Débruitage vidéo temps réel pour systèmes embarqués

Andrea Petreto

Laboratoire d'Informatique de Paris 6

Architecture et Logiciels pour Systèmes Embarqués sur Puce

Débruitage Vidéo Temps Réel pour Systèmes Embarqués

Auteur :
Andrea PETRETO

Thèse pour l'obtention du grade de :
Docteur en informatique et systèmes embarqués
soutenue le 29 juin 2020

Composition du jury :

Mme Corinne ANCOURT Directeur de Recherche, CRI - MINES ParisTech	Rapporteur
M. Daniel MÉNARD Professeur, IETR - INSA Rennes	Rapporteur
M. Daniel ETIEMBLE Professeur, LRI - Université Paris Saclay	Examinateur
M. Pierre SENS Professeur, LIP6 - Sorbonne Université	Examinateur
M. Lionel LACASSAGNE Professeur, LIP6 - Sorbonne Université	Directeur de thèse
M. Quentin MEUNIER Maître de Conférences, LIP6 - Sorbonne Université	Encadrant

Résumé

Beaucoup d'applications embarquées reposent sur le traitement ou la visualisation de vidéos. Dès lors, la présence de bruit dans les vidéos devient un problème majeur pour ce type d'applications. Des algorithmes de débruitage existent dans la littérature. La plupart sont qualitativement très efficaces mais au prix d'un temps de traitement trop important pour pouvoir envisager une implémentation temps réel embarquée. D'autres méthodes, plus rares, peuvent être exécutées en temps réel. Les applications de ces dernières sont cependant limitées. On remarque notamment que les méthodes les plus rapides gèrent mal les forts niveaux de bruit.

Le besoin de conserver une bonne qualité d'images, malgré des conditions très défavorables, reste cependant important pour bon nombre d'applications notamment de défense. Les contraintes d'embarquabilité y sont souvent également importantes. Dans ces travaux, nous cherchons à proposer une solution de débruitage vidéo qui permet un traitement en direct sur des systèmes embarqués. La solution proposée doit rester efficace même pour de forts niveaux de bruit. Ici, nous nous limitons à l'utilisation de CPU embarqués d'une consommation inférieure à 30W.

Les travaux menés dans le cadre de cette thèse ont permis la mise en place d'une chaîne de débruitage nommée RTE-VD (*Real-Time Embedded Video Denoising*). RTE-VD se décompose en trois étapes. La première, est une étape de stabilisation qui permet de compenser le mouvement de caméra. La deuxième étape correspond à un recalage temporel des images consécutives entre elles par estimation dense du flot optique. La dernière étape est l'application d'un filtre spatio-temporel qui permet la réjection finale du bruit. Sur un CPU embarqué (Jetson AGX), RTE-VD permet de traiter à une cadence de 30 images par seconde, des vidéos au format qHD (960×540 pixels). Afin de pouvoir atteindre ces performances, de nombreux compromis et optimisations ont dû être faits.

Dans un premier temps, nous rappelons des notions importantes d'optimisations architecturale et d'ingénierie algorithmique, indispensables pour la compréhension des différentes transformations apportées à la chaîne de traitement. Dans un deuxième temps, nous détaillons les trois principales étapes de RTE-VD. Pour chaque étape, nous détaillons la résolution mathématique associée, les optimisations spécifiques apportées et leur impact. Dans un troisième temps, nous présentons les performances obtenues avec la chaîne de traitement complète. Pour se faire, nous comparons RTE-VD à d'autres méthodes de référence de la littérature. La comparaison se fait à la fois en termes de qualité du débruitage et en termes de vitesse d'exécution. Nous montrons que RTE-VD apporte un nouveau positionnement particulièrement pertinent au niveau du rapport qualité/-vitesse. Enfin, une étude de la vitesse de traitement et de l'énergie consommée est faite pour différentes plateformes embarquées et à différentes fréquences de fonctionnement.

À Agathe.

Remerciements

Je tiens dans un premier temps à remercier l'ensemble des membres de mon jury : Corinne Ancourt et Daniel Ménard en qualité de rapporteurs ainsi que Daniel Etiemble et Pierre Sens en tant qu'examinateurs. Je vous remercie d'avoir accepté d'évaluer mes travaux et du temps que vous m'avez accordé.

Je souhaite également grandement remercier mon directeur de thèse, Lionel Lacassagne, sans qui je n'aurai jamais pu postuler à ce sujet de thèse passionnant. Je te remercie d'avoir encadré cette thèse avec un investissement et une disponibilité permanente ainsi que pour ces nombreuses réunions professionnelles qui pouvaient avoir tendance à dériver vers d'autres sujets qui nous passionnent tous les deux.

Merci aussi à Quentin Meunier, d'avoir participer à l'encadrement de ces travaux. Je te remercie notamment de ta disponibilité et particulièrement lors des différentes phases intenses de relecture d'articles ou du manuscrit. Je pense que mes rédactions ont beaucoup profité de ta rigueur et de ta précision.

J'ai bien conscience que cela ne sera pas bon pour ta modestie mais merci Boris Gaillard de l'aide que tu as pu me fournir en me partageant une partie de ton génie qui s'étend à de nombreux domaines. J'apprécie beaucoup nos débats, scientifiques ou pas, même si nous ne tombons pas toujours d'accord.

Je tiens également à remercier Patrice Ménard et Pascal Dupuy de m'avoir fait confiance pour mener ces travaux. Je vous remercie de m'avoir permis d'effectuer mon doctorat dans les meilleures conditions possible alors qu'il s'agissait de la première thèse effectuée chez LHERITIER. J'ai beaucoup apprécié la liberté qui m'a été accordée dans mes recherches malgré des objectifs industriels évidents.*

J'aimerai également remercier mes tuteurs DGA : Marie-Véronique Serfaty et Philipe Adam. Merci de m'avoir suivi et soutenu tout au long de cette thèse.

