu d'attendre la fin de traitement complet du tableau en entrée avant de passer à une itération suivante d'appel de l'opérateur.

Comme dans le cas 4 des exemples de bases de la fusion d'opérateurs, la figure 3.6 montre que la fusion de plusieurs itérations fait augmenter de manière considérable les dépendances de données pour le calcul d'un seul point :

— On a un opérateur stencil nécessitant 10 accès mémoires pour 1 appel.

— Pour la fusion de 2 itérations de cet opérateur, on passe de 20 accès mémoires et 2 appels aux opérateurs à 26 accès mémoires et 10 appels aux opérateurs.

— Pour la fusion de 3 itérations de cet opérateur, on passe de 30 accès mémoires et 3 appels aux opérateurs à 50 accès mémoires et 35 appels aux opérateurs.

En effet, nous avons une croissance cubique des dépendances données fonction du nombre d'itérations fusionnées.

Nous voulons calculer la complexité ponctuelle du nombre de pixels total à calculer/accéder pour mettre à jour une zone centrale de $w \times h$ pixels en fonction du nombre p d'itérations du stencil. Nous posons $C((w,h),p)$ cette complexité. La zone à calculer correspond à l'enchaînement de zones rectangulaire de données. Nous avons ainsi :

$$\begin{aligned} C((w,h),p) &= \sum_{j=1}^{p}(j+w+j)(j+h+j) \\ &= \sum_{j=1}^{p} 4j^2 + (2h+2w)j + wh \\ &= 4\sum_{j=1}^{p} j^2 + (2h+2w)\sum_{j=1}^{p} j + wh\sum_{j=1}^{p} 1 \\ &= \frac{4p(2p+1)(p+1)}{6} + (2h+2w)\frac{p(p+1)}{2} + whp \\ &= \frac{8p^3 + (6h+6w+12)p^2 + (6h+6w+4)p}{6} \\ &= \frac{4}{3}p^3 + (h+w+2)p^2 + (h+w+wh+\frac{2}{3})p. \end{aligned} \tag{3.3}$$

On obtient bien une croissance cubique.

Un pipeline d'opérateurs peut donc être intéressant pour plusieurs itérations de cet opérateur. On introduit le terme de *profondeur de pipeline* qui correspond aux nombres d'itérations totales réalisées par le pipeline sur les données en entrées. Le traitement va également se dérouler sur les lignes du tableau d'entrée. Il faut donc utiliser une variante de l'opérateur réalisant le traitement sur une ligne complète comme indiqué en figure 3.7.

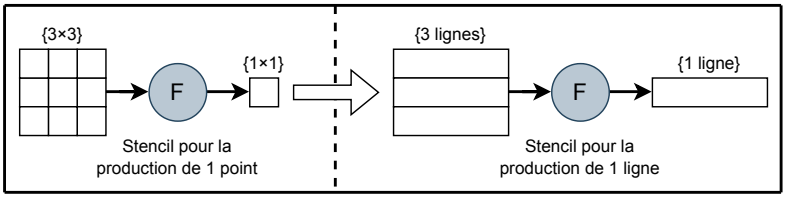


Fig. 3.7 : Transformations du stencil 3×3 en un opérateur sur 3 lignes.

Cet opérateur va donc s'enchaîner sur les lignes de façon à consommer les lignes produites au plus tôt sans traiter le tableau au complet avant de passer à l'appel suivant du stencil. La figure 3.8 représente les symboles utilisés dans la suite des schémas concernant le pipeline.

Le pipeline est décomposé en 3 parties :

— le *prologue* qui se charge d'initialiser le pipeline de manière spécifique de par les dépendances de données ;

— le *régime permanent* qui constitue la boucle itérative principale du pipeline : les dépendances nécessaires à la production d'une nouvelle ligne sont calculées dans le corps de la boucle, et la boucle itérée pour le traitement du tableau d'entrée complet ;

— l'*épilogue* éventuellement nécessaire pour le traitement des dernières lignes du tableau. Si ce dernier n'est pas nécessaire, le régime permanent est simplement étendu pour inclure toutes les lignes du tableau.

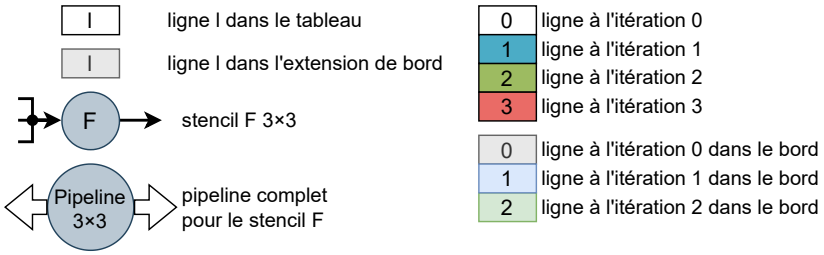


Fig. 3.8 : Légende utilisée dans les schémas des pipelines.

La figure 3.9 décrit les différentes étapes du pipeline de profondeur de 3 itérations pour le stencil 3×3. On peut noter qu'il n'est pas nécessaire d'utiliser un tableau auxiliaire pour chaque calcul d'itération. En prenant en compte les dépendances verticales, il est possible de factoriser le nombre de tableaux à 2. Le tableau de sortie contenant les lignes à la bonne itération sera soit le tableau initial dans le cas des itérations paires ou bien le tableau auxiliaire pour les itérations impaires. La figure 3.10 illustre cette factorisation. Le nombre d'étapes reste inchangé.

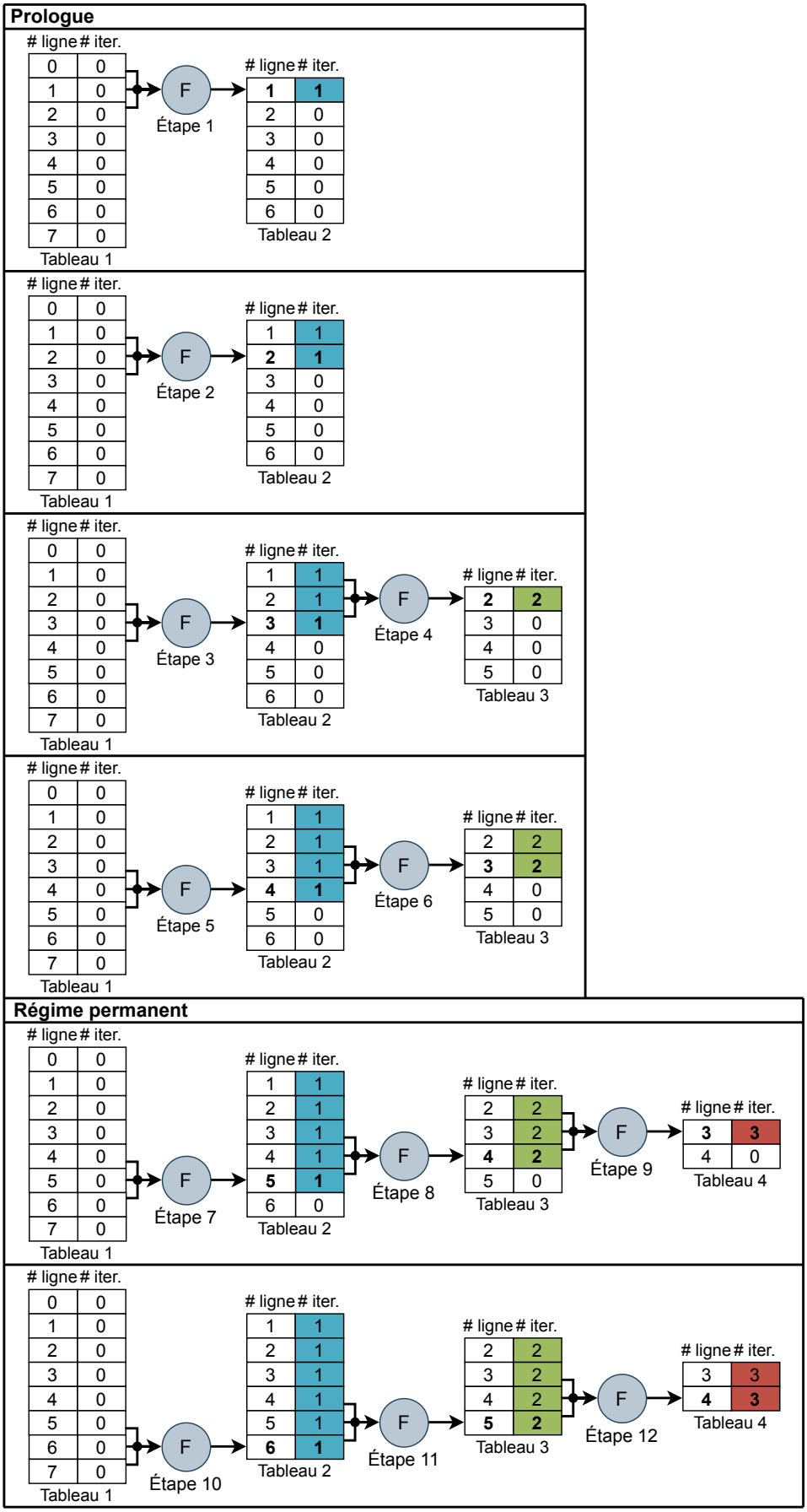


Fig. 3.9 : Étapes du pipeline de profondeur 3 itérations pour le stencil 3×3. Version simple avec un tableau par itérations.

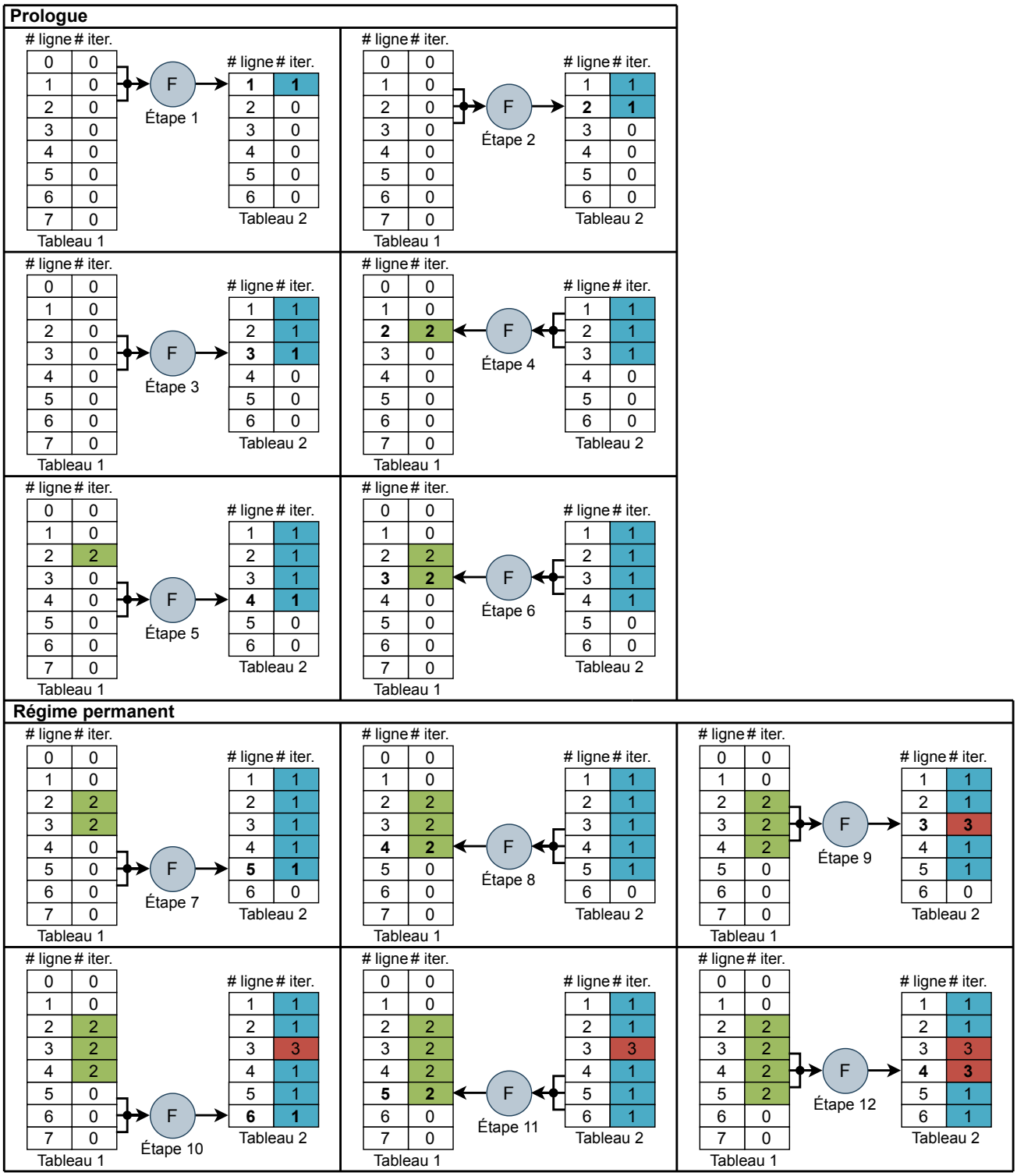
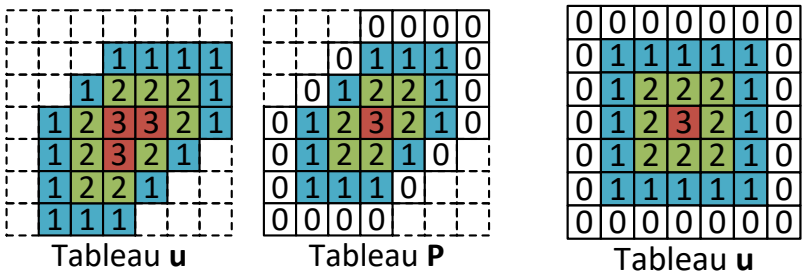


Fig. 3.10 : Étapes du pipeline de profondeur 3 itérations pour le stencil 3×3. Version optimisée avec 2 tableaux.

3.3.3 Gestion des bords

Comme on a pu le voir en figure 3.6, plus on augmente le nombre d'itérations calculées et plus les dépendances spatiales et temporelles pour le calcul d'un pixel vont augmenter. À titre d'exemple, nous représentons la croissance des dépendances spatiales pour 1, 2 et 3 itérations des algorithmes de TV-L^1 en figure 3.11a et de Horn-Schunck en figure 3.11b.



(a) TV-L^1 **(b)** Horn-Schunck

Fig. 3.11 : Croissance des dépendances spatiales pour 1, 2 et 3 itérations.

L'augmentation spatiale va ainsi nous entraîner à réaliser un traitement particulier en bord d'image pour traiter le cas des points en dehors de l'image. Une bonne gestion de cette extrapolation de données est nécessaire pour obtenir des résultats de bonne qualité [BA18] tout en gardant de bonnes performances [Tad+12]. Il est généralement préférable de réaliser cette extrapolation de données avant l'application d'un filtre stencil pour obtenir de meilleurs résultats [Bai11]. Comme listé dans l'article [BA18], il existe plusieurs manières de gérer les bords.

3.3.3.1 Serrage aux bords

Dans notre travail, nous avons choisi la technique de duplication de la valeur la plus proche, encore appelé serrage aux bords ou « border *clamping* » en anglais. Cette technique propose un bon compromis entre le temps d'exécution et la qualité en sortie de traitement [Ham15]. De plus, il existe plusieurs implémentations possibles de cette technique selon les besoins des algorithmes. Dans nos travaux, nous vérifions les indices d'accès aux pixels et nous les corrigeons s'ils tombent en dehors de l'image. On accède ainsi au pixel réel le plus proche du pixel demandé. Il n'y a pas besoin de modifier la structure initiale de stockage des données pour cette gestion des bords.

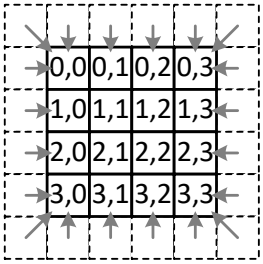


Fig. 3.12 : Technique de serrage aux bords. Les indices des pixels sont indiqués et les flèches représentent les pixels vers lesquels les indices « clampés » vont pointer. On représente ici un tableau de 4×4 pixel avec un bord de 1.

3.3.3.2 Duplication de bords

Lorsque suffisamment d'espace mémoire est à la disposition du programmeur, les tableaux sont étendus pour inclure des bords suffisamment grands pour gérer les dépendances de données du stencil. Cette technique demande cependant une réallocation et une recopie par rapport à un tableau initial sans bords. La taille en mémoire du nouveau tableau sera donc égale à la somme des dimensions du tableau d'origine avec les bords supplémentaires. Un décalage en mémoire est ensuite fait pour accéder aux pixels dans les bords de manière transparente. Pour une image de dimensions $w \times h$ pixels avec un bord b supplémentaire, ce décalage permet un accès aux pixels en lignes $[-1, -b]$ et $[h, h+b-1]$ et en colonnes $[-1, -b]$ et $[w, w+b-1]$.

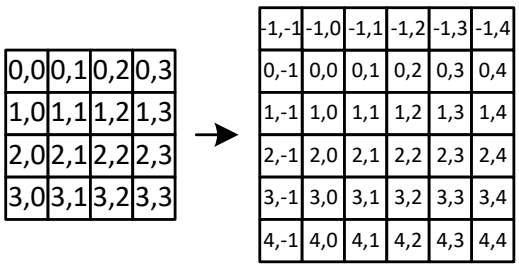


Fig. 3.13 : Technique de duplication bord. Un nouveau tableau plus grand est alloué contenant les pixels du tableau d'origine ainsi que les bords supplémentaires.

En reprenant l'exemple du stencil 3×3, les bords sont étendus en haut et en bas pour inclure p lignes supplémentaires chacune, avec p la profondeur du pipeline. La figure 3.14 illustre la pré-duplication des bords hauts et bas dans le tableau d'entrée du pipeline pour $p = 3$. Les données sont dupliquées dans ces lignes et le pipeline commence son traitement en ligne -2. Le pipeline est alors en état de traiter les 8 lignes complètes et de produire 8 lignes dans le tableau de sortie. Ce n'est pas le cas

dans le pipeline sans gestion de bord en figures 3.9 et 3.10. Dans ces deux figures, le pipeline n'est en mesure de produire que 2 lignes à la bonne itération dans le tableau de sortie. Cette gestion reste simple, les étapes du pipeline restent inchangées et ne nécessitent pas une différenciation du corps de la boucle en fonction de la position du stencil dans le tableau.

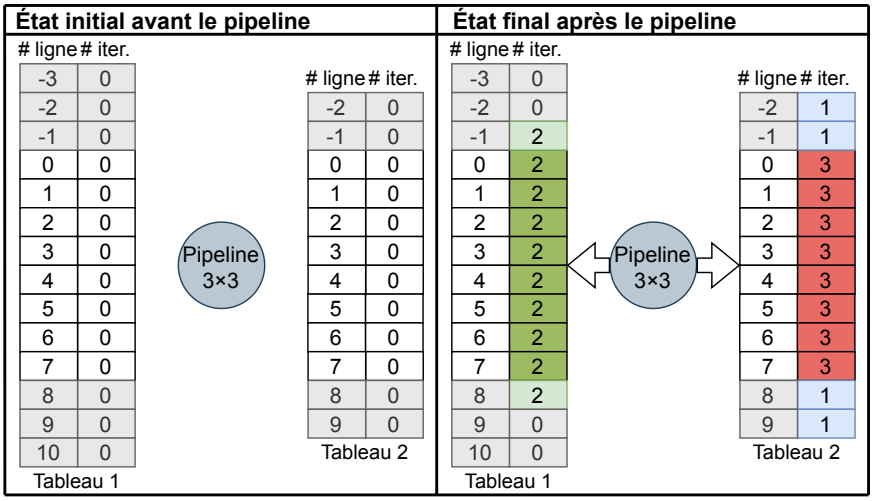


Fig. 3.14 : Effet de l'extension des bords hauts et bas sur le nombre de lignes traitées par le pipeline. Les lignes en gris clair représentent l'extension de lignes supplémentaire nécessaire pour traiter les 8 lignes en blanc.

3.3.3.3 Adressage modulaire

Lorsque l'espace mémoire dans lequel est réalisé le pipeline est limité, il n'est pas forcément possible de réaliser une duplication complète ou encore d'utiliser des tableaux auxiliaires de la taille du tableau d'entrée. Il serait possible de réaliser une redirection des lignes en bordures verticales, mais cela nécessite également suffisamment de mémoire pour contenir toutes les redirections des lignes du tableau. Dans ce cas, un calcul modulaire de l'indice des lignes peut être utilisé pour réduire l'empreinte des dépendances mémoire sur les lignes verticales.

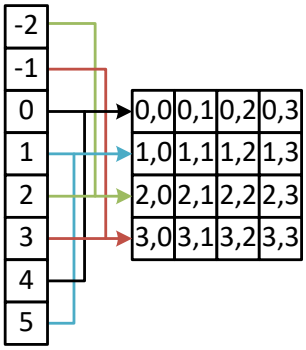


Fig. 3.15 : Technique d'adressage modulaire. On représente ici la redirection des accès pour les lignes -1, -2, 4 et 5 pour un tableau de 4×4 pixels.

Le bon choix du modulo utilisé pour le calcul des indices de lignes dans la mémoire intermédiaire où est exécuté le pipeline permet de réduire au maximum le nombre de lignes. Une fois qu'une ligne atteint le nombre d'itérations suffisant dans cette mémoire, celle-ci est écrite dans une autre mémoire externe. La ligne en mémoire intermédiaire est alors disponible pour contenir d'autres lignes nécessaires au calcul des lignes suivantes. Le calcul de ligne modulaire va donc utiliser un modulo correspondant au nombre de lignes total nécessaire à l'exécution du régime permanent sur 1 ligne.

En reprenant l'exemple du pipeline du stencil 3×3 pour 3 itérations, il faut $p+1=4$ lignes pour contenir toutes les lignes nécessaires au pipeline de $p=3$ itérations dans 1 seul tableau. Pour simplifier la représentation du déroulage du pipeline, on représente uniquement l'exécution du stencil sur 1 ligne par étape. Les lignes en entrée du stencil ne sont pas représentées. Pour la version du pipeline avec extension de bord, on obtient la représentation en figure 3.16.

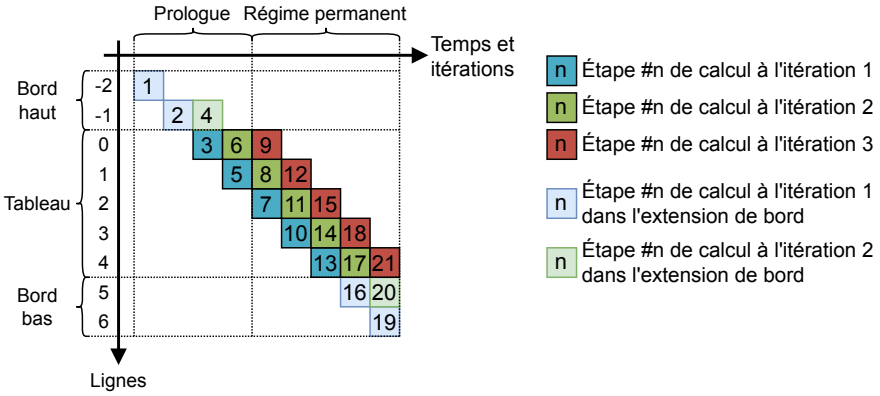


Fig. 3.16 : Ordre des étapes du pipeline pour le calcul de 3 itérations du stencil 3×3 sur 5 lignes avec extensions des bords.

Dans cette figure, une représentation de l'état des lignes est suffisante pour montrer le déroulement du pipeline. L'indice de la ligne calculée correspond à son indice dans le tableau servant aux calculs du pipeline. Les dépendances explicites entre lignes ne sont pas illustrées. Pour le pipeline utilisant un adressage modulaire, il faut ajouter à cette représentation l'indice de ligne calculé pour chaque étape étant donné l'absence de correspondance entre l'indice de la ligne calculé et son indice dans le tableau. On obtient ainsi la représentation en figure 3.17. Une représentation complète du pipeline modulaire est disponible en annexe et montre les chargements et les écritures entre les mémoires d'entrée, de sortie et de calcul.

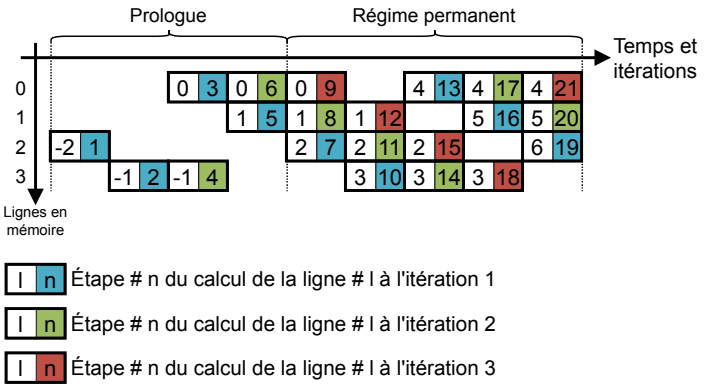


Fig. 3.17 : Ordre des étapes du pipeline pour le calcul de 3 itérations du stencil 3×3 sur 5 lignes en utilisant un adressage de lignes modulaire.

3.3.4 Synthèse

Dans cette section, nous avons décrit les deux principales optimisations haut-niveau mises en œuvre dans notre travail d'implémentation : la fusion d'opérateurs et le pipeline d'opérateurs. Le but premier de la fusion d'opérateurs est de supprimer les accès intermédiaires à la mémoire ainsi que dans certains cas réduire le nombre de calculs. L'objectif du pipeline est quant à lui d'améliorer la localité des accès aux données spatialement et/ou temporellement. Cette meilleure localité peut alors exploiter de manière plus efficace les mémoires rapides présentes dans le matériel pour réduire le temps passer à accéder aux données. Dans le cas d'algorithmes itératifs, le pipeline peut nécessiter l'ajout et la gestion de bords. Cet ajout de bords peut également être optimisé notamment grâce à la duplication de bords et leur empreinte mémoire réduite grâce à des accès modulaires et une réutilisation de l'espace.

Nous avons également présenté nos différentes gestions pour les bords verticaux et horizontaux de l'image. Les bords horizontaux sont les plus simples à gérer. Les pipelines implémentés dans nos travaux sont des pipelines sur les lignes. Les enchaînements particuliers précédemment présentés se font donc sur les lignes. Lorsqu'il s'agit d'accéder à des points en dehors de bords droit et gauche de l'image, un clamping de l'indice horizontal est suffisant pour obtenir de bons résultats. Ce n'est pas le cas pour les bords haut et bas de l'image. Dans ce cas, une duplication des valeurs dans une réelle extension allouée est nécessaire pour permettre un enchaînement de calcul juste qui respecte les dépendances de données. Deux types de gestions des bords verticaux sont réalisés dans nos travaux.

Il est ainsi important de bien adapter ces transformations et de régler leurs paramètres (nombre de points calculés, nombre de fonctions fusionnées, nombre d'itérations pipelinées, taille des bords, etc.) en fonction de l'architecture et du matériel utilisés. La plupart de ces transformations requièrent le stockage des données intermédiaires en registres, en cache ou en mémoire rapide. Dans le cadre du GPU, il faudra ainsi faire attention au nombre de cœurs de calculs utilisable en même temps ainsi qu'à la mémoire qui est beaucoup plus limitée par unité de calcul.

3.4 Transformations bas niveau (LLT)

Les transformations de codes dites de bas niveau – « Low-Level Transforms » en anglais – réalisent des optimisations au niveau du compilateur et de la génération des instructions machines. À la différence des transformations de haut niveau, elles ne changent pas la sémantique du code. Le but du programmeur est de guider le compilateur en ajoutant des indications explicites dans le code. L'impact en termes de transformations du code est donc variable, mais peut rester complexe, notamment dans le cas de la vectorisation de code. Les transformations bas niveau sont également beaucoup moins généralistes et dépendent en grande partie de l'architecture et du jeu d'instructions du ou des processeurs exécutant le code.

Pour centrer notre étude sur les optimisations bas niveau sur les GPU NVIDIA, nous utilisons comme référence le guide NVIDIA des bonnes pratiques de programmation CUDA [NVI22]. Trois axes d'optimisations se sont dégagés concernant les GPU :

— Un axe d'optimisation mémoire. L'objectif est de maximiser l'utilisation de la bande passante mémoire en maximisant l'utilisation du matériel. Pour ce faire, il faut utiliser les mémoires rapides à disposition, réutiliser un maximum de données déjà chargées et aligner les accès mémoire pour permettre une mutualisation des transactions ;

— Un axe d'optimisation d'exécution des threads. Cet axe concerne les techniques d'optimisation permettant une meilleure exécution de la totalité des threads sur les cœurs du GPU. Notre attention se porte ici sur l'utilisation d'instructions spécifiques rapides et l'utilisation des nombres flottants demi-précision permettant une diminution de l'empreinte mémoire du code, une exécution plus rapide du code et l'utilisation du parallélisme de sous-mot. L'élimination des instructions de contrôle (les instructions générant des branchements dans le code assembleur) quand cela est possible est également à prendre en compte ;

— Un axe d'optimisation de la configuration de lancement des kernels CUDA. Il faut répartir le traitement à faire sur l'ensemble des cœurs CUDA du GPU de manière la plus efficace possible. L'idée est ici de maintenir les cœurs actifs le plus longtemps possible est de tirer parti des ressources de calculs présentes sur le GPU. Pour ce faire, un dimensionnement fin de la taille des blocs, du nombre de blocs et du nombre de traitements réalisés par thread est à étudier.

Ces optimisations sont réalisées en parallèle des optimisations haut-niveau précédemment décrites. Pour chaque variante de haut-niveau de notre code (avec ou sans fusion, pipeline, extension de bords, adressage modulaire, etc.), l'implémentation associée comprend le même niveau d'optimisation bas-niveau. Nous décrivons maintenant les différentes optimisations selon leur axe.

3.4.1 Optimisation mémoire

Comme indiqué précédemment, le but de ces optimisations est d'obtenir la meilleure bande passante requise pour chaque kernel exécuté sur GPU. Le programmeur peut ainsi jouer sur plusieurs leviers pour maximiser cette bande passante.

Minimisation des transactions mémoire L'idée est ici de limiter au maximum le nombre d'octets transférés entre la mémoire externe et les registres des threads. Pour ce faire et quand cela ne pénalise pas les calculs, on peut réduire la dynamique des nombres flottants vers des nombres simple-précision, voire demi-précision. Les GPU NVIDIA considérés sont capables de gérer les formats de calculs double-précision sur 64 bits, simple-précision sur 32 bits et demi-précision sur 16 bits. Il faut également veiller à réutiliser un maximum les données déjà chargées (caches et registres).

Les requêtes mémoires entre les caches et la mémoire externe font 128 octets. Si la limitation de la taille des nombres flottants permet ainsi d'augmenter le nombre de nombres transmis en une seule transaction, il faut également faire attention à l'alignement des données demandées. Les requêtes mémoire d'un warp sont mutualisées pour essayer de ne former qu'une seule transaction. C'est le cas lorsque les 32 threads du même warp sollicitent l'accès à des données présentes dans la même zone contigüe et alignée sur 128 octets. On parle alors d'accès *coalescents*. Lorsque ce n'est pas le cas, autant de requêtes de 128 octets que nécessaire sont envoyées malgré une grande partie de données chargées en mémoire inutilisées. Cette utilisation inefficace du cache et de la localité spatiale des données entraine une dégradation des performances dans de nombreuses applications et benchmarks [Li+19].

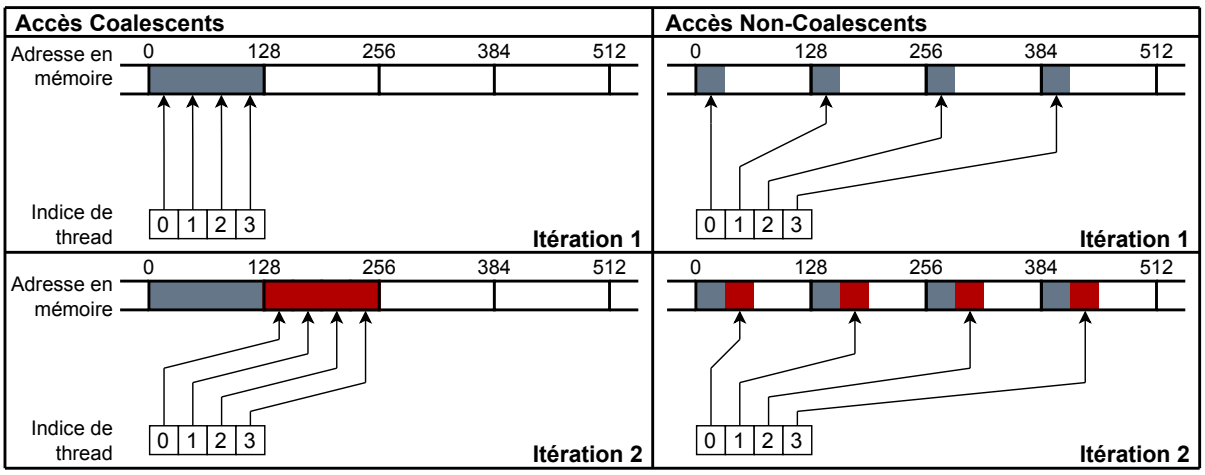


Fig. 3.18 : Exemples d'accès itératifs coalescés à gauche et non-coalescés à droite. La représentation utilise des warps de 4 threads pour diminuer la taille du schéma. Il en résulte une seule transaction de 128 octets par itération pour l'implémentation de gauche et 4 transactions par itération pour le schéma de droite.

Utilisation de la mémoire *Shared* L'utilisation de cette mémoire présente un double avantage. Cette mémoire se situe au même niveau que le cache L1 des SM du GPU et est donc beaucoup plus rapide que la mémoire globale. De plus, cette mémoire est allouée pour chaque bloc de threads. Tous les threads du même bloc ont ainsi accès aux mêmes données présentes dans cette mémoire. D'autres zones mémoires sont à disposition du programmeur mais se situent à l'extérieur du GPU, en mémoire globale externe. Pour tirer le meilleur parti de cette mémoire, on effectue des chargements coalescents depuis la mémoire externe jusque dans cette dernière. La bande passante entre ces mémoires est alors maximale si les données sont bien alignées. Les données sont ensuite utilisées par les threads. La mémoire *Shared* est divisée en 32 bancs, chacun capable d'émettre 32 bits par coup d'horloge lorsqu'il n'y a pas de conflit d'accès. Un conflit est généré lorsque plusieurs threads cherchent à accéder à des données présentes dans le même banc mais pas dans le même mot de 4 octets. Dans ce cas, les accès sont sérialisés. Si plusieurs threads cherchent à accéder à des données présentes dans le même mot de 4 octets dans le même banc, la donnée est alors diffusée à tous les threads qui en ont besoin (« broadcast » en anglais).

Instruction Shuffle Enfin, on peut également citer les instructions `shuffle` qui permettent d'échanger des données en registre entre les threads du même warp. L'utilisation de telles instructions permet des gains importants lorsqu'un thread doit échanger des informations avec ses voisins [WXC17]. C'est le cas pour les algorithmes réalisant de la diffusion large de données ou des réductions, comme pour

les algorithmes d'étiquetage en composante connexe [LHL21a ; LHL21b ; LHL20 ; Hen+18]. Ces instructions de shuffle permettent une réduction du temps d'exécution se situant entre $\times 4$ et $\times 10$ sur des cartes de calcul A100 de NVIDIA [LHL21b] ; et se situant entre $\times 1,8$ et $\times 2,7$ sur la plateforme embarquée TX2 [Hen+18], par rapport à l'état de l'art GPU. Pour les échanges entre threads de différents warps, il faut utiliser d'autres mécanismes ou d'autres zones mémoires, ce qui peut dégrader les performances des implémentations utilisant des shuffles. Il faut également noter le débit de 32 résultats par cycle et par SM d'une instruction shuffle. Le traitement de plusieurs warps doit donc être sérialisé.

3.4.2 Optimisation d'exécution

L'idée est ici de réduire la latence des instructions ainsi que d'augmenter le nombre d'instructions traitées en parallèle par le GPU.

Réduction de la latence d'instructions et parallélisme de sous-mot La principale optimisation pour réduire le temps d'exécution d'un kernel est d'utiliser les instructions et les nombres ayant le plus faible nombre de cycles associé. Pour ce faire, la première transformation consiste à utiliser les fonctions rapides intrinsèques à CUDA. Il faut cependant veiller à ce que leur précision et exactitude soient suffisantes pour les calculs.

En plus de fonctions rapides à disposition du programmeur, un format demi-précision sur 16 bit *__half* est disponible pour les calculs. Comme indiqué dans la section précédente, l'utilisation de ce format permet de diminuer par 2 la bande passante requise pour les transferts mémoire et de réaliser le traitement de 2 nombres demi-précision en même temps sur chaque cœur CUDA (parallélisme de sous-mot) [LE05 ; LEK05 ; EL04]. Le débit en termes de résultats produits à chaque cycle s'en retrouve ainsi doublé [NVI21]. Grâce au cumul des fonctions rapides demi-précision et de l'augmentation de débit grâce au parallélisme de sous-mot, des gains de facteur $\times 3$ sont possibles par rapport à l'emploi de nombres flottants simple précision [HW17]. La réduction de la bande passante par nombre permet des gains supplémentaires.

Diminution de la divergence entre threads Lorsqu'une structure de contrôle est présente dans le code d'un kernel GPU, son exécution peut entrainer une divergence dans l'exécution des threads d'un même bloc de threads. On désigne par structure

de contrôle toute instruction susceptible de générer un branchement dans le code machine généré par le compilateur. Les instructions `if`, `switch`, `do`, `for` et `while` sont ainsi concernées. Comme indiqué dans le chapitre 2, un warp de 32 threads exécute la même instruction lorsqu'il est alloué et envoyé sur les cœurs CUDA du sous-bloc de traitement associé. Tous les threads exécutent donc la même instruction. Si une partie des threads doit exécuter une instruction différente, les deux instructions sont sérialisées et les threads masqués en fonction du choix de l'instruction. Les threads vont donc exécuter le code de toutes les branches, peu importe si le branchement est pris ou non, ce qui peut allonger l'exécution et donc dégrader les performances du kernel. Des blocages peuvent également arriver si l'exécution d'un thread dépend de la fin de l'exécution d'un autre thread en état de divergence (utilisation d'instructions atomiques, locks et mutex). Depuis l'architecture de GPU Volta, chaque thread comporte désormais chacun un compteur ordinal d'instruction (registre PC, ou « Program Counter » en anglais) et une pile d'exécution. Cela permet de mitiger les problèmes de blocages dus à des dépendances inter-threads, car les threads peuvent désormais diverger et reconverger avant la fin de l'exécution de la branche conditionnelle complète. Il faut donc veiller à l'utilisation d'instructions réalisant des branchements conditionnels. Le compilateur CUDA `nvcc` réalise à ce titre des remplacements de branchements par des instructions avec prédicats. Si un gain en performance est possible en remplaçant les structures `if-then-else` par des opérateurs ternaires ou des opérations masquées, le compilateur réalise déjà ces transformations dans la plupart des cas [STA20].

3.4.3 Optimisation de configuration de lancement

Selon le kernel écrit par le programmeur et ses paramètres de lancement, un bloc de threads exécuté sur le GPU nécessite plus ou moins de ressources. Il est impératif de prendre en compte ces besoins dans la configuration de lancement du kernel sur le GPU. Les configurations de lancement du kernel désignent les 4 paramètres à définir lors de l'exécution du kernel à savoir :

1. un paramètre définissant les dimensions de la grille de blocs à lancer sur le GPU (l'unité de dimension est ici le nombre de blocs de threads) ;
2. un paramètre définissant les dimensions des blocs de threads qui sont lancés dans la grille (l'unité est ici le nombre de threads dans chaque dimension du bloc) ;
3. la quantité de mémoire *Shared* en octet à allouer par bloc de threads (on parle ici d'allocation dynamique à la différence de l'allocation statique où la quantité est connue à la compilation). Il faudra noter que selon l'architecture,

la mémoire *Shared* et le cache L1 occupent le même espace mémoire : une trop grande utilisation de l'un ou de l'autre peut réduire les performances ;

4. un paramètre *CUDA stream* définissant sur quel flot d'exécution CUDA la grille est exécutée sur le GPU. Ce paramètre permet l'exécution de plusieurs kernels simultanément quand les ressources du GPU le permettent.

Une bonne configuration de lancement permet de bien répartir la charge de travail sur tous les cœurs de calcul à disposition sur le GPU. Le but est de maximiser l'utilisation des ressources matérielles et de limiter les temps morts et les attentes lors de l'exécution des instructions des threads. Pour ce faire, il faut aussi faire attention aux facteurs limitant l'exécution des kernels sur le GPU. On parle de limitation de l'*occupancy*. L'occupancy est le rapport entre le nombre de warps *actifs* par SM et le nombre maximal de warps actifs possible.

— Un warp est *actif* s'il appartient à l'ensemble des warps résidant sur un SM lors de l'exécution du kernel. Le nombre de warps pouvant être alloués sur un SM dépend des limitations de registres et de mémoire *Shared* décrites précédemment. Il existe également un nombre maximum de warp actifs pouvant être alloués sur un SM dépendant de l'architecture GPU. Pour les architectures visées, un maximum de 64 warps (2048 threads) peut être alloué.

— Un warp est *bloqué* s'il est actif mais n'est pas en état d'émettre une instruction à exécuter sur les cœurs CUDA. C'est le cas quand un warp se trouve derrière une barrière de synchronisation et attend la fin de l'exécution d'autres warps ; quand il n'a pas encore récupéré sa prochaine instruction depuis la mémoire ; ou est en attente du résultat d'une instruction précédente.

— Un warp est *éligible* si, au contraire, il est capable de progresser dans son exécution et d'émettre une nouvelle instruction à exécuter. C'est le cas quand l'instruction a été récupérée depuis la mémoire, le cœur d'exécution disponible et que l'instruction n'a pas de dépendances en attente. Ce warp est alors prêt à être sélectionné par le *scheduler de warp* pour ensuite envoyer une instruction sur les cœurs CUDA via la *l'unité de dispatch* [Jia+18].

— Un warp est *sélectionné* parmi les warps éligibles par le scheduler de warp à chaque cycle pour exécuter une instruction. Cette instruction peut ainsi être une instruction indépendante du même warp, ou bien une instruction d'un autre warp.

Il faut également noter que le programmeur n'a pas la main sur le placement des warps sur les cœurs du GPU. Le scheduler de warps peut ainsi engendrer des variations dans l'exécution d'un kernel d'une exécution à une autre. Cela peut poser des problèmes d'arithmétique flottante de part des erreurs d'arrondis et d'annulation catastrophique [DC15].

Il faut donc veiller aux facteurs suivants pour maximiser le débit d'instructions traitées par les cœurs CUDA. Nous listerons en fin de section les limitations physiques réelles pour les architectures GPU visées dans ce projet.

Disponibilité des registres La taille du banc de registres est l'un des premiers facteurs physiques limitant dans l'exécution d'un grand nombre de threads. Cette mémoire interne à chaque SM contient les variables en registres de chaque thread. Il s'agit d'une petite mémoire de 65536 registres de 32 bits dans la plupart des architectures GPU qui est partagée avec tous les cœurs présents sur le même SM. En plus de cette limitation physique, il existe un nombre maximum de registres par bloc de thread – lui aussi variant selon l'architecture – ainsi qu'un nombre maximum de 255 registres 32 bits par thread. Lorsque qu'un thread individuel ou qu'un bloc de threads nécessite un nombre trop élevé de registres, une erreur est alors signalée et le kernel retourne de son exécution et signal une erreur. L'option de compilation `-maxrregcount` existe pour limiter le nombre maximum de registres par thread, mais force une allocation dans la mémoire locale se trouvant physiquement dans la mémoire globale externe pour les registres qui dépassent ce maximum. Une augmentation du temps d'accès pour les variables utilisant ces registres est donc à prévoir et dégrade ainsi les performances du kernel. En revanche, cette limitation des ressources permet à la fois l'allocation de plus de blocs sur le GPU mais également l'exécution de plus de threads en parallèles qui ont ainsi plus de ressources chacune (un seul thread n'occupe plus un grand nombre de ressources).

Disponibilité de la mémoire *Shared* Tout comme le banc de registres, la quantité de mémoire *Shared* est limitée sur le GPU. Comme présenté dans la section précédente, la quantité, mais aussi l'emplacement physique de cette mémoire varie selon l'architecture. Pour les architectures les plus anciennes (Maxwell et Pascal notamment), un emplacement physique se situant au même niveau que le cache de données L1 est dédié pour cette mémoire et varie entre 64 et 96 Ko par SM. À partir des architectures Volta, le cache L1 de données et la mémoire ont été unifiés en une seule mémoire. Volta présente ainsi un cache L1/mémoire *Shared* unifié de 128 Ko par SM dont 96 Ko pouvant être utilisé comme mémoire *Shared*. Dans le cas où le maximum de 96 Ko est alloué, il ne restera plus que 32 Ko par SM comme cache de données L1. De plus, un maximum de 48 Ko peut être alloué par bloc de threads.

Maximisation de l'exécution des instructions des warps sur un SM Tout comme les mémoires, il existe un nombre maximum de warps pouvant résider dans un SM. Il

faut ainsi veiller à deux limitations majeures des performances concernant le nombre de warps par SM déterminé par la dimension des blocs de threads.

La première concerne les dimensions maximales des blocs de threads en termes de warps. Sachant que les warps d'un même bloc de threads s'exécutent sur le même SM, il faut bien dimensionner les blocs de threads pour maximiser le nombre de blocs actifs sur un SM.

La deuxième concerne le scheduler de warp et la latence des instructions. Plusieurs cycles sont nécessaires à l'exécution d'une instruction complète pour un warp. Sur les architectures visées, il faut au minimum 4 cycles pour l'exécution d'instructions arithmétiques de base (addition, multiplication et multiplication-addition combinées) sur des nombres flottants 32-bits. Il faut alors au minimum 4 instructions par scheduler de warp pour cacher cette latence et ainsi produire un nouveau résultat à chaque cycle d'horloge. La plupart du temps, un warp est bloqué dans son exécution si ses opérandes d'entrées ne sont pas encore disponibles. Une bonne heuristique est donc de s'assurer de la présence de 4 warps actifs pour chaque scheduler de warp qui peuvent ainsi émettre chacun 4 instructions indépendantes à chaque cycle. Si un warp individuel présente des instructions indépendantes, on peut tirer parti du parallélisme d'instructions et moins de warps actifs sont ainsi nécessaires pour cacher la latence. Étant donné les 4 schedulers de warps présents dans les 4 sous-blocs de traitement pour un SM, il faut donc 16 warps actifs pour pouvoir cacher la latence des instructions dans une majorité de cas. Un travail d'affinage plus fin est cependant nécessaire pour prendre en compte les latences plus longues dues à l'utilisation d'autres instructions, d'autres types, ou du temps d'accès à la mémoire externe lors de miss dans les caches instructions ou données.

3.4.4 Synthèse

Dans cette section, nous avons présenté un autre niveau d'optimisation plus proche du compilateur : les transformations de bas-niveau (LLT). À la différence des transformations de plus haut niveau opérant sur la structure et l'organisation algorithmique du code, ces optimisations ne modifient pas la sémantique du code. Elles opèrent sur la génération du code par le compilateur. Le but du programmeur est donc d'informer le mieux ce dernier pour générer le code plus rapide possible. D'un point de vue GPU, cela passe par les 3 grands points d'optimisation mémoire, d'exécution des instructions et de configuration de lancement des kernels.

Le premier point cherche à réduire au maximum les échanges entre les mémoires lentes afin de réduire les temps d'attente d'accès aux données dans ces dernières.

Cela passe par une réécriture du code pour coalescer au maximum les accès en un minimum de transactions, utiliser des nombres ayant une empreinte mémoire plus petite et utiliser des zones ou des instructions rapides d'échange de données.

Le deuxième point cherche à réduire un maximum la latence des instructions. Cela se fait grâce à l'utilisation de fonctions spécialisées rapides mais moins précises, l'utilisation du parallélisme de sous-mot pour augmenter le débit de traitement et en réduisant les divergences de thread via une diminution des branchements dans le code.

Enfin, le troisième point cherche à maximiser l'utilisation des ressources GPU via un choix judicieux des paramètres de lancement d'un kernel. Les paramètres que sont les dimensions du bloc de thread, la quantité de mémoire *Shared* et le nombre de registres utilisés ont un impact direct sur le lancement ou non d'un kernel sur le GPU. De nombreux tests sont à faire afin de déterminer les paramètres optimaux permettant un temps d'exécution minimal du kernel.

3.5 Conclusion

Nous avons abordé dans ce chapitre les notions de l'état de l'art nécessaires à la compréhension des travaux de cette thèse.

Nous avons ainsi présenté l'état de l'art des algorithmes d'estimation du flot optique sur CPU et sur GPU. Nous avons ainsi décidé de concentrer nos efforts d'implémentation sur les algorithmes de TV-L^1 et de Horn-Schunck. TV-L^1 est un algorithme présentant un bon compromis entre qualité de flot et temps d'exécution est et utilisé dans de nombreuses applications. Bien que moins précis que TV-L^1, Horn-Schunck est l'un des deux premiers algorithmes d'estimation du flot optique et reste une référence dans le domaine pour des applications embarquées nécessitant un algorithme simple et rapide.

Nous avons ensuite présenté et décrit les détails architecturaux des GPU utilisés dans nos travaux. Il en dégage que chaque génération de GPU présente des spécificités qui rendent l'écriture d'un code optimal pour toutes les architectures difficiles. Ces GPU présentent différentes capacités mémoires, nombres et types d'unités de traitement ainsi que différentes fréquences de fonctionnement, technologie de gravure et donc différents profils de consommation énergétique. Il sera alors nécessaire de tester chaque optimisation sur chaque carte visée et de déterminer la meilleure

implémentation dans chaque cas, en termes de temps de traitement, qualité du flot calculé et d'énergie consommée.