Je souhaite bien sûr également remercier l'ensemble des personnes avec qui j'ai pu travailler ou évoluer tout au long de cette thèse que ce soit au LIP6 ou chez LHERITIER. Je pense tout particulièrement à Ian Masliah et Florian Lemaitre qui m'ont beaucoup aidé et surtout beaucoup appris, mais également à Thomas Romera qui a été obligé (et continuera de l'être) de partager son bureau avec le mien, ce qui, il faut l'admettre, peut s'avérer parfois fatiguant. Mais aussi Gaëlle Clot et Bertrand Hardy-Baranski qui ont accepté d'aller faire des allers-retours devant une caméra, parfois en plein milieu de la nuit dans le seul but de me fournir des séquences vidéo exploitables. C'est aussi le cas de Sébastien Perot qui, en plus de cela, est l'un des rares à avoir accepté de me suivre dans beaucoup de mes idées farfelues comme participer à une Spartan Race. Merci Jérôme Bondouy pour être la seule personne avec laquelle on peut parler de façon sérieuse et approfondie sur des sujets importants comme Star Wars ou le MCU.

Je remercie aussi ma belle-famille Catherine, Hervé, Ariane et Alexiane Lebrun pour tout le soutien que vous avez pu m'apporter que ce soit directement ou indirectement en apportant votre soutien à Céline.

Un immense merci à ma famille : le clan des Petreto. Merci Maman pour ton soutien indéfectible depuis toujours, même dans cette thèse pour laquelle tu as accepté de relire mon manuscrit alors que le sujet est particulièrement éloigné de tes domaines de prédilections. Merci Papa, pour m'avoir transmis le goût des sciences et ce besoin d'aller plus loin, merci aussi pour toutes ces questions techniques que tu m'as posé pour finir par essayer de m'expliquer ce que je venais de t'expliquer parce que je "n'avais rien compris". Merci aux deux meilleures sœurs du monde ! Alexandra, ma grande sœur préférée, la plus forte de nous, tu as su me faire part de ton expérience et de tes conseils même lorsque tu étais atterrée par ma façon de faire ; et Anna, ma petite sœur préférée, le second opossum du clan, qui a mis un point d'honneur comprendre ce que je fais dans la vie même si cela mène parfois à des raccourcis...surprenants. Merci Olivier pour être, bien que plus discret, sûrement aussi déjanté que moi : tout le monde ne participerai pas avec autant d'entrain à des joutes nautiques en paddleboard à coup de frites... Merci Paul-Arthur Sol pour tes blagues à tomber par terre... (Et aussi pour m'autoriser à accessoiriser ta voiture).

Enfin, je souhaite par-dessus tout remercier ma compagne Céline. Merci ma chérie d'avoir su me supporter dans tous les sens du terme pendant ces trois années... et quelques. Tu me complètes et es toujours là pour moi y compris dans les moments difficiles. Merci simplement d'être toi et promis : je n'en ferai pas une deuxième !

Découverte parallèle faite durant cette thèse

En rapport avec les différents compromis et optimisations que l'on peut trouver dans ces travaux de thèse il est intéressant de s'intéresser au problème du temps, de l'énergie consommée et de la qualité pour d'autres applications que celle considérée dans ce manuscrit. On peut notamment s'attarder sur le problème de l'alimentation et plus particulièrement celui de l'alimentation par les lentilles. L'expérience a en effet montré que, la réduction du temps de cuisson des lentilles par rapport à celui indiqué sur la boîte, permet un gain de temps significatif ainsi qu'une économie d'énergie substantielle au niveau de la plaque de cuisson sans que l'objectif premier n'en soit pour autant affecté : ça croque mais ça nourrit. Certaines personnes, probablement jalouses de cette découverte, appellent cela des "lentilles pas cuites ou ratées" mais je pense qu'il serait préférable et plus juste d'appeler cela "lentilles *al dente*".

Table des matières

1 Introduction **5**
- 1.1 Contexte . . . 5
- 1.2 État de l'art . . . 7

2 Architecture des processeurs et optimisations **13**
- 2.1 Architecture et parallélisme . . . 13
  - 2.1.1 Parallélisme MIMD . . . 14
  - 2.1.2 Parallélisme SIMD . . . 14
  - 2.1.3 Précision des calculs et parallélisme SIMD . . . 16
- 2.2 Transformations algorithmiques . . . 17
  - 2.2.1 Fusion d'opérateurs . . . 19
  - 2.2.2 Pipeline d'opérateurs . . . 21
- 2.3 Vecteurs de Iliffe et allocation modulaire . . . 23
  - 2.3.1 Vecteurs de Iliffe . . . 23
  - 2.3.2 Allocation modulaire . . . 24
- 2.4 Synthèse . . . 31

3 Chaîne de traitement : RTE-VD **33**
- 3.1 Présentation . . . 33
- 3.2 Stabilisation . . . 36
  - 3.2.1 Motivations et choix de la méthode . . . 36
  - 3.2.2 Description de l'algorithme . . . 37
  - 3.2.3 Implémentation et optimisations . . . 44
  - 3.2.4 Résultats et performances de StabLK . . . 52
  - 3.2.5 Synthèse : StabLK . . . 53
- 3.3 Recalage temporel par calcul de flot optique dense . . . 54
  - 3.3.1 Motivations et choix de la méthode . . . 54
  - 3.3.2 Description de l'algorithme . . . 58
  - 3.3.3 Paramétrage de TV-L^1 . . . 64
  - 3.3.4 Optimisation d'implémentation de TV-L^1 . . . 74
  - 3.3.5 Résultats et performances de TV-L^1 . . . 84
  - 3.3.6 Synthèse : TV-L^1 . . . 100
- 3.4 Filtrage spatio-temporel . . . 102