Enfin, nous avons exploré les différents types d'optimisations utilisés dans nos travaux. Ces optimisations se font sur 2 niveaux différents. Le premier niveau le plus haut consiste en une réécriture profonde du code afin de réutiliser un maximum les données déjà calculées dans des mémoires rapides. Ces types de transformations sont difficiles à implémenter en GPU de par la quantité de mémoire par thread limitée sur les GPU. Le deuxième niveau est un niveau plus bas, proche du code généré par le compilateur. Il est nécessaire de bien informer le compilateur afin que le code généré soit le plus rapide possible. Cela passe par l'utilisation de fonctions, de types de nombres et des mémoires spécialisées rapides. Il faut également veiller à ce qu'une sur-utilisation des ressources du GPU par bloc de thread n'entraîne pas une diminution des performances. Là encore, des tests et des benchmarks seront à faire pour déterminer les meilleures configurations possibles.

Nous allons ainsi pouvoir décrire notre travail d'implémentation dans le chapitre suivant.

4

Mise en œuvre des optimisations pour deux algorithmes d'estimation du flot optique

Dans ce chapitre, nous présentons les applications des techniques d'optimisations introduites dans le chapitre précédent pour les algorithmes itératifs d'estimation de flot optique. Dans le cadre des travaux de thèse d'Andrea Petreto [Pet20], une chaîne de traitement a été définie et étudiée pour du débruitage de bas niveau de lumière en temps réel : la chaîne de traitement « RTE-VD » pour *Real Time Embedded Video Denoising*.

Le bloc majeur de cette chaîne est l'estimation du mouvement apparent des pixels d'une image vidéo à une autre, autrement appelé *flot optique*. Il existe 3 types de flot optiques différents :

— le flot optique dense correspondant à l'estimation du mouvement pour tous les pixels d'une image à l'autre,
— le flot optique creux correspondant à l'estimation du mouvement pour un sous-ensemble de pixels dans les images (points d'intérêt, macro-pixels),
— le flot optique global correspondant à l'estimation du mouvement moyen global entre deux images d'une séquence vidéo (translation, rotation...).

la chaîne de traitement nécessite une estimation dense de flot optique sur toute l'image. L'obtention de ce flot optique permet un filtrage temporel des pixels sur des scènes en mouvement par compensation du mouvement d'une image à l'autre. Sans cette estimation, un filtrage temporel simple entraîne un flou de bougé trop important dans les séquences d'images filtrées.

Nous avons donc étudié deux grands algorithmes d'estimation de flot optique dense. Dans un premier temps, nous avons implémenté et optimisé la méthode d'estimation dite TV-L[1]. Il s'agit d'un algorithme de référence présentant un excellent compromis entre qualité du flot optique estimé, robustesse au bruit et complexité

de calcul. Cet algorithme a été optimisé sur CPU dans la version d'origine de RTE-VD. Nous avons également appliqué nos optimisations sur un autre algorithme d'estimation du flot optique : la méthode d'estimation de Horn et Schunck. Il s'agit d'un algorithme de référence également, plus simple et plus adapté à des plateformes embarquées ou des plateformes FPGA.

Dans la suite de ce chapitre, nous allons présenter les bases mathématiques sur lesquelles reposent ces deux algorithmes ainsi que les approximations numériques nécessaires afin d'obtenir un algorithme complet. Par la suite, nous présenterons les optimisations algorithmiques et d'implémentation réalisées sur ces deux algorithmes.

4.1 Modélisation mathématique

Afin de modéliser et par la suite estimer le flot optique dans une séquence vidéo, il est nécessaire de définir les objets mathématiques utilisés dans les différentes méthodes d'estimation.

Commençons par définir une séquence de T images successives dans le domaine image $\Omega \subset \mathbb{R}^2$ par la fonction :

$$I(x, y, t) : \Omega \times [0, T] \rightarrow \mathbb{R}. \tag{4.1}$$

Il s'agit de l'intensité lumineuse d'un point de l'espace qui se trouve aux coordonnées (x, y) d'une image à l'instant t. Le flot optique est la trajectoire des points lumineux, ou pixels, représentant les objets dans une séquence d'images au cours du temps. La trajectoire de ces pixels dans le plan image est définie par $(x(t), y(t))$. Déterminer le flot optique dense correspond donc à déterminer le champ de vecteur

$$\mathbf{u} = (u_1, u_2) = \left(\frac{dx}{dt}, \frac{dy}{dt}\right) \tag{4.2}$$

pour tous les pixels de l'image. Nous faisons l'hypothèse que l'intensité lumineuse d'un point reste constante le long de sa trajectoire. Il s'agit de la *contrainte fondamentale du flot optique*. Cette hypothèse est le point commun sur lequel repose une grande partie des méthodes variationnelles d'estimation du flot optique. Autrement dit, nous avons pour des petits déplacements spatiaux δx et δy et pour un petit intervalle temporel δt :

$$I(x + \delta x, y + \delta y, t + \delta t) - I(x, y, t) = 0. \tag{4.3}$$

On notera qu'afin de simplifier l'écriture de l'équation précédente, nous utilisons des abus de notation. En effet, les coordonnées (x, y) dépendent du temps. On devrait donc écrire $x(t)$, $y(t)$, $\delta x(x, y, t)$ et $\delta y(x, y, t)$.

En faisant le développement de Taylor au point (x, y) au temps t à l'ordre 1 on obtient :

$$I(x + \delta x, y + \delta y, t + \delta t) = I(x, y, t) + \delta x \frac{\partial I}{\partial x} + \delta y \frac{\partial I}{\partial y} + \delta t \frac{\partial I}{\partial t} + \epsilon, \tag{4.4}$$

où ϵ est un reste négligeable contenant des termes d'ordre supérieurs pour δx, δy et δt. En appliquant cette approximation dans l'équation de la contrainte du flot optique 4.3 et en divisant par dt, on obtient donc :

$$\frac{\partial I}{\partial x}\frac{\delta x}{dt} + \frac{\partial I}{\partial y}\frac{\delta y}{dt} + \frac{\partial I}{\partial t}\frac{\delta t}{dt} + \epsilon = 0. \tag{4.5}$$

En passant à la limite $dt \to 0$ on obtient finalement la formulation linéaire de la contrainte du flot optique :

$$\begin{aligned} \frac{\partial I}{\partial x}\frac{dx}{dt} + \frac{\partial I}{\partial y}\frac{dy}{dt} + \frac{\partial I}{\partial t}\frac{dt}{dt} &= 0 \\ \Leftrightarrow \\ I_x u_1 + I_y u_2 + I_t &= 0. \end{aligned} \tag{4.6}$$

avec I_x, I_y, I_t les dérivées partielles de I en x, y et t respectivement et avec (u_1, u_2) le flot optique à déterminer. On peut noter qu'il est également possible d'obtenir l'équation 4.6 en calculant la dérivée totale de I par rapport à t en utilisant la règle de dérivation en chaîne des fonctions composées de plusieurs variables. La formulation du problème peut également s'exprimer en utilisant le gradient $\nabla I = [I_x, I_y]^\intercal$ de I en x et y :

$$\mathbf{u} \cdot \nabla I + I_t = 0. \tag{4.7}$$

Il s'agit d'une équation à deux inconnues u_1 et u_2. Il est donc nécessaire d'introduire des contraintes supplémentaires afin de résoudre ce problème d'indétermination, autrement appelé le *problème d'ouverture*. Les méthodes TV-L[1] et d'Horn-Schunck introduisent deux types de contraintes supplémentaires et proposent des méthodes de résolutions.

4.2 Méthode TV-L^1

La méthode d'estimation de flot optique TV-L^1 fut introduite en 2007 par Zach, Pock et Bischof [ZPB07]. Comme Horn-Schunck, il s'agit également d'une méthode variationnelle se basant sur la minimisation d'une fonctionelle. Cette fonctionnelle utilise cependant la formulation non-linéaire de la contrainte du flot optique présentée en équation 4.3.

On commence par prendre deux images d'entrées I_0 et I_1 consécutives dans la séquence d'image. On prend ainsi une cadence d'image δt telle que $I(x, y, t)$ correspond à I_0 et $I(x, y, t + \delta t)$ correspond à I_1. On obtient ainsi la formulation suivante de la contrainte du flot optique :

$$\begin{aligned} I_1(x + u_1, y + u_2) - I_0(x, y) = 0 \\ \Leftrightarrow \\ I_1(\mathbf{x} + \mathbf{u}) - I_0(\mathbf{x}) = 0. \end{aligned} \tag{4.8}$$

Cette formulation a l'avantage d'être valide pour des déplacements arbitrairement grands. Les auteurs proposent donc de minimiser la fonctionnelle suivante :

$$E = \iint_{\Omega} \underbrace{\lambda \left| I_0(\mathbf{x}) - I_1(\mathbf{x} + u) \right|}_{\mathcal{D}} + \underbrace{|\nabla u_1| + |\nabla u_2|}_{\mathcal{R}} dxdy. \tag{4.9}$$

Cette fonctionnelle à minimiser est proche de celle de Horn-Schunck en équation 4.23 : il suffit de mettre le terme $\mathcal{D}$ et le terme $\mathcal{R}$ sous forme quadratique pour obtenir la proposition de Horn-Schunck. La proposition TV-L^1 quant à elle utilise une norme 1 pour le terme $\mathcal{D}$ et la variation totale de norme 1 également comme terme $\mathcal{R}$.

Afin de trouver un minimum à cette fonctionnelle, les auteurs du papier emploient le schéma numérique de Chambolle [Cha04] utilisé pour la résolution du modèle de débruitage de Rudin-Osher-Fatemi [ROF92] (ROF). Il faut alors adapter ce modèle à l'estimation du flot optique. On commence par linéariser le terme $I_1(\mathbf{x} + \mathbf{u})$ en $\mathbf{u}^0$, une estimation initiale du flot optique proche de $\mathbf{u}$. On obtient alors :

$$I_1(\mathbf{x}+\mathbf{u}) - I_0(\mathbf{x}) \approx I_1(\mathbf{x}+\mathbf{u}^0) + (\mathbf{u}-\mathbf{u}^0) \cdot \nabla I_1(\mathbf{x}+\mathbf{u}^0) - I_0(\mathbf{x}) = \rho(\mathbf{u}, \mathbf{u}^0). \tag{4.10}$$

On introduit ensuite une relaxation convexe dans la fonctionnelle à minimiser en équation 4.11 sous forme d'une variable auxiliaire $\mathbf{v} = (v_1, v_2)$. On obtient ainsi la fonctionnelle de la méthode TV-L^1 à minimiser présentée dans l'article d'origine :

$$E_{\text{TVL1}} = \iint_{\Omega} |\nabla u_1| + |\nabla u_2| + \frac{1}{2\theta} |\mathbf{u} - \mathbf{v}|^2 + \lambda |\rho(\mathbf{v})| \, dxdy, \tag{4.11}$$

avec θ une petite valeur telle que $\mathbf{v}$ est une approximation proche de $\mathbf{u}$.

La minimisation de cette fonctionnelle se fait tour à tour en fixant l'une des variables $\mathbf{u}$ ou $\mathbf{v}$ et en faisant varier l'autre :

1. Avec $\mathbf{v}$ fixe, on cherche la valeur de $\mathbf{u}$ qui minimise la fonctionnelle suivante :

$$\iint_{\Omega} |\nabla u_1| + |\nabla u_2| + \frac{1}{2\theta} |\mathbf{u} - \mathbf{v}|^2 \, dxdy. \tag{4.12}$$

2. Avec $\mathbf{u}$ fixe, on cherche la valeur de $\mathbf{v}$ qui minimise la fonctionnelle suivante :

$$\iint_{\Omega} \frac{1}{2\theta} |\mathbf{u} - \mathbf{v}|^2 + \lambda |\rho(\mathbf{v})| \, dxdy. \tag{4.13}$$

La première équation 4.12 correspond ainsi au modèle de débruitage ROF et peut donc être résolue grâce à l'algorithme de Chambolle. La deuxième équation 4.13 ne dépend pas des dérivées spatiales de $\mathbf{v}$ et peut ainsi être résolue en utilisant une fonction de seuillage point par point.

Schéma Numérique

Comme dit précédemment, le schéma numérique correspond à la résolution des deux contraintes par alternance en fixant l'une des variables $\mathbf{u}$ ou $\mathbf{v}$. Le schéma numérique se base sur l'implémentation de la méthode proposée par Sánchez, Meinhardt-Llopis et Facciolo en 2012 [SMF13].

La variable $\mathbf{v}$ est d'abord estimée en résolvant l'équation 4.13. La solution est donnée par seuillage :

$$\mathbf{v}^{k+1} := \mathbf{u}^k + \text{TH}(\mathbf{u}^k, \mathbf{u}^0), \tag{4.14}$$

avec la fonction de seuillage :

$$\mathrm{TH}(\mathbf{u},\mathbf{u}^0) := \begin{cases} \lambda\theta\nabla I_1(\mathbf{x}+\mathbf{u}^0) & \text{si } \rho(\mathbf{u},\mathbf{u}^0) < -\lambda\theta\left|\nabla I_1(\mathbf{x}+\mathbf{u}^0)\right|^2 \\ -\lambda\theta\nabla I_1(\mathbf{x}+\mathbf{u}^0) & \text{si } \rho(\mathbf{u},\mathbf{u}^0) > \lambda\theta\left|\nabla I_1(\mathbf{x}+\mathbf{u}^0)\right|^2 \\ -\rho(\mathbf{u},\mathbf{u}^0)\frac{\nabla I_1(\mathbf{x}+\mathbf{u}^0)}{|\nabla I_1(\mathbf{x}+\mathbf{u}^0)|^2} & \text{si } \left|\rho(\mathbf{u},\mathbf{u}^0)\right| < \lambda\theta\left|\nabla I_1(\mathbf{x}+\mathbf{u}^0)\right|^2 \end{cases}. \tag{4.15}$$

On notera ici l'utilisation de I_1 et de son gradient spatial ∇I_1, déformés par l'estimation initiale du flot $\mathbf{u}^0$. Dans notre implémentation, nous utilisons une interpolation bicubique pour réaliser ces transformations.

L'étape suivante consiste à estimer $\mathbf{u}$ en utilisant le $\mathbf{v}$ fixe précédemment calculé. La méthode de résolution de Chambolle de l'équation 4.12 utilise une formulation duale du problème et une recherche de point fixe. On introduit la variable duale $\mathbf{P} := (\mathbf{p_1}, \mathbf{p_2})$ telle que :

$$\mathbf{u}^{k+1} := \mathbf{v}^{k+1} + \theta \operatorname{div} \mathbf{P}^k. \tag{4.16}$$

Chacune des composantes du flot u_1 et u_2 sont mises à jour en utilisant respectivement la divergence du champ de vecteur $\mathbf{p}_1^k$ et du champ de vecteur $\mathbf{p}_2^k$. Les deux composantes $\mathbf{p}_1 := (p_{11}, p_{12})$ et $\mathbf{p}_2 := (p_{21}, p_{22})$ du double champ de vecteur $\mathbf{P}$ sont enfin elles aussi mises à jour selon l'équation suivante pour $d \in \{1, 2\}$:

$$\begin{aligned} \mathbf{p}_d^{k+1} &:= \frac{\mathbf{p}_d^k + \tau/\theta\nabla(v_d^{k+1} + \theta \operatorname{div} \mathbf{p}_d^k)}{1 + \tau/\theta\left|\nabla(v_d^{k+1} + \theta \operatorname{div} \mathbf{p}_d^k)\right|} \\ &\Leftrightarrow \\ \mathbf{p}_d^{k+1} &:= \frac{\mathbf{p}_d^k + \tau/\theta\nabla(u_d^{k+1})}{1 + \tau/\theta\left|\nabla(u_d^{k+1})\right|}, \end{aligned} \tag{4.17}$$

avec τ un paramètre de pas de temps du schéma numérique et θ un paramètre d'approximation relativement petit tel que $\mathbf{u}$ soit proche de $\mathbf{v}$. La convergence est assurée pour $\tau < 0.125$ mais il a été montré empiriquement que le schéma converge plus rapidement pour des valeurs autour de 0.25.

Le schéma numérique précédent fait l'emploi d'opérateurs devant être approximés. Les dérivées spatiales partielles pour le calcul du gradient $\nabla I_1(\mathbf{x} + \mathbf{u}^0)$ sont approximées en utilisant les différences finies centrées telles que :

$$\begin{aligned} \frac{\partial I_1}{\partial x}(i,j) &\approx \frac{I_1(i+1,j) - I_1(i-1,j)}{2}, \\ \frac{\partial I_1}{\partial y}(i,j) &\approx \frac{I_1(i,j+1) - I_1(i,j-1)}{2}. \end{aligned} \tag{4.18}$$

Pour les dérivées partielles utilisées pour le gradient du flot itéré $\nabla(u_d^{k+1})$, les différences finies décentrées en avant sont utilisées telles que :

$$\begin{aligned} \frac{\partial u_d}{\partial x}(i,j) &\approx u_d(i+1,j) - u_d(i,j), \\ \frac{\partial u_d}{\partial y}(i,j) &\approx u_d(i,j+1) - u_d(i,j). \end{aligned} \tag{4.19}$$

Enfin, pour l'estimation des divergences des variables duales $\mathbf{p}_1$ et $\mathbf{p}_2$, on utilise les différences finies décentrées en arrière telles que :

$$\operatorname{div} \mathbf{p}_d(i,j) \approx (p_{d1}(i,j) - p_{d1}(i-1,j)) + (p_{d2}(i,j) - p_{d2}(i,j-1)). \tag{4.20}$$

Le schéma itératif complet consiste donc à une mise à jour de la variable auxiliaire $\mathbf{v}^k$ en utilisant les équations 4.14 et 4.15, suivi d'une mise à jour du flot itéré $\mathbf{u}^k$ en utilisant l'équation 4.16 et enfin une mise à jour des variables duales $\mathbf{P}^k$ en utilisant l'équation 4.17. Ce schéma est itéré jusqu'à la convergence du flot vers une valeur stable.

Cette méthode présente certains avantages par rapport à la méthode de Horn-Schunck présentée précédemment.

De par l'utilisation de contraintes de norme 1 à la place d'une contrainte quadratique, la méthode TV-L^1 permet la présence de discontinuités dans l'image : les dérivées spatiales faibles et donc les transitions lisses sont favorisées dans le terme quadratique de régularisation $\mathcal{R}$. Ce terme de norme 1 permet également de réduire la sensibilité de la méthode au bruit dans les images d'entrées.

4.3 Méthode Horn-Schunck

Introduite en 1981, la méthode de Horn et de Schunck [HS81] propose de faire l'hypothèse supplémentaire que la meilleure solution est la solution la plus globalement régulière. Autrement dit, l'intensité lumineuse d'un pixel est proche de celle de ses voisins.

Le problème d'origine 4.7 est formulé sous forme d'une minimisation de fonctionnelle :

$$E_d = \iint_{\Omega} (\mathbf{u} \cdot \nabla I + I_t)^2 \, dxdy. \tag{4.21}$$

En effet, trouver le champ de vecteurs minimisant 4.21 revient à trouver $\mathbf{u}$ tel que l'équation de contrainte du flot optique 4.7 soit nulle. Horn-Schunck ajoute à cette fonctionnelle le critère de régularité du flot tel que :

$$E_r = \alpha^2 \iint_{\Omega} |\nabla u_1|^2 + |\nabla u_2|^2 \, dxdy, \tag{4.22}$$

avec $|\nabla u_1|^2$ et $|\nabla u_2|^2$ les carrés des magnitudes des gradients des composantes du flot et α^2 un coefficient de pondération utilisé pour corriger les problèmes causés par le bruit dans les images en entrée dans le calcul des dérivées partielles de I. Plus la valeur de α^2 est grande et plus le flot optique estimé sera lisse et uniforme. La fonctionnelle globale à minimiser est donc :

$$\begin{aligned} E_{\text{HS}} &= \iint_{\Omega} \underbrace{(\mathbf{u} \cdot \nabla I + I_t)^2}_{\mathcal{D}} + \underbrace{\alpha^2 \left(|\nabla u_1|^2 + |\nabla u_2|^2 \right)}_{R} dxdy \\ &= \iint_{\Omega} \underbrace{(I_x u_1 + I_y u_2 + I_t)^2}_{\mathcal{D}} + \underbrace{\alpha^2 \left[\left(\frac{\partial u_1}{\partial x} \right)^2 + \left(\frac{\partial u_1}{\partial y} \right)^2 + \left(\frac{\partial u_2}{\partial x} \right)^2 + \left(\frac{\partial u_2}{\partial y} \right)^2 \right]}_{\mathcal{R}} dxdy, \end{aligned} \tag{4.23}$$

avec $\mathcal{D}$ le terme d'adéquation aux données correspondant à la contrainte du flot optique et avec $\mathcal{R}$ le terme de régularisation supplémentaire.

Afin de trouver le minimum de cette fonctionnelle, les équations d'Euler-Lagrange sont utilisées et nous donnent que le minimum de cette fonctionnelle est atteint pour :

$$\begin{cases} I_x(I_x u_1 + I_y u_2 + I_t) - \alpha^2(\Delta u_1) = 0 \\ I_y(I_x u_1 + I_y u_2 + I_t) - \alpha^2(\Delta u_2) = 0 \end{cases}, \tag{4.24}$$

avec Δu_1 et Δu_2 les laplaciens des composantes du flot optique.

Schéma numérique

Pour l'application du schéma itératif, il est nécessaire d'approximer les dérivées partielles de I ainsi que les laplaciens de u_1 et u_2. Une approximation par différences finies est utilisée pour calculer les dérivées partielles autour d'un point en utilisant les deux images d'entrée I_0 et I_1 :

$$\begin{aligned}
I_x(i,j) \approx \frac{1}{4}(& I_0(i,j{+}1) - I_0(i,j) + I_0(i{+}1,j{+}1) - I_0(i{+}1,j) \\
& + I_1(i,j{+}1) - I_1(i,j) + I_1(i{+}1,j{+}1) - I_1(i{+}1,j)), \\
I_y(i,j) \approx \frac{1}{4}(& I_0(i{+}1,j) - I_0(i,j) + I_0(i{+}1,j{+}1) - I_0(i,j{+}1) \\
& + I_1(i{+}1,j) - I_1(i,j) + I_1(i{+}1,j{+}1) - I_1(i,j{+}1)), \\
I_t(i,j) \approx \frac{1}{4}(& I_1(i,j) - I_0(i,j) + I_1(i{+}1,j) - I_0(i{+}1,j) \\
& + I_1(i,j{+}1) - I_0(i,j{+}1) + I_1(i{+}1,j{+}1) - I_0(i{+}1,j)),
\end{aligned} \tag{4.25}$$

avec $I_k(i,j)$ le pixel (i,j) de la $k$ième image de la séquence d'images. Le laplacien est également approximé et discrétisé en utilisant les différences finies :

$$\begin{aligned}
\Delta u_1 &\approx (\overline{u_1}(i,j) - u_1(i,j)) \\
\Delta u_2 &\approx (\overline{u_2}(i,j) - u_2(i,j)),
\end{aligned} \tag{4.26}$$

avec $\overline{u_k}(i,j)$ une moyenne locale de la composante k du flot optique centrée en (i,j) et $u_k(i,j)$ la valeur de la composante k du flot optique en (i,j). La méthode d'origine utilise les coefficients suivants pour le calcul de la moyenne :

$$\begin{aligned}
\overline{u_1}(i,j) := & \frac{1}{6}(u_1(i{-}1,j) + u_1(i,j{+}1) + u_1(i{+}1,j) + u_1(i,j{-}1)) \\
& + \frac{1}{12}(u_1(i{-}1,j{-}1) + u_1(i{-}1,j{+}1) + u_1(i{+}1,j{+}1) + u_1(i{+}1,j{-}1)), \\
\overline{u_2}(i,j) := & \frac{1}{6}(u_2(i{-}1,j) + u_2(i,j{+}1) + u_2(i{+}1,j) + u_2(i,j{-}1)) \\
& + \frac{1}{12}(u_2(i{-}1,j{-}1) + u_2(i{-}1,j{+}1) + u_2(i{+}1,j{+}1) + u_2(i{+}1,j{-}1)).
\end{aligned} \tag{4.27}$$

Grâce aux approximations introduites précédemment, il est possible de réécrire le système d'équations 4.24 sous sa forme approximée :

$$\begin{cases}
(\alpha^2 + I_x^2 + I_y^2)(u_1 - \overline{u_1}) = -I_x(I_x\overline{u_1} + I_y\overline{u_2} + I_t) \\
(\alpha^2 + I_x^2 + I_y^2)(u_2 - \overline{u_2}) = -I_y(I_x\overline{u_1} + I_y\overline{u_2} + I_t)
\end{cases}. \tag{4.28}$$

Finalement, nous avons un système de deux équations à résoudre pour chaque point de l'image. Il est coûteux de résoudre tous ces systèmes en utilisant des méthodes directes comme la méthode du pivot de Gauss. En effet, la matrice correspondant au système linéaire complet pour toute une image contient deux fois plus de lignes et deux fois plus de colonnes que l'image d'origine. Horn et Schunck ont donc fait le choix de résoudre ce système en utilisant la méthode itérative de Jacobi. On obtient ainsi le schéma itératif final de la méthode de Horn-Schunck d'estimation du flot optique :

$$\begin{cases} u_1^{n+1} := \overline{u_1}^n - I_x \dfrac{I_x \overline{u_1}^n + I_y \overline{u_2}^n + I_t}{\alpha^2 + I_x^2 + I_y^2} \\ u_2^{n+1} := \overline{u_2}^n - I_y \dfrac{I_x \overline{u_1}^n + I_y \overline{u_2}^n + I_t}{\alpha^2 + I_x^2 + I_y^2} \end{cases}. \tag{4.29}$$

Ce schéma est itéré un certain nombre de fois, jusqu'à obtenir une convergence à des valeurs stables. Au final, la méthode d'Horn-Schunck est l'une des méthodes les plus simples pour l'estimation d'un flot optique dense.

4.4 Synthèse des méthodes

Nous avons décrit les schémas numériques pour les algorithmes TV-L^1 et de Horn-Schunck d'estimation du flot optique. Ces deux méthodes variationnelles se basent sur la contrainte fondamentale du flot optique et utilisent chacune des contraintes supplémentaires et des minimisations de fonctionnelles. Dans le cas de TV-L^1, la méthode utilise un terme de données de norme 1 et un terme de régularisation utilisant les variations totales du flot dans les images. Pour Horn-Schunck, le terme de données est de norme 2 et le terme de régularisation utilise le laplacien du flot. L'utilisation d'un terme de données de norme 1 permet une meilleure gestion des discontinuités dans le flot. Le terme de données de norme 2 dans la méthode de Horn-Schunck pénalise les discontinuités dans le flot optique : le flot optique estimé est supposé relativement lisse en fonction du paramètre de l'algorithme α^2. La méthode de Horn-Schunck est également plus sensible au bruit dans la séquence vidéo que la méthode TV-L^1 de par l'utilisation de la magnitude des gradients au carré des images d'entrées.

Ces deux méthodes nécessitent également l'emploi d'une pyramide d'images afin de gérer les déplacements supérieurs à 1 pixel. En effet, la linéarisation de la contrainte du flot optique 4.6 n'est valable que pour des petits déplacements. L'estimation des gradients spatiaux et temporels n'est également valable que pour des petits déplacements, la taille des noyaux faisant au maximum 3×3 pixels. Nous

décrivons dans la suite de ce chapitre les caractéristiques de la pyramide d'images à utiliser pour ces deux méthodes.

4.5 Pyramide multi-résolution

Comme vu précédemment, les méthodes de Horn-Schunck et TV-L^1 ne sont valides que pour des petits déplacements. Pour le cas de grands déplacements, il est alors nécessaire d'intégrer ces méthodes dans un schéma englobant pour dépasser cette limitation. Une stratégie d'affinage de la résolution est employée. Il existe deux grandes approches pour cette stratégie multi-résolution :

— la première consiste à créer une pyramide d'images sous-échantillonnées jusqu'à l'obtention de mouvements relatifs suffisamment petits dans les images,

— la deuxième consiste à créer un espace linéaire de mise à l'échelle consistant à appliquer un même filtre de lissage – en général un filtrage gaussien – sur les images d'entrées jusqu'à un lissage suffisant des images et des grands mouvements.

La méthode *mono-échelle* choisie peut alors être exécutée à chaque niveau de pyramide. La méthode retenue dans nos travaux est celle de la pyramide d'images. Il nous faut alors construire une suite d'images de résolutions de plus en plus petites. Le théorème de Shannon nous indique que pour ce faire, un filtre anti-repliement (« anti-aliasing » en anglais) adéquat doit être utilisé afin de restreindre la bande de fréquences des images et ainsi éviter l'apparition d'artefacts de crénelage.

Le flot obtenu est ensuite mis à l'échelle pour le niveau suivant de la pyramide. Le flot à ce niveau est utilisé pour compenser les grands mouvements précédemment obtenus et ainsi permettre l'exécution de la méthode choisie à ce niveau. Ces opérations de compensation du mouvement, calcul de flot à un niveau et de transmission à l'étage suivant sont ainsi répétées du niveau ayant la résolution la plus basse jusqu'au niveau de résolution le plus haut.

Il faut faire attention à la façon de raffiner le flot optique global d'un étage à un autre selon l'algorithme utilisé. Deux différentes stratégies sont utilisées dans nos travaux : une stratégie de raffinage *globale* et une stratégie *incrémentale*. La méthode TV-L^1 utilise une pyramide globale où les valeurs du flot estimé sont itérées directement d'étage en étage, alors que la méthode de Horn-Schunck fait ainsi usage d'une pyramide incrémentale où un étage estime localement la valeur du flot qui est ensuite ajouté à l'estimation précédente provenant des étages inférieurs.

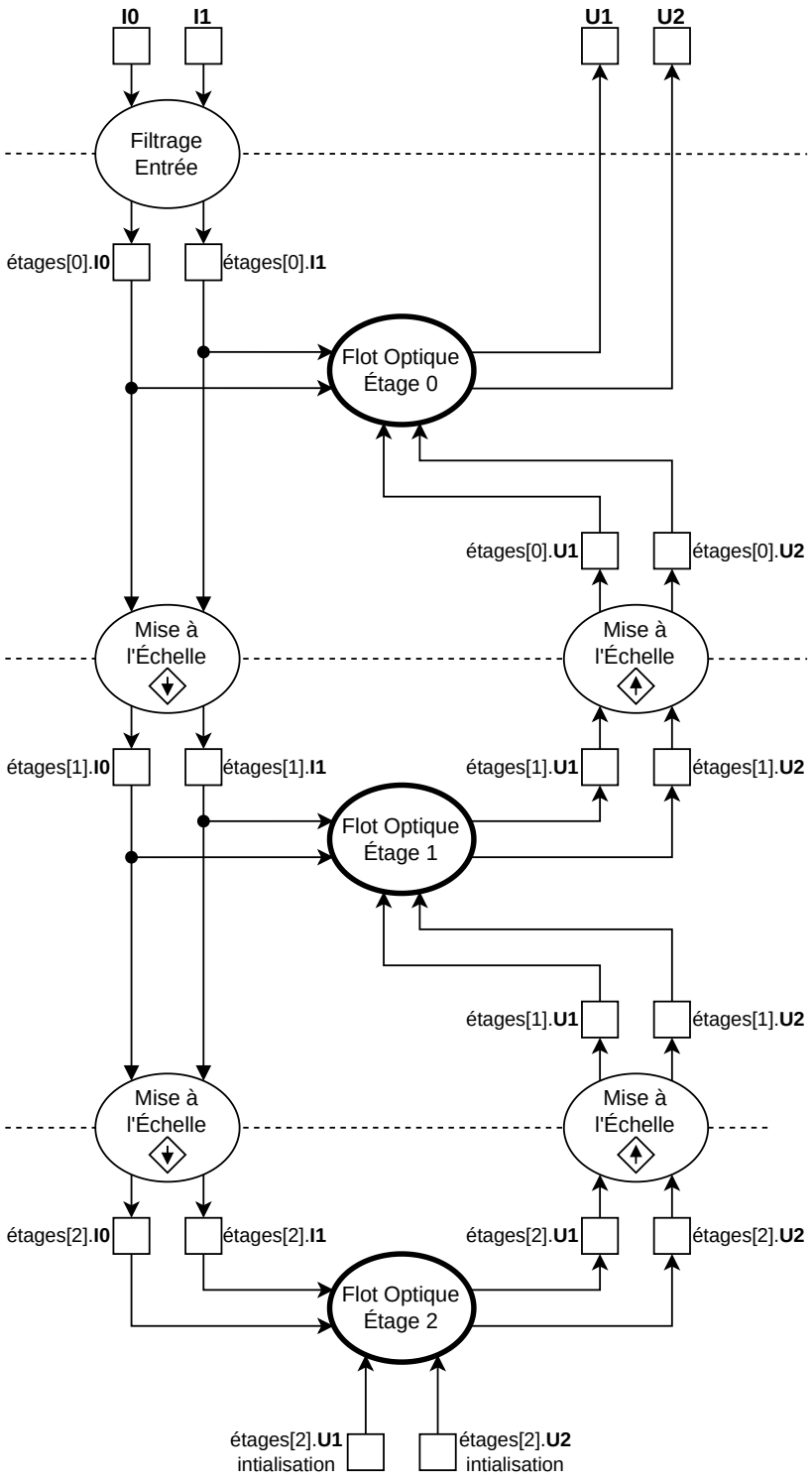


Fig. 4.1 : Schéma fonctionnel des algorithmes pyramidaux d'estimation du flot optique. La structure pyramidale est commune aux algorithmes. Les blocs « Flot Optique Étage n » sont eux propres à l'algorithme choisi.

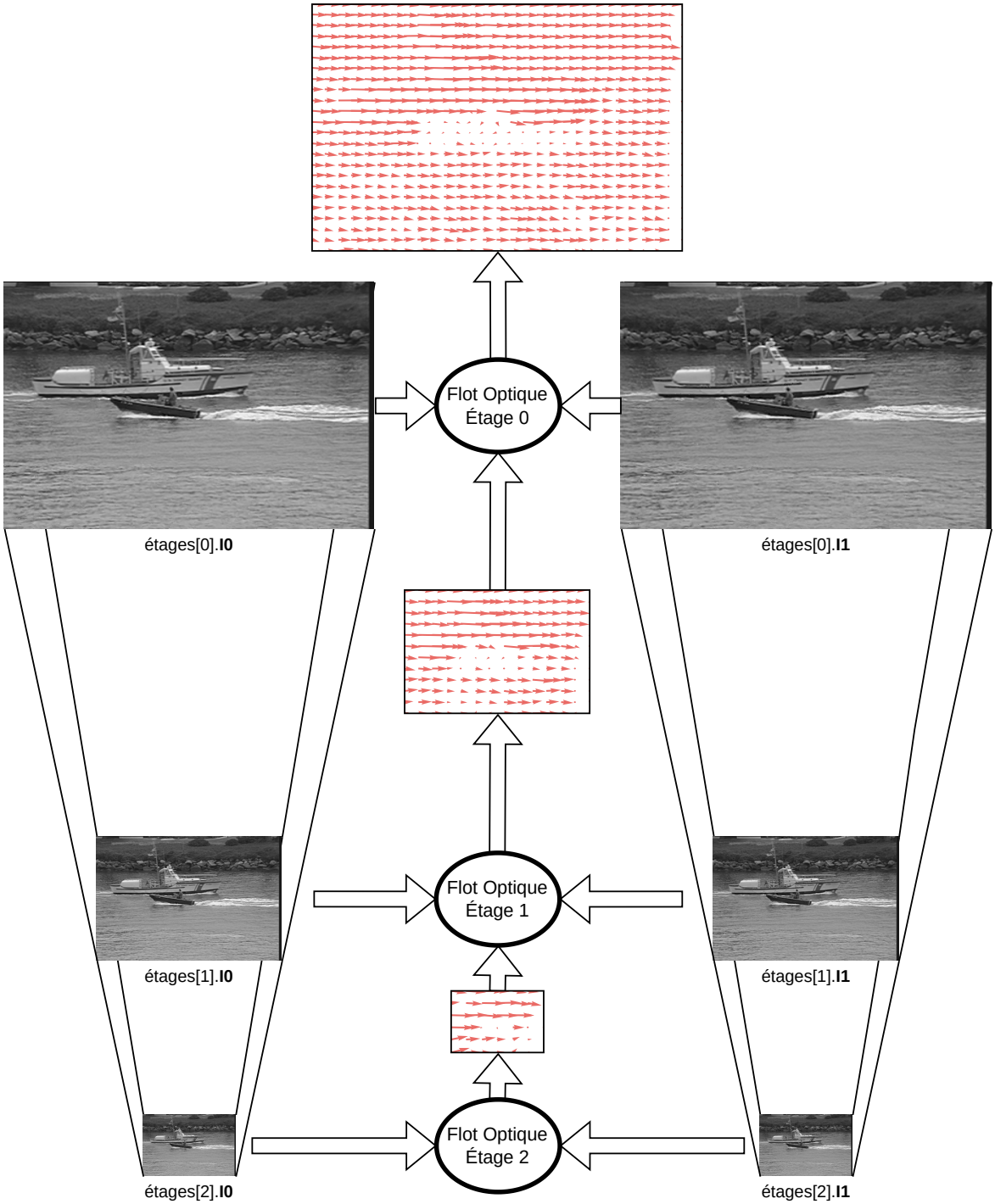


Fig. 4.2 : Représentation du champ de vecteurs en sortie et en entrée de chaque étage de la pyramide. Le résultat final est produit à la sortie de l'étage 0.

4.5.1 Méthode TV-L^1

Dans le cas de TV-L^1, le schéma fait directement usage de l'estimation initiale du flot optique $\mathbf{u}^0$ dans le calcul du gradient : on compense ce mouvement dans le gradient par warping. Ainsi, la version mono-échelle peut être directement utilisée à chaque étage sans avoir à calculer des flots locaux incrémentaux. C'est le cas puisque la méthode mono-échelle fait bien usage de l'estimation initiale du flot optique $\mathbf{u}^0$: l'étape de warping de compensation des grands mouvements fait partie de la méthode. Un flot global unique raffiné à chaque étage est donc utilisé. Le raffinement du flot optique à un étage s se déroule de la manière suivante :

1. Le flot optique global $\mathbf{u}$ provenant de l'étage inférieur $s+1$ est utilisé pour compenser les grands mouvements dans l'image I_1 d'entrée de cet étage et dans le gradient ∇I_1. Cette étape de « warping » utilise une interpolation bicubique pour estimer les pixels.
2. En utilisant ces nouvelles valeurs, le flot optique global est raffiné à cet étage : la méthode est itérée sur le flot optique global.
3. Ce flot est alors directement transmis à l'étage supérieur $s+1$ en faisant une mise à l'échelle par sur-échantillonnage du flot pour correspondre à la résolution de l'étage visé. Un sur-échantillonnage avec interpolation bilinéaire est utilisé. Il faut également multiplier la norme des vecteurs du flot optique par le facteur de réduction entre les images de la pyramide.

Des étapes de warping supplémentaires peuvent être utilisées à chaque étage pour améliorer la convergence de la méthode vers des résultats plus précis. Ces étapes sont répétées pour chaque étage de la pyramide en partant de l'étage de plus petite résolution, avec un flot optique $\mathbf{u}$ global initialisé à 0.

4.5.2 Méthode Horn-Schunck

Dans le cas de Horn-Schunck, le gradient présent dans le terme d'adéquation aux données en équation 4.21 ne prend pas en entrée une estimation initiale du flot optique $\mathbf{u}^0$. La compensation des grands mouvements calculés aux étages inférieurs de la pyramide est séparée de la méthode mono-échelle et ne s'opère que sur l'image d'entrée I_1. Plusieurs tableaux d'accumulation de flot optique sont donc nécessaires : un flot global à la pyramide dans lequel on obtiendra le résultat final et des flots incrémentaux locaux à chaque étage. La méthode mono-échelle va ainsi calculer un flot optique local initialisé à 0 à chaque étage. Ce flot local va ensuite être ajouté au flot optique global provenant des étages inférieurs, puis mis à l'échelle et propagé à

l'étage suivant. Le raffinement du flot optique à un étage s se déroule de la manière suivante :

1. Le flot optique global $\mathbf{u}$ provenant de l'étage inférieur $s+1$ est utilisé pour compenser les grands mouvements dans l'image I_1 d'entrée de cet étage. Cette étape de « warping » utilise une interpolation bicubique pour estimer les pixels.
2. Cette nouvelle image warpée $I_{1w} := I_1(\mathbf{x} - \mathbf{u})$ est utilisée en entrée de Horn-Schunck.
3. Un flot optique local à cet étage est alors calculé : l'algorithme ne prend pas en entrée $\mathbf{u}$ mais un flot optique local $\mathbf{du} := (du_1, du_2)$ initialisé à 0.
4. Une fois que l'algorithme a terminé, $\mathbf{du}$ contient donc les déplacements trouvés entre les images I_0 et I_{1w} de cet étage. Il faut alors ajouter ce $\mathbf{du}$ contenant les déplacements locaux relativement petits au flot global accumulateur $\mathbf{u}$.
5. Ce flot est alors transmis à l'étage supérieur s+1 en faisant une mise à l'échelle par sur-échantillonnage du flot pour correspondre à la résolution de l'étage visé. Un sur-échantillonnage avec interpolation bilinéaire est utilisé. Il faut également multiplier la norme des vecteurs du flot optique par le facteur de réduction entre les images de la pyramide.

Ces étapes sont répétées pour chaque étage de la pyramide en partant de l'étage de plus petite résolution, avec un flot optique $\mathbf{u}$ global initialisé à 0.

4.6 Exploration des optimisations sur GPU

Dans cette section, nous présentons les versions GPU qui ont été développées pour les algorithmes de TV-L^1 et de Horn-Schunck. Lors de nos travaux, nous avons commencé nos développements avec l'algorithme de TV-L^1. Une plus grande exploration a ainsi été réalisée et certaines versions n'ont pas été retenues pour les phases de tests. Les versions retenues ont ensuite servi de pistes d'exploration pour l'algorithme de Horn-Schunck. Seules les versions mono-échelles sont décrites ici. Elles sont toutes intégrées dans la même pyramide gaussienne. Seul le raffinage d'un étage à un autre change en fonction de l'utilisation de Horn-Schunck ou de TV-L^1 comme indiqué précédemment.

Nous commençons par décrire le pseudo-code et la représentation consommateur/producteur associée pour les deux algorithmes résultants de la première phase d'optimisation algorithmique. Cette première phase d'optimisation haut-niveau a pour but de minimiser le nombre de kernels appelés et de maximiser la réutilisation des données. Cette phase ne comprend donc que les optimisations algorithmiques.

La deuxième phase, comprenant des optimisations bas niveau plus proches de l'architecture GPU, sera décrite ensuite. Ces optimisations correspondent à la fois à notre historique de développement, mais aussi à la complexité du code écrit : les premières versions sont moins complexes que les dernières.

4.6.1 Méthode TV-L^1

4.6.1.1 Stratégies d'implémentation

Nous commençons par reprendre le schéma numérique de l'algorithme de TV-L^1 décrit en sous-section 4.2. Nous obtenons le pseudo-code en algorithme 1 avec une fonction de calcul par étape du schéma.

Algorithme 1 Algorithme TV-L^1 mono-échelle de base

Entrée: I_0, I_1, $\mathbf{u}^0$, Max_{warps}, Max_{iters}, τ, λ, θ
  $\mathbf{v} \leftarrow 0$, $\mathbf{P} \leftarrow 0$
  Calcul `gradient_central` pour ∇I_1 (équation 4.18)
  pour $n_w = 0$ **à** Max_{warps} **faire**
    Calcul `warping` pour $I_1(\mathbf{x}+\mathbf{u}^{n_w})$, $\nabla I_1(\mathbf{x}+\mathbf{u}^{n_w})$ et $|\nabla I_1(\mathbf{x}+\mathbf{u}^{n_w})|^2$ (par interpolation cubique)
    pour $n_i = 0$ **à** Max_{iters} **faire**
      Calcul `seuillage` pour $\mathbf{v}^{n_i+1}$ (équations 4.14 et 4.15)
      Calcul `divergence` pour $\mathrm{div}\,\mathbf{p}_d^{n_i}$ (équation 4.20)
      Calcul `estimation_u` pour $\mathbf{u}^{n_i+1}$ (équation 4.16)
      Calcul `gradient_avant` pour $\frac{\partial u_d^{n_i+1}}{\partial x}$ et $\frac{\partial u_d^{n_i+1}}{\partial y}$ (équation 4.19)
      Calcul `estimation_P` pour $\mathbf{P}^{n_i+1}$ (équation 4.17)
    fin pour
  fin pour

Le schéma producteur/consommateur des données est présenté en figure 4.3. Cette représentation met en évidence les dépendances des données de chaque sous-section du schéma itératif et nous informe sur les potentiels endroits de réutilisation de données. Le symbole $\circ$ symbolise la composition de fonctions : la sortie d'une étape utilisée en entrée de la suivante. Un carré blanc représente une valeur du tableau d'entrée centrée en (i, j) et les carrés gris les dépendances supplémentaires horizontales en $(i, j–1)$ et $(i, j+1)$, et verticales en $(i–1, j)$ et $(i+1, j)$ quand cela est requis. On retrouve les deux boucles itératives du schéma numérique : la première boucle s'occupe de l'étape de warping de compensation des mouvements et la deuxième boucle du schéma itératif. Nous nous concentrons sur la deuxième étape du schéma étant donné qu'il s'agit de l'étape la plus répétée et qu'elle ne dépend pas du type d'interpolation choisi pour l'étape de warping [ZPB07].

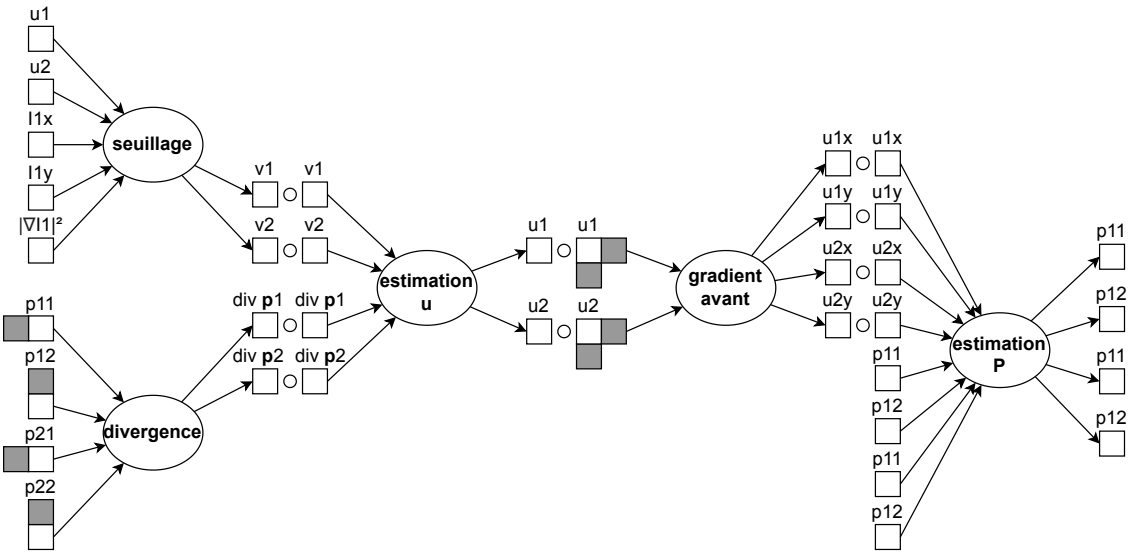


Fig. 4.3 : Schéma itératif interne de TV-L^1.

D'un point de vue structurel, les codes CUDA des implémentations mono-échelles suivront donc le motif présenté en algorithme 2. Les itérations externes sont implémentées de la manière suivante et utilisées dans toutes nos versions des implémentations :

1. Le kernel `gradient_central` qui se charge du calcul des différences finies centrées $\frac{\partial I_1}{\partial x}$ et $\frac{\partial I_1}{\partial y}$ pour le calcul du gradient de I_1 ;
2. Le kernel `warping` qui se charge du calcul des warpings bicubiques pour la compensation du flot optique dans I_1 et ses dérivées partielles.

La façon d'implémenter les itérations varieront d'une version à une autre. Nous présentons maintenant ces différentes versions. L'ordre de présentation correspond à la fois à la chronologie suivie dans notre travail mais également à la croissance en complexité du code.

Algorithme 2 Structure TV-L^1 mono-échelle commune

Entrée: I_0, I_1, $\mathbf{u}^0$, Max_{warps}, Max_{iters}, τ, λ, θ
  $\mathbf{v} \leftarrow 0$, $\mathbf{P} \leftarrow 0$
  Kernel `gradient_central` pour ∇I_1
  pour $n_w = 0$ **à** Max_{warps} **faire**
    Kernel `warping` pour $I_1(\mathbf{x}+\mathbf{u}^{n_w})$, $\nabla I_1(\mathbf{x}+\mathbf{u}^{n_w})$ et $|\nabla I_1(\mathbf{x}+\mathbf{u}^{n_w})|^2$
    pour $n_i = 0$ **à** Max_{iters} **faire**
      $\Rightarrow$ Implémentation des itérations internes de TV-L^1 $\Leftarrow$
    fin pour
  fin pour

4.6.1.2 Optimisations communes

Comme on peut le voir en figure 4.3, de nombreux opérateurs ponctuels peuvent être fusionnés. En tenant compte des dépendances de données spatiales, on obtient après fusion le schéma de dépendance montré sur la figure 4.4.