3.4.1 Motivations et choix de la méthode . . . 102
3.4.2 Description de l'algorithme . . . 103
3.4.3 Implémentation et optimisations . . . 104
3.4.4 Résultats . . . 105
3.4.5 Synthèse : Filtrage spatio-temporel . . . 107
3.5 Synthèse : Chaîne de traitement RTE-VD . . . 108

4 Mise en place de RTE-VD et résultats 109
4.1 Paramétrage de RTE-VD . . . 109
4.1.1 Configuration de TV-L^1 . . . 109
4.1.2 Paramétrage du filtrage . . . 115
4.2 Performance de RTE-VD : comparaison avec l'état de l'art . . . 120
4.2.1 Efficacité du débruitage . . . 121
4.2.2 Comparaison du temps de traitement . . . 126
4.2.3 Comparaison : synthèse . . . 127
4.3 Temps de calcul et consommation énergétique . . . 127
4.4 Projet VIRTANS . . . 131
4.5 Synthèse des résultats . . . 133

5 Conclusion et travaux futurs 135
5.1 Synthèse . . . 135
5.2 Critiques et travaux futurs . . . 138

Bibliographie 151

Glossaire 153

A Temps de calcul et consommation de TV-L^1 : Résultats complets 155

Chapitre 1

Introduction

1.1 Contexte

Les travaux de cette thèse on été initiés par la société LHERITIER et le Laboratoire d'Informatique de Paris 6 (LIP6) dans le cadre d'une thèse CIFRE défense. LHERITIER est une société spécialisée dans la conception de modules caméra hautes performances. Les capteurs utilisés fonctionnent dans les bandes spectrales du visible (VIS). Au sein du LIP6, c'est l'équipe ALSOC (Architecture et Logiciels pour Systèmes Embarqués sur Puce) qui est impliquée. Cette équipe est spécialisée dans la conception des systèmes multiprocesseurs intégrés sur puce ainsi que la conception et le déploiement d'algorithmes sur architectures parallèles. L'objectif du projet est de concevoir et développer un nouvel algorithme pour le débruitage vidéo en direct, sur systèmes embarqués.

Beaucoup d'applications embarquées s'appuient sur du traitement ou une visualisation de vidéos. Les systèmes sont souvent équipés d'une ou plusieurs caméras pour percevoir leur environnement ou retransmettre l'information à un opérateur. Dans certains cas, les conditions de prises de vue peuvent cependant s'avérer particulièrement défavorables. Parmi celles-ci, on retrouve notamment les conditions de Bas Niveau de Lumière (BNL) et de faibles contrastes (par exemple à cause du brouillard). Lorsque ces conditions sont trop mauvaises, la qualité des images est dégradée par une trop forte présence de bruit. Ce bruit peut prendre de l'importance jusqu'à rendre la vidéo difficilement voire totalement inexploitable.

Pour certaines applications, il est critique de pouvoir continuer à produire des vidéos exploitables même dans les conditions les plus défavorables. Les véhicules autonomes, par exemple, doivent être capables de rouler par tous les temps. Les applications de surveillance et de défense, impliquent également un fort besoin d'observations en conditions difficiles. Dans ces cas, les capteurs fonctionnant dans les bandes spectrales du proche infrarouge (NIR), de l'infrarouge ondes courtes (SWIR), moyennes ondes (MWIR) ou grandes ondes (LWIR), sont parfois préférés à ceux fonctionnant dans le visible. Les technologies infrarouges permettent en effet souvent des observations longues distances plus aisées et sont moins affectées par les conditions de luminosité ou climatiques.

Le visible présente cependant plusieurs avantages majeurs, en particulier pour les applications de défense. Tout d'abord, le prix des capteurs VIS est bien plus faible que celui des capteurs infrarouges. Les technologies infrarouges (surtout MWIR et LWIR) ont également souvent besoin d'être refroidies pour améliorer leur performances. Cela entraîne l'utilisation de machines à froid qui sont à la fois bruyantes et consommatrices en énergie. En outre, s'il est vrai que l'observation dans les domaines de l'infrarouge peut présenter de meilleures performances de détection et de reconnaissance, il en va différemment pour l'identification.

Le critère de la Détection Reconnaissance et Identification (DRI) est souvent utilisé pour qualifier un système d'observation. Si le système permet de signaler la présence d'un objet sans donner d'information sur sa nature exacte, il est détecté. Si les informations données permettent de définir la nature de l'objet détecté, il est alors reconnu : véhicule ou être vivant, avion ou véhicule roulant, camion ou voiture etc. Enfin, si le système permet de connaître le modèle, la marque, l'identité ou tout autre signe caractéristique d'un objet ou d'un individu, alors ce dernier est identifié. L'œil humain est naturellement sensible dans le visible. Dans l'infrarouge, les inversions d'albédos perturbent nos capacités d'observation et c'est pourquoi les technologies du visible permettent une meilleure identification.

Les technologies VIS, et infrarouge sont donc complémentaires. L'amélioration des systèmes fonctionnant dans le visible a par conséquent un intérêt majeur pour repousser les limites de DRI et plus particulièrement d'identification. Les capacités de DRI peuvent être limitées par différents facteurs comme le bruit ou les perturbations atmosphériques pour les observations à longues distances. Dans ces travaux, nous nous concentrons sur la réduction du bruit. Jusqu'ici, un moyen d'améliorer les performances d'un module caméra était de concevoir des capteurs de plus en plus performants avec un bruit de lecture toujours plus faible. De nos jours, les capteurs présentent cependant des caractéristiques proches d'un capteur "parfait" avec un bruit de lecture très faible. Le bruit prépondérant devient alors le bruit de photons. Le bruit de photons est un phénomène physique naturel et ne peut pas être évité.

Pour l'observation en BNL, il est également possible d'utiliser des optiques plus performantes avec une ouverture plus grande et des lentilles de taille supérieure. Cette solution a également ses limites étant donné l'encombrement et l'augmentation du poids que cela implique. Dans ces deux cas (changement de capteur ou d'optique), l'amélioration du matériel se traduit également par une augmentation significative du coup de production des systèmes.

Afin de dépasser les limites de la physique, la solution envisagée est d'effectuer un débruitage de la vidéo en direct via une chaîne de traitement logicielle. En plus d'être complémentaire des autres voies d'amélioration, cette méthode présente plusieurs avantages. Une fois développée, la chaîne de traitement a un surcout d'intégration quasiment nul. La chaîne de traitement peut être appliquée à un système existant pour améliorer ses performances. À performances équivalentes, il est possible de concevoir un dispositif plus compact, plus léger et moins cher.

Les algorithmes de débruitage vidéo requièrent cependant beaucoup de ressources de calculs. Ils sont généralement trop lents pour pouvoir envisager une implémentation en temps réel sur des systèmes embarqués légers. Les rares méthodes qui le permettent ne sont pas adaptées à des niveaux de bruits importants. Dans ces travaux, nous présentons une nouvelle chaîne de traitement pour le débruitage temps réel embarqué de vidéos fortement bruitées. Le terme de temps réel est ici employé comme un abus de langage par rapport à sa définition première. Nous considérons qu'un traitement est temps réel s'il peut être exécuté en direct avec une cadence minimum de 25 images par seconde, soit un temps de traitement maximum de 40 ms. Nous ne considérons pas un temps réel dur pour lequel la contrainte de temps est sûre : nous utilisons un systèmes d'exploitation généraliste et n'avons pas le contrôle de l'ordonnancement ou des interruptions. Le terme de système embarqué est également vague et peut correspondre à une gamme très large de systèmes. Pour nos travaux, nous nous limitons à des systèmes avec une consommation électrique de quelques dizaines Watts. Tous les systèmes que nous appelons embarqués dans ce manuscrit respectent cette contrainte ($\approx$ 30,5 W max pour l'AGX, la plateforme la plus puissante).

1.2 État de l'art

Le débruitage est un sujet majeur du traitement d'images et de la vision par ordinateur. De nombreuses méthodes existent et peuvent être classées en différentes catégories.

Les méthodes à base de filtrage point à point [1, 2, 3] permettent d'appliquer une fonction de pondération à l'ensemble des pixels d'une image. Généralement, la valeur du pixel de sortie est une valeur réévaluée en fonction de celles du pixel d'entrée et de son voisinage. Parmi ce type de filtre, on retrouve notamment les filtres moyenneur, gaussien, médian [4, 2] ou encore bi-latéral [5]. On trouve également beaucoup de méthodes basées sur la transformée en ondelettes [6, 7, 8]. Ce type d'approche est particulièrement populaire dans les applications de compression et de décompression d'images. Les filtres point à point et à base de transformées en ondelettes ont une approche locale du débruitage. Ces deux types de méthodes utilisent le postulat de similarité entre les pixels voisins. Elles permettent une réduction significative du bruit et sont capables de produire des images avec un niveau de PSNR (*Peak Signal to Noise Ratio*) élevé. En revanche, avec un telle approche on observe une perte importante des détails au sein des images [9].

Pour résoudre ce problème une nouvelle catégorie de méthode a été introduite en 2005 : les méthodes non locales. L'idée est de ne plus considérer le postulat de similarité de façon locale mais de considérer qu'il peut se trouver éloigné dans l'image. Cela permet de tirer parti de la redondance des structures souvent présentes dans les images naturelles. Les méthodes non locales ont été introduites pour la première fois avec l'algorithme *NL-means* (*Non Local Means*) dans [10]. Le principe est de pondérer la valeur d'un pixel non pas avec celles des pixels de son voisinage direct mais avec des pixels

similaires dans une fenêtre de recherche plus grande (pouvant aller jusqu'à l'image complète). La similarité entre 2 pixels p_1 et p_2 est évaluée en fonction de la différence de leur intensité mais également en fonction de la différence de valeur entre leur voisinages respectifs. On dit alors qu'on évalue la similarité des patchs. Un patch représente un pixel et son voisinage. Plus un patch est similaire à un autre, plus il sera pris en compte lors de la pondération. Dans le cas de *NL-means*, la similarité est donnée par la distance euclidienne entre les patchs : plus celle-ci est petite, plus la similarité est grande.

Cette méthode requiert un grand temps de calcul mais les résultats obtenus sont de meilleure qualité que ceux obtenus avec des filtrages point à point ou par transformée en ondelettes. Pour cette raison, de nombreuses autres méthodes de débruitage non locales ont été proposées depuis. Beaucoup s'appuient sur *NL-means* : en ajoutant une étape supplémentaire pour améliorer la robustesse de la méthode [11], en proposant des moyens de réduire le temps de calcul [12, 13, 14], ou en proposant de nouvelles métriques pour la similarité [15, 16]. Parmi les méthodes non locales présentées après *NL-means*, BM3D [9] est sans doute l'une des plus populaire.