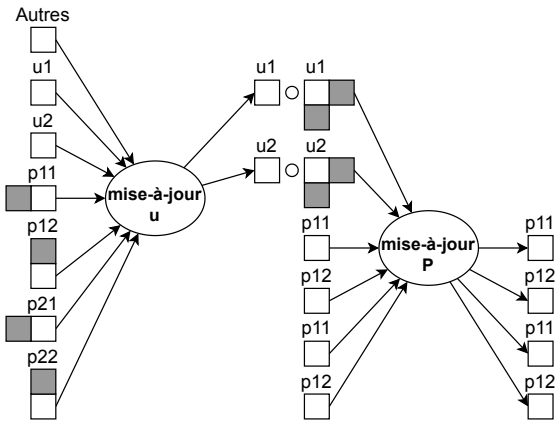


Fig. 4.4 : Schéma itératif interne de TV-L[1] après fusion des opérateurs ponctuels.

De par les dépendances à gauche et en haut pour la mise à jour de $\mathbf{u}$ et des dépendances à droite et en bas pour la mise à jour de $\mathbf{P}$, il n'est plus possible de réaliser une fusion directe supplémentaire de l'algorithme. Notre travail d'optimisation aura donc pour but de tester différentes stratégies d'enchaînement de ces deux étapes.

Version Global Hormis une implémentation directe de l'algorithme avec une fonction kernel par étape de calcul, nous commençons par présenter la version dite Global. La version Global est une implémentation directe du schéma présenté figure 4.4.

Cette implémentation ne contient désormais plus que 2 kernels et utilise uniquement la mémoire globale. Les accès à la mémoire sont réduits, la localisation des données et l'intensité arithmétique sont améliorées. La surcharge due au lancement de multiples kernels GPU est également réduite puisqu'il n'y a que deux noyaux séquentiels à exécuter par itération TV-L^1 : un pour mettre à jour $\mathbf{u}$ et un pour mettre à jour $\mathbf{P}$. Pour les accès à la bordure en dehors de l'image, l'index est bloqué et la valeur de la bordure est réutilisée (technique dite de « clamping » en anglais).

Nous détaillons maintenant la structure des 2 kernels résultants pour les 2 traitements séquentiels. Concernant les dépendances spatiales des données, afin de simplifier les représentations, nous représenterons uniquement l'union des espaces mémoire pour les entrées et les sorties des threads, blocs et kernels. Ces détails de représentation nous servent d'exemples pour le reste des implémentations. Nous ne représenterons que l'union finale obtenue pour un bloc de threads pour les autres implémentations.

La figure 4.5 a pour but de représenter à la fois l'état des mémoires en entrée et en sortie du bloc de thread CUDA mais également l'état et les dimensions du bloc de threads lui-même. Les dépendances de données spatiales en bord de thread et de blocs sont représentés en couleurs foncées. Ces données sont uniquement lues sans être modifiées par le bloc. Étant donné qu'elles ne sont pas modifiées, ces données peuvent être lues par plusieurs blocs en parallèle sans problème de dépendances de données. En effet, les dépendances spatiales sont sur un tableau différent que celui de sortie. Comme on peut le voir, l'union des données pour chaque thread du bloc forme un rectangle de données des dimensions du bloc de threads avec des bords supplémentaires variant d'un kernel à l'autre. Les threads actifs dans le bloc sont marqués par une croix. Tous les threads restent ici actifs dans l'exécution du kernel mais cela n'est pas le cas dans toutes les versions.

Pour cette version, une itération complète de TV-L^1 correspondra ainsi à un enchaînement du kernel mettant à jour $\mathbf{u}$ puis du kernel mettant à jour $\mathbf{P}$.

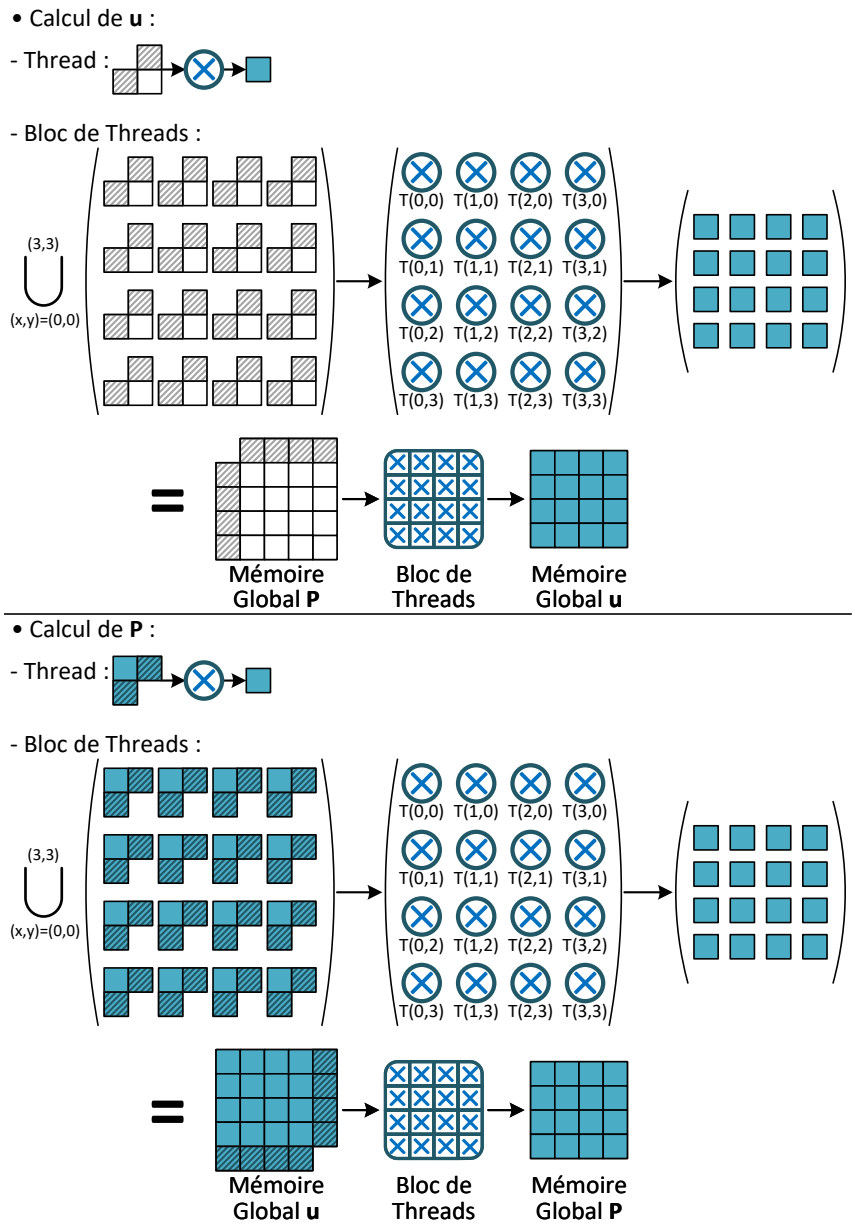


Fig. 4.5 : Représentation des deux kernels de calculs pour la version Global.

Version Shared Un thread n'a pas accès aux données de ses voisins sans utiliser de mécanismes particulier de l'API de CUDA. Comme on l'a vu précédemment, les dépendances spatiales font qu'il serait possible de réutiliser des données lues d'un thread à l'autre. Dans le cas de la version Global, cette réutilisation se fait grâce au cache de données dans le GPU. Le programmeur n'a cependant que peu de contrôle quant au mécanisme de réutilisation et d'évincement des données. Pour avoir un contrôle plus fin sur ce partage de données, il est possible d'utiliser la mémoire dite Shared. Cette mémoire est partagée par tous les threads du même bloc. Nous introduisons donc l'implémentation Shared qui fait usage de cette mémoire dans les deux kernels pour une meilleure réutilisation de données. Sa représentation schématique est illustrée en figure 4.6.

Comme on peut le voir sur le schéma, pour chaque kernels, il y a 2 étapes supplémentaires en plus de celle de calcul : une étape de chargement des données vers la mémoire Shared et une étape de stockage du résultat calculé en mémoire Global. Une synchronisation est nécessaire avant l'étape de calcul pour s'assurer que toutes les données sont chargées. On notera également que le rôle de certains threads est uniquement le chargement des données en Shared. Une fois ce chargement effectué, ces threads terminent leur exécution et ne participent pas au calcul. Ces threads sont indiqués par un **R** sur le schéma (pour indiquer un retour de fonction), et les données chargées par ces threads sont représentées par des hachures dans le schéma. Le cadre rouge indique la zone centrale calculée par les threads restants.

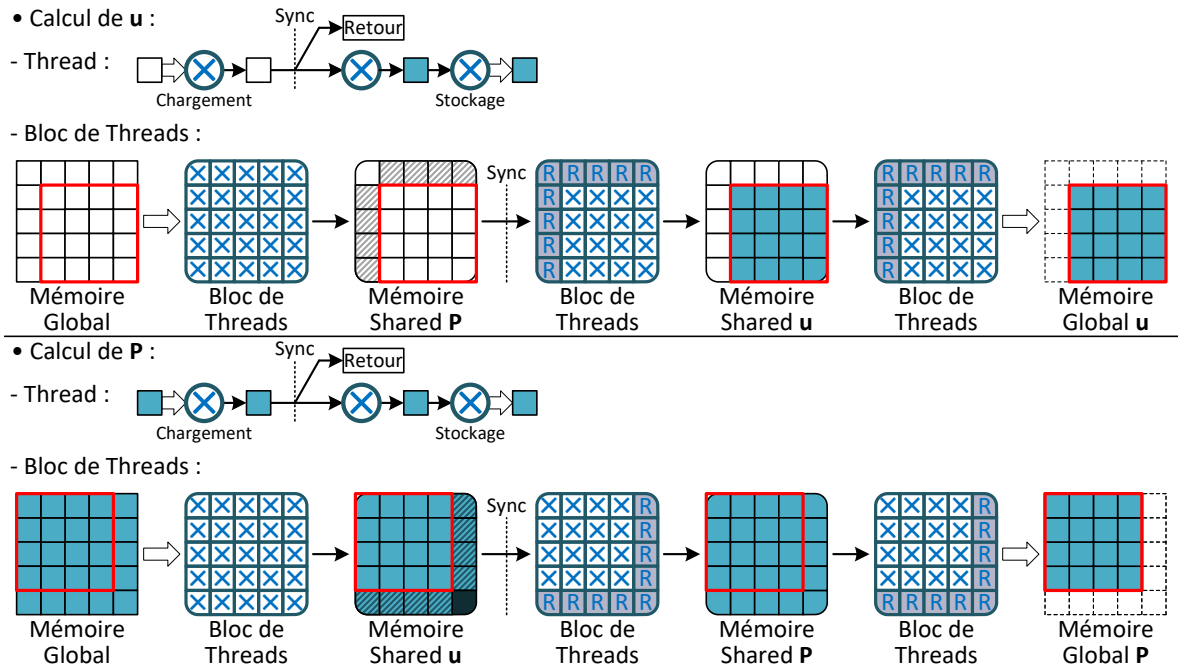


Fig. 4.6 : Représentation des deux kernels de calcul pour la version Shared.

Version Shuffle Il existe un autre mécanisme de partage de données dans l'API CUDA : les instructions « Shuffle ». Ces instructions permettent à un thread de récupérer une donnée dans un registre d'un autre thread appartenant au même warp de 32 threads. Cet échange n'est possible que dans le même warp et induit une synchronisation des threads. Pour des blocs plus grands, une gestion des bords de warps serait alors nécessaire. Nous introduisons donc l'implémentation Shuffle faisant usage de ces instructions. La représentation schématique de cette version est illustrée en figure 4.7. Les échanges de données par shuffle sont représentés par des flèches entre les threads du même bloc.

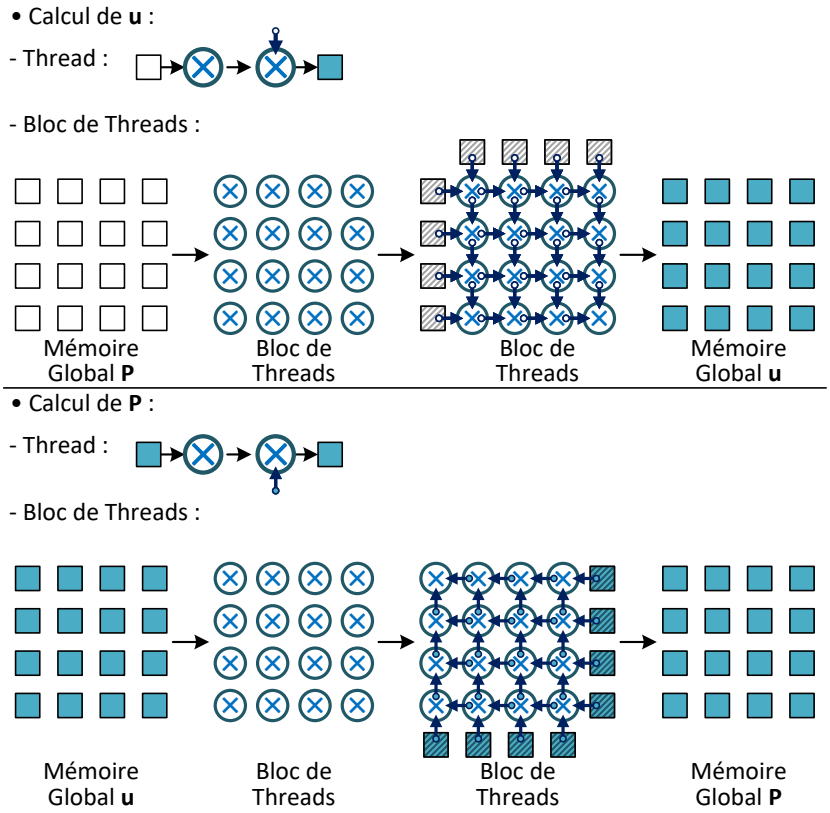


Fig. 4.7 : Représentation des deux kernels de calculs pour la version Shuffle.

Version Global_Fusion Dans les versions Shared et Shuffle, la réutilisation des données reste encore limitée par rapport à la quantité totale de données chargées pour le calcul d'un bloc. De plus, dans la version Shared, les performances sont encore réduites de par les threads en bordures ne réalisant qu'un chargement sans exécution. Il serait alors plus intéressant de réaliser une fusion complète de ces deux kernels quitte à recalculer certaines valeurs dans l'image complète afin d'améliorer la réutilisation de données et de minimiser les threads ne réalisant pas de calculs. Nous introduisons donc l'implémentation Global_Fusion en figure 4.8 réalisant cette fusion algorithmique supplémentaire et n'utilisant que la mémoire Global.

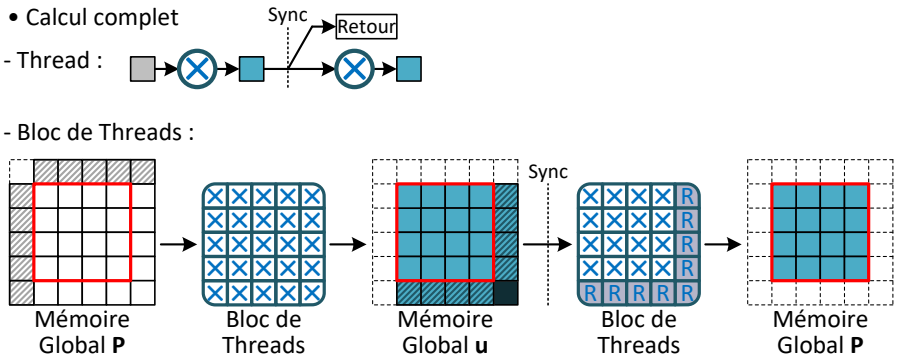


Fig. 4.8 : Représentation du kernel de calcul pour la version Global_Fusion.

Version Shared_Fusion L'évolution de la version Global_Fusion est de partager les données calculées par un thread au bloc complet via la mémoire Shared. Cette nouvelle version Shared_Fusion reprend les étapes de chargement en mémoire Shared, de calcul et de stockage en mémoire Global de la version Shared tout en réalisant la fusion de la version Global_Fusion. L'utilisation de la mémoire Shared permet de réduire les accès à la mémoire Global mais aussi une réutilisation des données à la fois au sein d'un thread, mais également au sein du bloc de threads. Tout comme la version Shared, une barrière de synchronisation est nécessaire pour garantir que toutes les données requises sont chargées dans la mémoire Shared avant le calcul de l'itération TV-L[1]. De plus, certains threads sont utilisés uniquement pour charger des données de bordure qui ne sont utilisées par les threads voisins. Ces threads terminent leur exécution après avoir effectué l'étape de chargement et n'effectuent pas l'étape de calcul. Enfin, comme la version Global_Fusion, il n'y a plus qu'un seul kernel à exécuter. Son schéma d'exécution est présenté en figure 4.9.

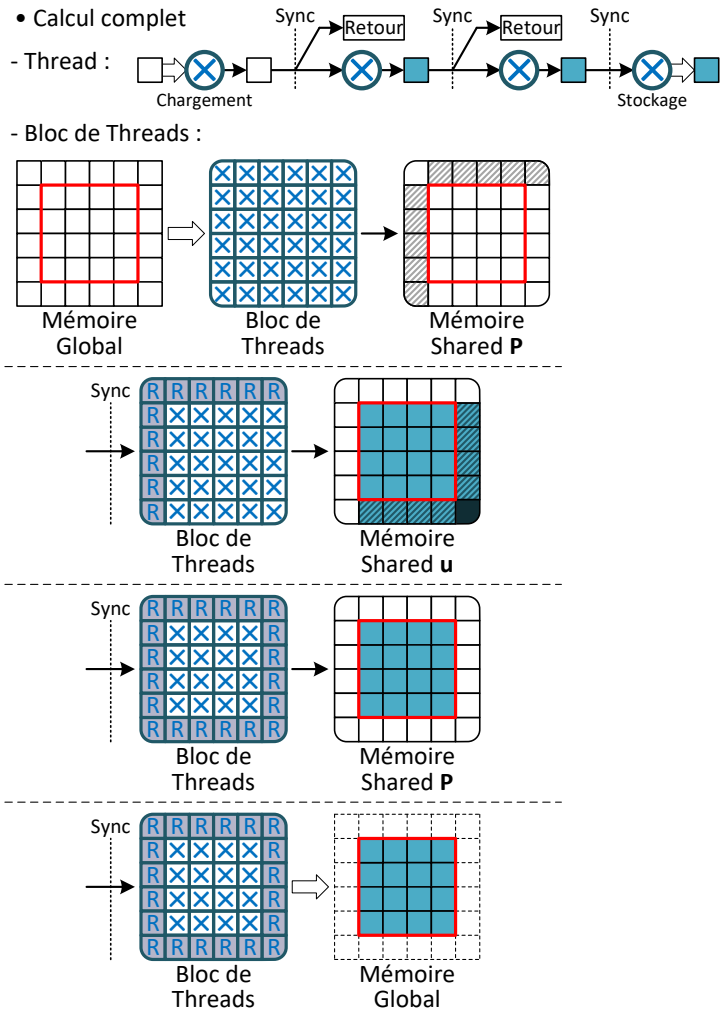


Fig. 4.9 : Représentation du kernel de calcul pour la version Shared_Fusion.

Version Global_Pipeline La version Global_Pipeline met en œuvre le pipeline d'opérateurs décrit dans le chapitre précédant en utilisant uniquement la mémoire Global. L'idée est ici d'utiliser une ligne mise à jour le plus tôt possible pour pouvoir calculer des itérations supplémentaires sur des lignes voisines. Ce pipeline itératif ou temporel permet une réduction des accès à la mémoire Global afin d'améliorer la localité des données dans les caches. Son schéma d'exécution est illustré en figure 4.10.

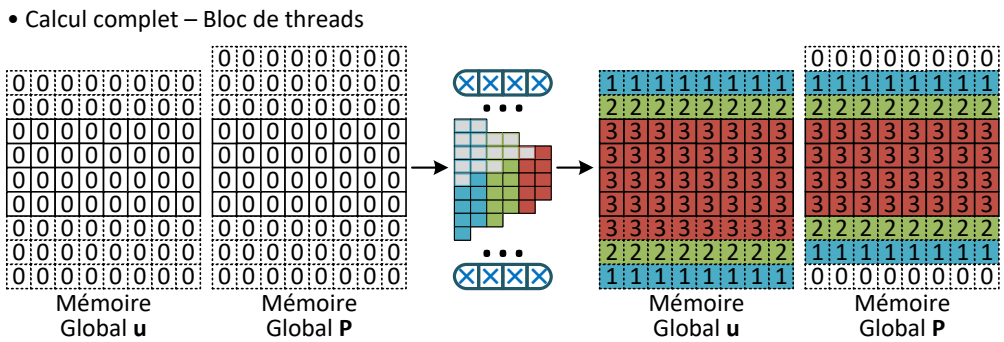


Fig. 4.10 : Représentation du kernel de calcul pour la version Global_Pipeline.

Le pipeline suit les dépendances verticales de TV-L[1] : une ligne complète est entièrement traitée et utilisée dès que possible pour calculer les étapes d'itération suivantes. Si le nombre de threads est inférieur à la largeur de la ligne, les lignes sont calculées en plusieurs étapes par le même bloc de threads. L'ordre de calcul de cette version est illustré à la figure 4.11. La figure 4.12 représente ce même schéma, mais sous forme chronographique simplifiée. Le prologue et l'épilogue sont exécutés dans des lignes frontalières allouées à cet effet pour les tableaux **u** et **P**. Comme vu depuis la version Shared, la taille des blocs et la taille de la zone traitée par un bloc sont différentes. Pour cette version, ces deux tailles ne sont pas liées : on fixe un nombre de blocs qui va traiter une paire d'images entières. Chaque bloc va traiter une sous-partie du flot à calculer.

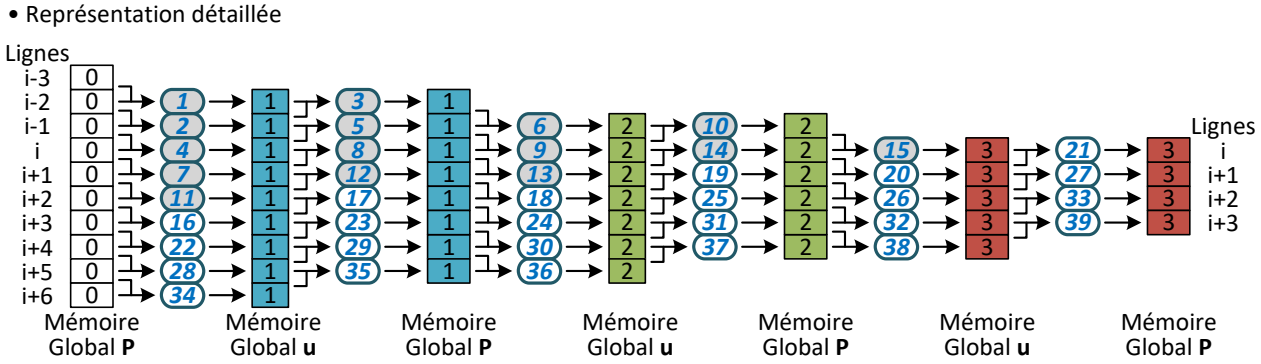


Fig. 4.11 : Ordre des calculs pour la version Global_Pipeline pour un pipeline de 3 itérations sur 4 lignes consécutives. On représente ici les accès aux lignes pour un bloc complet.

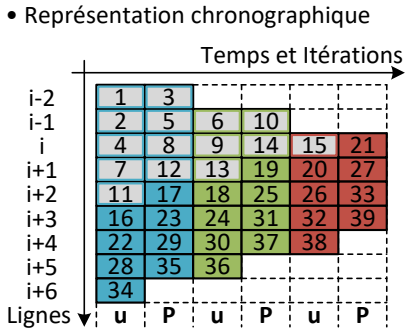


Fig. 4.12 : Schéma simplifié de l'ordre des calculs pour la version Global_Pipeline.

Version Shared_Pipeline La version Shared_pipeline implémente également le même pipeline, mais utilise la mémoire Shared pour partager les données déjà calculées par un thread avec les autres threads du même bloc. Les accès modulaires sont utilisés pour réduire le nombre de lignes nécessaires aux étapes de calcul. On représente en figure 4.13 son schéma d'exécution.

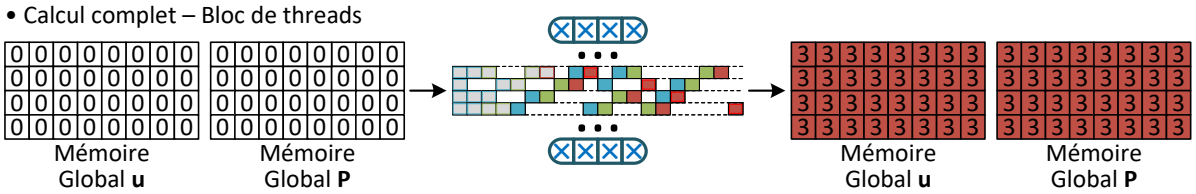


Fig. 4.13 : Représentation du kernel de calcul pour la version Shared_Pipeline.

Alors que la mémoire Global permet des blocs et des pipelines de taille quelconque (dans la limite de la mémoire vive), la quantité limitée de mémoire Shared nous limite dans les deux cas. La dynamique des nombres utilisés nous limite également en fonction de la taille de ces derniers :

— 16 bits par donnée pour un nombre scalaire flottant demi précision (*half*),
— 32 bits par donnée pour un nombre scalaire flottant simple précision (*float*),
— 32 bits par donnée pour un nombre regroupant deux sous-mots flottants demi précision (*half2*),
— 64 bits par donnée pour un nombre regroupant deux sous-mots flottants simple précision (*float2*).

Aussi, il n'est pas possible de traiter une ligne complète avant de passer à la suivante. Plusieurs passes verticales sont alors nécessaires pour traiter la zone attribuée à un bloc. La figure 4.14 détaille l'ordre des calculs et représente les dépendances pour le calcul de chaque ligne.

• Représentation détaillée

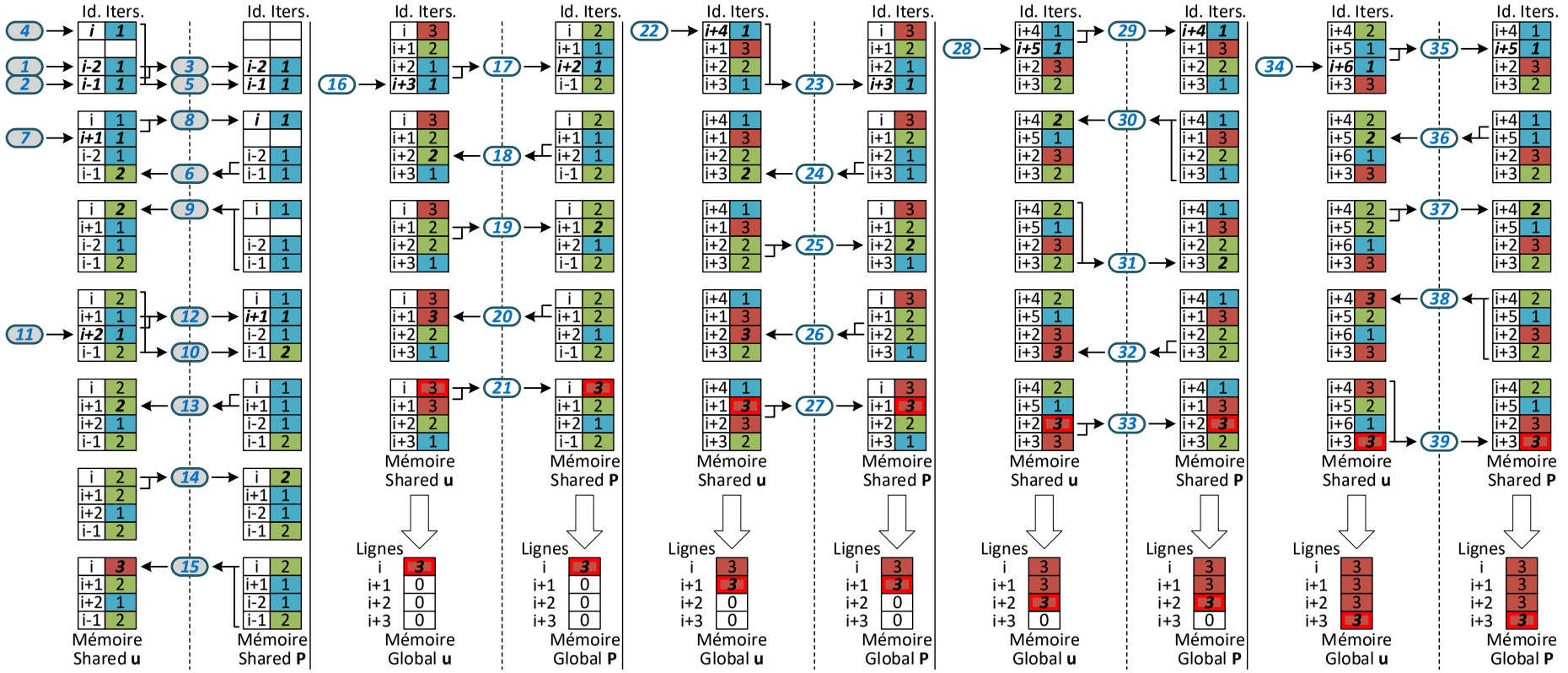


Fig. 4.14 : Détail de l'ordre des calculs pour la version Shared_Pipeline pour un pipeline de 3 itérations sur 4 lignes consécutives. On représente ici les accès modulaires dans les lignes de la mémoire Shared ainsi que les accès de stockage dans la mémoire Global.

Version Global_MItIteration Comme on vient de le voir, les versions implémentant un pipeline temporel visent à minimiser les accès à la mémoire Global tout en maximisant la localité temporelle des données dans le cache. On peut cependant remarquer que pour une profondeur de pipeline trop grande, les écarts entre les lignes pour une étape du régime permanent peuvent devenir trop grands, limitant ainsi la localité. Une autre technique permettant d'augmenter la localité des données consiste à fusionner des itérations multiples, comme fait dans les versions Fusion. Cette technique de *fusions multiples* (ou composition) consiste à allouer des blocs de thread contenant toutes les données nécessaires au calcul d'un certain nombre d'itérations pour une zone centrale dans le bloc [Cas14]. Cette implémentation permet ainsi aux mêmes données de rester plus longtemps en cache (« *Temporal Blocking* » en anglais [ZPM18]). Les threads du bloc vont donc itérer jusqu'à un certain point et arrêter leur exécution. Le halo de threads ne réalisant plus de calculs va devenir de plus en plus grand à mesure que le kernel va calculer de plus en plus d'itérations. De plus, un grand nombre de pixels vont être recalculé par plusieurs blocs en fonction du chevauchement de leurs halos de pixels. Ce chevauchement va également provoquer la lecture de pixels déjà mis à jour par les blocs voisins dont les halos se recoupent. Il faudra donc faire attention à ces effets de bords de blocs pour ce qui est de la qualité du flot calculé.

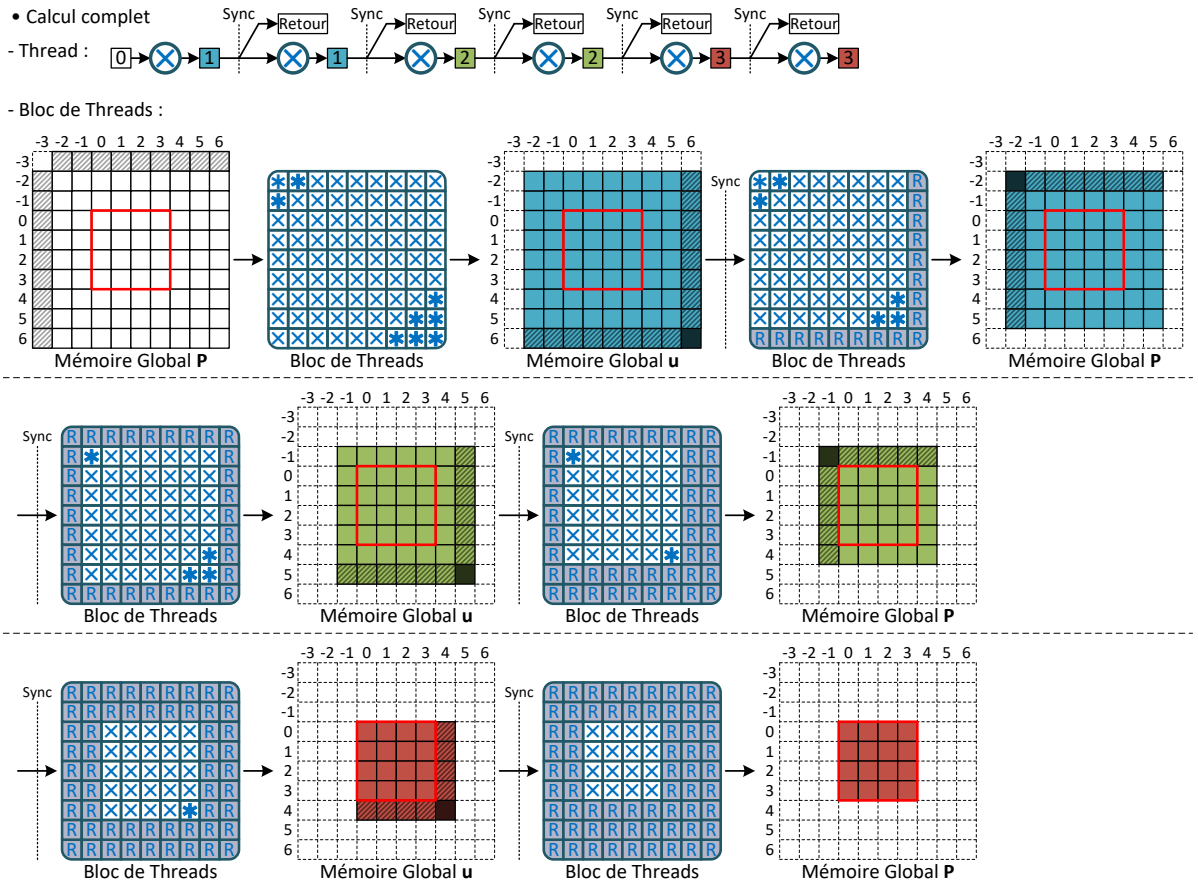


Fig. 4.15 : Représentation du kernel de calcul pour la version Global_MItIteration pour une fusion de 3 itérations et pour une zone de 4×4 pixels à calculer.

Version Shared_MItIteration La dernière version développée en date est l'implémentation Shared_MItIteration. Cette version implémente la même fusion multiple que dans la version Global_MItIteration mais utilise la mémoire Shared comme mémoire temporaire rapide. Comme dans la version Shared_Pipeline, l'utilisation de la mémoire Shared limite la taille des blocs ainsi que le nombre d'itérations fusionnées possibles. Cependant, là encore, on limite le nombre d'accès à la mémoire Global plus lente et l'on augmente la localité spatiale et temporelle des données. De plus, grâce à l'utilisation de cette mémoire qui est privée pour un bloc de threads donné, il n'y a plus d'effet de bords de blocs. Le résultat est ainsi identique à une version mono-échelle de base sans optimisations. Le schéma d'exécution et d'accès aux mémoires est représenté en figure 4.16.

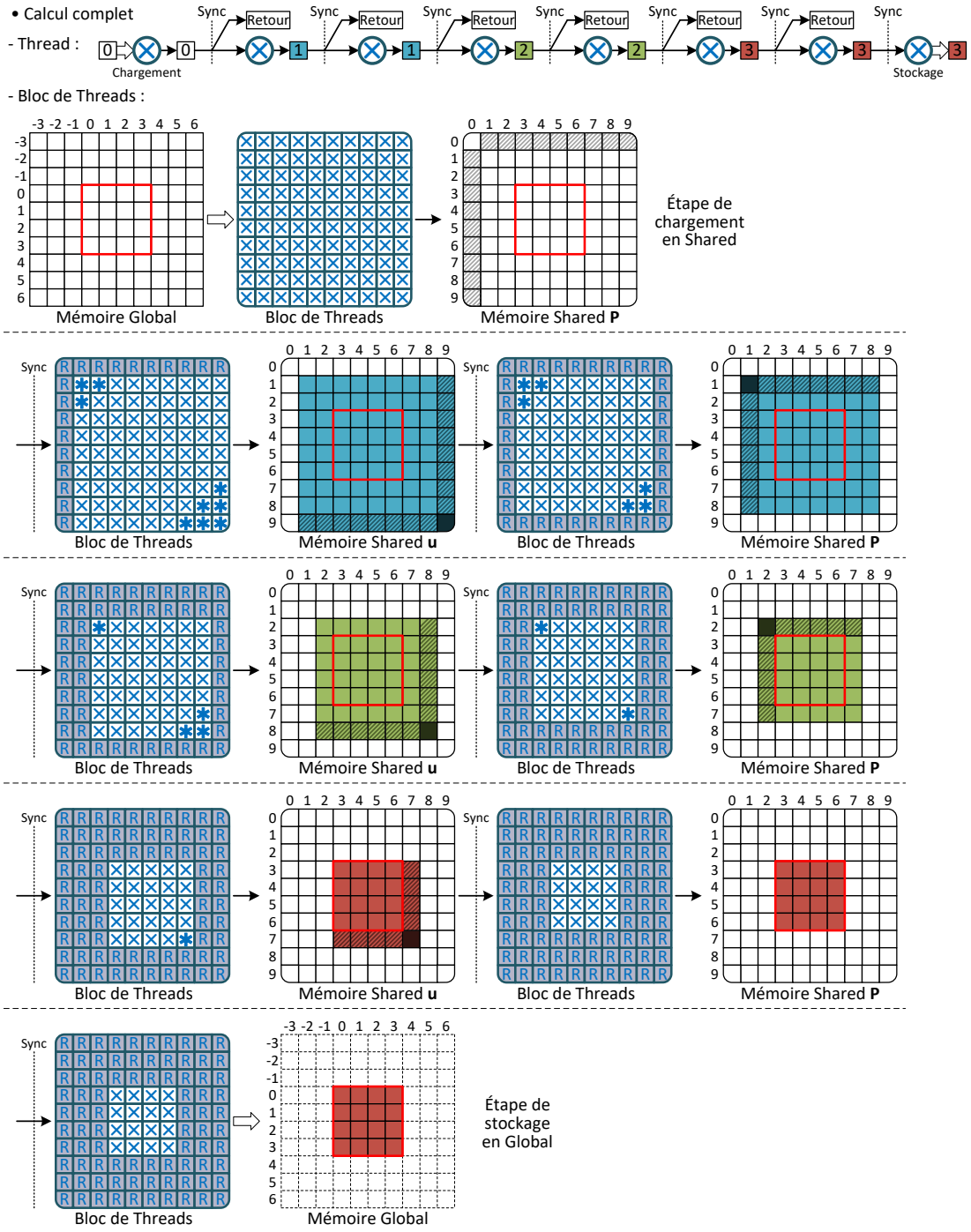


Fig. 4.16 : Représentation du kernel de calcul pour la version Shared_MltIteration pour une fusion de 3 itérations et pour une zone de 4×4 pixels à calculer.

4.6.2 Méthode de Horn et Schunck

Nous décrivons maintenant les différentes versions développées pour la méthode de Horn-Schunck. Nous nous sommes basés sur les versions développées pour TV-L^1. Étant donné que les deux algorithmes correspondent à des algorithmes de type stencils, il est naturel d'appliquer les mêmes optimisations aux deux algorithmes. En l'occurrence, l'idée est d'appliquer les mêmes patrons d'accès à l'algorithme de Horn-Schunck. Les changements fondamentaux correspondent aux calculs faits à l'intérieur des threads. De plus, forts de notre expérience avec l'algorithme de TV-L^1, nous avons retenu moins de versions à développer et à tester.

4.6.2.1 Stratégies d'implémentation

Là encore, nous commençons par reprendre le schéma numérique de l'algorithme de Horn-Schunck décrit en sous-section 4.3. Nous obtenons le pseudo-code en algorithme 3 avec une fonction de calcul par étape du schéma.

Algorithme 3 Algorithme Horn-Schunck mono-échelle de base

Entrée: I_0, I_1, $\mathbf{u}^0$, Max_{iters}, α
  Calcul `dérivées_partielles` pour I_x, I_y et I_t (équation 4.25)
  pour $n_i = 0$ **à** Max_{iters} **faire**
    Calcul `moyenne_locale` pour $\overline{u_1}$ et $\overline{u_2}$ (équation 4.27)
    Calcul `estimation_u` pour $\mathbf{u}^{n_i+1}$ (équation 4.29)
  fin pour

Le schéma producteur/consommateur de données associé est présenté en figure 4.3. On notera l'exclusion du point central du voisinage nécessaire au calcul de la moyenne locale. Cette exclusion est indiquée par une croix : on ne charge pas le point pour le calcul.

Étant donné qu'il n'y a qu'une seule boucle itérative, nous nous concentrerons sur l'optimisation de son contenu ainsi que l'enchaînement de plusieurs itérations de cette boucle.

D'un point de vue structurel, les codes CUDA des implémentations mono-échelle suivront donc le motif présenté en algorithme 4. Seule la fonction de calcul des dérivées partielles est commune à tous nos codes. Le code pour les itérations internes va varier d'une version à une autre. Nous présentons maintenant ces différentes versions.

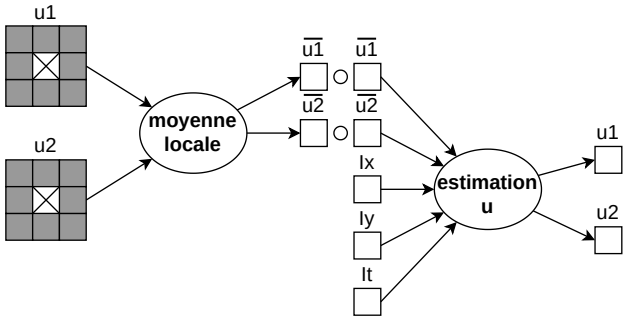


Fig. 4.17 : Schéma itératif interne de Horn-Schunck.

Algorithme 4 Structure Horn-Schunck mono-échelle commune

Entrée: I_0, I_1, $\mathbf{u}^0$, Max_{iters}, α
Kernel `dérivées_partielles` pour I_x, I_y et I_t
pour $n_i = 0$ **à** Max_{iters} **faire**
$\Rightarrow$ Implémentation des itérations internes de Horn-Schunck $\Leftarrow$
fin pour

4.6.2.2 Optimisations communes

Comme on peut le voir en figure 4.17, une partie des opérateurs ponctuels peuvent être fusionnés. On obtient le schéma de dépendances en figure 4.18 suivant après fusion. Il n'y a plus qu'un seul opérateur.

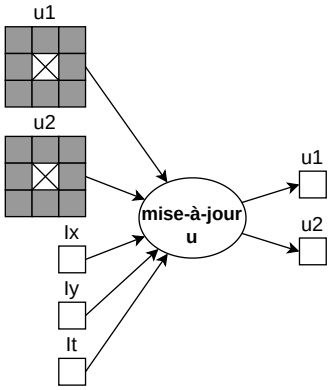


Fig. 4.18 : Schéma itératif interne de Horn-Schunck après fusion des opérateurs ponctuels.

Notre travail d'optimisation aura donc pour but de fusionner efficacement plusieurs itérations du schéma numérique.

Version Global Comme pour TV-L^1, nous commençons par la version Global. Cette version correspond à une implémentation directe du schéma numérique après les optimisations de fusions algorithmiques présentées en figure 4.18. Les schémas des accès pour un thread et pour un bloc sont présentés en figure 4.19.

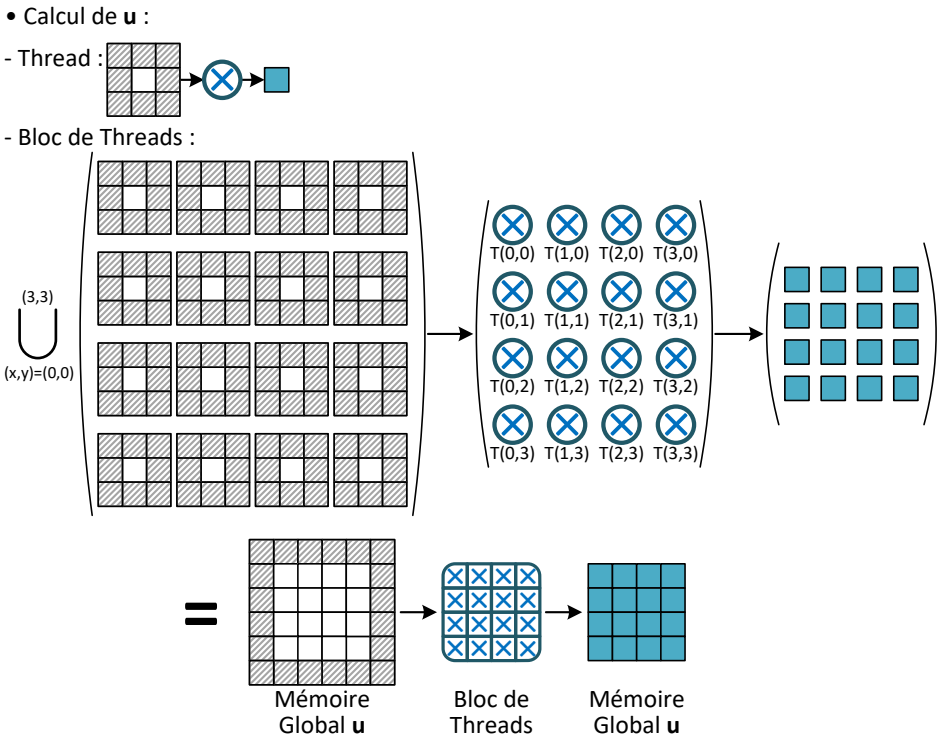


Fig. 4.19 : Représentation du kernel de calcul pour la version Global.

Version Shared_Fusion La version suivante Shared_Fusion fait usage de la mémoire Shared pour partager les données calculées à tous les threads du même bloc. Comme pour TV-L^1, un bord de 1 thread dans le bloc est nécessaire pour le chargement de toutes les données nécessaires au calcul d'une itération. Le schéma d'accès du bloc est présenté en figure 4.20.

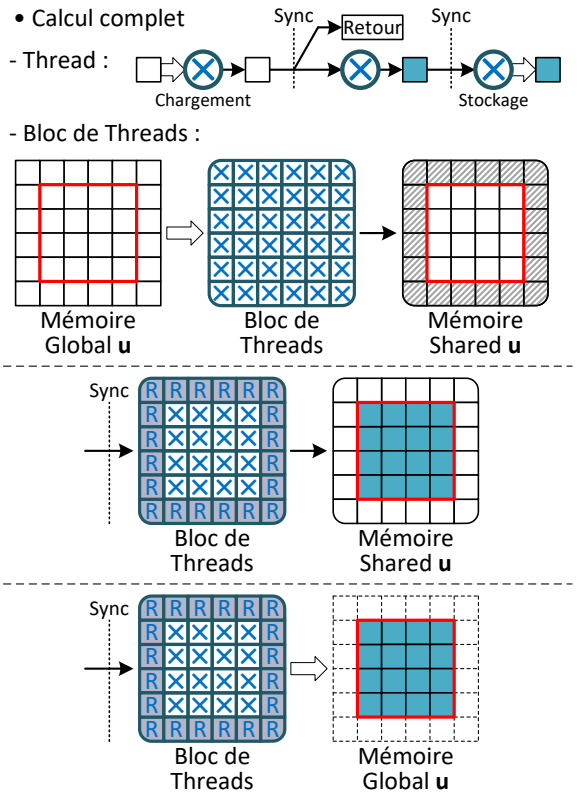


Fig. 4.20 : Représentation du kernel de calcul pour la version Shared_Fusion.

Version Global_Pipeline Comme pour TV-L[1] et tous les algorithmes de type stencils, il est possible d'établir un pipeline itératif pour le calcul au plus tôt des lignes du flot optique. Une extension et une duplication des bords verticaux sont réalisées pour pouvoir contenir suffisamment de lignes pour le prologue et le régime permanent. Les blocs sont unidimensionnels et sont chargés de la mise à jour des lignes tour à tour selon l'ordre de calcul présenté en figure 4.21 et en figure 4.22. Plusieurs blocs vont s'exécuter en parallèle sur des sous-bandes de l'image.

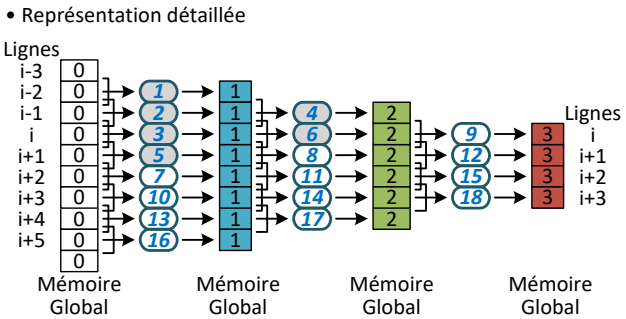


Fig. 4.21 : Ordre des calculs pour la version Global_Pipeline pour un pipeline de 3 itérations sur 4 lignes consécutives. On représente ici les accès aux lignes pour un bloc complet.

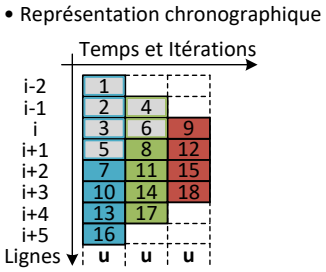


Fig. 4.22 : Ordre des calculs simplifié pour la version Global_Pipeline.

Le schéma d'accès pour un bloc et sa région complète à calculer est présenté en figure 4.23. On notera l'état des lignes à la fin de l'exécution dans l'extension des bords lié au prologue et au régime permanent du pipeline.