Une description plus détaillée que l'article originel, ainsi qu'une implémentation libre de BM3D sont fournis par Lebrun [17]. Cette méthode met en place deux passes de filtrage collaboratif par *block matching*. Le filtrage collaboratif est lui même composé de 4 étapes :

1. **Groupement de blocs :** À partir d'un patch de l'image, trouver un ensemble de patchs similaires et les stocker dans un bloc 3D. Dans ce cas, contrairement à *NL-mean*, tous les patchs ne sont pas pris en compte. Seuls les plus similaires sont conservés.
2. **Transformation linéaire 3D :** Cette transformation est appliquée au bloc de patchs 3D précédemment construit. La nature de la transformation peut varier : transformée en cosinus discrète (DCT) [18], transformée de fourrier discrète (DFT) [19], transformée de Walsh-Hadamard [20]... Le choix de la transformation utilisée influe sur la qualité des résultats et dépend de la passe en cours.
3. **Réduction des coefficients du spectre de la transformation :** À la première passe de BM3D la réduction est appliquée par une fonction de seuillage stricte tandis qu'un filtre de Wiener est appliqué lors de la seconde passe. Cette étape permet de filtrer chacun des patchs 2D du bloc 3D.
4. **Transformation linéaire 3D inverse :** Cette étape permet de transposer à nouveau les patchs 2D filtrés dans le domaine initial.

Chaque patch ainsi filtré est restitué à son emplacement initial. Le filtrage collaboratif étant appliqué pour chaque pixel, chaque patch sera à priori filtré plusieurs fois ainsi que d'autres patchs se chevauchant. Il est donc nécessaire de procéder à une phase d'agrégation des résultats du filtrage. Pour cette agrégation, un poids de pondération est appliqué à chaque patch en fonction de sa similarité avec le patch central qui a été considéré lors de sa génération.

Les deux passes de BM3D utilisent le filtrage collaboratif mais diffèrent surtout par le choix des méthodes choisies en étape 1 et 3 du filtrage collaboratif. Lors de la première passe de BM3D, le groupement de blocs se fait sur l'image d'origine et une fonction de seuillage est utilisée pour l'étape de réduction des coefficients. Lors de la seconde passe, le groupement de blocs se fait sur l'image d'origine mais à partir des similitudes mesurées pour des patchs issus de l'image filtrée obtenue précédemment. La réduction des coefficient du spectre de la transformation est obtenue en appliquant un filtre de Wiener.

Les méthodes non locales ont ainsi permis une amélioration significative de la qualité de débruitage envisageable en traitement des images. Des méthodes récentes à base d'apprentissage et plus particulièrement à base de réseaux de neurones convolutifs profonds (CNN) ont récemment permis d'atteindre voir de dépasser les performances des méthodes non locales [21, 22, 23, 24].

Si la littérature est particulièrement fournie concernant le débruitage d'images seules, il n'en est pas de même pour le débruitage de vidéos. Malgré un regain récent d'intérêt dans la littérature, le nombre de méthodes de débruitage adaptées aux vidéos est en effet bien moins important. La vidéo apporte pourtant une dimension temporelle supplémentaire qui peut s'avérer particulièrement intéressante dans le cadre de la réduction du bruit. Le bruit étant, un évènement transitoire à la fois spatialement et temporellement, il est possible, avec une vidéo, d'exploiter la redondance de l'information au cours du temps.

Les méthodes de débruitage vidéo sont généralement issues de méthodes initialement pensées pour les images seules et modifiées pour être adaptées aux vidéos. On retrouve notamment VBM3D [25] qui correspond à la version Vidéo de BM3D ainsi que sa variante VBM4D [26]. *NL-means* peu également être étendue à la vidéo [13]. Si VBM3D est une référence en matière de qualité de débruitage vidéo dans la littérature, d'autres méthodes plus récentes permettent d'obtenir de meilleurs résultats [27, 28, 29, 30]. Beaucoup de ces méthodes sont des méthodes par patchs. Si, dans le cas des vidéos, les méthodes à base de réseaux de neurones ne parvenaient pas dans un premier temps à rester compétitives avec les autres méthodes non locales, on constate la recrudescence de ce genre de méthodes dans la littérature récente [29, 30, 31].

La principale difficulté dans le traitement d'une vidéo par rapport au traitement d'une image seule, est l'exploitation de la dimension temporelle. Si l'on filtre la valeur d'un pixel au cours du temps, il est important de s'assurer que l'objet représenté par ce dernier, est bien le même qu'aux images précédentes. Dans le cas contraire, il est nécessaire de suivre l'objet pour savoir par quel pixel il est désormais représenté. Cela permet d'empêcher l'apparition de flou de bougé lorsqu'un objet se déplace. Pour assurer cette cohérence temporelle, plusieurs méthodes peuvent être employées : la recherche 3D de patchs [13, 32], l'estimation du flot optique (mouvement apparent entre deux images) [33] ou encore un filtre de Kalman pour les mouvements de faible amplitude [34, 35].

Les méthodes qui permettent d'obtenir les meilleurs résultats de l'état de l'art en termes de qualité de débruitage, exploitent la dimension temporelle soit par une approche non locale soit par une estimation du flot optique soit par une association des deux [30, 36, 29, 28, 27, 26, 25]. Ces chaînes de débruitage requièrent cependant un temps de calcul particulièrement long. Pour l'ensemble de ces méthodes, la cadence de traitement est inférieure à une image par seconde pour des architectures avec des puissances de calcul supérieures à celles que nous considérons pour ces travaux. Il n'est donc pas possible d'envisager une quelconque implémentation temps réel embarquée sans changer les algorithmes de façon significative.

Le problème du temps traitement trop long pour le débruitage de vidéos est abordé dans plusieurs travaux de la littérature [31, 37, 35, 38]. Les temps de traitement d'une image dépassant la minute pour certaines des méthodes les plus complexes [28], certains travaux récents ont tenté de proposer de nouvelles méthodes de débruitage efficaces dans des temps de calcul plus restreints. Deux méthodes de débruitage temps réel adaptées au systèmes embarqués ont notamment été proposées. La première introduite par Pfleger *et al.* en 2017, est appelée STMKF [35]. La seconde a été introduite par Ehmann *et al.* en 2018 [37], nous l'appellerons GLPD (*Gaussian Laplacian Pyramid Decomposition*). Si ces méthodes sont beaucoup plus rapides que la plupart des autres méthodes, leur spectre d'application est, du propre aveu des ses auteurs, assez limité. Ces méthodes sont essentiellement destinées à des séquences avec des mouvements de faibles amplitudes et dans un environnement fixe comme pour une caméra de surveillance. Les niveaux de bruits considérés sont également plus faibles que ceux que nous considérons dans ces travaux.

Si pour STMKF et GLPD, la vitesse de calcul prime sur la qualité du débruitage, les méthodes BNLK [39] et FastDVDnet [31] introduites en 2019 tentent d'accélérer le temps de traitement sans rogner de façon significative sur la qualité. Les résultats obtenus, bien qu'effectivement plus rapides, ne permettent cependant toujours pas d'envisager une implémentation temps réel embarquée. On peut noter toutefois une avancée majeure apportée par la méthode FastDVDnet, qui permet d'obtenir des résultats visuellement compétitifs (voire meilleurs que certaines des meilleures méthodes de l'état de l'art) pour un temps de traitement de 100ms par image qHD (960×540 pixels) sur un GPU Nvidia Titan Xp ($\approx$ 250 W). Cela demeure bien supérieur à 40 ms malgré une puissance de calcul et une consommation importantes.

On constate un trou dans le spectre des différentes méthodes de débruitage proposées. Il semble que les méthodes les plus efficaces en termes de qualité de débruitage, ont un temps de traitement largement supérieur aux 40 millisecondes requises pour un traitement en direct. Cela reste vrai même avec des puissances de calcul bien supérieures à celles que nous considérons dans ces travaux. D'un autre côté, les méthodes rapides existantes restent très limitées dans leurs applications. Ces méthodes ne sont pas adaptées aux très forts niveaux de bruit ni aux mouvements de trop forte amplitude.

L'objectif de ces travaux est d'apporter une réflexion sur les différents compromis possibles pour permettre un débruitage de vidéo temps réel sur des systèmes embarqués.

Les solutions proposées doivent permettre de rester efficace pour de forts niveaux de bruit. Dans un premier temps, et pour limiter la consommation des systèmes considérés, nous concentrons nos travaux sur une implémentation CPU de notre méthode. Nous proposons la chaîne de traitement RTE-VD (*Real-Time Embedded Video Denoising*). Nous montrons que cette méthode permet de traiter en temps réel des séquences vidéo au format qHD (960×540 pixels) avec le CPU embarqué de la plateforme Nvida Jetson AGX. Nous comparons également les performances de débruitage de RTE-VD à celles de STMKF, VBM3D et VBM4D. Par rapport à STMKF, notre méthode est 2,2 fois plus lente mais permet un débruitage significativement meilleur que ce soit sur des critères objectifs de PSNR que subjectifs. La qualité de débruitage de RTE-VD par rapport à celle de STMKF est d'autant meilleure, que le niveau de bruit est élevé. Par rapport à VBM3D (respectivement VBM4D) la qualité de débruitage proposée reste inférieure mais pour un temps de traitement 200 (respectivement 4600) fois inférieur.

Par rapport aux autres méthodes de la littérature, nous pensons que RTE-VD permet d'apporter un nouveau compromis pertinent entre la vitesse et la qualité de débruitage.

De nombreux compromis et transformations ont été appliqués à notre chaîne de traitement afin d'arriver aux performances énoncées. Dans le chapitre 2, nous présentons dans un premier temps, un panel d'optimisations architecturales et de transformations algorithmiques générales, appliqué pour chaque étape de RTE-VD. Ces optimisations, bien que faisant partie de l'état de l'art, ont un impact important sur les résultats présentés et sont donc présentées à titre pédagogique. Dans le chapitre 3, nous présentons et détaillons l'ensemble des étapes de RTE-VD. RTE-VD est une nouvelle chaîne de débruitage vidéo mise au point dans le cadre ces travaux de thèse. La première étape sert à la stabilisation globale de la vidéo et s'appuie sur une approche similaire au calcul de flot optique de Lucas-Kanade [40]. Cette méthode est inspirée de celle déjà implémentée dans les FPGA des caméras LHERITIER. Nous proposons ici, une implémentation optimisée sur CPU avec une nouvelle façon de paralléliser le calcul d'une somme sur un voisinage. La seconde étape effectue un recalage temporel entre deux images consécutives de la vidéo grâce à un calcul dense du flot optique par TV-L^1 [41]. D'importants efforts d'optimisation ont été nécessaires pour accélérer le calcul de TV-L^1. À notre connaissance, l'implémentation proposée ici est la plus rapide sur CPU de l'état de l'art avec une nouvelle façon de parcourir l'algorithme en "pipeline d'itérations". La troisième et dernière étape applique un filtrage spatio-temporel de type bilatéral [5] à partir des images recalées à l'étape précédente. Par rapport au filtrage bilatéral classique, nous intégrons un filtrage latéral temporel. Pour chaque étape, nous expliquons le choix de la méthode utilisée, nous décrivons la méthode en détail, ainsi que les différentes optimisations ou compromis appliqués. L'impact de ces optimisations est également discuté pour chaque étape. Enfin, dans le chapitre 4, nous discutons des performances générales de notre méthode que ce soit en termes de vitesse ou de qualité, et ce, par rapport à d'autres méthodes de l'état de l'art. Une étude du compromis entre la vitesse et la consommation d'énergie est également donnée pour différentes architectures embarquées. Nous concluons et présentons la suite des travaux envisagés dans le chapitre 5.