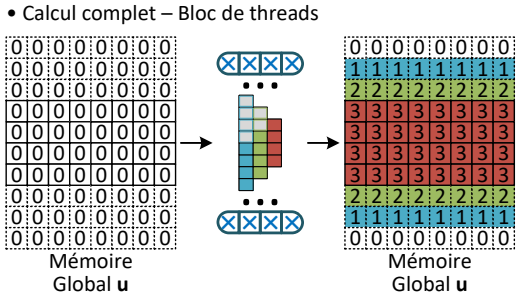


Fig. 4.23 : Représentation du kernel de calcul pour la version Global_Pipeline.

Version Shared_Pipeline Il est possible de réaliser le pipeline précédant en mémoire Shared pour bénéficier du partage des données aux threads du même bloc, d'un accès mémoire plus rapide ainsi qu'une meilleure localité spatiale et temporelle. Là encore, la quantité de mémoire Shared est limitée et un accès modulaire aux lignes est nécessaire. L'ordre des calculs est présenté en figure 4.25. Le schéma d'accès pour un bloc et sa région complète à calculer est présenté en figure 4.24.

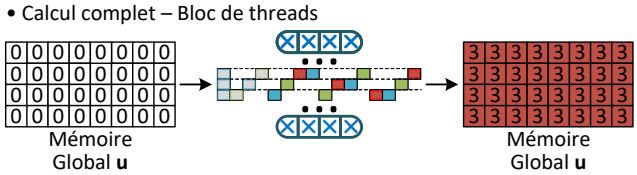


Fig. 4.24 : Représentation du kernel de calcul pour la version Shared_Pipeline.

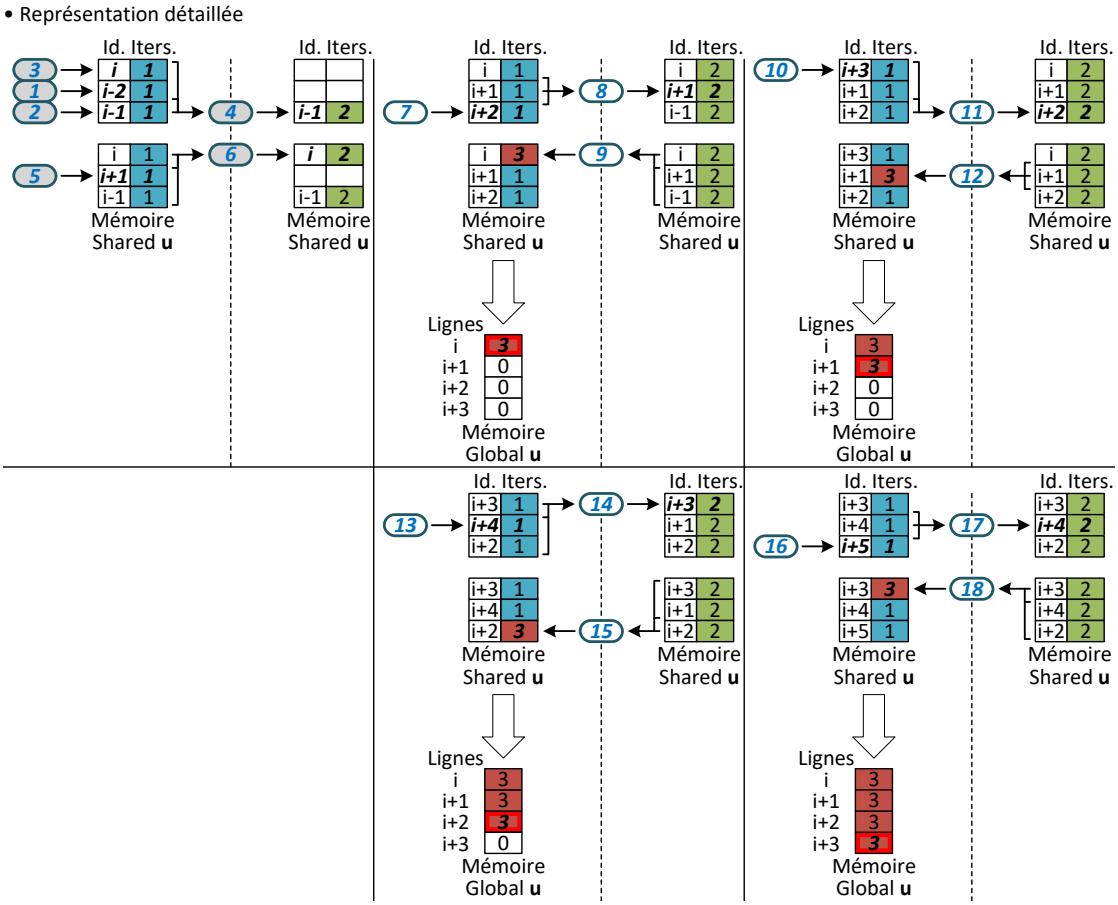


Fig. 4.25 : Détail de l'ordre des calculs pour la version Shared_Pipeline pour un pipeline de 3 itérations sur 4 lignes consécutives. On représente ici les accès modulaires dans les lignes de la mémoire Shared ainsi que les accès de stockage dans la mémoire Global.

Version Global_MltIteration La version Global_MltIteration reprend l'idée de fusion de plusieurs itérations introduite pour TV-L[1]. Les blocs de threads va s'exécuter plusieurs fois sur la même région de données afin de calculer plusieurs itérations dans le même bloc de threads. Les threads en bordure de bloc vont s'arrêter au fur et à mesure des itérations pour pouvoir respecter les dépendances spatiales et itératives. Le schéma d'accès du bloc de threads selon les itérations est représenté en figure 4.26.

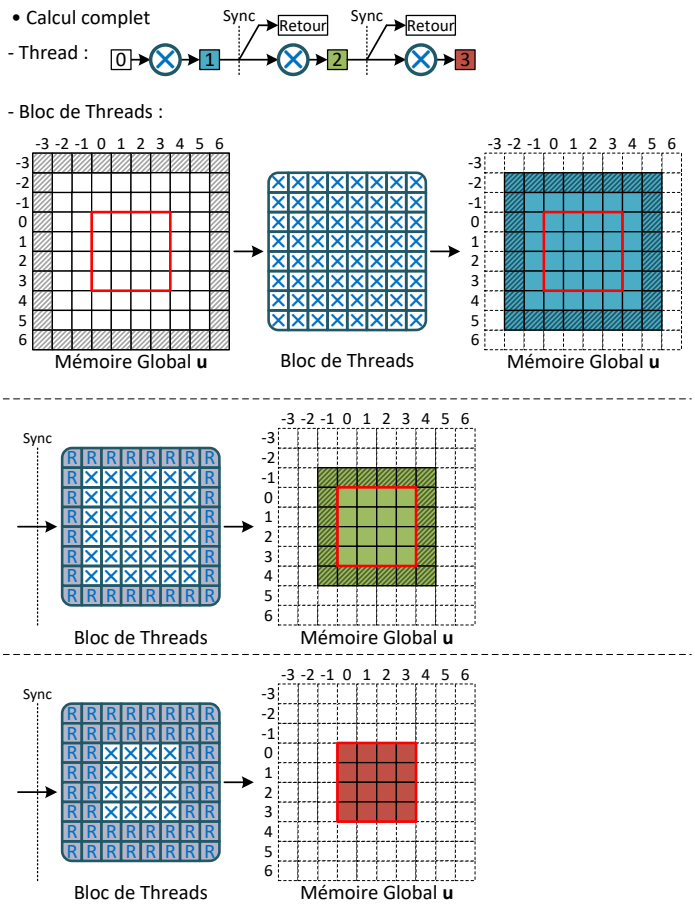


Fig. 4.26 : Représentation du kernel de calcul pour la version Global_MltIteration pour une fusion de 3 itérations et pour une zone de 4×4 pixels à calculer.

Version Shared_MltIteration Le schéma de fusion de plusieurs itérations est également réalisable en mémoire Shared. Une étape de chargement et une étape de stockage supplémentaires seront alors nécessaires. Le schéma d'accès de cette version est représenté en figure 4.27.

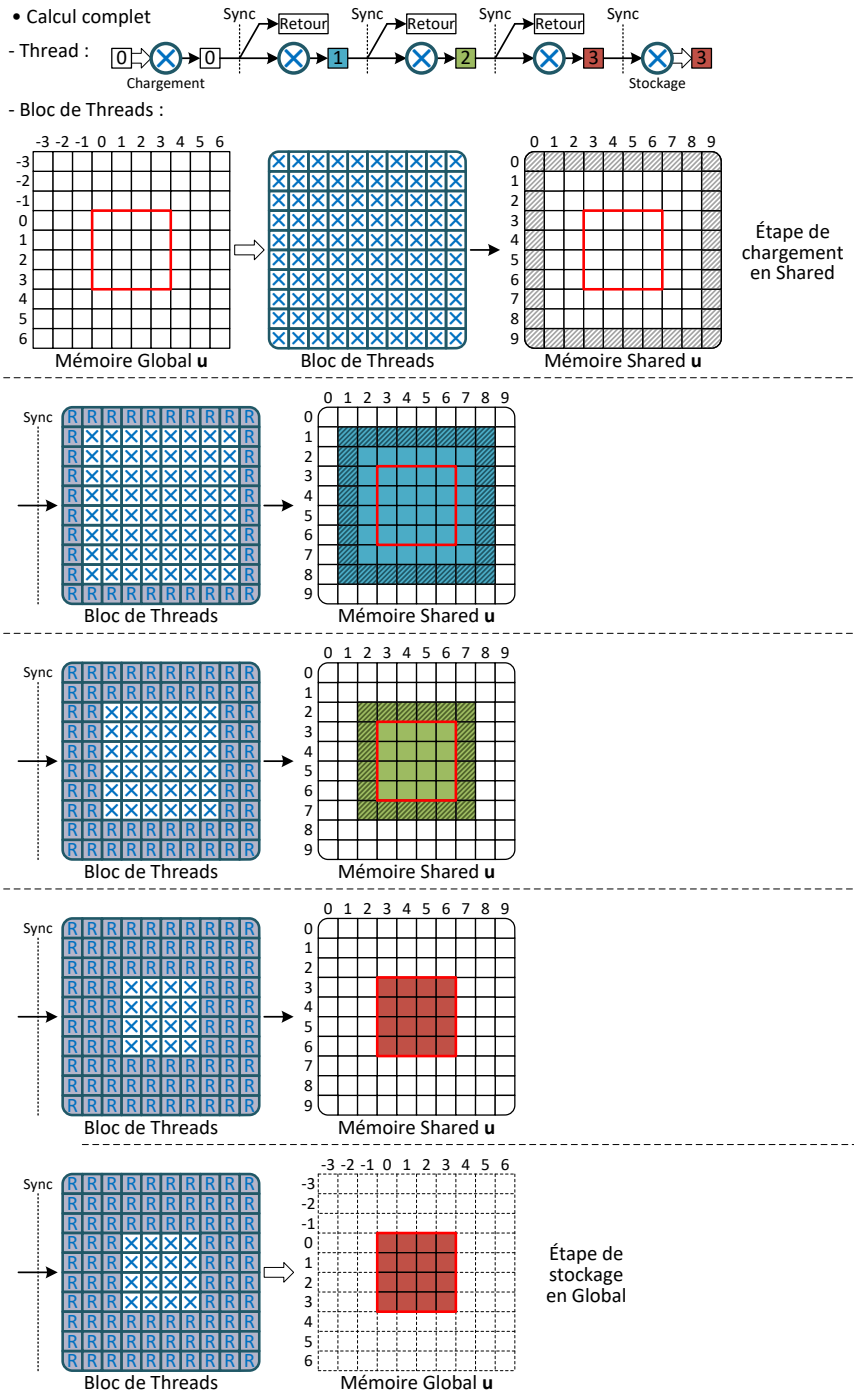


Fig. 4.27 : Représentation du kernel de calcul pour la version Shared_MltIteration pour une fusion de 3 itérations et pour une zone de 4×4 pixels à calculer.

4.7 Conclusion

Nous avons présenté dans ce chapitre les principales réalisations faites dans le cadre de cette thèse.

Nous avons commencé par présenter les concepts mathématiques nécessaires à l'écriture des schémas numériques et des algorithmes pour les méthodes d'estimation du flot optique TV-L[1] et Horn-Schunck. En plus de présenter les schémas de base, nous avons présenté le concept de pyramide multi-résolution. Cette structure permet aux méthodes précédentes d'estimer des mouvements plus grands, supérieurs à un pixel.

Nous avons ensuite présenté les points précis de notre travail d'optimisation. En partant des algorithmes de base issus des schémas numériques des deux méthodes, nous avons identifié un premier point d'optimisation algorithmique permettant de réduire autant que possible le nombre de fonctions à exécuter. Cette optimisation principale nous sert ensuite de base aux multiples implémentations GPU.

Ces implémentations GPU cherchent à tirer profit au maximum du matériel sous-jacent. La multiplicité des implémentations est liée aux multiples mécanismes de partage de données présents sur le GPU qui passent par les mémoires Global, Shared ainsi que par les instructions de Shuffle. En plus de ces mécanismes, plusieurs techniques d'exécutions plus ou moins complexes permettent d'augmenter la réutilisation des données déjà calculées et permettent ainsi une meilleure localité temporelle et spatiale.

Il est maintenant nécessaire d'évaluer ces différentes versions sur différentes plateformes afin de déterminer la ou les meilleures implémentations. Il n'est pas trivial de déterminer la version la plus rapide sans tests. En effet, ces versions diffèrent sur plusieurs points :

— Les mémoires utilisées ont des temps d'accès plus ou moins rapides, des bus plus ou moins larges ;
— Les données intermédiaires sont plus ou moins réutilisées ;
— Les dimensions des blocs varient ;
— Un thread va traiter dans certaines versions 1 seul pixel, dans d'autres plusieurs itérations du même pixel et dans d'autres encore plusieurs pixels d'une même ligne ou d'une même zone ;
— Il peut être nécessaire de recalculer des données.

Enfin, il est important de noter que la présentation de ces implémentations fait abstraction de la représentation utilisée pour les nombres flottants ainsi que de leur dynamique. Toutes ces versions ont en effet été déclinées en 4 sous-versions :

— Une version F32 utilisant des nombres *float* simple précision sur 32 bits ;
— Une version F32x2 utilisant le type vectoriel CUDA *float2* contenant 2 nombres flottants simple précision sur 32 bits chacun ;
— Une version F16 utilisant des nombres *half* demi-précision sur 16 bits ;
— Une version F16x2 utilisant le type vectoriel CUDA *half2* contenant 2 nombres flottants demi-précision sur 16 bits chacun.

Le type de nombre utilisé aura un impact à la fois sur le temps d'exécution, mais aussi sur la précision des calculs et ainsi sur la qualité du flot estimé.

Une analyse comparative des différentes versions est donc nécessaire et sera présentée dans le chapitre suivant.

5 Évaluation des Optimisations et Résultats

Nous avons présenté dans le chapitre précédent nos nombreuses implémentations mettant en œuvre différentes techniques d'optimisations pour les algorithmes TV-L[1] et Horn-Schunck. Ces implémentations sont plus ou moins complexes, et peuvent nécessiter l'utilisation d'unités de traitement et d'espaces mémoire variant d'une architecture GPU à une autre. De plus, l'utilisation de différents formats de nombre ayant une dynamique variable et possiblement vectoriel a une influence tant sur le temps d'exécution des différentes implémentations que sur la qualité du flot optique produit en sortie. En conséquence, il est impératif de réaliser de nombreux tests et évaluations pour garantir la qualité et la fiabilité des implémentations de ces algorithmes. Ce chapitre présentera ainsi les différentes métriques évaluées ainsi que les protocoles expérimentaux associés, en parallèle de l'analyse des résultats sur différentes plateformes GPU.

Il nous faudra dans un premier temps déterminer les bons paramètres spécifiques aux différentes implémentations. Ces paramètres correspondent au nombre de blocs de threads, à leurs dimensions, à la quantité de mémoire Shared utilisée, et au nombre d'itérations calculées par un thread dans le bloc. Cette exploration permettra de fixer ces paramètres afin d'obtenir l'exécution la plus efficace possible pour les expérimentations suivantes.

Dans un second temps, les différentes versions seront évaluées en termes de temps de traitement, normalisé pour trouver le temps moyen nécessaire pour calculer un pixel des images en entrée. Pour cette première évaluation de performances, les comparaisons qualitatives ne sont pas prises en compte, seule la vitesse de calcul des différentes optimisations des itérations est mesurée. Par conséquent, seules les versions mono-échelles seront testées en fixant les paramètres itératifs.

Les implémentations seront ensuite évaluées d'un point de vue énergétique. Une même configuration mono-échelle fixe sera utilisée sur des images suffisamment grandes pour assurer une charge suffisante sur le GPU. Cette étude sera particulièrement intéressante pour comparer les différentes plateformes GPU en termes

d'efficacité énergétique et temporelle. Elle a pour but de déterminer si les plateformes les plus récentes et les plus puissantes, bien que plus gourmandes en énergie, permettent à nos optimisations de fonctionner plus efficacement.

Enfin, nous terminerons ce chapitre par une étude qualitative de nos implémentations les plus rapides sur des collections de séquences d'images dont le flot optique est connu. Cette vérité terrain nous permettra ainsi de calculer différentes erreurs entre le flot estimé par les algorithmes et les références. Nous utiliserons ici des versions pyramidales permettant de calculer un flot optique sur des mouvements réalistes. Il nous sera également possible de déterminer et de comparer le temps d'exécution nécessaire pour obtenir une qualité de flot fixe pour nos meilleures implémentations. Le but de cette étude est de déterminer si la baisse de la dynamique des nombres et ainsi la baisse de la qualité de flot peut être compensée ou même dépassée par une réduction du temps par itération. En effet, il est possible pour une version plus rapide, mais moins précise de réaliser plus d'itérations du schéma itératif dans le même laps de temps qu'une version plus lente, mais plus précise.

5.1 Exploration des paramètres

Nous commençons donc par une exploration des paramètres intrinsèques pour chaque implémentation ainsi développée. Nous séparons notre travail d'exploration en deux. Dans un premier temps, nous nous occuperons de trouver une bonne dimension pour les blocs de threads dont la taille reste fixe peu importe le nombre d'itérations demandées. Il s'agit de tous les kernels exécutés avant les schémas itératifs des algorithmes, mais également des implémentations mono-itératives, à savoir les versions :

— Global
— Shared
— Global_Fusion
— Shared_Fusion

Dans un second temps, nous nous occuperons des versions réalisant le calcul de plusieurs itérations. Ce calcul multi-itératif à l'intérieur du kernel casse l'aspect générique du bloc étant donné ses dépendances de facteurs externes comme le nombre d'itérations à réaliser, le nombre de points traités par bloc et la quantité de mémoire nécessaire. C'est le cas pour les versions suivantes :

— Global_Pipeline

— Shared_Pipeline

— Global_MltIteration

— Shared_MltIteration

On notera que la version Shuffle a une taille de bloc fixe de 32 threads étant donné que les opérations de shuffle ne sont possibles que sur un seul warp de 32 threads.

5.1.1 Blocs de threads mono-itérations

Nous commençons par lister les paramètres des configurations d'explorations. Afin de solliciter suffisamment le GPU, nous fixons la taille d'image à 2048×2048 pixels. Nous limitons notre recherche à des blocs dont la taille est multiple de 32 threads pour assurer un bon découpage en warps de 32 threads. Nous nous intéressons aux kernels individuels. Aussi, nous les exécuterons seuls, hors de leur schéma numérique. Nous obtenons une série de temps d'exécution que nous représentons en deux dimensions en fonctions de la hauteur et de la largeur du bloc de threads. Nous utilisons l'échelle de couleur en figure 5.1 afin de mettre en avant les valeurs proches du temps minimum – au plus à 5% du minimum – atteint pour chaque kernel. Cette valeur de 5% nous sert visuellement afin de différencier les zones d'intérêt rapide de celles trop lentes.

Fig. 5.1 : Échelle de couleur pour les cartes de couleurs des dimensions de blocs.

Les figures 5.2, 5.4, , 5.3 et 5.5 représentent ces cartes de chaleur avec une graduation des temps d'exécution relatifs des différentes configurations de blocs. Les kernels testés sont ici ceux pour l'algorithme TV-L^1. Un bon compromis pour l'ensemble de ces blocs sont les dimensions (64×6) et (64×8). Le compromis (64×6) inclut les kernels auxiliaires aux deux schémas itératifs servant notamment pour la pyramide d'images. Le compromis (64×8) n'inclut que les kernels internes aux schémas itératifs. Des optimisations en taille de blocs plus fines sont possibles mais nécessiteraient une exploration et une gestion approfondie des dimensions des blocs pour chaque version et chaque kernel. Avec le compromis précédent, nous obtenons des temps d'exécution de kernels éloigné au pire à 15% du minimum pour les kernels du schéma numérique de TV-L^1 et au pire à 8% du minimum pour les kernels du schéma numérique de Horn-Schunck.

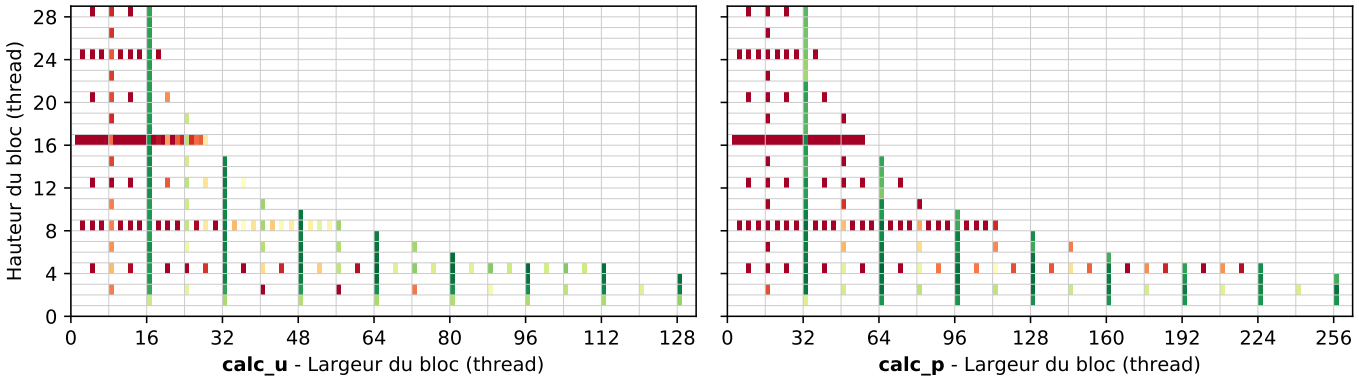


Fig. 5.2(a): **TV-L**[1] Global_F32 - **calc_u** min : 1,63 ms ; **calc_p** min : 1,40 ms

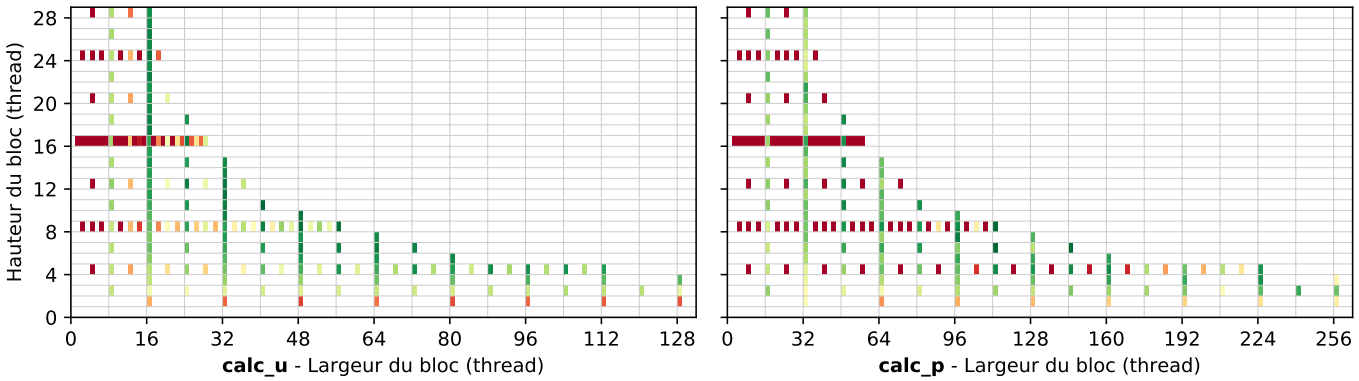


Fig. 5.2(b): **TV-L**[1] Global_F32x2 - **calc_u** min : 1,64 ms ; **calc_p** min : 1,40 ms

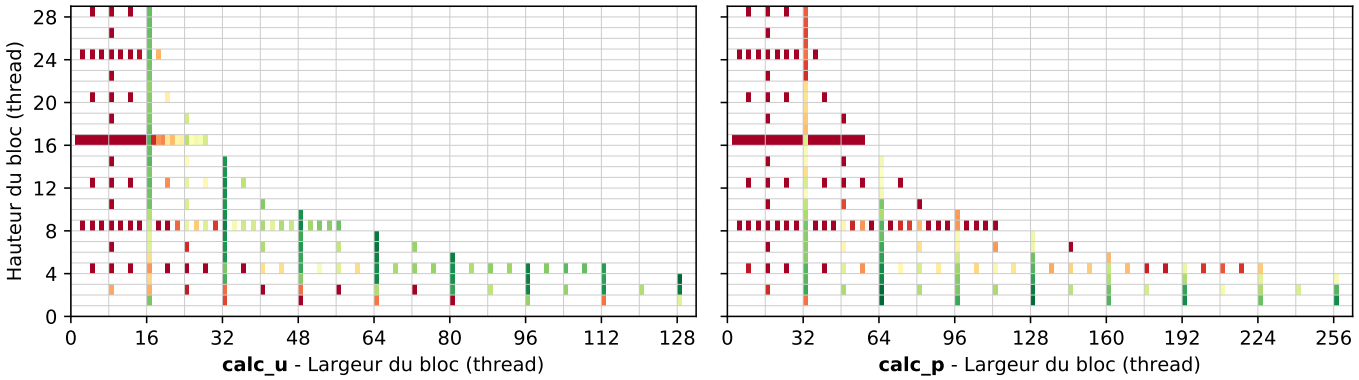


Fig. 5.2(c): **TV-L**[1] Global_F16 - **calc_u** min : 0,94 ms ; **calc_p** min : 1,02 ms

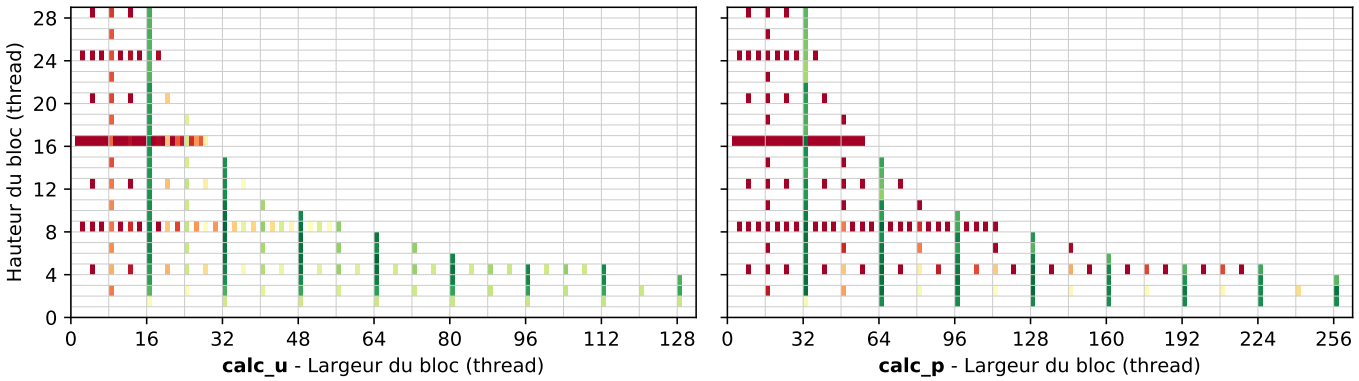


Fig. 5.2(d): **TV-L**[1] Global_F16x2 - **calc_u** min : 0,83 ms ; **calc_p** min : 0,72 ms

Fig. 5.2 : Temps d'exécution de TV-L[1] Global en fonction de la hauteur et de la largeur de bloc du GPU. La couleur indique le temps relatif au temps minimal, selon l'échelle de la figure 5.1

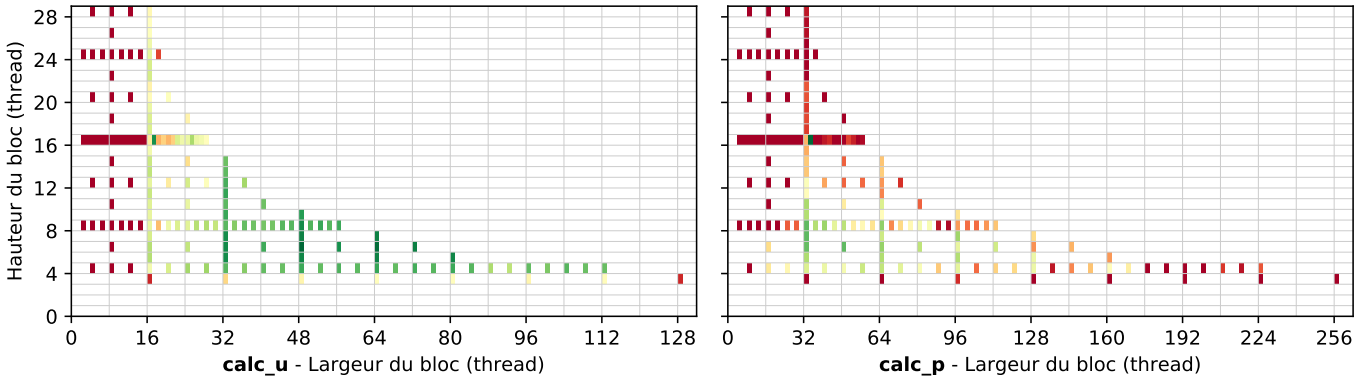


Fig. 5.3(a): **TV-L**[1] Shared_F32 - **calc_u** min : 1,96 ms ; **calc_p** min : 2,36 ms

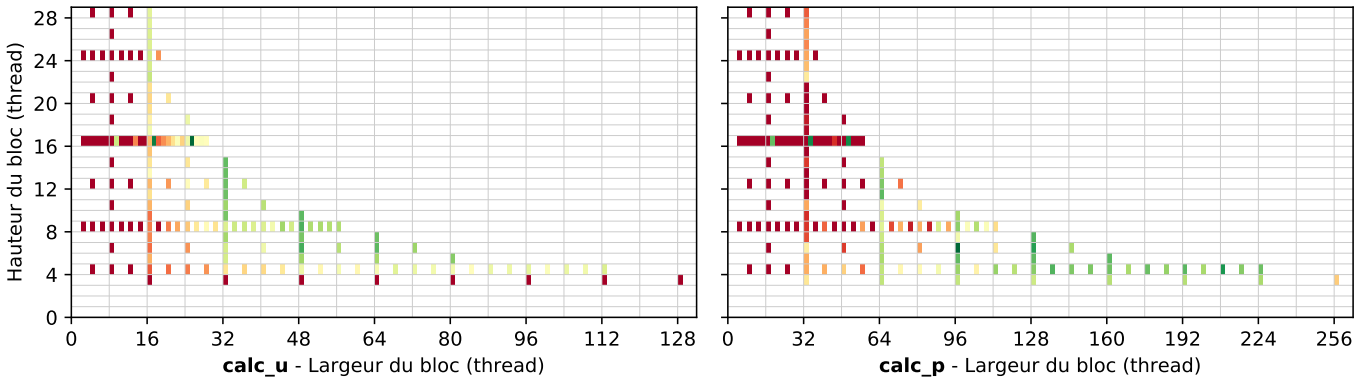


Fig. 5.3(b): **TV-L**[1] Shared_F32x2 - **calc_u** min : 1,84 ms ; **calc_p** min : 1,63 ms

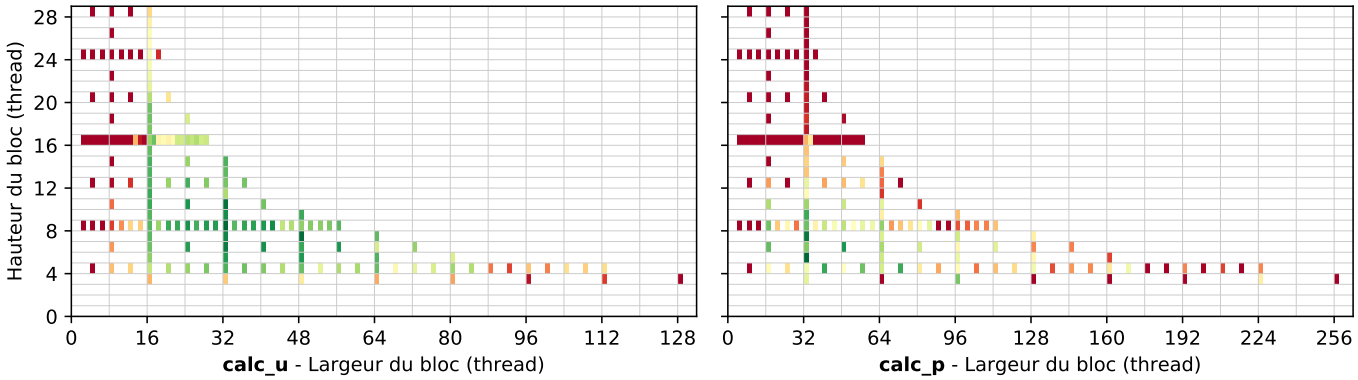


Fig. 5.3(c): **TV-L**[1] Shared_F16 - **calc_u** min : 1,16 ms ; **calc_p** min : 1,85 ms

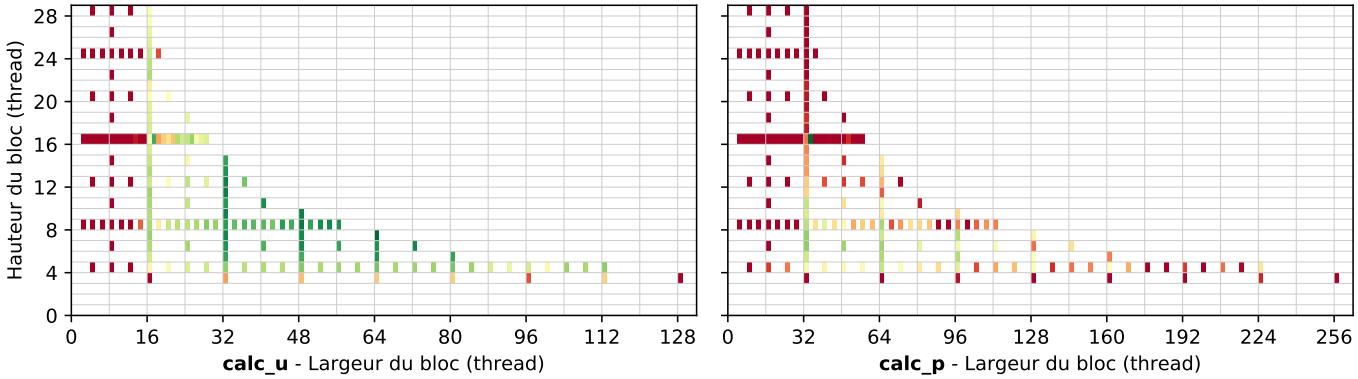


Fig. 5.3(d): **TV-L**[1] Shared_F16x2 - **calc_u** min : 0,97 ms ; **calc_p** min : 1,20 ms

Fig. 5.3 : Temps d'exécution des kernels de TV-L[1] Shared en fonction de la hauteur et de la largeur de bloc du GPU. La couleur indique le temps relatif au temps minimal, selon l'échelle de la figure 5.1

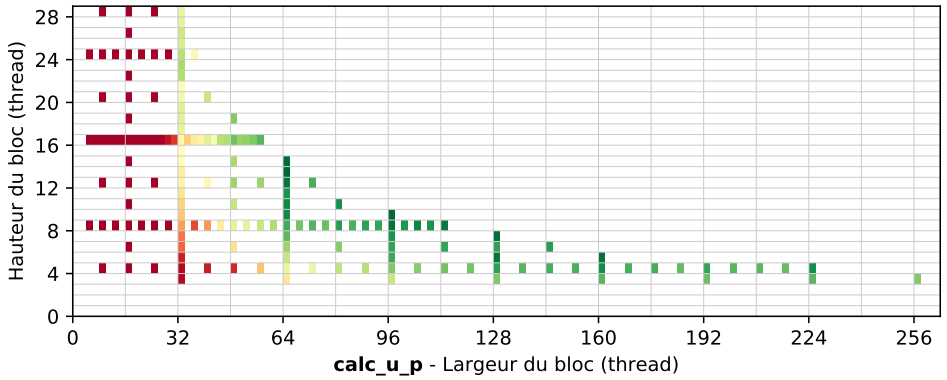


Fig. 5.4(a): **TV-L**[1] Global_Fusion_F32 - min : 2,86 ms

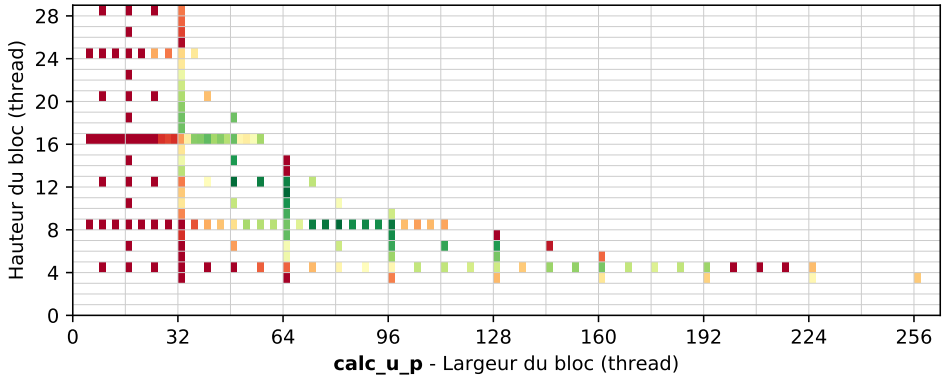


Fig. 5.4(b): **TV-L**[1] Global_Fusion_F32x2 - min : 2,73

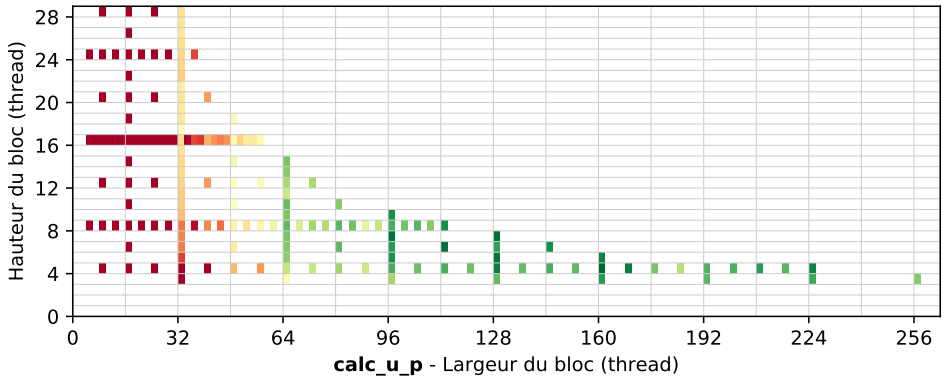


Fig. 5.4(c): **TV-L**[1] Global_Fusion_F16 - min : 1,94 ms

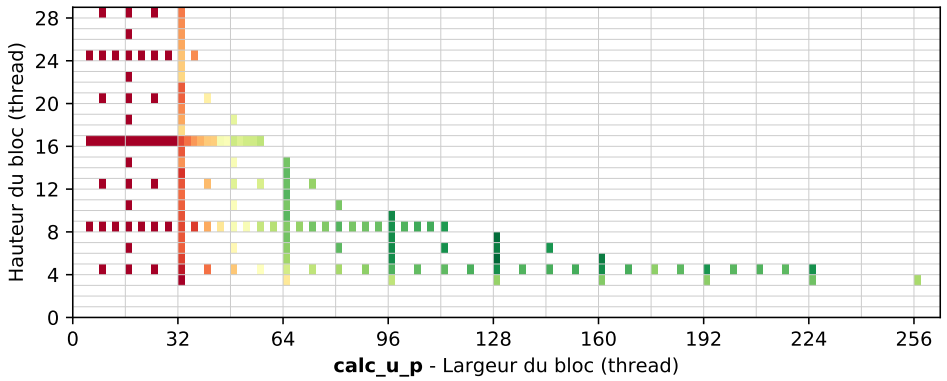


Fig. 5.4(d): **TV-L**[1] Global_Fusion_F16x2 - min : 1,37 ms

Fig. 5.4 : Temps d'exécution des kernels de TV-L[1] Global_Fusion en fonction de la hauteur et de la largeur de bloc du GPU. La couleur indique le temps relatif au temps minimal, selon l'échelle de la figure 5.1

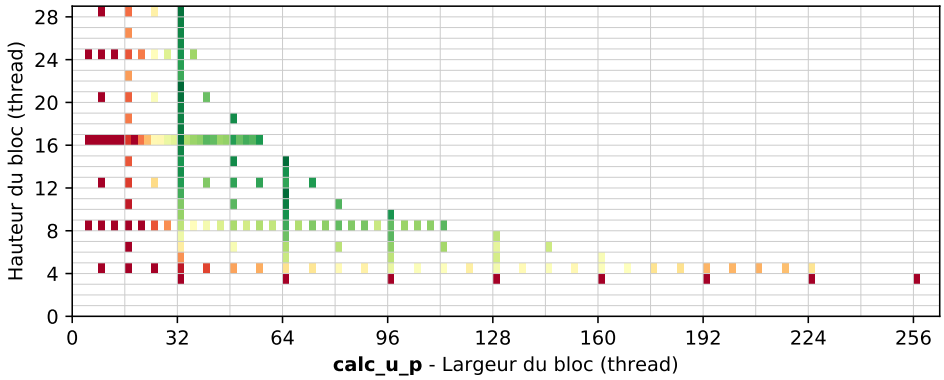


Fig. 5.5(a): **TV-L**[1] Shared_Fusion_F32 - min : 2,79 ms

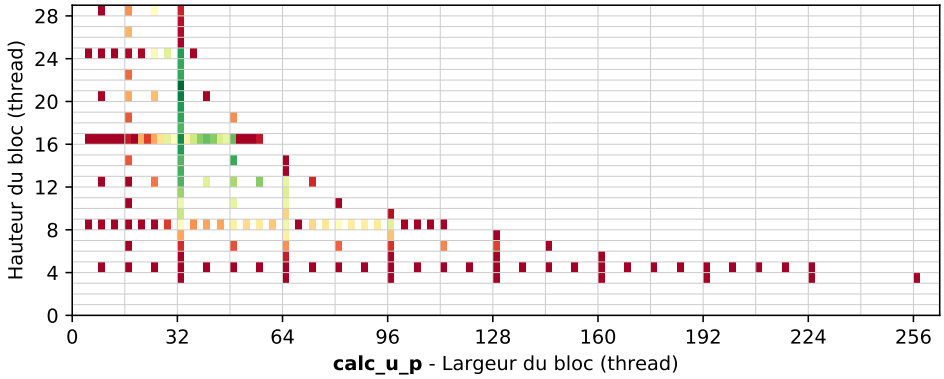


Fig. 5.5(b): **TV-L**[1] Shared_Fusion_F32x2 - min : 2,59 ms

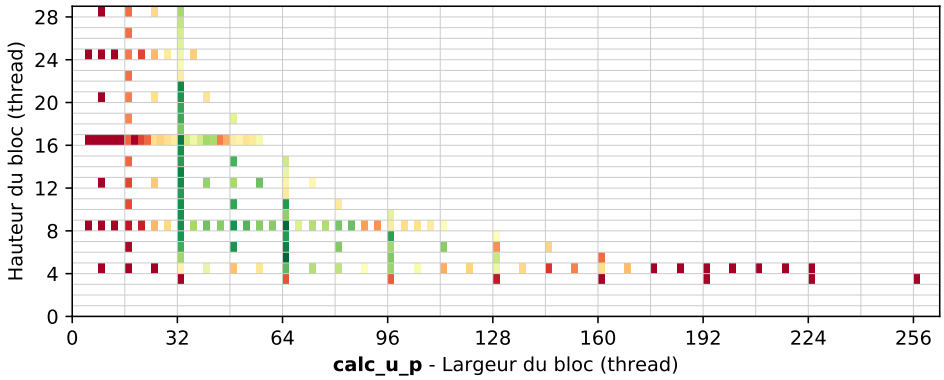


Fig. 5.5(c): **TV-L**[1] Shared_Fusion_F16 - min : 1,65 ms

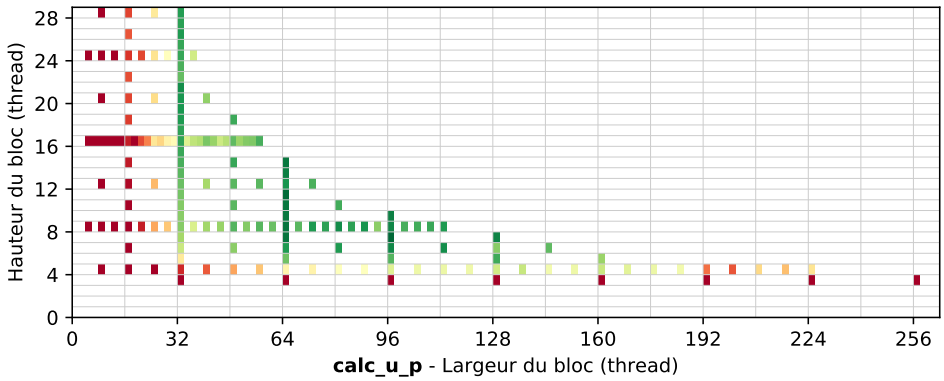


Fig. 5.5(d): **TV-L**[1] Shared_Fusion_F16x2 - min : 1.37 ms

Fig. 5.5 : Temps d'exécution des kernels de TV-L[1] Shared_Fusion en fonction de la hauteur et de la largeur de bloc du GPU. La couleur indique le temps relatif au temps minimal, selon l'échelle de la figure 5.1.

Pour des raisons de simplification des tests de performances, une seule configuration a été retenue. L'idéal serait d'avoir une bibliothèque de configurations pour chaque version et pour chaque GPU appelée automatiquement lors de l'exécution d'un kernel.

5.1.2 Blocs de threads multi-itérations

Pour ce qui est des kernels effectuant le calcul de plusieurs itérations, nous avons vu précédemment que leur empreinte mémoire et leur zone de calcul dépendent à la fois des dimensions des blocs mais aussi du nombre d'itérations interne au kernel. De plus, les dimensions maximales possibles dépendent des ressources disponibles sur les GPU. Une exploration plus fine de ces blocs est nécessaire pour déterminer les configurations idéales minimisant le temps d'exécution.

Nous testons donc nos implémentations Pipeline et MltIteration pour TV-L^1 et Horn-Schunck. Nous testons plusieurs tailles d'image et déterminons les dimensions optimales pour chacune. De par la croissance des dépendances de données selon le nombre d'itérations, il existe un nombre d'itérations maximales possible pour certaines versions. Ces limitations sont indiquées dans le tableau 5.1 pour la Jetson Xavier.

Algorithme	Version	Registres par bloc	Shared par bloc	Bloc Dim Max	Iter Max
TV-L^1	Global_Pipeline F32x2	49152	0	**1024**	511
	Global_Pipeline F16x2	49152	0	**1024**	511
	Shared_Pipeline F32x2	4608	**49152**	64	15
	Shared_Pipeline F16x2	4608	**49152**	64	31
	Global_MltIter F32x2	**61440**	0	768	13
	Global_MltIter F16x2	**61440**	0	768	13
	Shared_MltIter F32x2	65536	49152	**1024**	15
	Shared_MltIter F16x2	57344	24576	**1024**	15
HS	Global_Pipeline F32x2	57344	0	**1024**	511
	Global_Pipeline F16x2	57344	0	**1024**	511
	Shared_Pipeline F32x2	3584	**47104**	64	23
	Shared_Pipeline F16x2	6912	**47616**	96	31
	Global_MltIter F32x2	57344	0	**1024**	16
	Global_MltIter F16x2	49152	0	**1024**	16
	Shared_MltIter F32x2	40960	32768	**1024**	15
	Shared_MltIter F16x2	16384	16384	**1024**	15

Tab. 5.1 : Caractéristiques limitant le nombre d'itérations sur la Jetson Xavier.

Les figures 5.6, 5.7, 5.8 et 5.9 représentent cette étude pour TV-L^1.

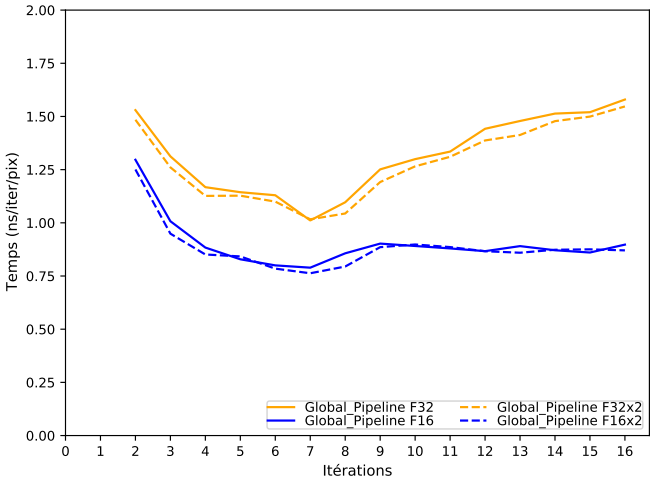


Image de 256×256 pixels :
(a) – min F32x2 : 7 itérations (1,02 ns/iter/pix)
– min F16x2 : 7 itérations (0,76 ns/iter/pix)

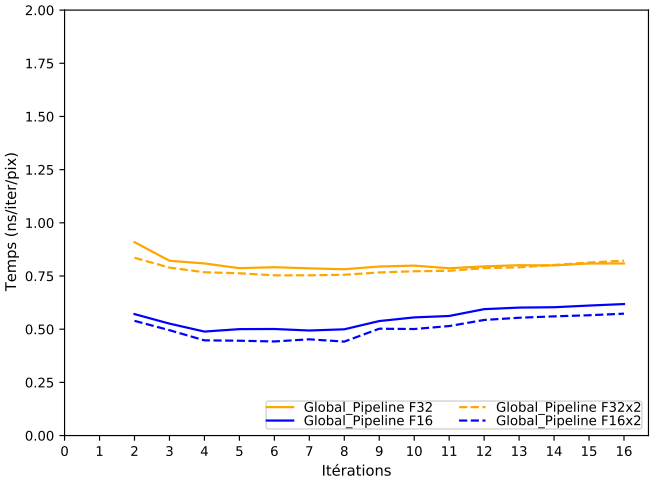


Image de 512×512 pixels :
(b) – min F32x2 : 6 itérations (0,75 ns/iter/pix)
– min F16x2 : 8 itérations (0,44 ns/iter/pix)

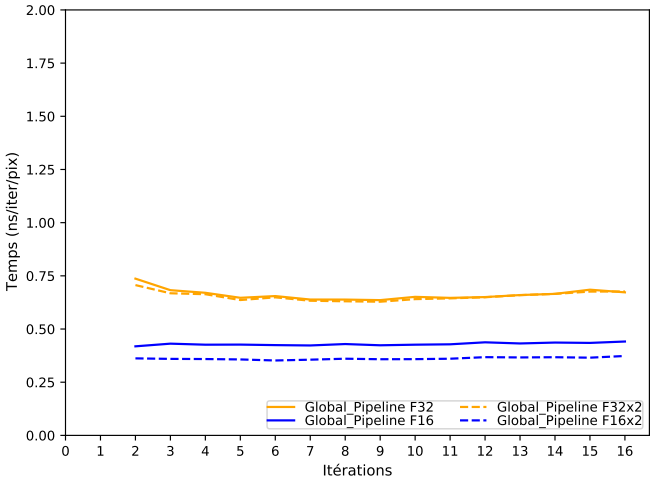


Image de 1024×1024 pixels :
(c) – min F32x2 : 9 itérations (0,63 ns/iter/pix)
– min F16x2 : 6 itérations (0,35 ns/iter/pix)

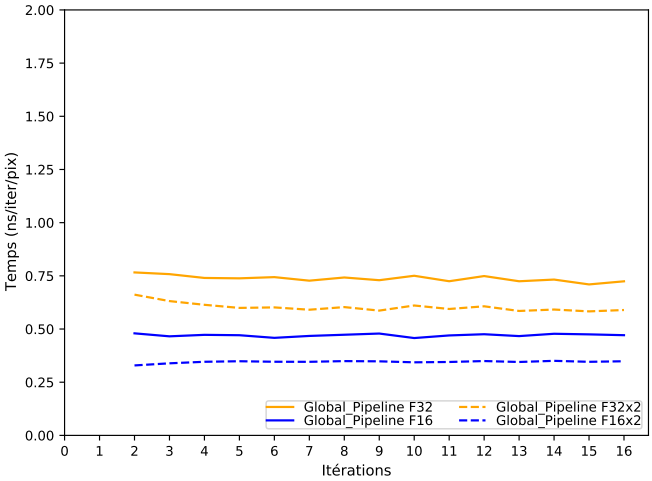
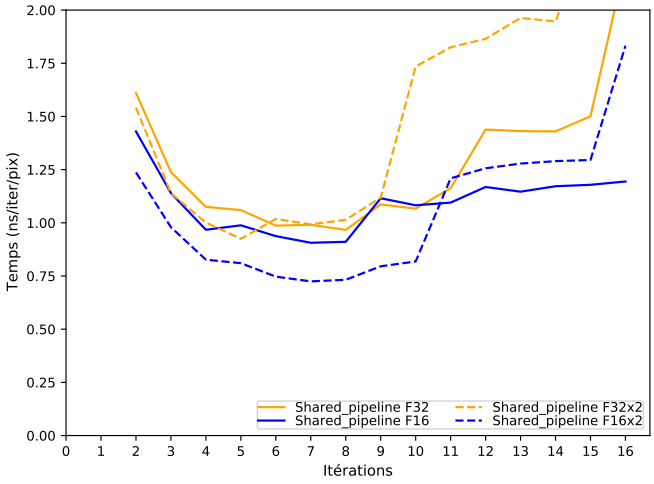
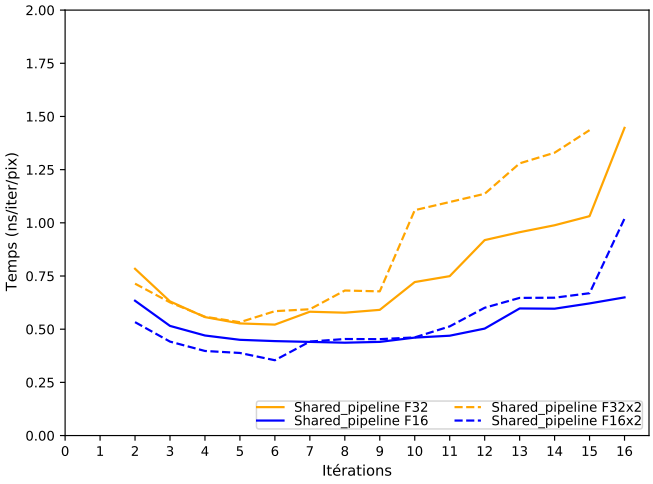


Image de 2048×2048 pixels :
(d) – min F32x2 : 15 itérations (0,58 ns/iter/pix)
– min F16x2 : 2 itérations (0,33 ns/iter/pix)

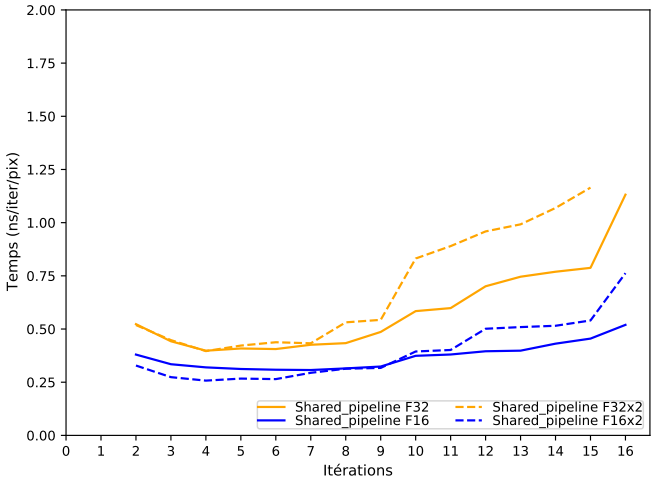
Fig. 5.6 : **TV-L**[1] Global_Pipeline - Jetson Xavier



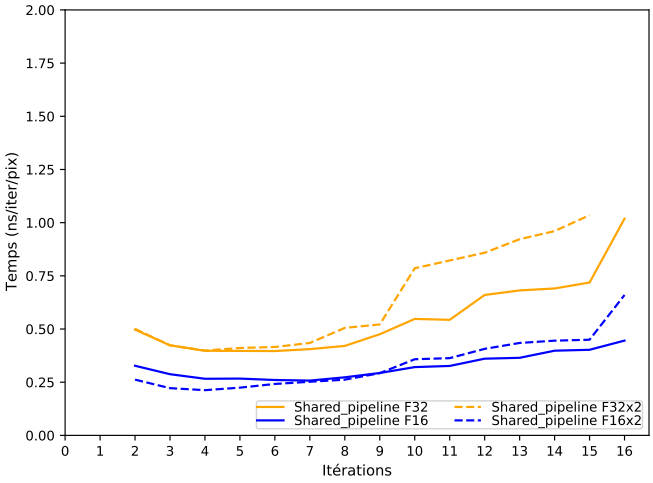
(a) Image de 256×256 pixels :
– min F32x2 : 5 itérations (0,93 ns/iter/pix)
– min F16x2 : 9 itérations (0,70 ns/iter/pix)



(b) Image de 512×512 pixels :
– min F32x2 : 5 itérations (0,53 ns/iter/pix)
– min F16x2 : 6 itérations (0,35 ns/iter/pix)



(c) Image de 1024×1024 pixels :
– min F32x2 : 4 itérations (0,40 ns/iter/pix)
– min F16x2 : 4 itérations (0,26 ns/iter/pix)



(d) Image de 2048×2048 pixels :
– min F32x2 : 4 itérations (0,40 ns/iter/pix)
– min F16x2 : 4 itérations (0,21 ns/iter/pix)

Fig. 5.7 : TV-L^1 Shared_Pipeline - Jetson Xavier

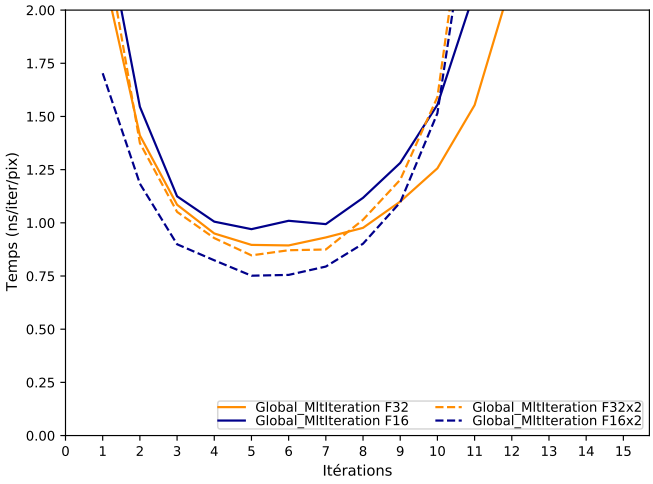


Image de 256×256 pixels :
(a) – min F32x2 : 5 itérations (0,85 ns/iter/pix)
– min F16x2 : 5 itérations (0,75 ns/iter/pix)

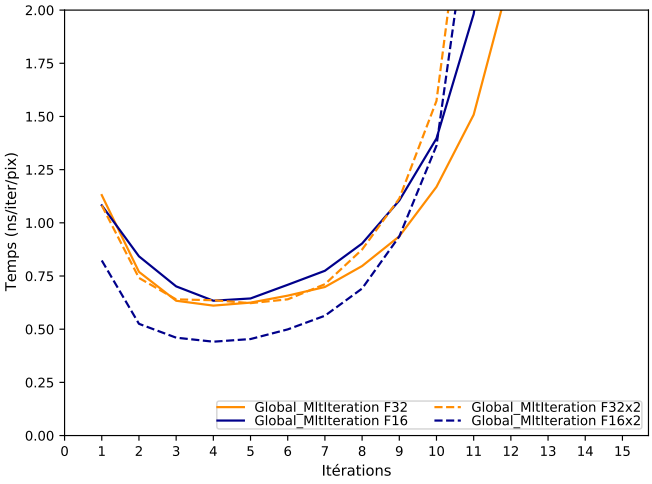


Image de 512×512 pixels :
(b) – min F32x2 : 5 itérations (0,62 ns/iter/pix)
– min F16x2 : 4 itérations (0,44 ns/iter/pix)

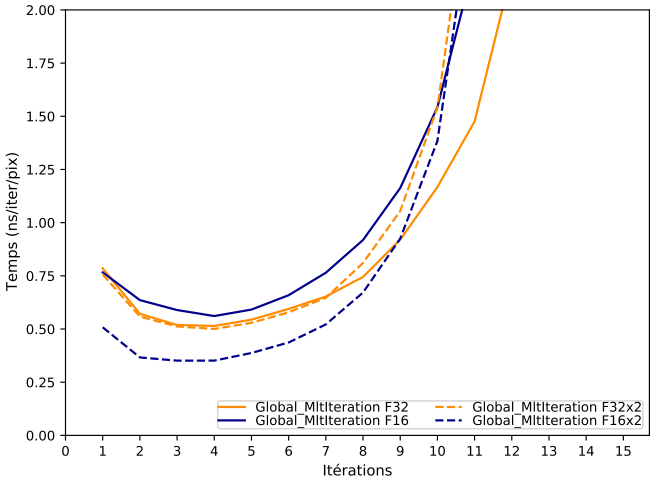


Image de 1024×1024 pixels :
(c) – min F32x2 : 4 itérations (0,50 ns/iter/pix)
– min F16x2 : 3 itérations (0,35 ns/iter/pix)

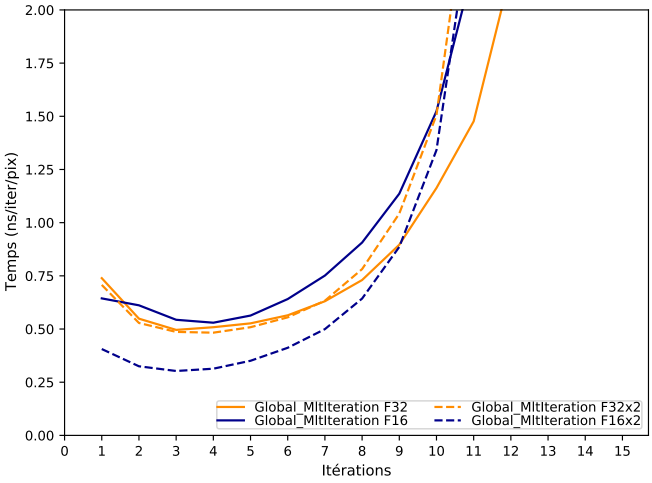


Image de 2048×2048 pixels :
(d) – min F32x2 : 4 itérations (0,48 ns/iter/pix)
– min F16x2 : 3 itérations (0,30 ns/iter/pix)

Fig. 5.8 : **TV-L**[1] Global_MltIteration - Jetson Xavier

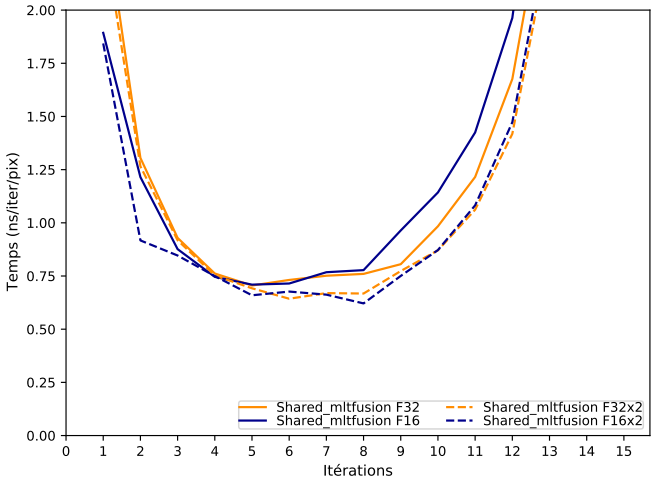


Image de 256×256 pixels :
(a) – min F32x2 : 6 itérations (0,63 ns/iter/pix)
– min F16x2 : 8 itérations (0,65 ns/iter/pix)

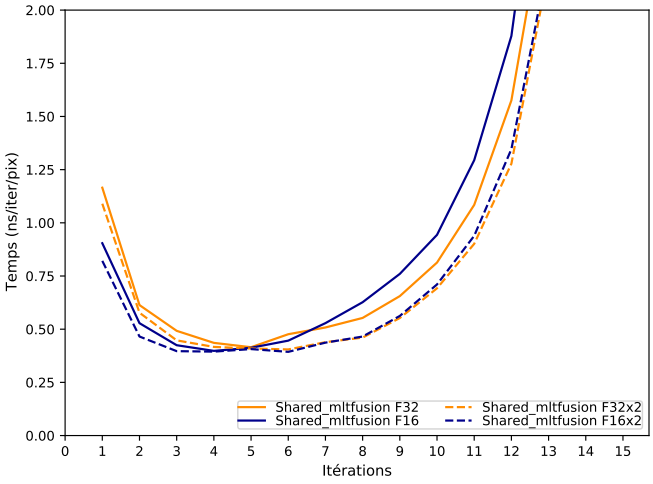


Image de 512×512 pixels :
(b) – min F32x2 : 6 itérations (0,39 ns/iter/pix)
– min F16x2 : 3 itérations (0,38 ns/iter/pix)

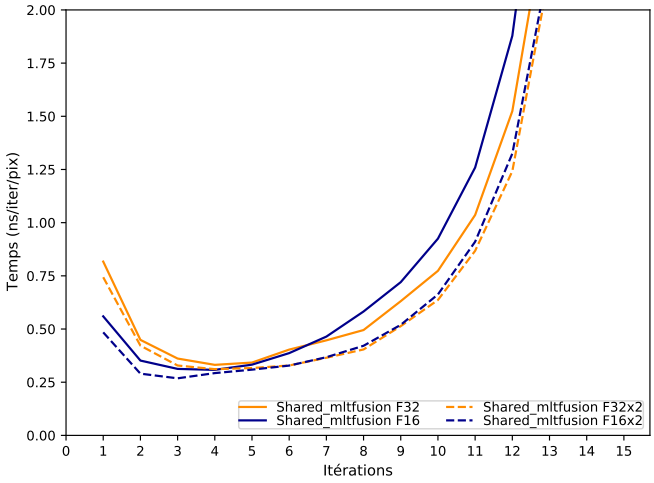


Image de 1024×1024 pixels :
(c) – min F32x2 : 4 itérations (0,30 ns/iter/pix)
– min F16x2 : 3 itérations (0,27 ns/iter/pix)

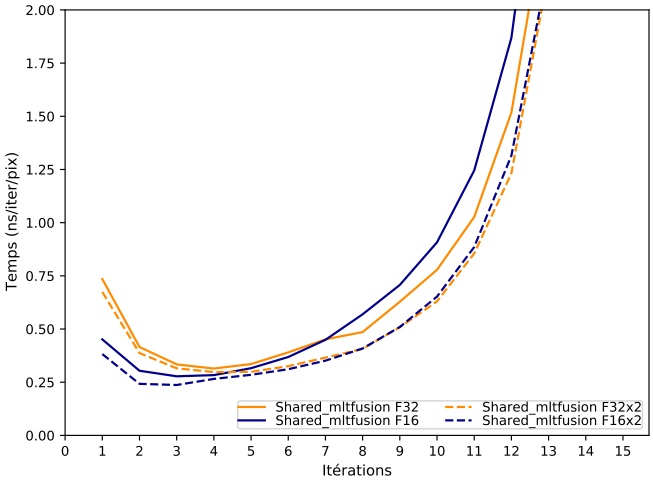


Image de 2048×2048 pixels :
(d) – min F32x2 : 5 itérations (0,29 ns/iter/pix)
– min F16x2 : 3 itérations (0,24 ns/iter/pix)

Fig. 5.9 : **TV-L**[1] Shared_MltIteration - Jetson Xavier

La première chose à remarquer sont les meilleures performances relatives des types vectoriels F32x2 et F16x2 par rapport aux types scalaires F32 et F16. Cela est dû à une meilleure mutualisation des accès coalescents faits par les threads du même warp. En effet, pour ces versions, moins d'accès mémoires sont réalisés et le débit mémoire est maximisé. L'utilisation du type F16x2 permet également l'utilisation d'instructions parallèles permettant le traitement de 2 nombres demi-précision en même temps.

La deuxième remarque concerne le nombre optimal d'itérations à réaliser à l'intérieur du kernel. Certaines versions comme l'implémentation Global_Pipeline sont constantes jusqu'à 16 itérations (figures 5.6c et 5.6d).

La version Shared_Pipeline présente un minimum pour 4 itérations. Cette version comporte un minimum de par la croissance des dépendances en fonction du nombre d'itérations. Le GPU comportant une quantité limitée de mémoire Shared par bloc de threads, un pipeline trop grand limite le nombre possible de blocs et donc de threads s'exécutant en parallèle. La quantité de mémoire nécessaire au calcul de P_i itérations d'un bloc de w threads est :

$$Q_{\text{Mémoire Shared}} = Q_{\text{Flottant}} \times N_a \times w \times (P_i + 1), \tag{5.1}$$

avec Q_{Flottant} la taille en octets des nombres à virgule flottante utilisés dans le kernel (8 octets pour les versions utilisant F32x2 et 4 octets pour les versions F16x2), et avec $N_a = 6$ le nombre de tableaux requis pour le calcul. De plus, sur la Jetson Xavier, le cache L1 de données et la mémoire Shared se partagent le même emplacement de mémoire physique. Allouer trop de mémoire Shared va donc limiter la quantité de cache disponible et donc dégrader davantage les performances du kernel quand celui-ci fait trop d'accès à la mémoire Global. On peut ainsi observer que les performances des versions vectorielles F32x2 et F16x2 se dégrade plus rapidement que les versions scalaires F32 et F16 : la mémoire Shared est 2 fois plus sollicitée avec les types vectoriels.

Une deuxième limitation de performance concerne le taux de threads actifs dans un bloc de threads. De par les dépendances de données, il faut aux versions Shared_Pipeline et MItIteration des halos de threads qui vont calculer plus ou moins d'itérations selon leur position dans le bloc. Ces halos correspondent à la croissance cubique des dépendances de données présentée en équation 3.3 des bords verticaux et horizontaux en fonction du nombre d'itérations à calculer. Ce n'est pas le cas pour la version Global_Pipeline : le bloc traite une ligne complète avant de passer à la ligne suivante dans le traitement. La croissance de dépendances se passe uniquement sur les dimensions verticales des tableaux de données et non sur les dimensions de la

zone à calculer par le bloc de threads. On observe ainsi une remontée du temps d'exécution pour toutes les autres versions quand trop d'itérations sont calculées. Cela est dû à une trop grande sollicitation de la mémoire Shared ainsi qu'à un mauvais taux de threads actifs dans le bloc de threads. En effet, à mesure que l'on réalise plus d'itérations, les threads présents dans les halos externes du bloc terminent leur exécution ou restent inactifs et ne participent plus à la production du résultat final. Pour la version Shared_Pipeline, la largeur de la zone finale calculée pour un bloc de taille $w \times 1$ et pour P_i itérations est de taille $w - 1 - 2 \times P_i$. On a ainsi les proportions suivantes dans la zone finale calculée pour un bloc de 9×1 pixels :

— 55,6% de threads calculant la zone finale de dimensions 5×1 pixels pour 2 itérations,
— 33,4% de threads calculant la zone finale de dimensions 3×1 pixels pour 3 itérations,
— 11,1% de threads calculant la zone finale de dimensions 4×1 pixels pour 4 itérations.

Ces proportions inférieures à 100% vont ainsi forcer le bloc à exécuter son pipeline plusieurs fois pour traiter la largeur complète de l'image.

Pour la version Global_MltIteration, la proportion de threads actifs en moyenne pour le traitement de F_i itérations dans un bloc de dimensions $w \times h$ threads est donné par la formule :

$$R_{\text{Threads Actifs}} = \frac{\sum_{k=0}^{k<2P_i}(w-k)(h-k)}{w \times h \times 2 \times P_i}. \tag{5.2}$$

Pour la version Shared_MltIteration, on a :

$$R_{\text{Threads Actifs}} = \frac{(w \times h) + \sum_{k=0}^{k<2P_i}(w-k-1)(h-k-1)}{w \times h \times (2 \times P_i + 1)}. \tag{5.3}$$

Ainsi pour un bloc de 9×9 threads, pour la version Global_MltIteration on a :

— 89,5% de threads actifs pour 1 itération (avec une zone calculée de dimensions 8×8 pixels),
— 71,0% de threads actifs pour 2 itérations (avec une zone calculée de dimensions 6×6 pixels),
— 55,8% de threads actifs pour 3 itérations (avec une zone calculée de dimensions 4×4 pixels),
— 43,8% de threads actifs pour 4 itérations (avec une zone calculée de dimensions 2×2 pixels).

Pour un bloc de 10×10 threads, pour la version Shared_MltIteration on a :

— 81,7% de threads actifs pour 1 itération (avec une zone calculée de dimensions 8×8 pixels),

— 66,0% de threads actifs pour 2 itérations (avec une zone calculée de dimensions 6×6 pixels),

— 53,0% de threads actifs pour 3 itérations (avec une zone calculée de dimensions 4×4 pixels),

— 42,7% de threads actifs pour 4 itérations (avec une zone calculée de dimensions 2×2 pixels).

Il faut de plus remarquer la diminution de la zone centrale calculée par le bloc de threads. Cette zone finale est de dimensions $(w - 2 \times P_i + 2) \times (h - 2 \times P_i + 2)$.

De manière générale, on trouve ainsi des minimaux en temps d'exécution compris entre 2 et 5 itérations lorsque le GPU est suffisamment sollicité sur des images de plus de 1024×1024 pixels.

5.1.3 Synthèse

Nous avons présenté dans cette section le travail préalable nécessaire aux évaluations en temps d'exécution, consommation énergétique et en qualité de flot optique estimé. Ce travail d'exploration est nécessaire pour trouver les meilleures configurations de nos différentes implémentations afin de pouvoir les comparer. Nous avons ainsi pu choisir et fixer les dimensions des blocs de threads ainsi que le nombre d'itérations à exécuter à l'intérieur des versions multi-itérations. Ces paramètres sont représentés en figure 5.2. Il n'est pas possible de trouver une configuration unique et idéale pour toutes les versions et pour toutes les tailles d'images. Nous avons ainsi limité notre exploration aux tailles d'images 256×256, 512×512, 1024×1024 et 2048×2048. Dans la suite de ce chapitre, lors des tests sur d'autres tailles d'images, nous choisirons la configuration la plus proche parmi ces tailles.

Algorithme	Version	Image 256 × 256		Image 512 × 512		Image 1024 × 1024		Image 2048 × 2048	
		Bloc Dim Optimal	Iter Optimal	Bloc Dim Optimal	Iter Optimal	Bloc Dim Optimal	Iter Optimal	Bloc Dim Optimal	Iter Optimal
TV-L^1	Global_Pipeline F32x2	256×1 (8 blocs)	7	512×1 (16 blocs)	6	512×1 (16 blocs)	9	1024×1 (8 blocs)	15
	Global_Pipeline F16x2	128×1 (16 blocs)	7	256×1 (32 blocs)	8	512×1 (16 blocs)	6	1024×1 (8 blocs)	2
	Shared_Pipeline F32x2	160×1 (16 blocs)	5	160×1 (16 blocs)	5	192×1 (16 blocs)	4	192×1 (16 blocs)	4
	Shared_Pipeline F16x2	192×1 (16 blocs)	9	288×1 (16 blocs)	6	192×1 (32 blocs)	4	192×1 (32 blocs)	4
	Global_MltIter F32x2	32×24	5	32×24	5	32×24	4	32×24	4
	Global_MltIter F16x2	32×24	5	32×24	4	32×24	3	32×24	3
	Shared_MltIter F32x2	32×32	6	32×32	6	32×32	4	32×32	5
	Shared_MltIter F16x2	32×32	8	32×16	3	32×16	3	32×16	3
HS	Global_Pipeline F32x2	256×1 (16 blocs)	7	256×1 (32 blocs)	8	256×1 (32 blocs)	5	256×1 (32 blocs)	4
	Global_Pipeline F16x2	256×1 (16 blocs)	8	256×1 (32 blocs)	8	512×1 (60 blocs)	9	1024×1 (16 blocs)	2
	Shared_Pipeline F32x2	160×1 (16 blocs)	6	288×1 (16 blocs)	5	256×1 (32 blocs)	3	256×1 (32 blocs)	2
	Shared_Pipeline F16x2	224×1 (16 blocs)	13	320×1 (16 blocs)	8	224×1 (32 blocs)	5	192×1 (32 blocs)	5
	Global_MltIter F32x2	32×28	4	32×32	4	32×32	3	32×16	2
	Global_MltIter F16x2	32×32	7	32×20	5	32×20	5	32×20	5
	Shared_MltIter F32x2	32×32	5	32×24	3	32×24	3	32×24	2
	Shared_MltIter F16x2	32×32	8	32×24	6	32×24	5	32×24	5

Tab. 5.2 : Dimensions des blocs et itérations optimales pour TV-L^1 et Horn-Schunck sur la Jetson Xavier. Pour les versions pipeline, on indique également le nombre de blocs alloués dans la grille.

5.2 Évaluation des performances

5.2.1 Temps de traitement

En premier lieu, les différentes versions sont évaluées en termes de temps de traitement, normalisé pour trouver le temps moyen nécessaire pour calculer un pixel depuis les images en entrée. Pour cette première série de tests, les comparaisons qualitatives ne sont pas prises en compte. Seule la vitesse de calcul des différentes optimisations des itérations TV-L^1 et de Horn-Schunck sont mesurées. Nous testons ainsi des versions mono-échelle sur des images vides allant de 128×128 à 2048×2048 pixels. Nous normalisons également le nombre d'itérations afin de pouvoir utiliser les versions multi-itérations dans leurs meilleures configurations. Nous obtenons des temps en nanosecondes par itération et par pixel (ns/iter/pix).

5.2.1.1 TV-L^1

Nous commençons par présenter nos résultats pour TV-L^1. Afin d'être complets dans notre étude, nous avons testé toutes nos versions avec les 2 types scalaires F32 et F16 et les 2 types vectoriels F32x2 et F16x2 sur les 3 cartes Jetson visées. Nous présentons les résultats temporels normalisés pour la Jetson Xavier en figure 5.10, les résultats de la Jetson TX2 en figure 5.11 et les résultats de la Jetson Nano en figure 5.12.

Jetson Xavier Globalement, l'implémentation la plus rapide est la version Shared_MltIteration. Les optimisations nous permettent ainsi d'atteindre 0,21 ns/iter/pix en F16x2, 0,26 ns/iter/pix en F16, 0,30 ns/iter/pix en F32 et 0,31 ns/iter/pix en F32x2 pour des résolutions allant jusqu'à 2048×2048 pixels. En F16 et F16x2, la version Shared_Pipeline est cependant légèrement plus rapide que la version Shared_MltIteration. Grâce à la fusion de plusieurs itérations dans le même bloc de threads, la localité des données est améliorée et le nombre de transferts de mémoire avec la mémoire Global diminue. En outre, l'utilisation de la mémoire Shared permet une diminution de la latence mémoire (similaire au cache L1) et une bande passante mémoire plus importante (128 octets par cycle d'horloge et par SM) par rapport à la mémoire Global.

L'utilisation de la seule mémoire Shared, comme dans les versions Shared, n'est cependant pas suffisante pour obtenir un temps d'exécution plus court. La quantité de données réutilisées dans la mémoire partagée dans les versions ne réalisant qu'une seule itération par appel de kernel est très limitée. Les versions Global_MltIteration n'utilisent pas la mémoire partagée, mais peuvent réutiliser davantage de données dans la mémoire Global grâce aux caches de données L1 et L2. Elle est plus rapide que les versions Shared ainsi que les versions Shared_Fusion en F32 et F32x2. Ce n'est plus le cas pour la version Shared_Fusion lors de l'utilisation des nombres F16 et F16x2.

Concernant le format de nombre employé, l'utilisation des types vectoriels présente certains avantages. Il y a peu de différences entre le F32 et le F32x2 : les versions plafonnent aux mêmes niveaux. On observe cependant une réduction de l'étalement des courbes : les versions les plus lentes bénéficient le plus de cette optimisation. Cette réduction de l'étalement est encore plus notable entre le F16 et le F16x2.

On notera que notre implémentation utilisant les instructions Shuffle est la version la plus lente dans tous nos cas. Cela est dû à la taille de blocs fixe faisant la

taille d'un warp de 32 threads. Sans gestion des bords de warp, il n'est pas possible d'utiliser les instructions Shuffle pour partager des registres vers d'autres warps. De plus, l'utilisation de ces instructions introduit des barrières de synchronisation supplémentaires qui limitent encore plus les performances.

Enfin on notera qu'il est nécessaire d'exécuter nos versions sur des images suffisamment grandes pour solliciter suffisamment le GPU. Les versions commencent à se différencier à partir d'images de taille 512×512 pixels.

Jetson TX2 L'analyse des résultats pour la Jetson TX2 est proche de celle de la Jetson Xavier. On atteint ainsi 0,84 ns/iter/pix en F16x2, 1,08 ns/iter/pix en F16, 1,14 ns/iter/pix en F32 et 1,20 ns/iter/pix en F32x2 pour des résolutions allant jusqu'à 2048×2048 pixels. Les temps obtenus sont plus lents que la Jetson Xavier, le GPU de la TX2 ayant une puissance de calcul plus petite. La meilleure version est la version Shared_MItIteration. La version Shared_Pipeline est également l'une des meilleures versions F16x2, voire légèrement plus rapide en F16. On voit également un net avantage à l'utilisation des F16x2 qui permet un gain de 20% sur la meilleure version.

Pour cette carte et ce GPU, on peut observer une dégradation des performances pour des images entre 1024×1024 et 1280×1280 pixels. Cela est dû au fait que des images plus grandes vont requérir plus de blocs pour leur traitement, et donc une quantité totale de mémoire Shared plus importante. Cela limite donc le nombre de blocs qui peuvent être exécutés en parallèle. Alternativement, il aurait été possible d'allouer plus de Shared au détriment du cache afin de conserver le même niveau de parallélisme, mais cela est encore moins souhaitable d'un point de vue performance. Ce phénomène n'était pas observable sur la Jetson Xavier de par ses quantités totales de cache et de Shared (512 Ko de cache L2 et 1024 Ko de cache L1/Shared) plus grandes que sur la Jetson TX2 (512 Ko de cache L2 et 48 Ko de cache L1 et 128 Ko de mémoire Shared). Pour la mémoire Shared, cette quantité mémoire limite également le nombre de blocs de threads pouvant être lancés en parallèle.

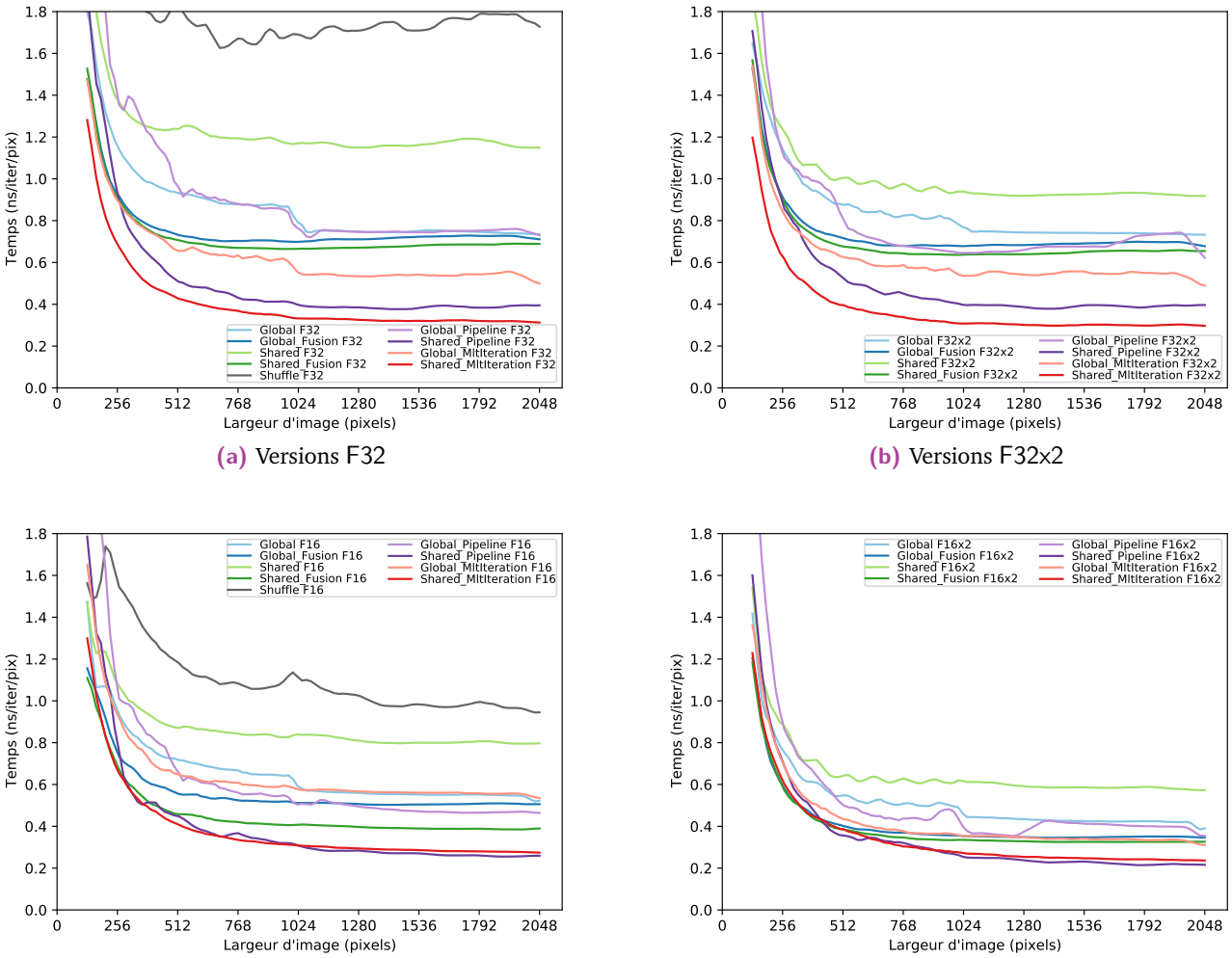
Les performances de la version Global_MItIteration sur TX2 sont également très dégradées par rapport à son exécution sur la Jetson Xavier. Cette version nécessite un grand nombre de registres pour contenir les données chargées par un bloc. La TX2 ayant un nombre maximum de registres par bloc 2 fois moins grand que la Jetson Xavier (32768 registres sur TX2 et 65536 sur Xavier), il n'est pas possible d'allouer des blocs suffisamment grands pour réutiliser un maximum de données. Lorsque les ressources sont correctement sollicitées et que le matériel ne limite pas

le lancement des kernels, on retrouve la même répartition de versions que sur la Jetson Xavier. C'est le cas pour les versions utilisant le type F16x2.

Jetson Nano Sur la Jetson Nano, on observe des résultats similaires à la Jetson TX2. On atteint ainsi 2,28 ns/iter/pix en F16x2, 2,74 ns/iter/pix en F16, 3,14 ns/iter/pix en F32x2 et 3,17 ns/iter/pix en F32 pour des résolutions allant jusqu'à 1024×1024 pixels. Cette architecture ayant encore plus de limitations physiques que sur la Jetson Xavier, nous n'avons pas testé d'implémentation sur des images allant au-delà de 1024×1024 pixels. La version Shared_MltIteration reste globalement la meilleure version, mais est cette fois devancée par la version Shared_Pipeline en F16 et F16x2.

La dégradation des performances commence cette fois-ci autour de 768×768 pixels, mais n'est pas visible en F16 et F16x2.

Hormis la version Shuffle, la version Global_MltIteration est l'implémentation la moins performante sur cette carte. Là encore, cette version est limitée par les registres, qui limitent à la fois la taille des blocs possibles, mais aussi le nombre de blocs pouvant être lancés en parallèle.



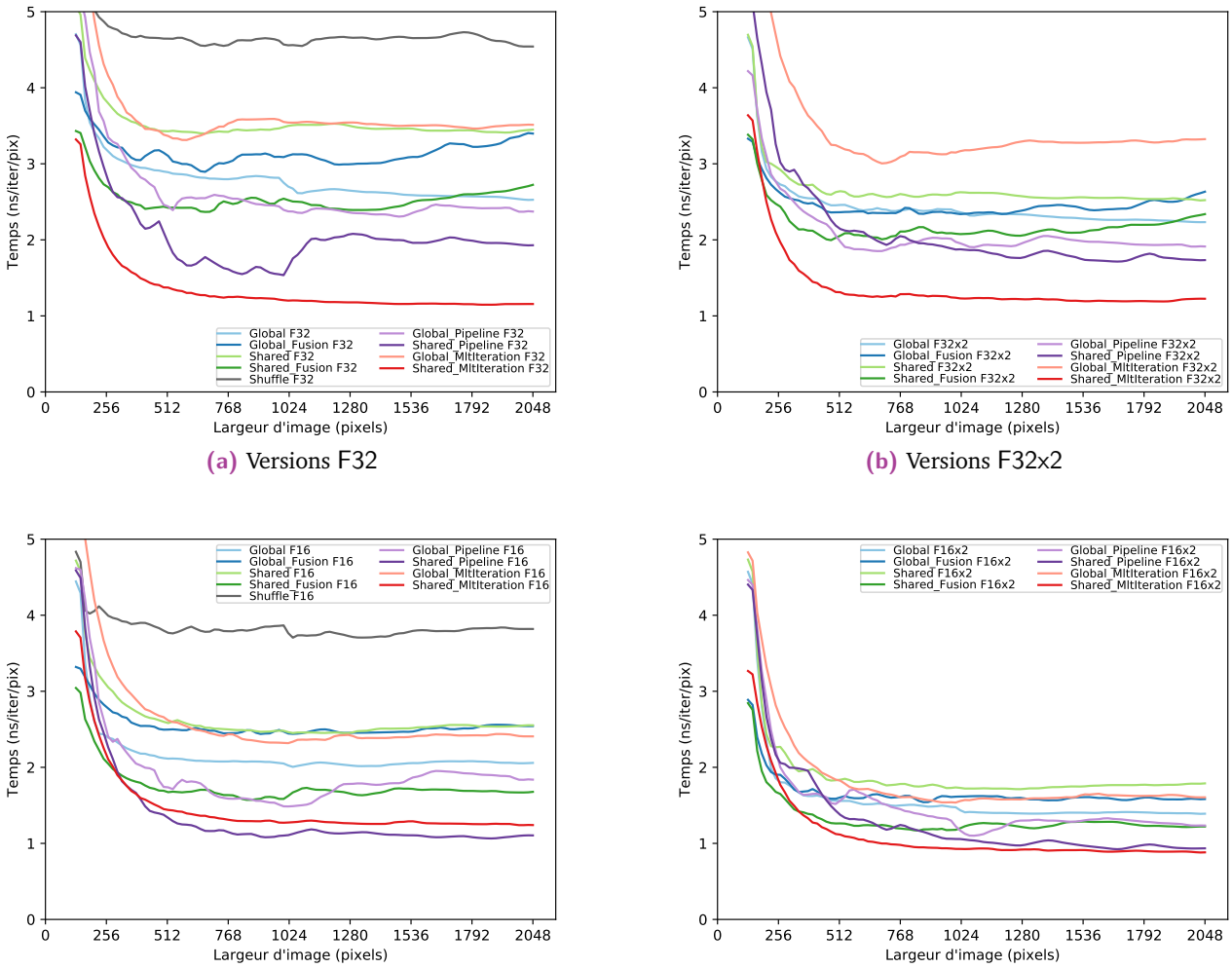
(a) Versions F32

(b) Versions F32x2

(c) Versions F16

(d) Versions F16x2

Fig. 5.10 : Temps d'exécution du schéma itératif de **TV-L**[1] sur Jetson Xavier



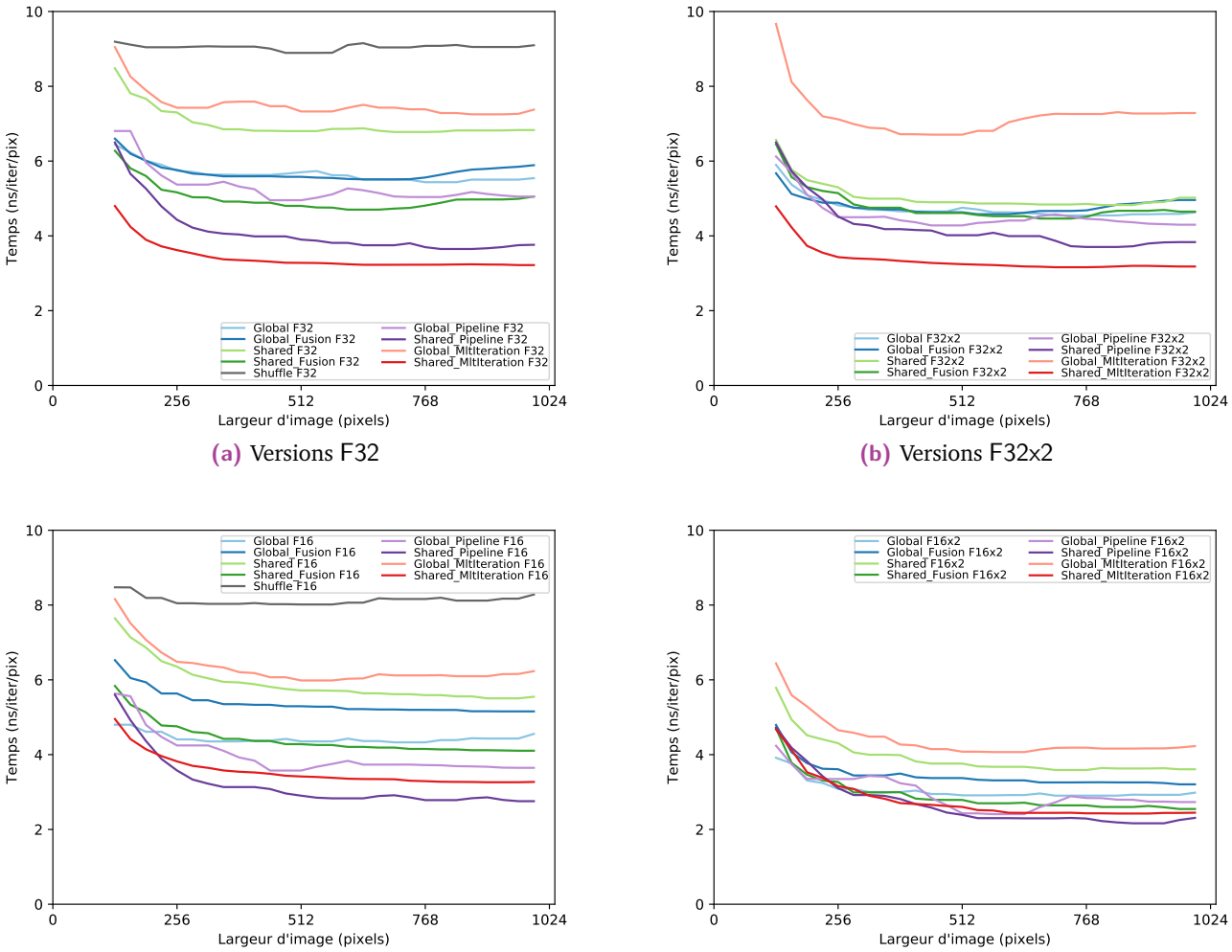
(a) Versions F32

(b) Versions F32x2

(c) Versions F16

(d) Versions F16x2

Fig. 5.11 : Temps d'exécution du schéma itératif de **TV-L**[1] sur Jetson TX2



(a) Versions F32

(b) Versions F32x2

(c) Versions F16

(d) Versions F16x2

Fig. 5.12 : Temps d'exécution du schéma itératif de **TV-L**[1] sur Jetson Nano

5.2.1.2 Horn-Schunck

Nous réalisons la même série de tests pour les implémentations de Horn-Schunck sur les trois Jetson, en faisant varier les tailles d'images de 128×128 à 2048×2048 pixels. Les temps sont de nouveau normalisés pour obtenir là encore des ns/iter/pix. Les résultats de la Jetson Xavier sont présentés sur la figure 5.13, ceux de la Jetson TX2 sur la figure 5.14 et ceux de la Jetson Nano sur la figure 5.15.

Jetson Xavier Pour cette plateforme, il faut distinguer le comportement des implémentations simple-précision F32 et F32x2 des implémentations demi-précision F16 et F16x2. Dans le premier cas, la version Global est la plus rapide. Les versions multi-itératives ne sont pas efficaces et la réutilisation de données en mémoire Shared n'offre pas de gains de performances. Ce n'est pas le cas pour les versions demi précision. Dans ce cas, les versions multi-itératives utilisant la mémoire Shared sont les plus rapides (les implémentations Shared_Pipeline et Shared_MltIteration). Il faut cependant noter que ces versions ne sont efficaces qu'à partir d'une certaine taille d'image (environ 1152×1152 pixels en F16 et 896×896 en F16x2). Les optimisations nous permettent ainsi d'atteindre 0,12 ns/iter/pix en F16x2, 0,17 ns/iter/pix en F16, 0,62 ns/iter/pix en F32x2 et 0,62 ns/iter/pix en F32 pour des résolutions allant jusqu'à 2048×2048 pixels.

L'algorithme de Horn-Schunck est moins intensif au niveau de la mémoire que l'algorithme TV-L^1 : pour le calcul d'un point de flot optique pour l'algorithme de TV-L^1, il faut accéder à 46 points dans les différents tableaux alors qu'il n'en faut que 19 pour Horn-Schunck. Les optimisations visant à maximiser la réutilisation des données et à augmenter la localité temporelle et spatiale en cache auront donc moins d'effet. C'est pour cela que la version la plus rapide est la version Global la plus simple. Les versions Shared_MltIteration et Shared_Pipeline sont efficaces uniquement en F16 et F16x2 et à partir d'une taille d'image de 768×768 pixels. Là encore, la réutilisation des données dans ce cas n'est pas suffisante pour compenser la perte de threads traitant des données au fur et à mesure des itérations internes des kernels.

Jetson TX2 Pour la Jetson TX2, on retrouve des conclusions similaires à la comparaison entre la Xavier et la TX2 pour TV-L^1. Les optimisations nous permettent ainsi d'atteindre 0,45 ns/iter/pix en F16x2, 0,67 ns/iter/pix en F16, 1,28 ns/iter/pix en F32x2 et 1,30 ns/iter/pix en F32.