Chapitre 2

Architecture des processeurs et optimisations

Dans ce chapitre, nous introduisons les techniques d'optimisation et les transformations algorithmiques qui ont été appliquées à l'ensemble des maillons de la chaîne de débruitage. D'autres optimisations plus spécifiques à chaque étape seront abordées lors de la présentation de l'étape en question. Ces méthodes d'ingénierie algorithmique permettent de réduire le temps d'exécution et la consommation énergétique d'une application. Ces optimisations ont notamment été beaucoup étudiées dans le cas du filtre de Harris [42] : [43, 44, 45, 46, 47, 48, 49], mais également pour d'autres algorithmes de traitement d'images [50, 51, 52, 53, 54, 55, 56, 57, 58, 59]. Le traitement d'images et de vidéos est un domaine qui se prête particulièrement à ce type de transformations. C'est pourquoi, nous les reprenons et les adaptons à notre application.

Bien que les techniques d'optimisation présentées ici fassent partie de l'état de l'art [60, 61, 62, 63, 64, 65, 66], nous y consacrons un chapitre car la compréhension fine des différents termes les référençant nous semble primordiale pour appréhender les développements, mais aussi les résultats présentés dans la suite de cette thèse. En particulier, si les exemples présentés dans la suite ont une vocation pédagogique, ils laissent entrevoir ce que ces optimisations peuvent induire en complexité sur le code des applications.

2.1 Architecture et parallélisme

Dans cette section, nous présentons deux niveaux de parallélisme différents : MIMD (*Multiple Instruction on Multiple Data*) et SIMD (*Single Instruction on Multiple Data*). Pour ces travaux nous cherchons dans un premier temps à implémenter notre chaîne de traitement sur des processeurs généralistes, ou CPU, embarqués. L'utilisation d'architectures différentes, et plus particulièrement de processeurs graphiques (GPU), est envisagée comme futurs travaux. C'est pourquoi, nous nous concentrons ici uniquement sur les parallélismes présents dans la plupart des CPU modernes. Des outils existent

pour faciliter le déploiement d'applications sur architectures parallèles [61, 67, 68, 69]. On peut notamment noter les outils PIPS [70, 71] et GeCoS [64] qui permettent de profiter du parallélisme et apportent des transformations pour améliorer l'utilisation de la mémoire. Ils permettent un temps de développement souvent plus court qu'une optimisation manuelle. Certains permettent également de cibler plusieurs types d'architecture à la fois : CPU, GPU, FPGA... S'il est possible, grâce à ces outils, de gagner du temps et de faciliter l'utilisation d'architectures parallèles pour un plus grand nombre de personnes, les performances atteintes ne sont en générale pas au niveau de celles obtenues avec des optimisations manuelles [72]. Dans ces travaux nous cherchons avant tout à obtenir l'implémentation la plus rapide possible pour pouvoir faire des compromis sur la consommation énergétique. En outre, nous ne disposons pas de la maîtrise de la plupart de ces outils. C'est pourquoi, nous avons décidé de ne pas les utiliser.

2.1.1 Parallélisme MIMD

Le MIMD est souvent désigné par les termes de parallélisme de tâches ou multi-tâches. Il s'agit d'un des modèles de parallélisme définit dans la taxonomie de Flynn [60]. Le principe est d'exécuter plusieurs tâches en parallèle et de façon indépendante grâce aux architectures multi-processeurs ou multicœurs. Les architectures multicœurs étant aujourd'hui extrêmement répandues, le parallélisme MIMD est courant mais la parallélisation peut être difficile. Dans ces travaux, nous parallélisons un même programme de façon similaire sur l'ensemble des processeurs. On parle alors de SPMD (*Single Program on Multiple Data*). Le terme de SPMD est plus précis et plus répandu que celui de MIMD c'est pourquoi c'est celui que nous employons dans la suite de ces travaux. Les interfaces de programmation de parallélisation les plus utilisées sur CPU sont *Pthread* [73] et OpenMP [63]. Étant donné que nous maîtrisons mieux *OpenMP*, c'est l'interface que nous utilisons dans ces travaux pour paralléliser les algorithmes sur plusieurs cœurs.

Si la gestion du parallélisme de tâche est mal effectuée, celui-ci peut s'avérer pas ou peu efficace. Notamment, l'exécution de plusieurs tâches indépendantes mais nécessitant des données en commun est susceptible d'entraîner des accès simultanés sur une même donnée. L'accès à cette donnée se fait alors de façon séquentielle et les performances sont réduites. Pour éviter cela, il est nécessaire de s'assurer que les différentes tâches en parallèles accèdent le moins possible à des données identiques. On fait alors la différence entre le parallélisme de tâches : TLP (*Thread Leval Parallelism*) et le parallélisme de mémoire : MLP (*Memory Level Parallelism*). Lorsque l'on optimise plus particulièrement les accès à la mémoire cache, on parle de CLP (*Cache Level Parallelism*).

2.1.2 Parallélisme SIMD

Le SIMD est également un des modèles de parallélisme définit dans la taxonomie de Flynn. Avec ce modèle, une même instruction est exécutée sur plusieurs données à la fois. Ce type de parallélisme s'applique particulièrement bien aux traitements réguliers comme le traitement d'images ou de vidéos. L'utilisation du SIMD repose sur l'exploitation de l'ensemble de la largeur du jeu d'instructions d'un processeur. Il existe différents jeux

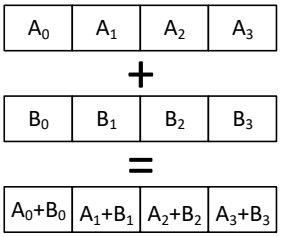


FIGURE 2.1 – Addition SIMD pour des registres de cardinal 4

d'instructions car chaque constructeur peut proposer le sien. Les plus utilisés sur CPU sont :