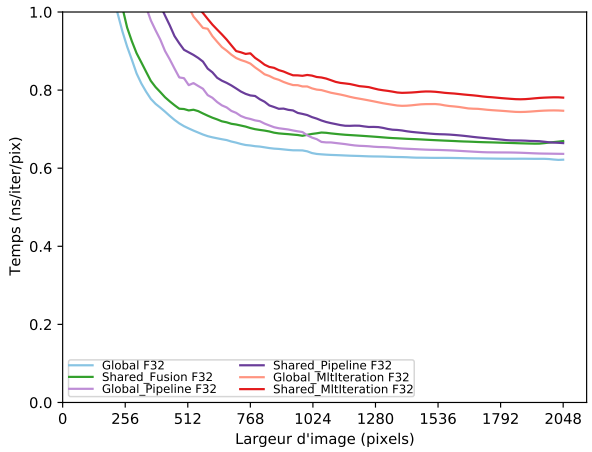
Similairement aux implémentations de TV-L^1, les versions Global_MltIteration sont les plus lentes. Là encore, la réutilisation des données en mémoire Global

ne compense pas les pertes des threads réalisant le traitement à chaque itération interne du kernel. Il n'y a que la version utilisant la mémoire Shared qui présente les meilleures performances en F16x2.

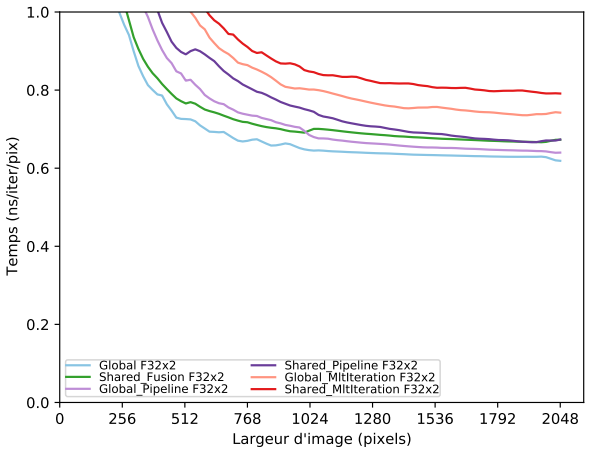
Comme pour TV-L[1], en F32 et F32x2, les versions Global_Pipeline et Shared_Pipeline sont les plus performantes après la version Global. La version Shared_Pipeline est également l'une des deux versions les plus rapides en F16 et F16x2 après la version Shared_MltIteration

Jetson Nano Pour la Jetson Nano, nous obtenons 1,30 ns/iter/pix en F16x2 (version Shared_MltIteration), 1,94 ns/iter/pix en F16 (version Shared_Pipeline), 3,41 ns/iter/pix en F32 (version Global) et 3,90 ns/iter/pix en F32x2 (version Global_Pipeline). Parmi les implémentations en F32, la version Global reste la meilleure comme précédemment. Pour les versions F32x2, la version Global_Pipeline est la plus rapide. La Jetson Nano est la moins puissante des 3 cartes testées. L'utilisation du type F32x2 induit un surcoût d'utilisation similaire à des instructions de mélanges et de décalages *shuffle* SIMD. À la différence du type F16x2, les cœurs de calculs CUDA ne sont pas capables de produire 2 nombres flottants de 32 bits par cycle d'horloge. On observe ici ce surcoût, qui est en partie compensé dans la version Global_Pipeline qui sollicite davantage les caches de données pour la réutilisation que la version Global.

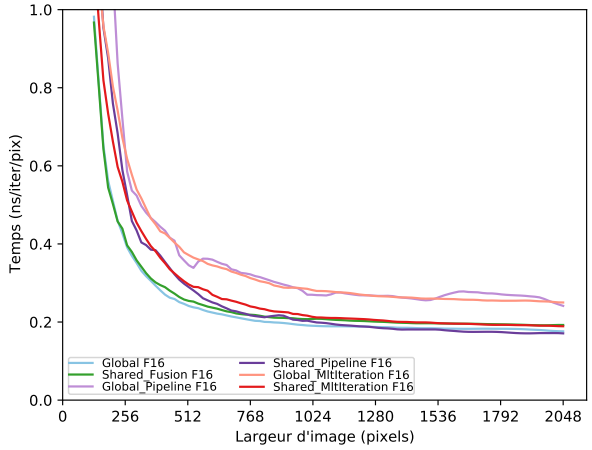
Concernant les types F16 et F16x2, les versions Global, Shared_Pipeline et Shared_MltIteration produisent les meilleurs résultats.



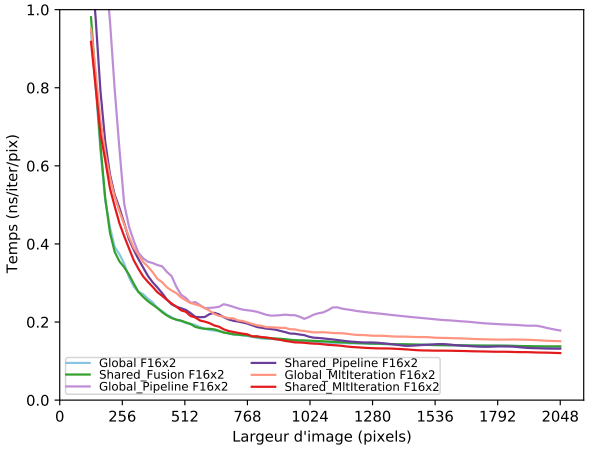
(a) Versions F32



(b) Versions F32x2

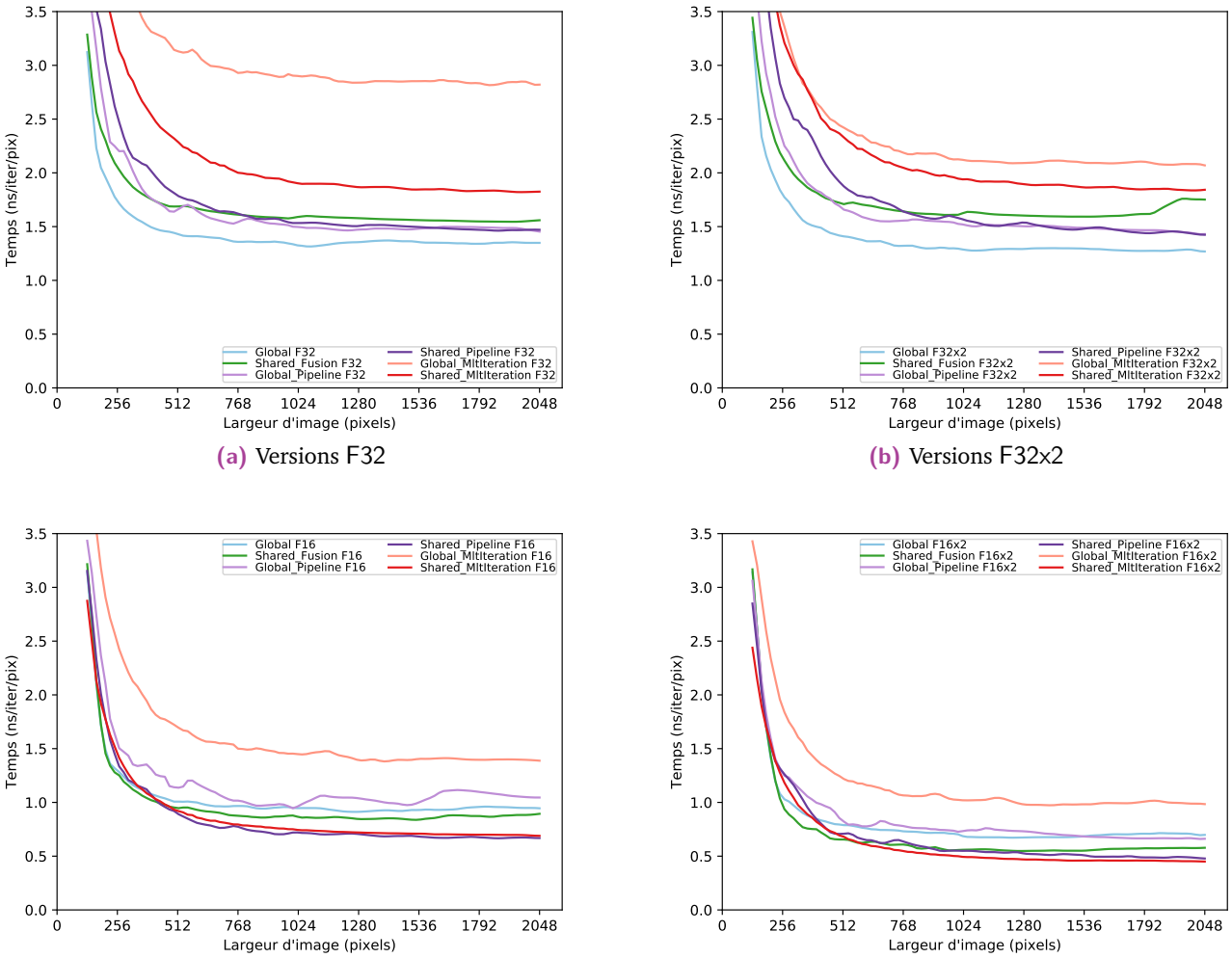


(c) Versions F16



(d) Versions F16x2

Fig. 5.13 : Temps d'exécution du schéma itératif de **Horn-Schunck** sur Jetson Xavier.



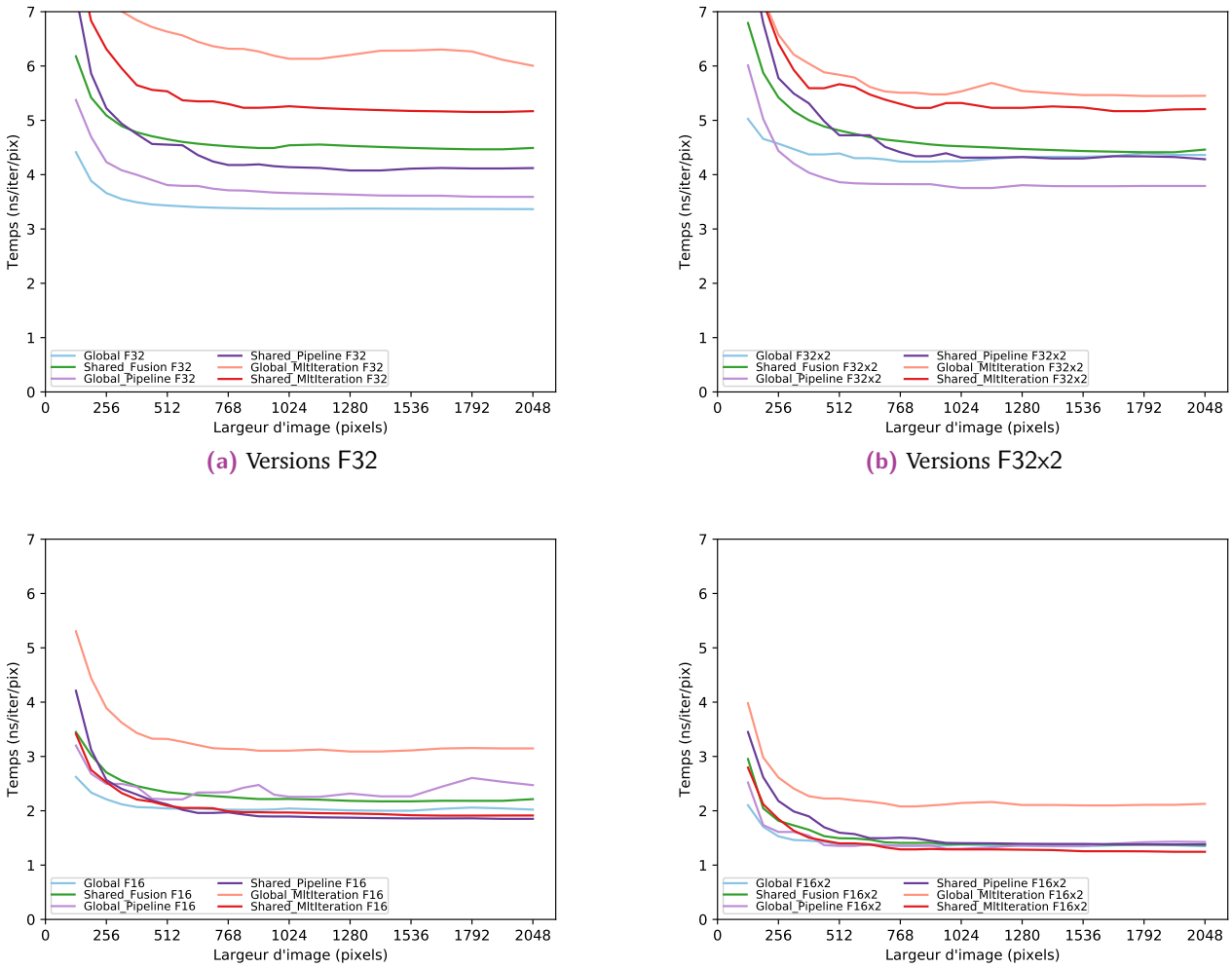
(a) Versions F32

(b) Versions F32x2

(c) Versions F16

(d) Versions F16x2

Fig. 5.14 : Temps d'exécution du schéma itératif de **Horn-Schunck** sur Jetson TX2.



(a) Versions F32

(b) Versions F32x2

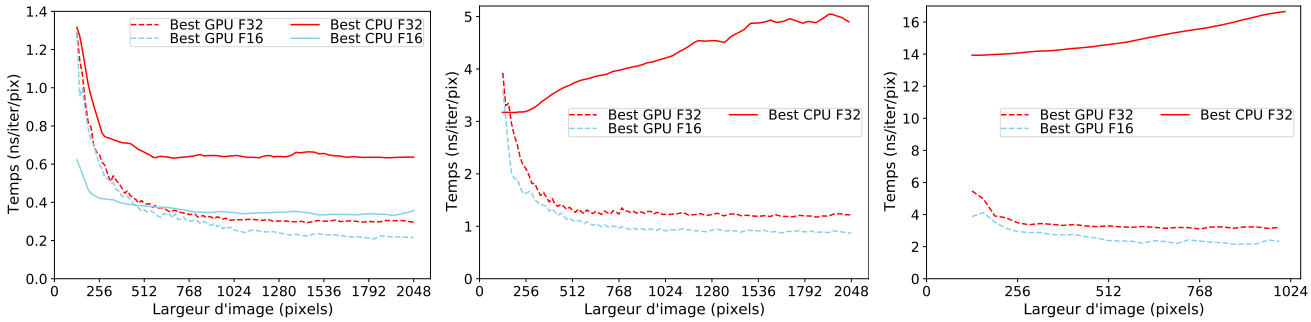
(c) Versions F16

(d) Versions F16x2

Fig. 5.15 : Temps d'exécution du schéma itératif de **Horn-Schunck** sur Jetson Nano.

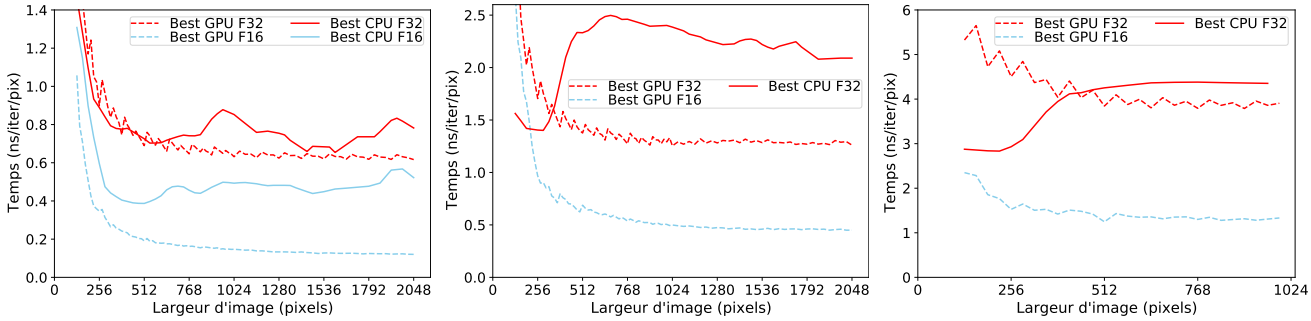
5.2.1.3 Pré-résultat GPU-CPU

Pour ce qui est de la comparaison entre le GPU et le CPU, nous comparons en figures 5.16 et 5.17 les versions optimisées CPU SIMD Neon de TV-L^1 et de Horn-Schunck avec nos meilleures versions, déclinées en nombres flottants simple précision et en demi précision. Ces implémentations sont issues de travaux de la thèse d'Andrea Petreto [Pet20] pour TV-L^1 et de la thèse en cours de Maxime Millet [Mil+23] pour Horn-Schunck. Là encore, cette étude a été réalisée sur des versions mono-échelle complètes n'incluant que la boucle itérative du schéma numérique de TV-L^1 (voir algorithme 1) et de Horn-Schunck(voir algorithme 3). Ces tests sont réalisés pour une configuration de 10 itérations du schéma numérique et les résultats normalisés en nanosecondes par itérations et par pixel (ns/iter/pix).



(a) Xavier (b) TX2 (c) Nano

Fig. 5.16 : Temps d'exécution des implémentations optimisées GPU et CPU de TV-L^1 simple-précision (*float*) et demi-précision (*half*).



(a) Xavier (b) TX2 (c) Nano

Fig. 5.17 : Temps d'exécution des implémentations optimisées GPU et CPU de Horn-Schunck simple-précision (*float*) et demi-précision (*half*).

Sur les 3 cartes et pour les 2 algorithmes, il y a un avantage à réaliser le traitement sur GPU plutôt que sur CPU à partir d'une taille d'image suffisamment grande.

Pour TV-L^1 et pour Horn-Schunck, sur la Jetson Xavier on sollicite suffisamment le GPU à partir de taille d'image d'environ 512×512 pixels. En dessous de ces tailles d'images, le CPU est plus rapide pour des implémentations SIMD Neon optimisées. La seule implémentation toujours plus rapide que le CPU sur ces cartes et la version F16x2 de notre meilleure implémentation GPU de Horn-Schunck. Pour des grandes tailles d'images, à la convergence des courbes, l'implémentation CPU F32 de TV-L^1 atteint 0,64 ns/iter/pix, la version F16 atteint 0,34 ns/iter/pix. Les implémentations GPU atteignent elles 0,31 ns/iter/pix pour la meilleure version F32 (la version Shared_MltIteration F32x2 et 0,21 ns/iter/pix pour la meilleure version F16 l'implémentation Shared_Pipeline F16x2). Pour Horn-Schunck, l'implémentation CPU SIMD Neon F32) de TV-L^1 atteint 0,77 ns/iter/pix et la version F16 atteint 0,51 ns/iter/pix pour des tailles d'images à la convergence des courbes (2048×2048 pixels). Les implémentations GPU atteignent 0,62 ns/iter/pix pour la meilleure version F32 (la version Global_F32x2 et 0,12 ns/iter/pix pour la meilleure version F16 (l'implémentation Shared_MltIteration F16x2).

Il faut également noter que pour Horn-Schunck sur la Jetson Xavier, la version F32 sur CPU reste proche du temps d'exécution de l'implémentation F32 sur GPU. On observe un pic du temps d'exécution sur CPU pour des tailles d'images autour de 1024×1024 puis une redescente de ce temps pour obtenir des temps proche pour des tailles d'images comprise entre environ 1500×1500 et 1750×1750. Ces pics sont liés aux tailles d'images en entrée, induisant des alignements de tableaux en mémoire défavorable pour les caches : il y a un pic des évincements de données, ce qui réduit les performances considérablement.

Sur Jetson TX2, la version CPU Neon F32 de TV-L^1 est plus efficace que la meilleure version GPU F32 pour les résolutions inférieures à 140×140 pixels, et plus efficace que la meilleure version F16 pour les résolutions inférieures à 170×170 pixels. On notera que sur le CPU de cette carte, il n'y a pas d'unité de calcul F16. Pour Horn-Schunck, la version CPU Neon F32 est plus efficace que la meilleure version GPU F32 pour les résolutions inférieures à 350×350 pixels, et plus efficace que la meilleure version F16 pour les résolutions inférieures à 210×210 pixels. À une résolution de 2048×2048 pixels, la version F32 CPU perd en efficacité et atteint 4,9 ns/iter/pix. Pour cette résolution, la meilleure version GPU F32 atteint 1,20 ns/iter/pix et la meilleure version F16 0,84 ns/iter/pix. Pour Horn-Schunck sur cette même carte et pour cette résolution, la version CPU atteint un temps de 2,1 ns/iter/pix. La meilleure version GPU F32 atteint elle 1,28 ns/iter/pix et la meilleure version GPU F16 0,45 ns/iter/pix.

Enfin, sur Jetson Nano, la version CPU Neon F32 de TV-L^1 n'est jamais plus rapide que les meilleures versions GPU F32 et F16. De plus, l'augmentation des tailles d'images fait perdre encore davantage en efficacité pour la version CPU alors que c'est l'inverse pour les versions GPU : elle gagne en efficacité. À une résolution de 1024×1024 pixels, la version CPU atteint 16,6 ns/iter/pix. La meilleure version GPU F32 atteint 3,2 ns/iter/pix pix et la meilleure version GPU F16 atteint 2,3 ns/iter/pix. Pour Horn-Schunck, la version CPU Neon F32 est plus efficace que la meilleure version GPU F32 pour les résolutions inférieures à 440×440 pixels. La meilleure version GPU F16 est elle plus rapide pour toutes les tailles d'images testées. Sur la même carte et pour une résolution de 1024×1024 pixels, la version CPU atteint un temps d'exécution de 4,35 ns/iter/pix. À cette résolution, la version GPU F32 atteint 3,9 ns/iter/pix et la version GPU F16 1,3 ns/iter/pix.

5.2.1.4 Synthèse

Version	Ops. flottantes par pixel	Accès mémoire par pixel	Intensité arithmétique	Temps (ns/pix)	Vitesse de traitement (GFLOPS)	Débit mémoire généré (Go/s)
Global F32x2	640	181	0,44	8,17	78	165
Global F16x2	508	181	0,70	4,36	117	155
Shared_fusion F32x2	640	251	0,32	8,13	79	230
Shared_fusion F16x2	488	261	0,47	4,31	113	226
Global_Pipeline F32x2	640	181	0,44	6,65	96	203
Global_Pipeline F16x2	508	181	0,70	4,02	126	168
Shared_Pipeline F32x2	660	219	0,38	5,10	129	320
Shared_Pipeline F16x2	553	219	0,63	**2,86**	193	285
Global_MltIteration F32x2	902	165	0,68	5,85	154	210
Global_MltIteration F16x2	680	165	1,03	4,02	169	153
Shared_MltIteration F32x2	895	182	0,61	*3,62*	**247**	**374**
Shared_MltIteration F16x2	675	182	0,93	3,35	202	203

(a) Résultats des performances pour **TV-L^1**.

Version	Ops. flottantes par pixel	Accès mémoire par pixel	Intensité arithmétique	Temps (ns/pix)	Vitesse de traitement (GFLOPS)	Débit mémoire généré (Go/s)
Global F32x2	420	133	0,39	*6,77*	62	143
Global F16x2	254	133	0,48	**1,73**	146	280
Shared_fusion F32x2	420	153	0,34	7,25	58	154
Shared_fusion F16x2	254	153	0,42	1,96	129	285
Global_Pipeline F32x2	420	133	0,39	6,87	61	141
Global_Pipeline F16x2	254	133	0,48	2,28	111	213
Shared_Pipeline F32x2	422	169	0,31	7,52	56	165
Shared_Pipeline F16x2	276	155	0,45	1,77	156	320
Global_MltIteration F32x2	433	138	0,39	8,48	51	118
Global_MltIteration F16x2	324	185	0,44	2,03	159	332
Shared_MltIteration F32x2	430	155	0,35	9,14	47	124
Shared_MltIteration F16x2	319	193	0,41	1,92	**166**	**369**

(b) Résultats des performances pour **Horn-Schunck**

Tab. 5.3 : Accès mémoire et opérations en virgule flottante par pixel pour nos implémentations de TV-L^1 et de Horn-Schunck sur le Jetson Xavier. Le temps d'exécution, le débit de calcul et le débit mémoire généré associés ont été calculés pour des images de 2048×2048 pixels et pour les meilleures configurations mono-échelle, 1 warp et 10 itérations.

Dans cette sous-section, nous avons évalué les performances en temps d'exécution de nos différentes versions de TV-L^1 et de Horn-Schunck. Les tableaux sur les figures 5.3 résument les meilleures performances atteintes pour ces versions pour une taille d'image de 2048×2048 pixels pour une configuration mono-échelle complète de 10 itérations sur Jetson Xavier. Grâce à nos optimisations, nous obtenons 2,86 ns/pix en F16x2 et 3,62 ns/pix en F32x2 pour TV-L^1 et 1,73 ns/pix en F16x2 et 6,77 ns/pix en F32x2 pour Horn-Schunck.

La première chose à noter est la différence de performance entre les implémentations des deux algorithmes. Entre TV-L^1 et Horn-Schunck, en moyenne, il faut 72% plus d'opérations flottantes et 34% plus d'accès mémoires en F32x2 et 103% plus d'opérations flottantes et 25% plus d'accès mémoires en F16x2. TV-L^1 est un algorithme plus complexe nécessitant plus de calculs et plus d'accès à des données en mémoire. Il va donc beaucoup plus solliciter le GPU que Horn-Schunck. Cela explique ces différences de performances.

La deuxième différence majeure concerne le gain lié à un passage de nombres F32x2 vers des nombres F16x2. Dans le cas de TV-L^1, un passage vers des nombres flottants demi-précision entraîne une augmentation moyenne des performances en GFLOPS de 17% et une diminution du débit mémoire généré en Go/s de 26%. Dans le cas de Horn-Schunck, on observe une augmentation des performances de calcul de 259% et une augmentation du débit mémoire de 212%. Là encore, TV-L^1 nécessite plus de calculs et plus d'accès mémoire : il sature le GPU en F32x2 ainsi qu'en F16x2. Ce n'est pas le cas pour Horn-Schunck : cet algorithme étant plus léger, il bénéficie mieux du passage vers des nombres demi-précision. Ceux-ci permettent ainsi à la fois un traitement parallèle par paquet de deux nombres, mais aussi une réduction de la quantité de mémoire d'un facteur 2, diminuant ainsi le temps d'attente à la mémoire, et augmentant de fait le débit mémoire.

Enfin, concernant la comparaison entre les versions GPU et CPU, le GPU est plus efficace que le CPU qu'à partir d'une taille d'image sollicitant suffisamment les cœurs de calcul et la mémoire du GPU. Cette taille dépend de la plateforme de calcul et de l'algorithme TV-L^1 ou Horn-Schunck, mais elle se situe au maximum à 512×512 pixels. Au dessus de cette résolution, le GPU est plus efficace et la disponibilité d'unité de calcul demi précision F16 non disponible sur 2 des 3 CPU testés permet une deuxième réduction des temps de calcul. Pour des tailles suffisamment grandes, pour TV-L^1 et par rapport aux implémentations CPU les plus rapides, les meilleures versions GPU sont $1,62\times$ plus rapides sur la Jetson Xavier, $5,8\times$ plus rapide sur la Jetson TX2 et $7,2\times$ plus rapide sur la Jetson Nano. Pour Horn-Schunck, les meilleures versions GPU sont $4,3\times$ plus rapides sur la Jetson Xavier, $4,7\times$ plus

rapides sur la Jetson TX2 et $3,3\times$ plus rapides sur la Jetson Nano, que les meilleures implémentations CPU.

5.2.2 Consommation énergétique

Maintenant que nous avons déterminé les meilleures configurations pour les meilleures versions pour les deux algorithmes et pour chaque carte, nous évaluons la consommation énergétique de nos implémentations. Pour ce faire, des cartes de mesure électronique ont été spécifiquement développées à cet égard par l'ingénieur de notre équipe Manuel Bouyer [Pet+18a]. Elles sont insérées entre la plateforme de calcul et son alimentation électrique. Elles mesurent ainsi la tension et le courant du système complet et envoient cette mesure vers un ordinateur hôte avec une fréquence d'échantillonnage de 5000 échantillons à la seconde (5 kHz). Les sorties GPIO des plateformes Jetson permettent d'associer une mesure de consommation avec une exécution spécifique du programme à tester. Ces outils ont été développés dans le cadre du projet Météorix, projet de nano-satellite réalisant un traitement embarqué de détection temps-réel de météores et de débris spatiaux [Cio+23a ; Kan+23b ; Cio+23b ; Kan+23a ; Mil+22a ; Mil+22c ; Vau+22 ; Mil+22b ; Kan+ ; Mil+21 ; Col+20 ; Ram+19 ; Ram+18].

Avec ces outils de mesures, nous pouvons déterminer les points de fonctionnement de nos différentes versions en fonction du temps d'exécution et de la consommation énergétique. Ces mesures sont normalisées par le nombre de pixels des images testées pour obtenir ainsi des nanosecondes par pixel (ns/pix) ainsi que des nanojoules par pixel (nJ/pix). Nous testons ici les versions mono-échelle complètes et non pas uniquement le schéma itératif interne de nos algorithmes. Nous fixons les mêmes paramètres pour toutes les versions, à savoir 1 seul warp de compensation de mouvement et 10 itérations, pour des images de 2048×2048 pixels. De plus, il faut noter que nous ne gardons désormais que les versions vectorielles F32x2 et F16x2 [Rom+23].

5.2.2.1 TV-L^1

La figure 5.18 présente les résultats dans l'espace (temps par pixel, énergie par pixel) pour les implémentations de TV-L^1 sur GPU ainsi que les versions CPU Neon optimisées et la version OpenCV GPU. Les points multiples sur une courbe correspondent à des fréquences d'horloge différentes présentes sur le front de Pareto

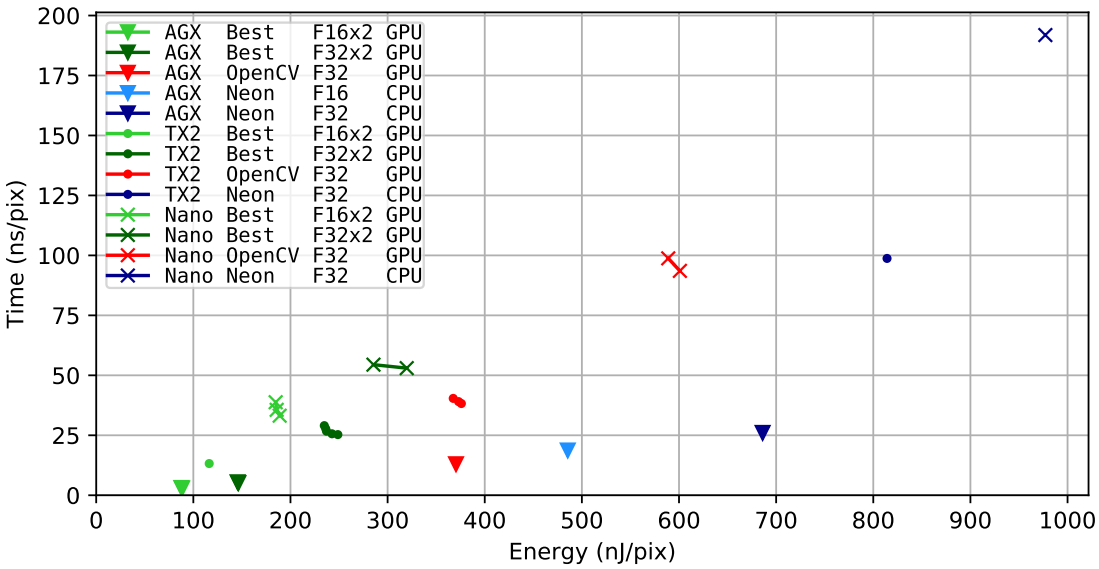


Fig. 5.18 : Points de fonctionnement en temps (ns/pix) et en énergie (nJ/pix) pour les meilleures implémentations de **TV-L**[1] sur chaque carte Jetson. Plusieurs fréquences d'horloge sont testées pour chaque version pour une image de 2048×2048 pixels.

dans cet espace. Seule la fréquence maximale est indiquée pour les versions CPU SIMD Neon.

La Jetson Xavier obtient les meilleures performances en termes de temps d'exécution et de consommation énergétique, suivie par la TX2 et la Nano. L'implémentation GPU la plus rapide testée est la version Shared_pipeline_F16x2 et elle est $7.4\times$ plus rapide et $6.4\times$ plus économe en énergie que la version optimisée sur CPU Neon_F16. Elle est également $5,2\times$ plus rapide et $4,9\times$ plus économe en énergie que la version OpenCV F_{32} GPU. Cette différence d'efficacité s'explique par les optimisations réalisées qui permettent une meilleure maîtrise de l'architecture sous-jacente qu'un code plus générique. La meilleure implémentation F_{32} GPU testée est la version Shared_pipeline_F32x2 et elle est $5,7\times$ plus rapide et $5,3\times$ plus économe en énergie que la version Neon_F16 sur CPU. Toujours par rapport à la version GPU F_{32}, elle est $2,9\times$ plus rapide et $2,8\times$ plus économe en énergie.

La Jetson TX2 est la deuxième meilleure carte en termes de temps d'exécution et de consommation d'énergie. La meilleure implémentation sur cette carte est également la version Shared_pipeline_F16x2. Par rapport à la Jetson Xavier, elle est $4,6\times$ plus lente et $1,4\times$ moins économe en énergie.

La Jetson Nano est la carte la moins bien classée. Là encore, parmi les versions évaluées, la version Shared_pipeline_F16x2 est la plus rapide. Cette carte est $11,6\times$ plus lente et $2,1\times$ moins économe en énergie que la Jetson Xavier.

Un matériel plus récent et plus puissant, bien que plus gourmand en énergie, permet à nos optimisations de fonctionner plus efficacement. Par rapport aux deux plus petites et anciennes cartes, la Jetson Xavier a une consommation d'énergie totale réduite et permet à la version optimisée Shared_pipeline_F16x2 de fonctionner le plus rapidement possible.

5.2.2.2 Horn-Schunck

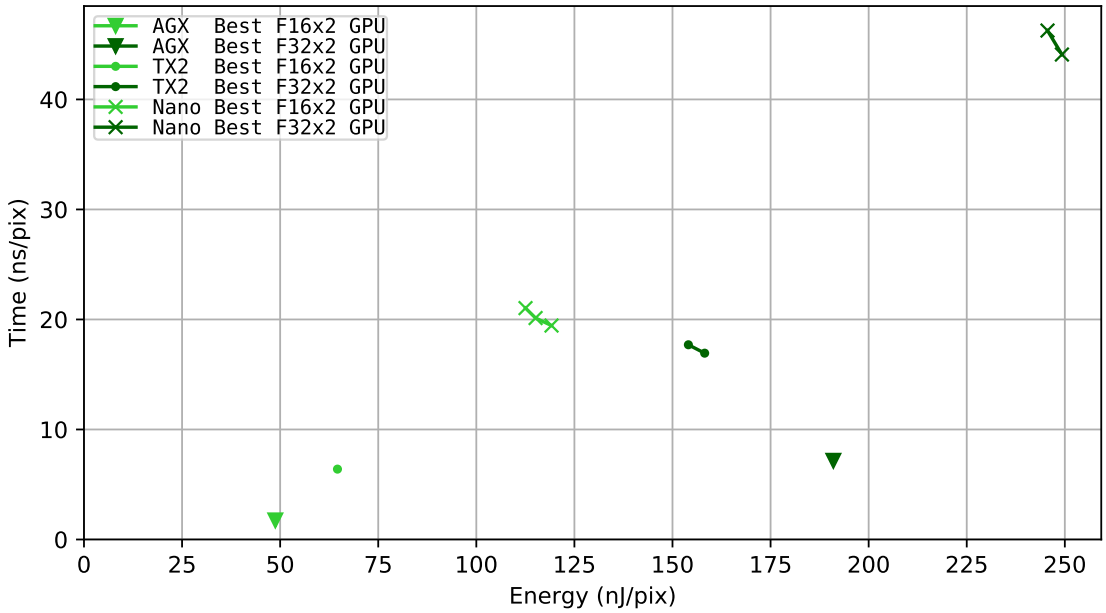


Fig. 5.19 : Points de fonctionnement en temps (ns/pix) et en énergie (nJ/pix) pour les meilleures implémentations de **Horn-Schunck** sur chaque carte Jetson. Plusieurs fréquences d'horloge sont testées pour chaque version pour une image de 2048×2048 pixels.

La figure 5.19 présente les résultats pour les implémentations de Horn-Schunck sur GPU. Il n'existe pas d'implémentation de Horn-Schunck dans la bibliothèque OpenCV. Là encore, l'implémentation et les évaluations de l'algorithme de Horn-Schunck viennent accompagner les résultats pour l'algorithme TV-L[1] à titre de référence et d'exemple d'algorithme de type stencil itératif plus simple pour le calcul de flot optique. Une version optimisée SIMD Neon est en cours de développement dans le cadre d'une nouvelle thèse : « Irregular & Architecture : déploiement d'algorithmes irréguliers sur architectures parallèles hétérogènes CPU+GPU » et nous permettra ainsi de compléter notre nuage de point.

La Jetson Xavier est la carte la plus rapide en F16x2 et F32x2 mais n'est plus la plus efficace partout. La version F32x2 sur cette carte est $2,4\times$ plus rapide que celle de la Jetson TX2 mais est $1,2\times$ moins énergiquement efficace que cette dernière. Cela est dû à une sous-utilisation du GPU de la Jetson Xavier pour ce format et pour

cet algorithme. Baisser la dynamique du format permet de libérer suffisamment de ressources pour permettre une meilleure allocation des blocs de threads sur le GPU de cette carte et permettre de gagner en efficacité.

La Jetson TX2 est donc la deuxième carte en termes de temps de traitement et d'efficacité énergétique pour le F16x2 mais est la première carte en efficacité pour le F32x2. Pour ce qui est du F16x2, on trouve des résultats temporels $3,7\times$ plus lents pour une consommation $1,3\times$ supérieure que la Jetson Xavier.

La Jetson Nano est la carte la plus lente et la plus gourmande en énergie pour nos implémentations de Horn-Schunck. Par rapport à la Jetson Xavier, pour la version F16x2 la plus rapide, la Nano est $11,1\times$ plus lente et consomme $2,4\times$ plus d'énergie.

5.2.2.3 Synthèse

Dans cette sous-section, nous avons examiné les performances énergétiques des implémentations de TV-L^1 et Horn-Schunck sur les cartes Jetson visées. Dans l'ensemble, la Jetson Xavier se distingue en termes de temps d'exécution et de consommation énergétique, obtenant les meilleures performances parmi les cartes testées. Nos optimisations permettent à TV-L^1 d'être $7,4\times$ plus rapide et $6,4\times$ plus économe en énergie sur GPU que la meilleure version CPU Neon F16. Elle est également $5,2\times$ plus rapide et $4,9\times$ plus économe en énergie que la version OpenCV F32 GPU Pour Horn-Schunck, la Jetson Xavier est la carte la plus rapide pour les versions F16x2 et F32x2, mais n'est pas la plus efficace dans tous les cas. La Jetson TX2 se classe deuxième en termes de temps de traitement et d'efficacité énergétique pour les versions F16x2, mais est la plus efficace pour les versions F32x2. La Jetson Nano est la carte la moins performante. Elle est la carte la plus lente et la plus gourmande en énergie pour les implémentations de nos deux algorithmes.

5.2.3 Qualité du flot optique

Nous avons étudié jusqu'à présent des paramètres quantitatifs pour classer nos différentes versions en termes de vitesse de traitement et de consommation énergétique sur des images vides. Dans cette section, nous évaluons expérimentalement nos meilleures implémentations simple-précision F_{32} et demi-précision F_{16} sur les collections d'images pour flot optique Middlebury [Bak+11] et MPI Sintel [But+12]. Le but est de déterminer l'impact qu'a la baisse de dynamique sur la qualité du flot produit. Il s'agit de la seule optimisation pouvant modifier la qualité du flot.

Pour ces évaluations, nous avons utilisé la carte Jetson Xavier. La configuration multi-échelle pour TV-L[1] et Horn-Schunck utilisée pour les évaluations de référence est une pyramide à 3 échelles pour les ensembles de données Middlebury et une pyramide à 5 échelles pour les ensembles de données MPI Sintel. Chaque niveau de pyramide exécute le même nombre d'itérations. Nous faisons varier le nombre d'itérations jusqu'à 100 afin de générer un ensemble de points de temps et de précision. Nous comparons le flot optique calculé avec une vérité terrain fournie en erreur moyenne point à point (AEPE) et en erreur angulaire moyenne (AAE).

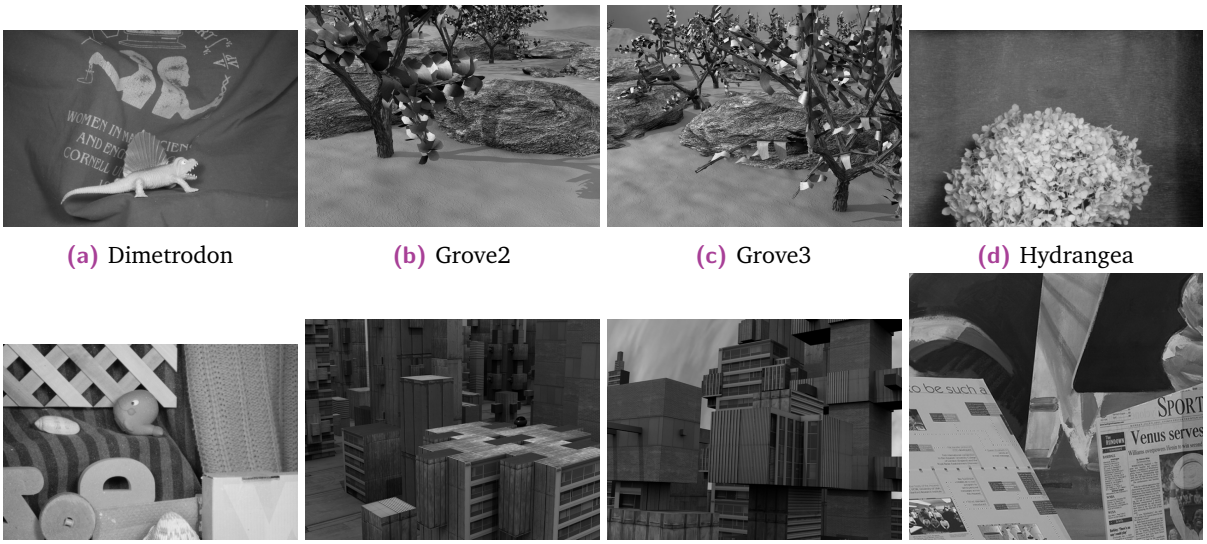
On calcule l'AEPE entre un flot vérité terrain $(u_1^{\text{gt}}, u_2^{\text{gt}})$ et un flot calculé (u_1, u_2) de dimensions $w \times h$ de la manière suivante :

$$\text{AEPE} = \frac{1}{w \cdot h} \sum_{i=0}^{w-1} \sum_{j=0}^{h-1} \sqrt{(u_1^{\text{gt}}(i,j) - u_1(i,j))^2 + (u_2^{\text{gt}}(i,j) - u_2(i,j))^2}. \tag{5.4}$$

L'AAE est calculée de la manière suivante :

$$\text{AAE} = \frac{1}{w \times h} \sum_{i=0}^{w-1} \sum_{j=0}^{h-1} \cos^{-1}\left(\frac{u_1^{\text{gt}}(i,j) \cdot u_1(i,j) + u_2^{\text{gt}}(i,j) \cdot u_2(i,j) + 1}{\sqrt{u_1^{\text{gt}}(i,j)^2 + u_2^{\text{gt}}(i,j)^2 + 1} \cdot \sqrt{u_1(i,j)^2 + u_2(i,j)^2 + 1}} \right). \tag{5.5}$$

Le flot optique est obtenu à partir des images représentées en figures 5.20 et 5.21. Nous représentons ici des images de toutes les séquences de la base de données Middlebury mais seulement 3 images de la base MPI-Sintel. L'encodage du flot se fait grâce à la roue chromatique présentée en figure 5.22. Il s'agit de la représentation accompagnant la base de donnée Middlebury.



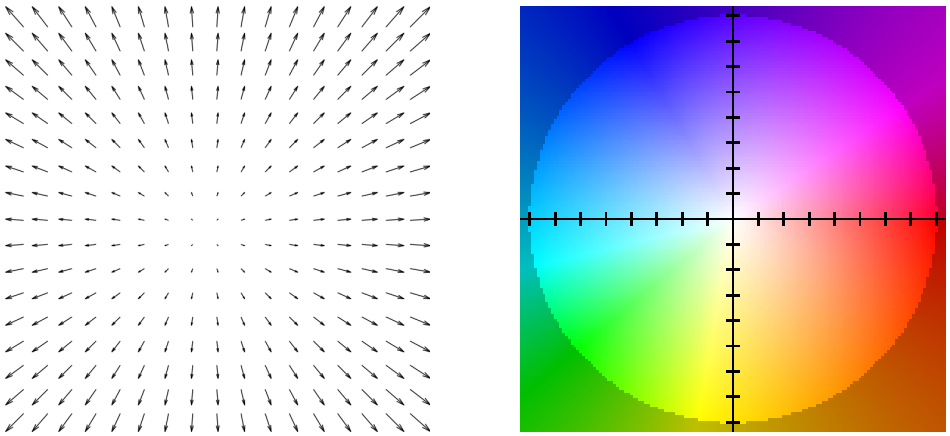
(a) Dimetrodon (b) Grove2 (c) Grove3 (d) Hydrangea

(e) RubberWhale (f) Urban2 (g) Urban3 (h) Venus

Fig. 5.20 : Images du jeu de données Middlebury.

(a) alley_2 (b) sleeping_1 (c) cave_4

Fig. 5.21 : Exemples d'images du jeu de données MPI Sintel.



(a) Champ de vecteurs. (b) Roue chromatique.

Fig. 5.22 : Relation entre le champ de vecteurs du flot optique et le schéma de couleur de la roue chromatique utilisée.

5.2.3.1 TV-L^1

Jeu de données Middlebury Le jeu de données de Middlebury consiste en 8 paires d'images en niveaux de gris de différentes résolutions (3 séquences de résolution 584×388 pixels, 4 séquences de résolution 640×480 pixels et 1 séquence de résolution 420×380 pixels).

La figure 5.23 montre l'impact de la précision des nombres demi-précision F16x2 sur la précision du flot calculé. Tout d'abord, les deux versions convergent vers la même qualité de flot au bout d'un certain temps (entre 1,3 et 1,4 pixel d'erreur moyenne point à point et 7,2 et 7,9 degrés d'erreur angulaire moyenne). Deuxièmement, pour une même durée de calcul, la version F16x2 a une erreur moyenne plus faible que la version F32x2. Comme les versions F16x2 sont plus rapides que les versions F32x2, il est possible de calculer plus d'itérations dans le même laps de temps.

Dataset	2,5 ms				5 ms				10 ms			
	AEPE		AAE		AEPE		AAE		AEPE		AAE	
	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}
Dimetrodon	0,91	**0,55**	19,60	**10,74**	0,29	**0,25**	5,47	**4,52**	0,20	**0,19**	3,43	**3,36**
Grove2	1,52	**0,74**	24,54	**10,28**	0,40	**0,26**	5,74	**3,78**	**0,22**	0,24	**3,33**	3,45
Grove3	2,47	**1,71**	27,15	**15,99**	1,34	**1,10**	11,71	**9,73**	1,01	**0,98**	8,88	**8,74**
Hydrangea	1,69	**0,93**	17,88	**7,78**	0,37	**0,33**	3,52	**3,35**	**0,30**	0,32	**2,92**	3,13
RubberWhale	0,42	**0,34**	13,39	**10,84**	0,28	**0,27**	9,05	**8,82**	**0,24**	0,25	**7,74**	7,87
Urban2	7,44	**6,77**	40,04	**28,32**	6,38	**5,92**	22,36	**17,23**	5,59	**5,30**	14,12	**12,08**
Urban3	6,34	**5,58**	50,08	**35,48**	5,03	**4,33**	27,70	**20,65**	3,95	**3,53**	17,89	**15,62**
Venus	1,86	**1,42**	21,51	**15,88**	0,89	**0,72**	10,77	**9,68**	0,52	**0,52**	**8,05**	8,12
Moyenne des datasets	2,83	**2,25**	26,77	**16,91**	1,87	**1,65**	12,04	**9,72**	1,50	**1,42**	8,30	**7,80**

Tab. 5.4 : AEPE et AAE pour nos implémentations TV-L^1 comparées à la vérité terrain sur Jetson Xavier à 2,5, 5 et 10 ms de temps d'exécution pour les séquences Middlebury.

Le tableau 5.4 détaille l'erreur point à point et l'erreur angulaire pour les durées d'exécution de 2,5 ms, 5 ms et 10 ms. Pour presque toutes les séquences, le F16x2 est toujours meilleur pour les deux mesures d'erreur pour chaque durée. À 2,5 ms de temps d'exécution, la version F16x2 a une erreur point à point en moyenne 20% inférieure et une erreur angulaire en moyenne 37% inférieure que la version F32x2. À 5 ms, la version F16x2 a une erreur point à point en moyenne 12% inférieure et une erreur angulaire inférieure de 19% en moyenne. Pour certaines séquences telles que « Grove 2 » ou « Hydrangea », l'erreur est encore plus faible et atteint 50% à 2,5 ms et 35% à 5 ms.

Dans l'ensemble, l'utilisation de nombres demi-précisions F16x2 sur le jeu de données de Middlebury est bénéfique et permet à TV-L^1 d'effectuer plus d'itérations

dans le même laps de temps que l'implémentation en simple précision F32x2. La baisse de la dynamique et de la précision des nombres F16x2 est compensée et le flot calculé est aussi précis, voire plus, que lorsque l'on utilise les nombres F32x2.

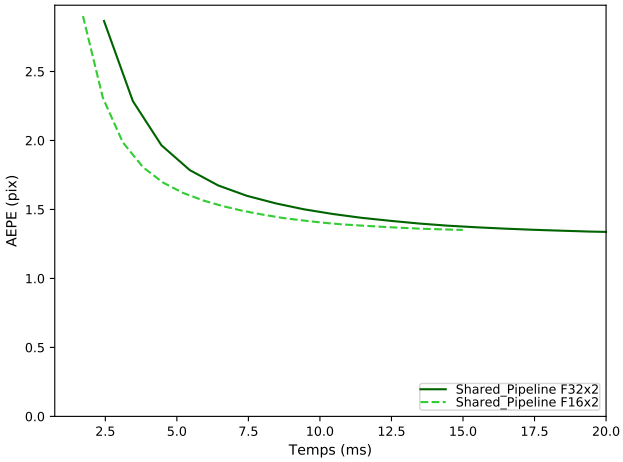
Jeu de données MPI Sintel Ce jeu de données est composé de 23 séquences avec vérité terrain associée et de 12 autres séquences de test. Ces séquences contiennent entre 20 et 50 images d'une résolution de 1024×436 pixels. Le flot calculé est comparé au flot de référence fourni pour chaque paire d'images consécutives de chaque séquence.

La figure 5.5 montre l'impact de la précision F16x2 sur la précision du flot calculé. Comme pour la base de données Middlebury, les versions F16x2 ont une erreur moyenne plus faible que les versions F32x2 pour les deux métriques considérées. Ce phénomène est moins prononcé car les mouvements sont plus grands et plus complexes et l'ensemble de données contient environ $3\times$ le nombre de séquences et plus de $10\times$ le nombre d'images par séquence.

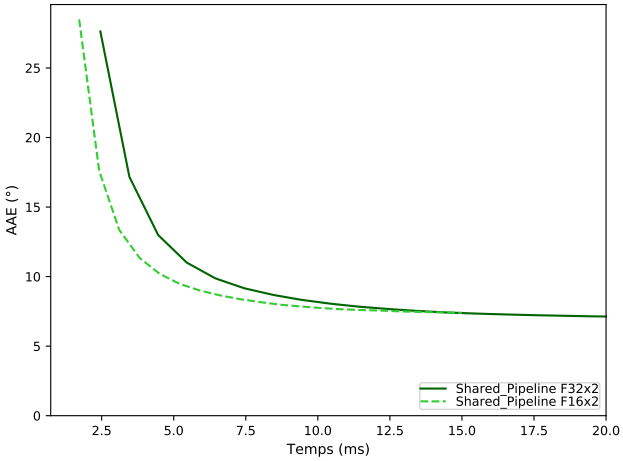
Ainsi, le tableau 5.5 se concentre sur les 3 séquences présentant les erreurs moyennes les plus faibles pour les durées d'exécution de 2,5 ms, 5 ms et 10 ms. Pour ces 3 séquences, la version F16x2 a une erreur point à point et une erreur angulaire inférieure d'environ 30% par rapport à la version F32x2. C'est le cas pour des durées allant jusqu'à 5 ms. Passé le point de convergence à environ 10 ms, les versions F16x2 et F32x2 présentent des erreurs similaires. En moyenne, la version F16x2 présente une erreur point à point moyenne inférieure de 7% et une erreur angulaire inférieure de 15% pour les temps d'exécution de 2,5 ms et 5 ms. Certaines séquences présentent des mouvements plus importants et plus complexes, faisant augmenter les deux mesures d'erreur en moyenne pour l'ensemble du jeu de données de MPI Sintel.

Dataset	2,5 ms				5 ms				10 ms			
	AEPE		AAE		AEPE		AAE		AEPE		AAE	
	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}
alley_2	4,48	**3,07**	25,65	**18,37**	1,44	**0,94**	9,00	**6,27**	0,79	**0,79**	**5,16**	5,16
sleeping_1	1,48	**1,05**	19,09	**13,63**	0,54	**0,38**	7,07	**4,95**	0,28	**0,27**	3,66	**3,52**
cave_4	15,39	**12,73**	51,80	**40,25**	9,57	**7,92**	25,77	**19,79**	7,07	**6,84**	17,39	**16,95**
Moyenne des datasets	14,99	**13,91**	37,38	**31,85**	12,33	**11,47**	22,74	**19,37**	10,65	**10,35**	16,38	**15,99**

Tab. 5.5 : AEPE et AAE pour nos implémentations TV-L[1] comparées à la vérité terrain sur Jetson Xavier à 2,5, 5 et 10 ms de temps d'exécution pour les séquences MPI Sintel.

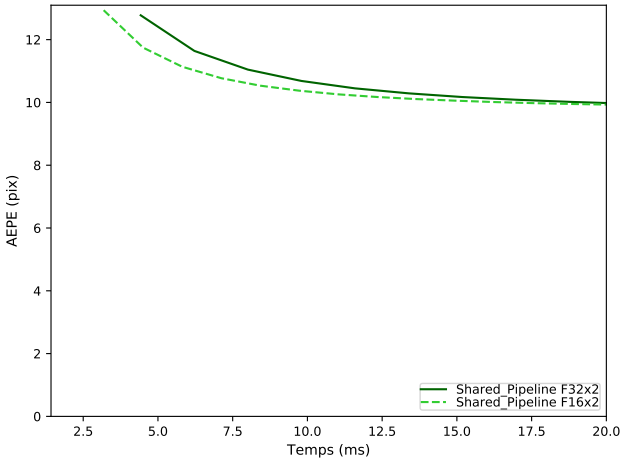


(a) TV-L^1 Middlebury AEPE

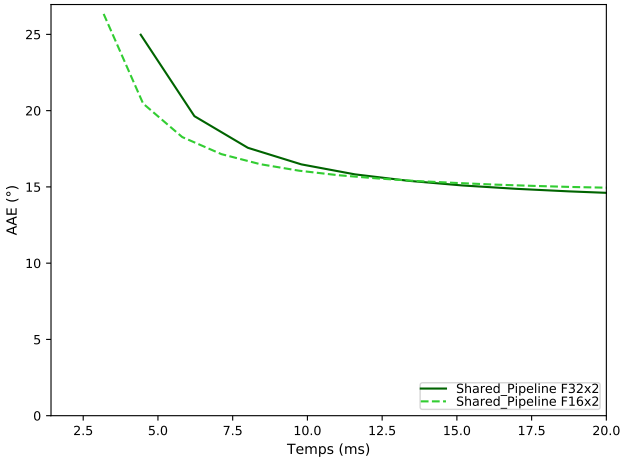


(b) TV-L^1 Middlebury AAE

Fig. 5.23 : Erreur point à point moyenne (AEPE) et erreur angulaire moyenne (AAE) obtenues pour TV-L^1 pour le jeu de données Middlebury en fonction du temps d'exécution.



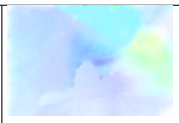
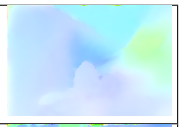
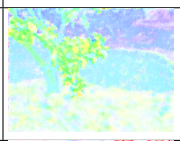
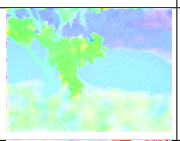
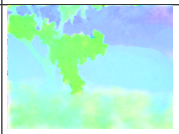
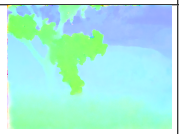
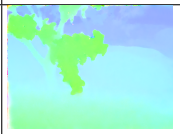
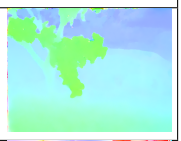
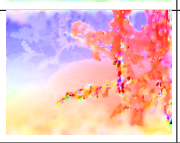
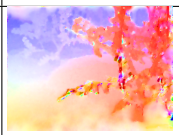
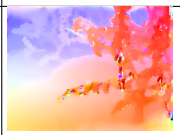
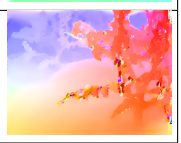
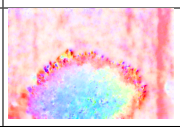
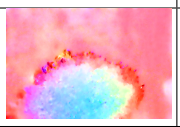
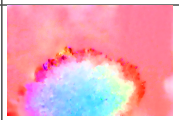
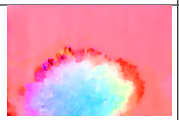
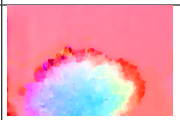
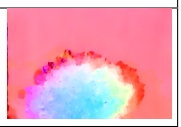
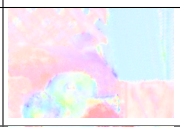
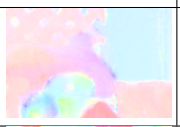
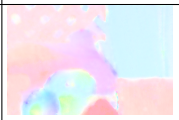
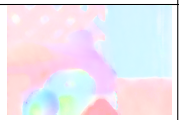
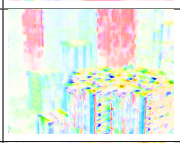
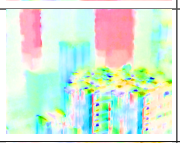
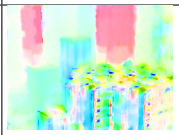
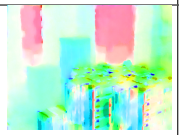
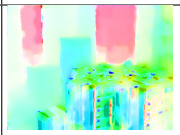
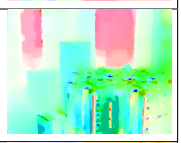
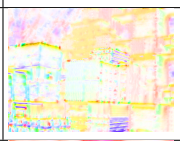
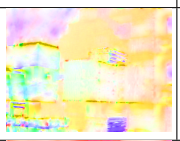
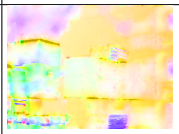
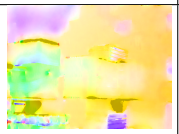
(a) TV-L^1 MPI Sintel AEPE



(b) TV-L^1 MPI Sintel AAE

Fig. 5.24 : Erreur point à point moyenne (AEPE) et erreur angulaire moyenne (AAE) obtenues pour TV-L^1 pour le jeu de données MPI Sintel en fonction du temps d'exécution.

Nous représentons dans le tableau 5.6 les flots optiques issus de TV-L^1 pour les jeux de données Middlebury et MPI Sintel pour les temps d'exécutions 2,5 ms, 5 ms et 10 ms sur Jetson Xavier.

Séquence	2,5 ms		5 ms		10 ms	
	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}
Dimetrodon						
Grove2						
Grove3						
Hydrangea						
RubberWhale						
Urban2						
Urban3						
Venus						
alley_2						
sleeping_1						
cave_4						

Tab. 5.6 : Représentation des estimations du flot optique par TV-L^1 pour les images de la base de donnée Middlebury et pour 3 images de la base de données MPI Sintel.

5.2.3.2 Horn-Schunck

Jeu de données Middlebury La figure 5.25 présente cette même comparaison entre les nombres demi-précision F16x2 et les nombres simple-précision F32x2 sur la précision du flot calculé pour l'algorithme de Horn-Schunck pour le jeu de données de Middlebury. Premièrement, on remarque qu'en 20 ms de temps d'exécution, les deux versions F16x2 et F32x2 de Horn-Schunck n'atteignent pas la même valeur. La version F16x2 atteint la valeur de 1,57 pour l'AEPE et 11,76 pour l'AAE alors que la version F32x2 attend les valeurs respectives de 2,23 et 20,10. Ce n'était pas le cas pour TV-L^1 : les deux versions convergeaient vers les mêmes valeurs avant la durée maximale de temps d'exécution de 20 ms. Ainsi et là encore, pour un temps de calcul équivalent, la version F16x2 a une erreur moyenne plus faible que la version F32x2 : on peut exécuter plus d'itérations pendant la même durée. Cet effet est notablement plus prononcé pour nos implémentations de Horn-Schunck : le passage du F32x2 vers F16x2 permet une diminution du temps d'exécution d'un facteur $\times 3,5$ comme vu dans la première section du chapitre.

Dataset	2,5 ms				5 ms				10 ms			
	AEPE		AAE		AEPE		AAE		AEPE		AAE	
	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}
Dimetrodon	1,57	**1,06**	43,47	**26,25**	1,27	**0,76**	32,93	**17,26**	0,97	**0,50**	23,54	**10,07**
Grove2	2,08	**1,19**	39,87	**19,54**	1,41	**0,78**	25,23	**10,54**	1,03	**0,47**	16,02	**6,14**
Grove3	2,72	**1,85**	35,81	**18,62**	1,99	**1,49**	22,01	**13,16**	1,62	**1,25**	15,53	**10,94**
Hydrangea	2,49	**1,55**	38,70	**16,24**	1,93	**0,85**	24,13	**7,50**	1,33	**0,50**	12,86	**5,05**
RubberWhale	0,81	**0,50**	28,38	**16,32**	0,58	**0,40**	19,60	**12,57**	0,45	**0,34**	14,56	**10,89**
Urban2	7,68	**6,29**	49,90	**28,90**	6,79	**5,31**	35,07	**20,04**	5,90	**4,83**	25,05	**16,96**
Urban3	6,51	**5,45**	57,93	**38,97**	5,82	**4,62**	45,10	**27,83**	5,14	**3,91**	34,80	**21,87**
Venus	2,54	**1,57**	36,53	**20,22**	1,90	**1,11**	24,99	**14,54**	1,40	**0,97**	17,97	**13,14**
Moyenne des datasets	3,29	**2,42**	41,33	**23,14**	2,71	**1,91**	28,78	**15,42**	2,23	**1,57**	20,10	**11,76**

Tab. 5.7 : Erreurs AEPE et AAE pour nos implémentations de Horn-Schunck comparées à la vérité terrain sur Jetson Xavier à 2,5, 5 et 10 ms de temps d'exécution pour les séquences Middlebury.

Le tableau 5.7 détaille l'erreur point à point et l'erreur angulaire pour les durées d'exécution de 2,5 ms, 5 ms et 10 ms. Comme pour TV-L^1, il y a ici un intérêt réel à utiliser des nombres demi-précision. Par rapport à la version simple précision, nous avons en moyenne pour la version demi-précision :

— une AEPE inférieure de 36% et une AAE inférieure de 79% pour un temps d'exécution de 2,5 ms,

— une AEPE inférieure de 42% et une AAE inférieure de 87% pour un temps d'exécution de 5 ms,

— une AEPE inférieure de 42% et une AAE inférieure de 71% pour un temps d'exécution de 10 ms.

Peur importe la séquence calculée, la version F16x2 présente une erreur toujours inférieure quelle que soit la durée de calcul.

Jeu de données MPI Sintel La figure 5.8 montre l'impact de la précision F16x2 sur la précision du flot calculé. Là encore, ce jeu de données est beaucoup plus grand et beaucoup plus complexe que celui de Middlebury. On observe ainsi que les deux courbes convergent vers une valeur beaucoup plus proche, mais que la version F16x2 reste légèrement en dessous pour des temps d'exécution jusqu'à 20 ms.

Dataset	2,5 ms				5 ms				10 ms			
	AEPE		AAE		AEPE		AAE		AEPE		AAE	
	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}
alley_2	5,75	**2,41**	51,17	**14,43**	2,84	**1,30**	18,01	**8,22**	1,56	**1,01**	9,63	**6,61**
sleeping_1	2,90	**1,25**	50,02	**15,08**	1,44	**0,65**	18,23	**7,62**	0,80	**0,48**	9,39	**5,95**
cave_4	14,83	**11,24**	68,86	**34,41**	11,80	**8,78**	38,27	**23,92**	9,70	**7,71**	27,84	**20,06**
Moyenne des datasets	15,17	**13,14**	51,88	**27,95**	13,23	**11,78**	29,41	**21,95**	12,06	**11,15**	23,04	**19,91**

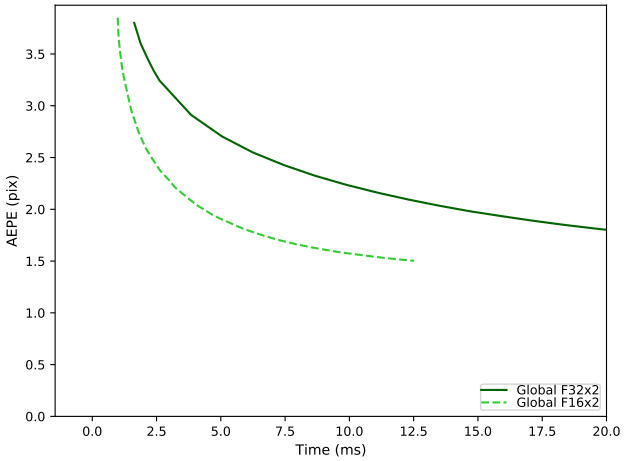
Tab. 5.8 : Erreurs AEPE et AAE pour nos implémentations de Horn-Schunck comparées à la vérité terrain sur Jetson Xavier à 2,5, 5 et 10 ms de temps d'exécution pour les séquences MPI Sintel.

On se concentre à nouveau sur les erreurs moyennes pour des durées d'exécution de 2,5 ms, 5 ms et 10 ms dans le tableau 5.5. Comme pour Middlebury, peu importe la séquence ou la métrique, à durée d'exécution identique, la version F16x2 demi-précision a une erreur inférieure que la version F32x2. Par rapport à la version simple précision, nous avons en moyenne pour la version demi-précision :

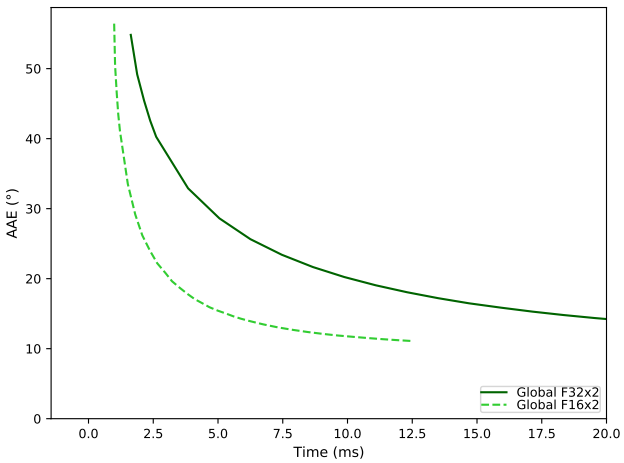
— une AEPE inférieure de 16% et une AAE inférieure de 86% pour un temps d'exécution de 2,5 ms,

— une AEPE inférieure de 12% et une AAE inférieure de 34% pour un temps d'exécution de 5 ms,

— une AEPE inférieure de 8% et une AAE inférieure de 16% pour un temps d'exécution de 10 ms.

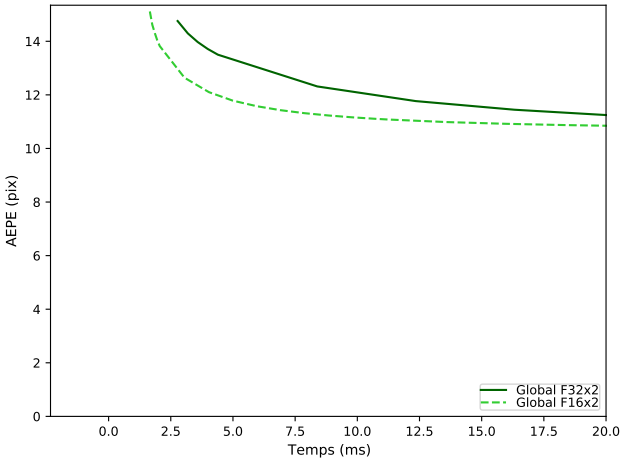


(a) Horn-Schunck Middlebury AEPE

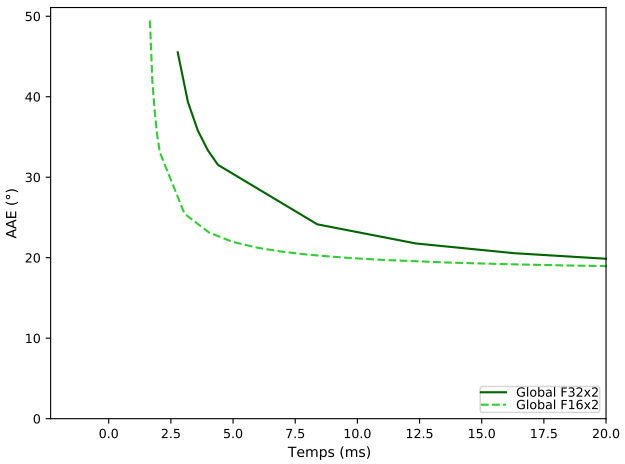


(b) Horn-Schunck Middlebury AAE

Fig. 5.25 : Erreur point à point moyenne (AEPE) et erreur angulaire moyenne (AAE) obtenues pour Horn-Schunck pour le jeu de données Middlebury en fonction du temps d'exécution.



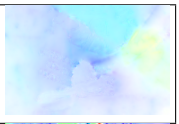
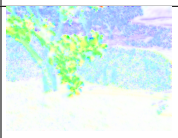
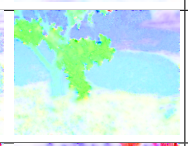
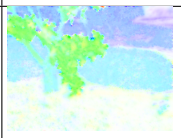
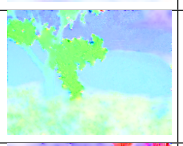
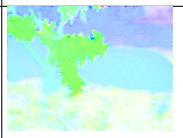
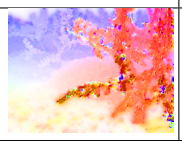
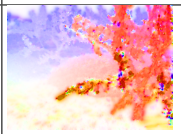
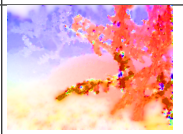
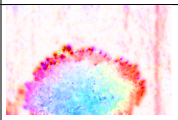
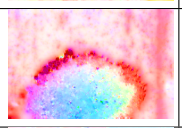
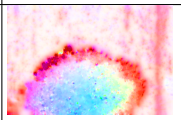
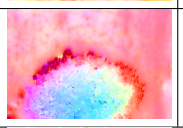
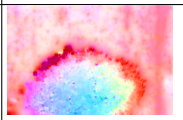
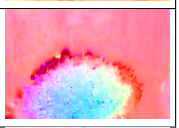
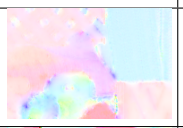
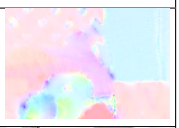
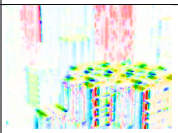
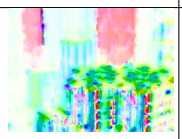
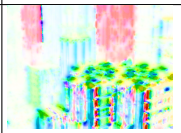
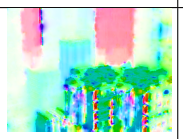
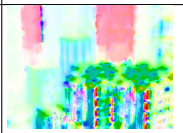
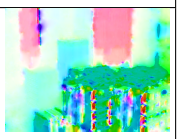
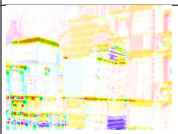
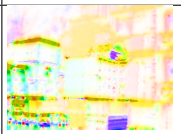
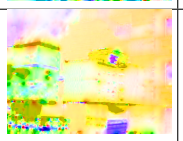
(a) Horn-Schunck MPI Sintel AEPE



(b) Horn-Schunck MPI Sintel AAE

Fig. 5.26 : Erreur point à point moyenne (AEPE) et erreur angulaire moyenne (AAE) obtenues pour Horn-Schunck pour le jeu de données MPI Sintel en fonction du temps d'exécution.

Nous représentons dans le tableau 5.9 les flots optiques issus de Horn-Schunck pour les jeux de données Middlebury et MPI Sintel pour les temps d'exécutions 2,5 ms, 5 ms et 10 ms sur Jetson Xavier.

Séquence	2,5 ms		5 ms		10 ms	
	F_{32}	F_{16}	F_{32}	F_{16}	F_{32}	F_{16}
Dimetrodon						
Grove2						
Grove3						
Hydrangea						
RubberWhale						
Urban2						
Urban3						
Venus						
alley_2						
sleeping_1						
cave_4						

Tab. 5.9 : Représentation des estimations du flot optique par Horn-Schunck pour les images de la base de donnée Middlebury et pour 3 images de la base de données MPI Sintel.

5.2.3.3 Synthèse

Nous avons vu dans cette section l'impact de la précision des nombres sur la qualité du flot optique calculé. Les résultats montrent que pour l'algorithme TV-L^1, la version F16x2 converge plus rapidement vers la même qualité de flot que la version F32x2. Pour le jeu de données Middlebury, à des temps d'exécution de 2,5 ms, 5 ms et 10 ms, la version F16x2 présente des erreurs point à point et angulaires inférieures que la version F32x2. Des améliorations similaires sont observées sur le jeu de données MPI Sintel. Pour l'algorithme de Horn-Schunck, les erreurs de la version F16x2 convergent également plus rapidement, mais vers une valeur inférieure à celle de la version F32x2. Cette observation est particulièrement visible pour le jeu de données Middlebury et également présente pour le jeu de données MPI Sintel. L'utilisation de nombres en demi-précision permet d'obtenir des résultats de flot optique de qualité similaire, voire supérieure par rapport aux nombres en simple précision. Cette diminution de la dynamique de calcul entraîne une diminution du temps d'exécution permettant ainsi d'exécuter plus d'itérations de nos algorithmes dans le même laps de temps.

5.3 Conclusion

Dans ce chapitre, nous avons présenté de nombreux résultats cherchant à qualifier et évaluer nos différentes optimisations. Ces évaluations sont scindées en deux groupes. Le premier groupe de tests cherchent à déterminer les paramètres d'exécution optimaux pour chacun des kernels CUDA dans nos implémentations.

La première partie de ce groupe de tests exploratoire nous a ainsi permis dégager un bon compromis de taille de bloc pour les kernels de nos versions mono-itératives. Notre exploration a ainsi principalement visé nos implémentations Global et Shared_Fusion déclinées en F32, F16, F32x2, F16x2. Pour TV-L^1, d'autres versions ont également été testées. Nous avons ainsi choisi les tailles de blocs 64×6 et 64×8 selon l'architecture. Avec ces dimensions, nous obtenons des temps d'exécution de kernels éloigné au maximum de 15% du minimum pour TV-L^1 et au maximum de 8% du minimum pour Horn-Schunck. Il s'agit d'un compromis. Une gestion plus fine des tailles de blocs pour chaque kernel permettrait un gain de performances pour nos versions les plus lentes. Les paramètres des versions les plus rapides sont en effet optimisées individuellement dans la deuxième partie de ce groupe de tests.

La deuxième partie de notre exploration de paramètres vise ainsi les versions multi-itératives. Selon l'implémentation, nous avons dû déterminer une bonne taille de bloc et un bon nombre de blocs dans la grille (pour les implémentations Pipeline) ou bien juste une bonne taille de bloc (pour les versions MltIteration). Cela doit être fait pour une taille d'image et pour un nombre d'itérations données. Nous avons ainsi pu dresser une bibliothèque interne de paramètres pour les conditions suivantes :

— pour chacune de nos plateformes embarquées (Jetson Xavier, Jetson TX2, Jetson Nano),
— pour chacune de nos versions (Global_Pipeline, Shared_Pipeline, Global_MltIteration, Shared_MltIteration),
— pour chaque type de nombre utilisé (F32, F16, F32x2, F16x2),
— pour une taille d'images fixe (256×256, 512×512, 1024×1024, 2048×2048),
— et pour un nombre d'itérations fixe (jusqu'à 13 ou 15 dans la plupart des cas).

Nous avons ainsi pu déjà dégager un premier résultat concernant les performances des types vectoriels F32x2 et F16x2 par rapport aux types scalaires F32 et F16. Cela est dû à une meilleure utilisation de la bande passante grâce à la mutualisation des requêtes mémoires et dans le cas des F16x2, au traitement parallèle de deux nombres à la fois. Deuxièmement, nous avons pu déterminer les configurations optimales pour chacune de nos versions. Ces configurations nous serviront pour le deuxième groupe de tests et d'évaluations.

La deuxième partie de ce groupe de tests et d'évaluations cherche à déterminer les performances de nos implémentations. Nous avons réalisé 3 études concernant le temps d'exécution, la consommation énergétique et la qualité des résultats produits.

La première étude nous a permis d'évaluer la réduction du temps d'exécution apportée par nos optimisations. Il s'agit ici de déterminer la tendance de nos implémentations en fonction de la taille d'images sur chacune des plateformes Jetson. Nous avons ainsi pu dégager de nos résultats les bonnes performances des versions multi-itératives Shared_MltIteration et Shared_Pipeline sur les Jetsons pour l'algorithme TV-L^1. Grâce à nos optimisations, la version Shared_MltIteration atteint 0,21 ns/iter/pix en F16x2 et 0,31 ns/iter/pix en F32x2 sur la Jetson Xavier. Sur la Jetson TX2, cette version atteint 0,84 ns/iter/pix en F16x2 et 1,20 ns/iter/pix en F32x2. Sur la Jetson Nano, la version Shared_Pipeline atteint 2,28 ns/iter/pix en F16x2 et 3,14 ns/iter/pix en F32x2. Pour Horn-Schunck, les résultats sont différents. Cet algorithme est moins complexe que TV-L^1 et nécessite moins d'accès mémoires. Sur la Jetson Xavier, l'implémentation Shared_MltIteration est la plus rapide F16x2 et atteint 0,61 ns/iter/pix. La version Global est légèrement plus lente en F16x2. En F32x2 la version

Global est la plus rapide et atteint 0,12 ns/iter/pix. On remarque un bien meilleur gain en passant de F32x2 vers F16x2, cela étant dû au code moins intensif en mémoire et en calcul qui sature moins les unités de calculs et la bande passante en F16x2. On remarque la même chose sur la Jetson TX2 : la version Shared_MltIteration est la plus rapide en F16x2 et atteint 0,42 ns/iter/pix et la version Global est la plus rapide en F32x2 et atteint 1,28 ns/iter/pix. Enfin, sur la Jetson Nano, on remarque que la version la plus rapide des implémentations en nombre simple précision est la version Global_F32 scalaire. Le surcoût lié à l'utilisation du type vectoriel F32x2 (mélanges et décalages pour obtenir des vecteurs prêts à une exécution vectoriel) n'est pas compensé sur cette carte, carte la moins puissante des 3 Jetson visées. Nous obtenons 1,30 ns/iter/pix en F16x2 (version Shared_MltIteration) et 3,41 ns/iter/pix en F32 (version Global).

La deuxième étude concerne la consommation énergétique de nos meilleures versions sur chaque carte Jetson. On s'intéresse ici à la consommation en nanojoules par pixel. Lorsque des versions CPU sont disponibles ou que d'autres implémentations sont disponibles sur GPU, nous les ajoutons à notre comparaison afin de se positionner. Nous avons également fait varier la fréquence horloge afin de trouver les meilleurs compromis en consommation énergétique et en temps d'exécution. Nous avons ainsi pu générer deux nuages de points dans l'espace Temps/Énergie, un pour TV-L^1 et un pour Horn-Schunck.

Pour TV-L^1 et pour Horn-Schunck, bien qu'étant la carte la plus puissante, la Jetson Xavier est également la plus efficace et également la plus rapide en F16x2. En F32x2, elle est la plus rapide pour TV-L^1 mais pas pour Horn-Schunck. Là encore, la charge de calcul et l'intensité des transactions mémoires de Horn-Schunck ne permettent pas une utilisation efficace du GPU. Baisser la dynamique de calcul permet cependant de libérer suffisamment de ressources pour permettre une meilleure allocation des blocs de threads sur le GPU. La Jetson TX2 et la Jetson Nano suivent ensuite en termes d'efficacité énergétique et temporelle. On notera également une meilleure efficacité de nos versions GPU par rapport à des implémentations optimisées Neon de TV-L^1 sur CPU.

Finalement, notre troisième étude a concerné la qualité du flot produit. Les résultats présentés dans cette section démontrent l'impact de la précision des nombres sur la qualité et la vitesse de calcul du flot optique. Dans le cas de l'algorithme TV-L^1, la version F16x2 converge plus rapidement vers un niveau de qualité équivalent à celui de la version F32x2 à la fois pour les jeux de données Middlebury et MPI Sintel. En conséquence, pour le même temps d'exécution, la version demi-précision F16x2 présente une meilleure qualité de flot que la version simple-précision F32x2. Il y va

de même pour l'algorithme Horn-Schunck où la version F16x2 converge même vers des valeurs d'erreurs inférieures à la version F32x2, particulièrement pour le jeu de données de Middlebury et plus modérement pour le jeu de données MPI Sintel. Cette étude souligne l'importance de la prise en compte de la précision des nombres dans le choix et l'optimisation des algorithmes de flot optique. Une baisse de dynamique permet contre-intuitivement de produire des flots de meilleure qualité grâce à la réalisation de plus d'itérations dans le même laps de temps.

Nous dressons le tableau synthétique 5.10 afin de résumer nos résultats temporels et énergétiques pour nos meilleures versions déclinées en simple et demi-précision.

	5 ms		10 ms		20 ms		40 ms	
Implémentation	Résolution maximale	W	Résolution maximale	W	Résolution maximale	W	Résolution maximale	W
OpenCV TV-L^1 GPU	320×320	23.8	616×616	24.9	1000×1000	26.2	1484×1484	27.8
Best GPU TV-L^1 F_{32}	792×792	23.3	1144×1144	25.9	1648×1648	28.2	2320×2320	29.9
Best GPU TV-L^1 F_{16}	**960 × 960**	25.1	**1440 × 1440**	31.1	**2048 × 2048**	35.6	**2880 × 2880**	38.6
Best GPU HS F_{32}	584×584	22.5	896×896	23.9	1280×1280	27.0	1856×1856	28.4
Best GPU HS F_{16}	**1216 × 1216**	26.1	**1792 × 1792**	31.3	**2576 × 2576**	35.1	**3616 × 3616**	40.0

Tab. 5.10 : Résolutions maximales atteignables en 5, 10, 20 et 40 ms de temps d'exécution pour TV-L^1 et Horn-Schunck sur Jetson Xavier. La configuration multi-échelle utilisée comporte 3 niveaux de pyramide, 1 warp par niveau et 10 itérations par niveau.

6 Conclusion, limitations et travaux futurs

6.1 Synthèse

Les travaux présentés dans cette thèse visent à détailler les processus d'optimisation algorithmiques et d'implémentations pour deux méthodes d'estimation du flot optique pour des plateformes GPU embarqués. Bien que l'on vise ici les algorithmes TV-L[1] et Horn-Schunck d'estimation du flot optique, ces optimisations peuvent s'appliquer à tout autre algorithme itératif de type stencil. Plusieurs niveaux et types d'optimisations ont été présentés et couvrent les transformations algorithmiques haut niveau, les transformations bas niveau propre aux architectures GPU visées, mais aussi le choix du type de nombres utilisés pour les calculs, à savoir la dynamique du type, mais aussi la cardinalité pour des types vectoriels.

Nous avons ensuite évalué ces transformations de manière quantitative en temps et en énergie, mais aussi de manière qualitative en évaluant nos implémentations en termes d'erreur point à point moyenne et en termes d'erreur angulaire moyenne. Ces évaluations nous ont ainsi permis de répondre aux interrogations posées au début de cette thèse.

Quel est le gain apporté par une utilisation efficace du GPU par rapport au CPU ?
Comme on l'a pu voir dans les sections 5.2.1 et 5.2.2, il y a un réel avantage à utiliser le GPU pour estimer le flot optique. Nous avons commencé par décrire en chapitre 3 les principales différences architecturales entre les CPU et les GPU embarqués utilisés dans nos travaux. Les principales différences concernent la quantité de cœurs de calculs et la quantité de mémoires cache et partagées disponibles. Nous avons ensuite décrit les transformations et les optimisations de code haut et bas niveaux propres ou non à ces architectures. Ce chapitre descriptif nous ainsi permis de comprendre et de décrire les implémentations plus ou moins optimisés des algorithmes TV-L[1] et Horn-Schunck en chapitre 4. Le but est ici d'identifier comment se comportent certaines transformations de codes sur les architectures GPU embarquées visées. Comme on l'a vu, un travail d'optimisations des paramètres intrinsèques à nos

versions et nécessaire afin d'en tirer les meilleures performances et ainsi pouvoir réaliser de bonnes comparaisons entre versions. La première partie du chapitre 5 concerne ce travail d'optimisations de paramètres. La seconde concerne les résultats comparatifs de nos versions en termes de temps d'exécution, de consommation énergétique et de qualité de flot optique produit. Nous avons ainsi pu déterminer sur chaque carte quelle est la meilleure version en temps de calcul et en consommation énergétique. Il n'existe pas une seule et unique meilleure version pour chaque carte et pour chaque taille d'images. Pour obtenir les meilleures performances possibles, il est nécessaire d'ajuster la version GPU de nos algorithmes de flot optique utilisée en fonction de la taille d'image traitée en pixels et du GPU embarqué utilisé. Nous avons ainsi pu réaliser la comparaison entre nos implémentations optimisées sur CPU avec celle sur GPU.

Sur les cartes Jetson, l'utilisation du GPU embarqué permet de réduire le temps de traitement uniquement lorsque la taille de l'image est suffisamment grande. Il est nécessaire de surmonter les coûts de lancement des fonctions kernels CUDA, les coûts de synchronisation des données, et d'avoir suffisamment de données à traiter sur tous les cœurs du GPU afin de solliciter pleinement ses capacités de calcul et de mémoire. Ces tailles d'images se situent entre 140×140 et 512×512 pixels, selon l'algorithme choisi et la plateforme de calcul utilisée. Lorsque ces tailles d'images sont dépassées, le GPU devient plus efficace tandis que le CPU perd en performance. Nous avons observé des gains allant de $\times 1,62$ à $\times 7,22$. Il convient de noter ici les limites des gains observés sur le GPU. Pour le cas de Horn-Schunck F32, la version CPU présente un temps d'exécution très proche, avec moins de 10% de différence, par rapport à la version GPU pour certaines tailles d'images supérieures à la résolution Full HD de 1920×1080 pixels. Nous nous attendions à des temps d'exécutions plus bas par rapport à ceux sur CPU, en particulier sur la carte Jetson contenant le GPU le plus récent et le plus puissant. Alors que les GPU testés contiennent jusqu'à 512 cœurs de calcul permettant une puissance de calcul supérieure au CPU de la même carte, nous avons vu en chapitre 3 les limitations de ces derniers en termes de mémoires locale rapide (la mémoire Shared) mais aussi en cache de données. Ces contraintes matérielles limitent le champ des optimisations agissant sur la localité spatiale et temporelles des données nécessaires au calcul : le pipeline d'opérateur et la fusion d'itérations multiples. Nous avons vu dans ce même chapitre ainsi que dans le chapitre 4 qu'à mesure que la profondeur du pipeline augmente ou que l'on fusionne des itérations algorithmiques, le coût mémoire croit de manière cubique en fonction du nombre d'itérations. Ces limitations défavorisent les GPU embarqués par rapport aux CPU.

Enfin, d'un point de vue de la consommation énergétique, bien que nécessitant plus de puissance, les estimations du flot optique sont plus efficaces sur GPU que sur CPU. Nous avons ainsi noté une diminution d'un facteur de $\times 6,4$ de la consommation énergétique en nanojoules par pixels en passant du CPU au GPU. Nous noterons également que la Jetson Xavier qui est la plateforme la plus puissante et la plus récente est la plus efficace des 3 cartes Jetson testées.

Ainsi, nos implémentations optimisées des algorithmes de TV-L^1 et de Horn-Schunck sur GPU nous ont permis d'obtenir des gains à la fois temporels, mais également énergétique par rapport à nos implémentations sur CPU.

Dans le cadre des algorithmes itératifs d'estimation du flot optique, une baisse de la dynamique des nombres utilisés permet-elle une accélération suffisante pour améliorer la convergence des algorithmes ?

C'est ici un résultat contre-intuitif que l'on a réussi à dégager de nos résultats en section 5.2.3. Comme nous avons pu le voir en chapitre 2, le passage de nombre simple-précision sur 32 bits vers des nombres demi-précision sur 16 bits entraîne invariablement à la fois une diminution de l'intervalle des valeurs représentables, mais aussi de la granularité des valeurs fractionnelles représentables. Ce changement de type a également un impact sur le traitement des nombres dans le matériel lui-même. Comme nous l'avons vu en chapitre 3, il y a des différences entre le traitement de nombres simple précision des nombres demi précision selon l'architecture GPU ou CPU visées. Le nombre de cycles peuvent changer selon la version de l'architecture ainsi que la possibilité de traiter plusieurs nombres à la fois si les types vectoriels F32x2 et F16x2 sont utilisés. En plus du temps de traitement, ce passage a un effet sur la bande passante nécessaire au traitement d'une image entière. Grâce aux tests et aux comparaisons entre versions réaliser en chapitre 5, nous avons pu dresser un tableau comparatif cherchant à résumer les besoins de chacune de nos implémentations de TV-L^1 et de Horn-Schunck selon le type numérique utilisé. Ce tableau synthétique ainsi que toutes les comparaisons dans ce chapitre de résultats nous montrent que le gain temporel du passage d'une implémentation d'une version simple précision vers une version demi précision dépend à la fois de l'algorithme choisi, mais également de l'implémentation et des transformations réalisées. En passant d'une implémentation simple précision vers une implémentation demi précision, nous observons ainsi pour nos meilleures implémentations une diminution du temps de traitement de 27% pour TV-L^1 et une diminution de 391% pour Horn-Schunck sur Jetson Xavier. Cette différence d'accélération s'explique par les différences en ressources nécessaires au traitement des deux algorithmes. La version la plus rapide de TV-L^1 fait également usage de la transformation haut niveau dîtes MltIteration qui

va traiter plusieurs itérations lors d'un seul appel au GPU. Les threads d'un bloc de calcul vont terminer leur exécution tour à tour en fonction de la décroissance de la zone à calculer en fonction du nombre d'itérations réalisé par le bloc de threads. Ce type d'implémentations est beaucoup moins impacté qu'une version plus classique ne réalisant que le traitement d'une seule itération d'un seul point du flot optique final à calculer.

Cette première comparaison entre nos implémentations simple-précision et demi précision n'est pas une comparaison qualitative. Elle nous sert cependant pour celle-ci, présenté en dernière partie du chapitre 5 en section 5.2.3. Le but de cette section est de déterminer l'impact de ce changement de type numérique et s'il est possible de le compenser ou même de le dépasser dans les cas d'algorithmes itératifs d'estimation du flot optique. La première observation de cette section concerne les erreurs obtenues lorsque nos algorithmes atteignent la convergence. Nous avons estimé l'erreur point à point moyenne ainsi que l'erreur angulaire moyenne de notre flot optique estimé par rapport à la vérité terrain de deux jeux de données de références, à savoir les banques d'images et de flot Middlebury et MPI-Sintel. Nous avons observé peu ou pas de différences pour ces deux métriques à la convergence. Notre deuxième observation concerne la qualité atteinte à temps constant entre nos implémentations simple et demi-précision. Étant donné que nos implémentations demi précision sont plus rapides, elles peuvent réaliser plus d'itérations pendant le même laps de temps que nos versions simple-précision. Pour TV-L^1, nous avons ainsi pu observer une baisse de l'erreur point à point moyenne de 20% et une baisse de l'erreur angulaire en moyenne de 37% pour le jeu de données de Middlebury. Pour le jeu de données MPI Sintel qui contient des mouvements plus grands et plus complexes, nous observons une baisse de l'erreur point à point moyenne (AEPE) de 7% et une baisse de l'erreur angulaire en moyenne (AAE) de 15%. Cette baisse des erreurs est encore plus accrue pour l'algorithme de Horn-Schunck étant donné qu'il bénéficie beaucoup plus du passage des nombres simple-précision vers demi précision. Ainsi pour Horn-Schunck, nous observons ainsi une baisse de l'AEPE 36% et une baisse de l'AAE de 79% pour Middlebury. Même pour la base MPI-Sintel, nous observons une baisse de l'AEPE de 16% et une baisse de l'AAE de 86%.

Ainsi, les optimisations en demi précision permettent à nos implémentations des algorithmes itératifs TV-L^1 et de Horn-Schunck de produire des résultats de meilleure qualité que nos implémentations simple-précision à temps égal.

Toutes ces optimisations et considérations sont-elles valables pour d'autres algorithmes itératifs de flot optique ?
Notre étude s'est concentré sur l'optimisation de deux algorithmes itératifs de types stencils d'estimation du flot optique TV-L^1 et Horn-Schunck. Ces algorithmes appartiennent à la même famille d'algorithmes itératifs d'estimation de flot optique dense. Nous avons décrit dans le chapitre 4 leur structure interne ainsi que les optimisations réalisées sur chacun. Le chapitre 5 a permis de déterminer des différences de comportement face à ces optimisations et ces comportements. Alors que des optimisations visant à augmenter la localité temporelle et spatiale des données dans les caches et dans les mémoires du GPU permettent une réduction du temps d'exécution pour TV-L^1, ce n'est pas le cas pour Horn-Schunck. Nous avons noté des temps d'exécutions sur GPU proche voir plus bas que ceux sur CPU pour Horn-Schunck. Le passage d'une version simple précision vers une version demi précision induit également différents gains entre les meilleures versions de TV-L^1 et celle de Horn-Schunck. Les capacités matérielles des architectures GPU ont également un impact sur le bon comportement ou non de nos transformations.

Ces considérations et ces limitations algorithmiques et matérielles nous ont ainsi poussé à réaliser de nombres test et de nombreuses comparaisons afin de déterminer les meilleures configurations de nos meilleures versions sur nos architectures, pour les algorithmes TV-L^1 et Horn-Schunck décliné en simple et demi précision. Cette liste d'optimisations et de transformations seront à tester dans le cas de l'implémentation de nouveaux algorithmes itératifs d'estimation du flot optique.

6.2 Limitations et travaux futurs

Nous présentons dans cette section les limitations de notre travail d'optimisations d'algorithmes d'estimation du flot optique sur GPU embarqué ainsi que les axes de recherches à suivre pour poursuivre notre travail.

Limitation de nos implémentations à TV-L^1 et Horn-Schunck Nous avons présenté dans notre travail deux algorithmes d'estimation du flot optique. Or, nous avons vu que le comportement de nos optimisations dépend en grande partie de la charge de calcul et des données à lire et écrire propre au schéma numérique de nos algorithmes. Afin de compléter notre étude, il faudrait l'étendre à d'autres algorithmes tels que Lucas-Kanade [LK+81], eFolki [PLC16], ou RLOF [SES12]. Ces algorithmes précédemment cités ont les bonnes propriétés d'être populaire et utilisé dans la

bibliothèque OpenCV ou d'avoir été le sujet d'implémentation sur GPU (dans le cas de eFolki). D'autres variantes et extensions de TV-L^1 et de Horn-Schunck sont également disponibles afin de renforcer la qualité du flot estimé [Wed+09 ; BFB94].

De par l'architecture ARM de cartes Jetson et de la nature embarquée du GPU de cette carte, nous avons également été limités dans la possibilité d'exécuter des algorithmes disponibles en ligne, notamment des algorithmes reposant sur des réseaux de neurones. Certaines nouvelles publications se penchent sur cette question d'implémentation de réseaux de neurones pour l'estimation du flot optique sur plateformes embarquées [Sez+22a]. Des fonctions disponibles dans certaines API telles que Caffe, TensorRT ou PyTorch ne sont pas disponibles et nécessitent un travail de réimplémentation sur les plateformes embarquées.

Enfin, un travail concernant l'approche pyramidale de nos versions pourrait également être fait. D'autres types que les pyramides gaussiennes d'images sont possibles comme les pyramides laplaciennes, autrement appelées pyramides de Burt [Bur81] ou encore les espaces d'échelle [ASW99].

Au-delà de l'emploi de nouvelles techniques, il serait également intéressant de tester et de modifier d'autres paramètres de nos algorithmes d'estimation de flot optique :

— le mode de rééchantillonnage dans la pyramide (linéaire, cubique, splines ...),
— le taille et le type de filtrage pour éviter le repliement de spectres (filtrage Gaussien, filtrage de Lanczos ...),
— le facteur de changement d'échelle,
— le calcul ou non à certains étages de la pyramide,
— une étude sur les paramètres propre de chaque algorithme ayant un impact sur la qualité du flot (τ, λ et θ pour TV-L^1, α pour Horn-Schunck).

Une nouvelle thèse est en cours pour tester une approche hétérogène CPU-GPU où certains étages de la pyramide sont réalisés sur CPU (les plus petits) et d'autres sur GPU (les plus gros). Il sera crucial de déterminer la bonne répartition des charges entre CPU et GPU afin de tirer les meilleures performances du système complet tout en limitant le temps d'exécution, la consommation et maximisant la qualité de traitement [Rou+17].

Ouverture à d'autres transformations Dans nos travaux de thèses, nous nous sommes limités à des transformations n'ayant pas d'impact sur les dépendances de données. Notre code le plus optimisé donne des résultats quasi-identique à notre code de base, les seules différences étant introduites par l'ordre des calculs en utilisant des

nombres flottants. D'autres transformations sont possibles, mais ne respecte plus l'ordre strict du schéma numérique des algorithmes. Ce mélange des itérations dans le chargement des données pour le calcul d'un stencil converge également. Cela revient à prendre une moins grande pente de descente de gradient qu'avec la version itérative de base.

La gestion des bords est également un autre sujet. Afin d'accélérer le code et de limiter la croissance des dépendances, il serait possible d'envisager un calcul restant dans les blocs, sans dépendances spatiales ayant un impact sur les blocs voisins. Une gestion des effets de bords devrait également être envisagée afin de limiter les transitions brutales dans le flot produit entre deux blocs (variations de la taille des blocs, passe de lissage...). L'exploration d'autres types numériques est également un sujet, que cela soit une comparaison à d'autres formats flottants plus grands (sur 64 bits) ou plus petits (sur 16, 13 ou 8 bits), ou à des formats à virgule fixe.

Limitations matérielles Depuis le début de cette thèse, de nouvelles cartes Jetson de Nvidia sont désormais disponibles comme la Jetson AGX Orin qui est désormais la carte la plus puissante de la gamme. La sous-gamme Orin est également déclinée en d'autres cartes contenant plus ou moins de cœurs de calcul GPU et CPU. Le format des cartes Jetson TX2 a également changé, cela pouvant avoir un impact sur la consommation énergétique totale de la carte.

Il serait également intéressant de tester nos optimisations sur des GPU non embarqués dits discrets. De nouvelles considérations, notamment les temps de transferts entre les mémoires CPU et les mémoires du GPU, seront à prendre en compte dans la mise-à-l'échelle de nos implémentations. Les différences de consommation entre le CPU et le GPU seront à nouveau à prendre en compte, les GPU discrets pouvant nécessiter plusieurs centaines de watts.

Ouverture à d'autres tests qualitatifs Nous nous sommes limités dans nos travaux à deux jeux de données contenant des vérités terrains pour pouvoir qualifier le flot optique produit par nos implémentations optimisées. D'autres banques d'images existent notamment les jeux de données KITTI 2012 [GLU12] et KITTI 2015 [MG15]. Il serait également intéressant de tester nos algorithmes de flot optique sur de véritables séquences vidéos. La difficulté est ici de trouver une bonne métrique pour qualifier la qualité du flot produit du point de vue visuel humain, comme notamment les métriques SSIM et FSIM [SAU19].

Publications

Ces travaux ont donné lieu aux publications suivantes :

1. Th. Romera, A. Petreto, F. Lemaitre, M. Bouyer, Quentin L. Meunier, L. Lacassagne, D. Etiemble : “Optical flow algorithms optimized for speed, energy and accuracy on embedded GPUs”, in *Journal of Real-Time Image Processing (JRTIP)*, 2023
2. Th. Romera, A. Petreto, F. Lemaitre, M. Bouyer, Q. Meunier, L. Lacassagne : “Implementations Impact on Iterative Image Processing for Embedded GPU”, in *European Signal Processing Conference (EUSIPCO)*, 2021

Ces travaux ont donné lieu aux publications suivantes en co-auteur :

1. A. Petreto, Th. Romera, F. Lemaitre, M. Bouyer, B. Gaillard, P. Menard, Q. Meunier, L. Lacassagne : “Real-time embedded video denoiser prototype”, in *Optronics in Defense and Security (Optro)*, 2020
2. A. Petreto, Th. Romera, F. Lemaitre, I. Masliah, B. Gaillard, M. Bouyer, Q. Meunier, L. Lacassagne : “Débruitage temps réel embarqué pour vidéos fortement bruitées”, in *Conférence d’informatique en Parallélisme, Architecture et Système (COMPAS)*, 2019
3. A. Petreto, Th. Romera, I. Masliah, B. Gaillard, M. Bouyer, Q. Meunier, L. Lacassagne, F. Lemaitre : “A New Real-Time Embedded Video Denoising Algorithm”, in *Conference on Design and Architectures for Signal and Image Processin (DASIP)*, 2019
4. A. Petreto, A. Hennequin, Th. Koehler, Th. Romera, Y. Fargeix, B. Gaillard, M. Bouyer, Q. Meunier, L. Lacassagne : “Energy and Execution Time Comparison of Optical Flow Algorithms on SIMD and GPU Architectures”, in *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2018
5. A. Petreto, A. Hennequin, Th. Koehler, Th. Romera, Y. Fargeix, B. Gaillard, M. Bouyer, Q. Meunier, L. Lacassagne : “Comparaison de la consommation énergétique et du temps d’exécution d’un algorithme de traitement d’images optimisé sur des architectures SIMD et GPU”, in *Conférence d’informatique en Parallélisme, Architecture et Système (COMPAS)*, 2018
6. A. Petreto, A. Hennequin, Th. Koehler, Th. Romera, Y. Fargeix, B. Gaillard, M. Bouyer, Q. Meunier, L. Lacassagne : “Comparaison de la consommation énergétique et du temps d’exécution d’un algorithme de traitement d’images optimisé sur des architectures SIMD et GPU”, in *Colloque du GdR SOC2*, 2018

Bibliographie

[AM16] Juan David ADARVE et Robert MAHONY. « A filter formulation for computing real time optical flow ». In : *IEEE Journal of Robotics and Automation Letters (RA-L)* 1.2 (2016), p. 1192-1199 (cf. p. 31, 32).

[All+95] Vicki H ALLAN, Reese B JONES, Randall M LEE et Stephen J ALLAN. « Software pipelining ». In : *ACM Computing Surveys (CSUR)* 27.3 (1995), p. 367-432 (cf. p. 49).

[ASW99] Luis ALVAREZ, Javier SÁNCHEZ et Joachim WEICKERT. « A scale-space approach to nonlocal optical flow calculations ». In : *Scale-Space Theories in Computer Vision : Second International Conference, Scale-Space'99 Corfu, Greece, September 26–27, 1999 Proceedings 2*. Springer. 1999, p. 235-246 (cf. p. 166).

[Amd67] Gene M AMDAHL. « Validity of the single processor approach to achieving large scale computing capabilities ». In : *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, p. 483-485 (cf. p. 19).

[AM17] Jérémy ANGER et Enric MEINHARDT-LLOPIS. « Implementation of local Fourier burst accumulation for video deblurring ». In : *Image Processing On Line* 7 (2017), p. 56-64 (cf. p. 34).

[AM18] Pablo ARIAS et Jean-Michel MOREL. « Video denoising via empirical Bayesian estimation of space-time patches ». In : *Journal of Mathematical Imaging and Vision* 60.1 (2018), p. 70-93 (cf. p. 34).

[Asa+06] Krste ASANOVIĆ, Ras BODIK, Bryan Christopher CATANZARO, Joseph James GEBIS, Parry HUSBANDS, Kurt KEUTZER, David A. PATTERSON, William Lester PLISHKER, John SHALF, Samuel Webb WILLIAMS et Katherine A. YELICK. *The Landscape of Parallel Computing Research : A View from Berkeley*. Rapp. tech. UCB/EECS-2006-183. EECS Department, University of California, Berkeley, déc. 2006 (cf. p. 11).

[ARS12] Alper AYVACI, Michalis RAPTIS et Stefano SOATTO. « Sparse occlusion detection with optical flow ». In : *International journal of computer vision* 97.3 (2012), p. 322-338 (cf. p. 33).

[Bai11] Donald G BAILEY. « Image border management for FPGA based filters ». In : *2011 Sixth IEEE International Symposium on Electronic Design, Test and Application*. IEEE. 2011, p. 144-149 (cf. p. 55).

[BA18] Donald G BAILEY et Anoop S AMBIKUMAR. « Border handling for 2D transpose filter structures on an FPGA ». In : *Journal of Imaging* 4.12 (2018), p. 138 (cf. p. 55).

[BM01] Simon BAKER et Iain MATTHEWS. « Equivalence and efficiency of image alignment algorithms ». In : *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*. T. 1. IEEE. 2001, p. I-I (cf. p. 30).

[Bak+11] Simon BAKER, Daniel SCHARSTEIN, JP LEWIS, Stefan ROTH, Michael J BLACK et Richard SZELISKI. « A database and evaluation methodology for optical flow ». In : *International Journal of Computer Vision (IJCV)* 92.1 (2011), p. 1-31 (cf. p. 10, 145).

[Bao+14] Linchao BAO, Hailin JIN, Byungmoon KIM et Qingxiong YANG. « A comparison of TV-L1 optical flow solvers on GPU ». In : *GTC Posters* 6 (2014) (cf. p. 32, 33).

[BFB94] John L BARRON, David J FLEET et Steven S BEAUCHEMIN. « Performance of optical flow techniques ». In : *International journal of computer vision* 12.1 (1994), p. 43-77 (cf. p. 10, 166).

[Bät+15] Michel BÄTZ, Andrea EICHENSEER, Jürgen SEILER, Markus JONSCHER et Andre KAUP. « Hybrid super-resolution combining example-based single-image and interpolation-based multi-image reconstruction approaches ». In : *2015 IEEE international conference on image processing (ICIP)*. IEEE. 2015, p. 58-62 (cf. p. 34).

[BT09] Amir BECK et Marc TEBOULLE. « A fast iterative shrinkage-thresholding algorithm for linear inverse problems ». In : *SIAM journal on imaging sciences* 2.1 (2009), p. 183-202 (cf. p. 33).

[Big+06] M BIGAS, Enric CABRUJA, Josep FOREST et Joaquim SALVI. « Review of CMOS image sensors ». In : *Microelectronics journal* 37.5 (2006), p. 433-451 (cf. p. 2).

[Boa21] OpenMP Architecture Review BOARD. *OpenMP Application Programming Interface Version 5.2 November 2021*. Nov. 2021. URL : `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf` (cf. p. 15).

[BCL21] Ilias BOURNIAS, Roselyne CHOTIN et Lionel LACASSAGNE. « FPGA Acceleration of the Horn and Schunck Hierarchical algorithm ». In : *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE. 2021, p. 1-5 (cf. p. 26, 46).

[BCL22] Ilias BOURNIAS, Roselyne CHOTIN et Lionel LACASSAGNE. « Using HLS for Designing a Parametric Optical Flow Hierarchical Algorithm in FPGAs ». In : *IEEE International Symposium on Circuits and Systems (ISCAS 2022)*. 2022 (cf. p. 46).

[Bra00] Gary BRADSKI. « The OpenCV library. » In : *Dr. Dobb's Journal : Software Tools for the Professional Programmer* 25.11 (2000), p. 120-123 (cf. p. 34).

[Bro+04] Thomas BROX, Andrés BRUHN, Nils PAPENBERG et Joachim WEICKERT. « High accuracy optical flow estimation based on a theory for warping ». In : *Proceedings of the 2004 European Conference on Computer Vision (ECCV)*. 2004, p. 25-36 (cf. p. 31, 33).

[BLM16] Antoni BUADES, Jose-Luis LISANI et Marko MILADINOVIĆ. « Patch-based video denoising with optical flow estimation ». In : *IEEE Transactions on Image Processing* 25.6 (2016), p. 2573-2586 (cf. p. 34).

[Bur81] Peter J BURT. « Fast filter transform for image processing ». In : *Computer graphics and image processing* 16.1 (1981), p. 20-51 (cf. p. 166).

[But+12] Daniel J BUTLER, Jonas WULFF, Garrett B STANLEY et Michael J BLACK. « A naturalistic open source movie for optical flow evaluation ». In : *European conference on computer vision*. Springer. 2012, p. 611-625 (cf. p. 10, 145).

[BB84] Bernard F BUXTON et Hilary BUXTON. « Computation of optic flow from the motion of edge features in image sequences ». In : *Image and Vision Computing* 2.2 (1984), p. 59-75 (cf. p. 30).

[BSB84] BF BUXTON, BK STEPHENSON et H BUXTON. « Parallel computations of optic flow in early image processing ». In : *IEE Proceedings F (Communications, Radar and Signal Processing)*. T. 131. 6. IET. 1984, p. 593-602 (cf. p. 30).

[CL14] L. CABARET et L. LACASSAGNE. « A Review of World's Fastest Connected Component Labeling Algorithms : Speed and Energy Estimation ». In : *IEEE International Conference on Design and Architectures for Signal and Image Processing (DASIP)*. 2014, p. 1-8 (cf. p. 5).

[CLE15] L. CABARET, L. LACASSAGNE et D. ETIEMBLE. « Parallel Light Speed Labeling : an efficient connected component labeling algorithm for multi-Core processors ». In : *IEEE International Conference on Image Processing (ICIP)*. 2015, p. 1-4 (cf. p. 5).

[Cam97] Ted CAMUS. « Real-time quantized optical flow ». In : *Real-Time Imaging* 3.2 (1997), p. 71-86 (cf. p. 30).

[CZU22] Humberto CARVALHO, Pavel ZAYKOV et Asim UKAYE. « Leveraging the HW/SW Optimizations and Ecosystems that Drive the AI Revolution ». In : *arXiv preprint arXiv :2208.02808* (2022) (cf. p. 33).

[Cas14] Adrien CASSAGNE. « Étude et implémentation d'une méthode de calcul pour la simulation numérique sur des architectures modernes ». In : (2014) (cf. p. 98).

[Cha04] Antonin CHAMBOLLE. « An algorithm for total variation minimization and applications ». In : *Journal of Mathematical Imaging and Vision* 20.1 (2004), p. 89-97 (cf. p. 74).

[Cha+11] Frédéric CHAMPAGNAT, Aurélien PLYER, Guy LE BESNERAIS, Benjamin LECLAIRE, Samuel DAVOUST et Yves LE SANT. « Fast and accurate PIV computation using highly parallel iterative correlation maximization ». In : *Experiments in fluids* 50.4 (2011), p. 1169-1182 (cf. p. 9).

[Che+17] Stefano CHERUBIN, Giovanni AGOSTA, Imane LASRI, Erven ROHOU et Olivier SENTIEYS. « Implications of Reduced-Precision Computations in HPC : Performance, Energy and Error ». In : *International Conference on Parallel Computing (ParCo)*. 2017 (cf. p. 27).

[Cio+23a] C. CIOCAN, M. KANDEEPAN, A. CASSAGNE, J. VAUBAILLON, F. ZANDER et L. LACASSAGNE. « A new meteor detection application robust to camera movements ». In : *arXiv :2309.06027*. 2023 (cf. p. 142).

[Cio+23b] C. CIOCAN, M. KANDEEPAN, A. CASSAGNE, J. VAUBAILLON, F. ZANDER et L. LACASSAGNE. « Une nouvelle application de détection de météores robuste aux mouvements de caméra ». In : *GRETSI*. 2023 (cf. p. 142).

[Cio+23c] Clara CIOCAN, Mathuran KANDEEPAN, Adrien CASSAGNE, Jeremie VAUBAILLON, Fabian ZANDER et Lionel LACASSAGNE. « Une nouvelle application de détection de météores robuste aux mouvements de caméra ». In : *Groupe de Recherche et d'Études de Traitement du Signal et des Images (GRETSI)*. 2023 (cf. p. 8).

[Col+20] F. COLAS, B. ZANDA, S. BOULEY et al. « FRIPON : a worldwide network to track incoming meteoroids ». In : *Astronomy and Astrophysics (A & A)* 644 (2020), p. 1-23 (cf. p. 142).

[dAn+11] Emmanuel D'ANGELO, Johan PARATTE, Gilles PUY et Pierre VANDERGHEYNST. « Fast TV-L1 optical flow for interactivity ». In : *Proceedings of the 18th IEEE International Conference on Image Processing (ICIP)*. 2011, p. 1885-1888 (cf. p. 32, 33).

[DFE07] Kostadin DABOV, Alessandro FOI et Karen EGIAZARIAN. « Video denoising by sparse 3D transform-domain collaborative filtering ». In : *2007 15th European Signal Processing Conference*. IEEE. 2007, p. 145-149 (cf. p. 4).

[DC15] David DEFOUR et Caroline COLLANGE. « Reproducible floating-point atomic addition in data-parallel environment ». In : *2015 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE. 2015, p. 721-728 (cf. p. 65).

[Def+20] David DEFOUR, Pablo de OLIVEIRA CASTRO, Matei IŞTOAN et Eric PETIT. « Custom-precision mathematical library explorations for code profiling and optimization ». In : *2020 IEEE 27th Symposium on Computer Arithmetic (ARITH)*. IEEE. 2020, p. 121-124 (cf. p. 26).

[DP13] David DEFOUR et Eric PETIT. « GPUburn : A system to test and mitigate GPU hardware failures ». In : *2013 International Conference on Embedded Computer Systems : Architectures, Modeling, and Simulation (SAMOS)*. IEEE. 2013, p. 263-270 (cf. p. 29).

[DS15] Mauricio DELBRACIO et Guillermo SAPIRO. « Hand-held video deblurring via efficient fourier aggregation ». In : *IEEE Transactions on Computational Imaging* 1.4 (2015), p. 270-283 (cf. p. 34).

[Den+20] Zilong DENG, Dongxiao YANG, Xiaohu ZHANG, Yuguang DONG, Chengbo LIU et Qiang SHEN. « Real-time image stabilization method based on optical flow and binary point feature matching ». In : *Electronics* 9.1 (2020), p. 198 (cf. p. 9).

[Dos+15] Alexey DOSOVITSKIY, Philipp FISCHER, Eddy ILG, Philip HAUSSER, Caner HAZIRBAS, Vladimir GOLKOV, Patrick VAN DER SMAGT, Daniel CREMERS et Thomas BROX. « Flownet : Learning optical flow with convolutional networks ». In : *Proceedings of the IEEE international conference on computer vision*. 2015, p. 2758-2766 (cf. p. 33).

[EPL06] D. ETIEMBLE, S. PISKORSKI et L. LACASSAGNE. « Performance evaluation of Altera C2H compiler on image processing benchmarks ». In : *Workshop on Tools And Compiler for Hardware Acceleration (TCHA)*. 2006 (cf. p. 25).

[EL04] Daniel ETIEMBLE et Lionel LACASSAGNE. « 16-bit FP sub-word parallelism to facilitate compiler vectorization and improve performance of image and media processing ». In : *International Conference on Parallel Processing, 2004. ICPP 2004.* IEEE. 2004, p. 540-547 (cf. p. 25, 63).

[Far03] Gunnar FARNEBÄCK. « Two-frame motion estimation based on polynomial expansion ». In : *Proceedings of the 2003 Scandinavian Conference on Image Analysis (SCIA)*. 2003, p. 363-370 (cf. p. 31).

[Fil+15] Jiří FILIPOVIČ, Matúš MADZIN, Jan FOUSEK et Luděk MATYSKA. « Optimizing CUDA code by kernel fusion : application on BLAS ». In : *The Journal of Supercomputing* 71.10 (2015), p. 3934-3957 (cf. p. 46).

[FLI19] Inc. FLIR SYSTEMS. *SENSOR REVIEW : MONO CAMERAS*. Rapp. tech. 2019 (cf. p. 2).

[Fly72] Michael J FLYNN. « Some computer organizations and their effectiveness ». In : *IEEE transactions on computers* 100.9 (1972), p. 948-960 (cf. p. 11).

[Fly66] Michael J FLYNN. « Very high-speed computing systems ». In : *Proceedings of the IEEE* 54.12 (1966), p. 1901-1909 (cf. p. 11).

[Fos93] Eric R FOSSUM. « Active pixel sensors : Are CCDs dinosaurs ? » In : *Charge-Coupled Devices and Solid State Optical Sensors III*. T. 1900. SPIE. 1993, p. 2-14 (cf. p. 1).

[Gar+14] Antonio GARCIA-DOPICO, José Luis PEDRAZA, Manuel NIETO, Antonio PÉREZ, Santiago RODRÍGUEZ et Juan NAVAS. « Parallelization of the optical flow computation in sequences from moving cameras ». In : *EURASIP Journal on Image and Video Processing* 2014.1 (2014), p. 1-19 (cf. p. 30).

[GLU12] Andreas GEIGER, Philip LENZ et Raquel URTASUN. « Are we ready for autonomous driving ? the kitti vision benchmark suite ». In : *2012 IEEE conference on computer vision and pattern recognition*. IEEE. 2012, p. 3354-3361 (cf. p. 10, 167).

[Gro20] Khronos GROUP. *OpenCL C 3.0 revision V3.0.11*. Sept. 2020. URL : `https://www.khronos.org/opencl/` (cf. p. 15).

[GIK10] Kai GUO, Prakash ISHWAR et Janusz KONRAD. « Action recognition using sparse representation on covariance manifolds of optical flow ». In : *2010 7th IEEE international conference on advanced video and signal based surveillance*. IEEE. 2010, p. 188-195 (cf. p. 9).

[Gus88] John L GUSTAFSON. « Reevaluating Amdahl's law ». In : *Communications of the ACM* 31.5 (1988), p. 532-533 (cf. p. 20).

[Gwo+10] Pascal GWOSDEK, Henning ZIMMER, Sven GREWENIG, Andrés BRUHN et Joachim WEICKERT. « A highly efficient GPU implementation for variational optic flow based on the Euler-Lagrange framework ». In : *European Conference on Computer Vision*. Springer. 2010, p. 372-383 (cf. p. 32).

[Hag+19a] Olfa HAGGUI, Claude TADONKI, Fatma SAYADI et Bouraoui OUNI. « Efficient GPU Implementation of Lucas-Kanade through OpenACC. » In : *VISIGRAPP (5 : VISAPP)*. 2019, p. 768-775 (cf. p. 15).

[Hag+19b] Olfa HAGGUI, Claude TADONKI, Fatma SAYADI et Bouraoui OUNI. « Memory efficient deployment of an optical flow algorithm on GPU using OpenMP ». In : *International Conference on Image Analysis and Processing*. Springer. 2019, p. 477-487 (cf. p. 15).

[Ham15] Leonard GC HAMEY. « A functional approach to border handling in image processing ». In : *2015 International Conference on Digital Image Computing : Techniques and Applications (DICTA)*. IEEE. 2015, p. 1-8 (cf. p. 55).

[HL19] A. HENNEQUIN et L. LACASSAGNE. « A New Direct Connected Component Labeling and Analysis Algorithm for GPUs ». In : *GPU Technology Conference (GTC)*. 2019 (cf. p. 5).

[Hen+18] Arthur HENNEQUIN, Lionel LACASSAGNE, Laurent CABARET et Quentin MEUNIER. « A new direct connected component labeling and analysis algorithms for GPUs ». In : *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE. 2018, p. 76-81 (cf. p. 63).

[HML19] Arthur HENNEQUIN, Ian MASLIAH et Lionel LACASSAGNE. « Designing efficient simd algorithms for direct connected component labeling ». In : *Proceedings of the 5th Workshop on Programming Models for SIMD/Vector Processing*. 2019, p. 1-8 (cf. p. 5).

[HW17] Nhut-Minh HO et Weng-Fai WONG. « Exploiting half precision arithmetic in Nvidia GPUs ». In : *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2017, p. 1-7 (cf. p. 39, 40, 63).

[HS81] Berthold KP HORN et Brian G SCHUNCK. « Determining optical flow ». In : *Journal of Artificial Intelligence (AIJ)* 17.1-3 (1981), p. 185-203 (cf. p. 9, 78).

[HL20] Tak-Wai HUI et Chen Change LOY. « LiteFlowNet3 : Resolving Correspondence Ambiguity for More Accurate Optical Flow Estimation ». In : *Proceedings of the 2020 European Conference on Computer Vision (ECCV)*. 2020, p. 169-184 (cf. p. 33).

[HTL20] Tak-Wai HUI, Xiaoou TANG et Chen Change LOY. « A lightweight optical flow CNN—Revisiting data fidelity and regularization ». In : *IEEE transactions on pattern analysis and machine intelligence* 43.8 (2020), p. 2555-2569 (cf. p. 33).

[HTL18] Tak-Wai HUI, Xiaoou TANG et Chen Change LOY. « Liteflownet : A lightweight convolutional neural network for optical flow estimation ». In : *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, p. 8981-8989 (cf. p. 33).

[HM15] Tae HYUN KIM et Kyoung MU LEE. « Generalized video deblurring for dynamic scenes ». In : *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, p. 5426-5434 (cf. p. 9).

[Ilg+17] Eddy ILG, Nikolaus MAYER, Tonmoy SAIKIA, Margret KEUPER, Alexey DOSOVITSKIY et Thomas BROX. « Flownet 2.0 : Evolution of optical flow estimation with deep networks ». In : *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017, p. 2462-2470 (cf. p. 33).

[Int+15] Giuseppe INTERMITE, Aongus MCCARTHY, Ryan E WARBURTON, Ximing REN, Federica VILLA, Rudi LUSSANA, Andrew J WADDIE, Mohammad R TAGHIZADEH, Alberto TOSI, Franco ZAPPA et al. « Fill-factor improvement of Si CMOS single-photon avalanche diode detector arrays by integration of diffractive microlens arrays ». In : *Optics express* 23.26 (2015), p. 33777-33791 (cf. p. 2).

[JHG99] Bernd JÄHNE, Horst HAUSSECKER et Peter GEISSLER. *Handbook of computer vision and applications*. T. 2. Citeseer, 1999 (cf. p. 8).

[Jia+18] Zhe JIA, Marco MAGGIONI, Benjamin STAIGER et Daniele P SCARPAZZA. « Dissecting the NVIDIA volta GPU architecture via microbenchmarking ». In : *arXiv preprint arXiv :1804.06826* (2018) (cf. p. 65).

[Kan+23a] M. KANDEEPAN, C. CIOCAN, A. CASSAGNE et L. LACASSAGNE. « Parallélisation d'une nouvelle application embarquée pour la détection automatique de météores ». In : *COMPAS*. 2023 (cf. p. 142).

[Kan+23b] M. KANDEEPAN, C. CIOCAN, A. CASSAGNE, J. VAUBAILLON, F. ZANDER et L. LACASSAGNE. « Parallelization of a new embedded application for automatic meteor detection ». In : *arXiv :2307.10632*. 2023 (cf. p. 142).

[Kan+] M. KANDEEPAN, C. CIOCAN, M. MILLET, M. BOUYER, A. CASSAGNE et L. LACASSAGNE. « Fast Meteor Detection Toolbox ». In : *Workshop AFF3CT* (cf. p. 142).

[Kan+23c] Mathuran KANDEEPAN, Clara CIOCAN, Adrien CASSAGNE et Lionel LACASSAGNE. « Parallélisation d'une nouvelle application embarquée pour la détection automatique de météores ». In : *COMPAS 2023-Conférence francophone d'informatique en Parallélisme, Architecture et Système*. 2023 (cf. p. 8).

[KDH10] Kamran KARIMI, Neil G DICKSON et Firas HAMZE. « A performance comparison of CUDA and OpenCL ». In : *arXiv preprint arXiv :1005.2581* (2010) (cf. p. 15).

[KAB15] Margret KEUPER, Bjoern ANDRES et Thomas BROX. « Motion trajectory segmentation via minimum cost multicuts ». In : *Proceedings of the IEEE international conference on computer vision*. 2015, p. 3271-3279 (cf. p. 9).

[KPM19] Musaab KHALID, Lionel PÉNARD et Etienne MÉMIN. « Optical flow for image-based river velocity estimation ». In : *Flow Measurement and Instrumentation* 65 (2019), p. 110-121 (cf. p. 9).

[Kla+09] Jens KLAPPSTEIN, Tobi VAUDREY, Clemens RABE, Andreas WEDEL et Reinhard KLETTE. « Moving object segmentation using optical flow and depth information ». In : *Pacific-Rim Symposium on Image and Video Technology*. Springer. 2009, p. 611-623 (cf. p. 9).

[Koz+98] Lester J KOZLOWSKI, J LUO, WE KLEINHANS et T LIU. « Comparison of passive and active pixel schemes for CMOS visible imagers ». In : *Infrared Readout Electronics IV*. T. 3360. SPIE. 1998, p. 101-110 (cf. p. 2).

[Kro+16] Till KROEGER, Radu TIMOFTE, Dengxin DAI et Luc Van GOOL. « Fast optical flow using dense inverse search ». In : *European conference on computer vision*. Springer. 2016, p. 471-488 (cf. p. 30).

[Kug19] Thanushan KUGATHASAN. « SISSA : Review on depleted CMOS ». In : *PoS* (2019), p. 042 (cf. p. 3).

[LE05] L LACASSAGNE et D ETIEMBLE. « 16-bit floating point operations for low-end and high-end embedded processors ». In : *Digests of ODES-3, March* (2005) (cf. p. 63).

[LZ09] L. LACASSAGNE et B. ZAVIDOVIQUE. « Light Speed Labeling for RISC architectures ». In : *IEEE International Conference on Image Analysis and Processing (ICIP)*. 2009, p. 3245-3248 (cf. p. 5).

[Lac+14] Lionel LACASSAGNE, Daniel ETIEMBLE, Ali HASSAN ZAHRAEE, Alain DOMINGUEZ et Pascal VEZOLLE. « High level transforms for SIMD and low-level computer vision algorithms ». In : *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing (WPMVP)*. 2014, p. 49-56 (cf. p. 46).

[LEK05] Lionel LACASSAGNE, Daniel ETIEMBLE et SA Ould KABLIA. « 16-bit floating point instructions for embedded multimedia applications ». In : *Seventh International Workshop on Computer Architecture for Machine Perception (CAMP'05)*. IEEE. 2005, p. 198-203 (cf. p. 63).

[Lac+09] Lionel LACASSAGNE, Antoine MANZANERA, Julien DENOULET et Alain MÉRIGOT. « High performance motion detection : some trends toward new embedded architectures for vision systems ». In : *Journal of Real-Time Image Processing* 4.2 (2009), p. 127-146 (cf. p. 46).

[LFE23] Jamy LAFENETRE, Gabriele FACCIOLO et Thomas EBOLI. « Implementing handheld burst super-resolution ». In : *Image Processing On Line* 2.3 (2023), p. 4 (cf. p. 9).

[Lam88] Monica LAM. « Software pipelining : An effective scheduling technique for VLIW machines ». In : *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. 1988, p. 318-328 (cf. p. 46).

[LR19] Vanel LAZCANO et Francisco RIVERA. « GPU based horn-schunck method to estimate optical flow and occlusion ». In : *International Conference on Theory and Applications of Models of Computation*. Springer. 2019, p. 424-437 (cf. p. 32).

[LML22] F. LEMAITRE, N. MAURICE et L. LACASSAGNE. « An efficient run-based Connected Component Labeling algorithm for processing holes ». In : *IEEE International Workshop Binary is the new Black (and White) workshop (BNBW @ ICIAP)*. 2022 (cf. p. 5).

[LHL20] Florian LEMAITRE, Arthur HENNEQUIN et Lionel LACASSAGNE. « How to speed Connected Component Labeling up with SIMD RLE algorithms ». In : *Proceedings of the 2020 Sixth Workshop on Programming Models for SIMD/Vector Processing*. 2020, p. 1-8 (cf. p. 63).

[LHL21a] Florian LEMAITRE, Arthur HENNEQUIN et Lionel LACASSAGNE. « Taming Voting Algorithms on GPUs for an Efficient Connected Component Analysis Algorithm ». In : *GPU Technical Conference (GTC)*. 2021 (cf. p. 63).

[LHL21b] Florian LEMAITRE, Arthur HENNEQUIN et Lionel LACASSAGNE. « Taming voting algorithms on GPUs for an efficient Connected Component Analysis Algorithm ». In : *ICASSP 2021-2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE. 2021, p. 7903-7907 (cf. p. 63).

[Li+19] Bingchao LI, Jizeng WEI, Jizhou SUN, Murali ANNAVARAM et Nam Sung KIM. « An efficient GPU cache architecture for applications with irregular memory access patterns ». In : *ACM Transactions on Architecture and Code Optimization (TACO)* 16.3 (2019), p. 1-24 (cf. p. 61).

[Low15] David LOWE. *The Computer Vision Industry*. 2015. URL : `https://www.cs.ubc.ca/~lowe/vision.html` (visité le 25 août 2022) (cf. p. 8).

[LK+81] Bruce D LUCAS, Takeo KANADE et al. « An iterative image registration technique with an application to stereo vision ». In : *Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI)*. 1981 (cf. p. 9, 30, 31, 165).

[Mag+12] Matteo MAGGIONI, Giacomo BORACCHI, Alessandro FOI et Karen EGIAZARIAN. « Video denoising, deblocking, and enhancement through separable 4-D nonlocal spatiotemporal transforms ». In : *IEEE Transactions on image processing* 21.9 (2012), p. 3952-3966 (cf. p. 4).

[Mau+22a] N. MAURICE, F. LEMAITRE, J. SOPENA et L. LACASSAGNE. « Etiquetage en Composantes Connexe par segments pour volumes 3D ». In : *COMPAS*. 2022 (cf. p. 5).

[Mau+22b] Nathan MAURICE, Florian LEMAITRE, Julien SOPENA et Lionel LACASSAGNE. « LSL3D : a run-based Connected Component Labeling algorithm for 3D volumes ». In : *International Conference on Image Analysis and Processing*. Springer. 2022, p. 132-142 (cf. p. 5).

[Mem+17] Suejb MEMETI, Lu LI, Sabri PLLANA, Joanna KOŁODZIEJ et Christoph KESSLER. « Benchmarking OpenCL, OpenACC, OpenMP, and CUDA : programming productivity, performance, and energy consumption ». In : *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*. 2017, p. 1-6 (cf. p. 15).

[MG15] Moritz MENZE et Andreas GEIGER. « Object scene flow for autonomous vehicles ». In : *Proceedings of the 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, p. 3061-3070 (cf. p. 10, 167).

[Mil+22a] M. MILLET, N. RAMBAUX, A. CASSAGNE, M. BOUYER, A. PETRETO et L. LACASSAGNE. « High performance computer vision application for Meteor detection from a cubesat ». In : *44th Assembly of Committee on Space Research (COSPAR)*. 2022 (cf. p. 142).

[Mil+22b] M. MILLET, N. RAMBAUX, A. PETRETO, F. LEMAITRE et L. LACASSAGNE. « Meteorix — A new processing chain for real-time detection and tracking of meteors from space ». In : *WGN, Journal of the International Meteor Organization (IMO)* 49.6 (2022), p. 1-5 (cf. p. 142).

[Mil+21] M. MILLET, N. RAMBAUX, A. PETRETO, F. LEMAITRE et L. LACASSAGNE. « Meteorix : Détection temps réel de météores à bord d'un nanosatellite ». In : *ORASIS*. 2021 (cf. p. 142).

[Mil+23] Maxime MILLET, Adrien CASSAGNE, Nicolas RAMBAUX et Lionel LACASSAGNE. « Real-time and approximate iterative optical flow implementation on low-power embedded CPUs ». In : *2023 IEEE 34rd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2023 (cf. p. 138).

[Mil+22c] Maxime MILLET, Nicolas RAMBAUX, Andrea PETRETO, Florian LEMAITRE et Lionel LACASSAGNE. « Meteorix - A new processing chain for real-time detection and tracking of meteors from space ». In : *WGN, Journal of the International Meteor Organization* 49.6 (2022) (cf. p. 142).

[Mit+09] Dennis MITZEL, Thomas POCK, Thomas SCHOENEMANN et Daniel CREMERS. « Video super resolution using duality based TV-L1 optical flow ». In : *Joint Pattern Recognition Symposium*. Springer. 2009, p. 432-441 (cf. p. 34).

[NVI22] NVIDIA CORPORATION. *CUDA C++ Best Practices Guide Version 11.8.0*. Oct. 2022. URL : `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/` (cf. p. 60).

[NVI21] NVIDIA CORPORATION. *CUDA C++ Programming Guide Version 11.5.0*. Oct. 2021. URL : `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html` (cf. p. 15, 63).

[NVI20] NVIDIA CORPORATION. *NVIDIA Jetson Linux Developer Guide 34.1*. Oct. 2020. URL : `https://docs.nvidia.com/jetson/archives/r34.1/DeveloperGuide/index.html#page/Tegra%5C%20Linux%5C%20Driver%5C%20Package%5C%20Development%5C%20Guide/power_management_jetson_xavier.html` (cf. p. 42).

[Org22] OpenACC ORGANIZATION. *The OpenACC Application Programming Interface Version 3.3*. Nov. 2022. URL : `https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.3-final.pdf` (cf. p. 15).

[Pet20] Andrea PETRETO. « Débruitage vidéo temps réel pour systèmes embarqués ». Thèse de doct. Sorbonne université, 2020 (cf. p. 3, 30, 33, 71, 138).

[Pet+18a] Andrea PETRETO, Arthur HENNEQUIN, Thomas KOEHLER, Thomas ROMERA, Yohan FARGEIX, Boris GAILLARD, Manuel BOUYER, Quentin MEUNIER et Lionel LACASSAGNE. « Comparaison de la consommation énergétique et du temps d'exécution d'un algorithme de traitement d'images optimisé sur des architectures SIMD et GPU ». In : *Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS 2018)*. 2018 (cf. p. 142).

[Pet+18b] Andrea PETRETO, Arthur HENNEQUIN, Thomas KOEHLER, Thomas ROMERA, Yohan FARGEIX, Boris GAILLARD, Manuel BOUYER, Quentin L MEUNIER et Lionel LACASSAGNE. « Energy and execution time comparison of optical flow algorithms on SIMD and GPU architectures ». In : *Proceedings of the 2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. 2018, p. 25-30 (cf. p. 30, 33).

[Pet+20] Andrea PETRETO, Thomas ROMERA, Florian LEMAITRE, Manuel BOUYER, Boris GAILLARD, Patrice MENARD, Quentin MEUNIER et Lionel LACASSAGNE. « Real-time embedded video denoiser prototype ». In : *9th International Symposium-Optronics in Defense and Security (Optro)*. 2020 (cf. p. 4).

[Pet+19] Andrea PETRETO, Thomas ROMERA, Florian LEMAITRE, Ian MASLIAH, Boris GAILLARD, Manuel BOUYER, Quentin L MEUNIER et Lionel LACASSAGNE. « A New Real-Time Embedded Video Denoising Algorithm ». In : *Proceedings of the 2019 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. 2019, p. 47-52 (cf. p. 9, 30, 34).

[Pfl+19] Sergio G PFLEGER, Patricia DM PLENTZ, Rodrigo CO ROCHA, Alyson D PEREIRA et Márcio CASTRO. « Real-time video denoising on multicores and gpus with kalman-based and bilateral filters fusion ». In : *Journal of Real-Time Image Processing* 16.5 (2019), p. 1629-1642 (cf. p. 4).

[PLE06] S. PISKORSKI, L. LACASSAGNE et D. ETIEMBLE. « Instruction SIMD flottantes 16 bits pour réduire la consommation dans les processeurs embarqués à jeux d'instructions spécialisables ». In : *SYMPA*. 2006 (cf. p. 25).

[Pis+06] Stephane PISKORSKI, Lionel LACASSAGNE, Samir BOUAZIZ et Daniel ETIEMBLE. « Customizing CPU instructions for embedded vision systems ». In : *Proceedings of the 2006 International Workshop on Computer Architecture for Machine Perception and Sensing (CAMPS)*. 2006, p. 59-64 (cf. p. 25).

[PLC16] Aurélien PLYER, Guy LE BESNERAIS et Frédéric CHAMPAGNAT. « Massively parallel Lucas Kanade optical flow for real-time video processing applications ». In : *Journal of Real-Time Image Processing (JRTIP)* 11.4 (2016), p. 713-730 (cf. p. 31, 32, 165).

[Poc+08] Thomas POCK, Markus UNGER, Daniel CREMERS et Horst BISCHOF. « Fast and exact solution of total variation models on the GPU ». In : *Proceedings of the 2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*. 2008, p. 1-8 (cf. p. 32).

[Ram+18] N. Rambaux, D. Galayko, G. Guignan, J. Vaubaillon, L. Lacassagne, P. Keckhut, A.C. Levasseur-Regourd, A. Hauchecorne, M.Birlan, G. Augarde, S. Barnier, S. Ben Kemmoum, A.Bigot, P. Boisse, M. Capderou, A. Chu, F. Colas, F. Deshours, Y. Fargeix, A. Hennequin, T. Koehler, M. Lumbroso, J.-F. Mariscal, D. Portela-Moreira, J. Raffard, J.-L. Rault, T. Romera, C. Tob et B. Zanda. « Meteorix : a cubesat mission dedicated to the detection of meteors ». In : *42nd Assembly of Committee on Space Research (COSPAR)*. 2018 (cf. p. 142).

[Ram+21] N. Rambaux, J. Vaubaillon, S. Derelle, M. Jacquart, M. Millet, L. Lacassagne, A. Petreto, P. Simoneau, K. Baillie, J. Desmars, D. Galayko et R. Chotin. « Meteorix camera tests for space-based meteor observations ». In : *WGN, Journal of the International Meteor Organization (IMO)* 49.5 (2021), p. 1-3 (cf. p. 9).

[Ram+19] N. Rambaux, J. Vaubaillon, L. Lacassagne, D. Galayko, G. Guignan, M. Birlan, M. Capderou, F. Colas, F. Deleflie, F. Deshours, A. Hauchecorne, P. Keckhut, A.C. Levasseurd-Regourd, J.L. Rault et B. Zanda. « Meteorix : a cubesat mission dedicated to the detection of meteors and space debris ». In : *ESA Space Safety Programme Office, NEO and Debris Detection Conference (ESA NDDC)*. 2019, p. 1-9 (cf. p. 142).

[Ren+90] D Renshaw, PB Denyer, G Wang et M Lu. « ASIC image sensors ». In : *IEEE International Symposium on Circuits and Systems*. IEEE. 1990, p. 3038-3041 (cf. p. 1).

[Rip+19] Oren Rippel, Sanjay Nair, Carissa Lew, Steve Branson, Alexander G Anderson et Lubomir Bourdev. « Learned video compression ». In : *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, p. 3454-3463 (cf. p. 9).

[Rom+23] T. Romera, A. Petreto, F. Lemaitre, M. Bouyer, Q. Meunier, L. Lacassagne et D. Etiemble. « Optical flow algorithms optimized for speed, energy and accuracy on embedded GPUs ». In : *Journal of Real-Time Image Processing (JRTIP)* 20,2.32 (2023), p. 1-12 (cf. p. 142).

[Rom+21] Thomas Romera, Andrea Petreto, Florian Lemaitre, Manuel Bouyer, Quentin Meunier et Lionel Lacassagne. « Implementations Impact on Iterative Image Processing for Embedded GPU ». In : *2021 29th European Signal Processing Conference (EUSIPCO)*. IEEE. 2021, p. 736-740 (cf. p. 34).

[Rou+17] Baptiste Roux, Matthieu Gautier, Olivier Sentieys et Jean-Philippe Delahaye. « Fast and Energy-driven Design Space Exploration for Heterogeneous Architectures ». In : *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society. 2017, p. 83-83 (cf. p. 166).

[ROF92] Leonid I Rudin, Stanley Osher et Emad Fatemi. « Nonlinear total variation based noise removal algorithms ». In : *Physica D : nonlinear phenomena* 60.1-4 (1992), p. 259-268 (cf. p. 74).

[S51] H. W. S. « The Perception of the Visual World ». In : *Journal of Philosophy* 48.25 (1951), p. 788 (cf. p. 9).

[STA20] Alvaro SALINAS, Claudio TORRES et Orlando AYALA. « A fast and efficient integration of boundary conditions into a unified CUDA Kernel for a shallow water solver lattice Boltzmann Method ». In : *Computer Physics Communications* 249 (2020), p. 107009 (cf. p. 64).

[SMF13] Javier SÁNCHEZ PÉREZ, Enric MEINHARDT-LLOPIS et Gabriele FACCIOLO. « TV-L1 Optical Flow Estimation ». In : *Image Processing On Line* 3 (2013), p. 137-150 (cf. p. 34, 75).

[SAU19] Umme SARA, Morium AKTER et Mohammad Shorif UDDIN. « Image quality assessment through FSIM, SSIM, MSE and PSNR—a comparative study ». In : *Journal of Computer and Communications* 7.3 (2019), p. 8-18 (cf. p. 167).

[Sat+10] Nadathur SATISH, Changkyu KIM, Jatin CHHUGANI, Anthony D NGUYEN, Victor W LEE, Daehyun KIM et Pradeep DUBEY. « Fast sort on CPUs and GPUs : a case for bandwidth oblivious SIMD sort ». In : *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 2010, p. 351-362 (cf. p. 30).

[SES12] Tobias SENST, Volker EISELEIN et Thomas SIKORA. « Robust local optical flow for feature tracking ». In : *IEEE Transactions on Circuits and Systems for Video Technology* 22.9 (2012), p. 1377-1387 (cf. p. 165).

[SM22] Olivier SENTIEYS et Daniel MENARD. « Customizing Number Representation and Precision ». In : *Approximate Computing Techniques : From Component-to Application-Level*. Springer, 2022, p. 11-41 (cf. p. 26).

[Sez+22a] Mickael SEZNEC, Agathe ARCHET, Nicolas GAC, François ORIEUX et Alvin Sashala NAIK. « MobileFlow : modèle et mise en œuvre pour une inférence de flot optique efficace ». In : *28eme Colloque GRETSI Traitement du Signal & des Images*. 2022 (cf. p. 33, 166).

[Sez+22b] Mickael SEZNEC, Nicolas GAC, François ORIEUX et Alvin Sashala NAIK. « Real-time optical flow processing on embedded GPU : an hardware-aware algorithm to implementation strategy ». In : *Journal of Real-Time Image Processing* 19.2 (2022), p. 317-329 (cf. p. 32).

[She+21] Dev Yashpal SHETH, Sreyas MOHAN, Joshua L VINCENT, Ramon MANZORRO, Peter A CROZIER, Mitesh M KHAPRA, Eero P SIMONCELLI et Carlos FERNANDEZ-GRANDA. « Unsupervised deep video denoising ». In : *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2021, p. 1759-1768 (cf. p. 34).

[Sun+18a] Deqing SUN, Xiaodong YANG, Ming-Yu LIU et Jan KAUTZ. « Pwc-net : Cnns for optical flow using pyramid, warping, and cost volume ». In : *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, p. 8934-8943 (cf. p. 33).

[Sun+18b] Shuyang SUN, Zhanghui KUANG, Lu SHENG, Wanli OUYANG et Wei ZHANG. « Optical flow guided feature : A fast and robust motion representation for video action recognition ». In : *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, p. 1390-1399 (cf. p. 9).

[SBK10] Narayanan SUNDARAM, Thomas BROX et Kurt KEUTZER. « Dense point trajectories by gpu-accelerated large displacement optical flow ». In : *European conference on computer vision*. Springer. 2010, p. 438-451 (cf. p. 32, 33).

[Tad20] Claude TADONKI. « OpenMP Parallelization of Dynamic Programming and Greedy Algorithms ». In : *arXiv preprint arXiv :2001.07103* (2020) (cf. p. 46).

[Tad+12] Claude TADONKI, Lionel LACASSAGNE, Elwardani DADI et Mostafa El DAOUDI. « Accelerator-based implementation of the Harris algorithm ». In : *International Conference on Image and Signal Processing*. Springer. 2012, p. 485-492 (cf. p. 55).

[Tav+18] Gemma TAVERNI, Diederik Paul MOEYS, Chenghan LI, Celso CAVACO, Vasyl MOTSNYI, David San Segundo BELLO et Tobi DELBRUCK. « Front and back illuminated dynamic and active pixel vision sensors comparison ». In : *IEEE Transactions on Circuits and Systems II : Express Briefs* 65.5 (2018), p. 677-681 (cf. p. 2).

[TD20] Zachary TEED et Jia DENG. « RAFT : Recurrent all-pairs field transforms for optical flow ». In : *European conference on computer vision*. Springer. 2020, p. 402-419 (cf. p. 33).

[Vau+22] J. VAUBAILLON, C. LOIR, M. MILLET, C. CIOCAN, M. KANDEEPAN, A. CASSAGNE, L. LACASSAGNE, P. Da FONSECA, F. ZANDER, D. BUTTSWORTH, S. LEOHLE, J. TOTH, S. GRAY, A. MOINGEON et N. RAMBAUX. « A 2022 t-Herculids meteor cluster ». In : *International Meteorix Conference (IMC)*. 2022 (cf. p. 142).

[WKC94] Dan S WALLACH, Sharma KUNAPALLI et Michael F COHEN. « Accelerated MPEG compression of dynamic polygonal scenes ». In : *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. 1994, p. 193-196 (cf. p. 9).

[WXC17] Jie WANG, Xinfeng XIE et Jason CONG. « Communication optimization on GPU : A case study of sequence alignment algorithms ». In : *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE. 2017, p. 72-81 (cf. p. 62).

[Wed+09] Andreas WEDEL, Thomas POCK, Christopher ZACH, Horst BISCHOF et Daniel CREMERS. « An improved algorithm for TV-L1 optical flow ». In : *Statistical and geometrical approaches to visual motion analysis*. Springer, 2009, p. 23-45 (cf. p. 32, 166).

[Wer+09] Manuel WERLBERGER, Werner TROBIN, Thomas POCK, Andreas WEDEL, Daniel CREMERS et Horst BISCHOF. « Anisotropic Huber-L1 Optical Flow. » In : *BMVC*. T. 1. 2. 2009, p. 3 (cf. p. 32).

[Wro+19] Bartlomiej WRONSKI, Ignacio GARCIA-DORADO, Manfred ERNST, Damien KELLY, Michael KRAININ, Chia-Kai LIANG, Marc LEVOY et Peyman MILANFAR. « Handheld multi-frame super-resolution ». In : *ACM Transactions on Graphics (ToG)* 38.4 (2019), p. 1-18 (cf. p. 9).

[Ye+12] H YE, Lionel LACASSAGNE, Daniel ETIEMBLE, Laurent CABARET, Joel FALCOU, A ROMERO et Olivier FLORENT. « Impact of high level transforms on high level synthesis for motion detection algorithm ». In : *Proceedings of the 2012 Conference on Design and Architectures for Signal and Image Processing*. IEEE. 2012, p. 1-8 (cf. p. 46).

[Ye+13] H YE, Lionel LACASSAGNE, Joël FALCOU, Daniel ETIEMBLE, Laurent CABARET et Olivier FLORENT. « High level tranforms to reduce energy consumption of signal and image processing operators ». In : *2013 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. IEEE. 2013, p. 247-254 (cf. p. 46).

[ZPB07] Christopher ZACH, Thomas POCK et Horst BISCHOF. « A duality based approach for realtime TV-L1 optical flow ». In : *Proceedings of the 29th DAGM Conference on Pattern Recognition (DAGM GCPR)*. 2007, p. 214-223 (cf. p. 30-32, 74, 86).

[ZPM18] Hamid Reza ZOHOURI, Artur PODOBAS et Satoshi MATSUOKA. « Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL ». In : *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2018, p. 153-162 (cf. p. 98).

Table des figures

Liste des tableaux

Liste des Algorithmes