— chez Intel : SSE, AVX, AVX512
— chez ARM : NEON, SVE
— chez IBM : Altivec

Le nombre de données traitées en parallèle dépend de la largeur des registres utilisés par le jeu d'instruction - typiquement 64, 128, 256 ou 512 bits - et de la taille des données elles mêmes - typiquement 8, 16, 32 ou 64 bits. Un jeu d'instruction 128 bits permet, par exemple, de manipuler 4 données flottantes 32 bits. On parle alors d'un registre SIMD de cardinal 4. Autre exemple, un jeu d'instruction de 128 bits permet de manipuler 16 données entières 8 bits. Le cardinale du registre SIMD est alors 16. La figure 2.1 illustre l'exemple de l'addition de 2 registres SIMD A et B de cardinal 4. Par abus de langage on utilise également les termes de vecteurs et de programmation vectorielle, en opposition avec la programmation classique dite scalaire. Il est important de noter cependant, que les calculs vectoriels et SIMD sont légèrement différents. Lors d'un calcul SIMD, une même instruction est appliquée en même temps à plusieurs données. Pour un calcul vectoriel, une même instruction est appliquée de façon pipelinée à plusieurs données.

Le calcul régulier facilite l'utilisation du SIMD. Dans certains cas, les compilateurs sont susceptibles de générer automatiquement des instructions SIMD et de profiter de ses avantages. On appelle cela la vectorisation. En revanche, dans d'autres cas plus complexes, les transformations du compilateur s'avèrent inefficaces voire inexistantes, et il revient au développeur d'apporter les transformations nécessaires et d'implémenter lui même le code en SIMD. On appelle cela la SIMDisation. Dans ces travaux, la SIMDisation a systématiquement permis une accélération des traitement par rapport aux seules optimisations faites par le compilateur.

Par construction, le SIMD ne parallélise pas seulement les calculs mais également les accès mémoire. En effet, l'ensemble des données d'un registre SIMD sont chargées en une fois et ce, sans pénalité par rapport au chargement d'une seule donnée. Ce dernier point est vrai seulement si le jeu d'instruction du processeur et les instructions SIMD ont effectivement la même largeur. Certains processeurs, comme les architectures ARMv8-A par exemple, sont compatibles avec du SIMD 128 bits alors que leur jeu d'instruction est réellement sur 64 bits. Dans ce cas, l'instruction 128 bits est émulée et l'apport du SIMD, bien que réel, est moins important. Ce parallélisme introduit sur les accès mémoire

permet d'améliorer l'utilisation de la bande passante. Cette dernière est en effet connue pour être souvent le principal facteur limitant la vitesse d'exécution d'une application. L'accélération introduite avec le SIMD peut également être limitée de la même façon, par la largeur du bus mémoire si ce dernier ne permet pas d'accéder au registre complet en une passe.

Outre le parallélisme, les jeux d'instructions SIMD intègrent souvent des instructions spécifiques qui permettent d'accélérer les calculs. On peut noter par exemple les instructions dites de FMA (*Fused Multiply-Add* ou *Fused Multiply-Accumulate*) qui permettent en une seule instruction d'effectuer une opération du type $A \times B + C$. De même, l'approximation rapide de l'inverse de la racine carrée permet d'accélérer les calculs de racinées carrées si l'approximation est suffisamment précise pour l'application.

2.1.3 Précision des calculs et parallélisme SIMD

Comme nous avons pu le voir dans la section précédente, la taille des données utilisées, influe directement sur le cardinal des registres SIMD. Ainsi, en réduisant la dynamique des calculs, il est possible d'augmenter le parallélisme SIMD. Cela permet non-seulement d'augmenter le parallélisme mais également de réduire la bande passante nécessaire puisque les données sont de taille inférieure.

Considérons un jeu d'instruction sur 128 bits utilisé pour effectuer des calculs sur des flottants de 32 bits (F32). Le parallélisme SIMD est alors de 4. Si l'application et le processeur le permettent, il est possible de choisir d'effectuer des calculs en utilisant plutôt des flottants demi-précision codés sur 16 bits. Le cardinal des registres SIMD est alors de 8. Le parallélisme SIMD est dans ce cas 2 fois plus important. Il est aussi rapide d'accéder à 8 données de 16 bits qu'à 4 données de 32 bits. Il est toutefois primordial de s'assurer que la réduction de précision des données est possible pour l'application visée. Si la précision des résultats n'est pas suffisante ou si la dynamique des calculs est trop grande, il n'est pas possible de réduire la taille des données pour augmenter le parallélisme SIMD. Un résumé des plages de valeurs possibles et de la précision des calculs en flottants 32 et 16 bits est donné à titre indicatif dans le tableau 2.1.

Format	Signe	Exposant	Mantisse	Min (≥ 0)	Max	Précision	Entiers exacts
Float 32	1 bit	8 bits	23 bits	2^{-126}	$\approx 3,4 \times 10^{38}$	24bits	$\pm 1 \times [0, 16777216]$
Float 16	1 bit	5 bits	10 bits	2^{-14}	65504	11bits	$\pm 1 \times [0, 2048]$

TABLE 2.1 – Plage de valeurs et précision des données codées en flottants 32 et 16 bits. Dans le cas de valeurs normalisées avec le mode d'arrondi au plus proche.

Le besoin de précision est lié à l'application et peut faire l'objet d'un choix du développeur. Dans le cas du calcul de flot optique par exemple, l'amplitude des mouvements apparents dans la scène se mesure en nombre de pixels. Pour l'œil humain, il n'y aura pas de différence significative à la reconstruction si les vecteurs vitesse ont été estimés au dixième ou au centième de pixel près. Les résultats n'auront donc pas besoin d'être

très précis. En revanche, la dynamique nécessaire aux calculs peut également influer sur le format de calcul utilisé. Les calculs intermédiaires peuvent en effet nécessiter une plus grande dynamique que le résultat final. Prenons l'exemple du calcul de la moyenne sur un voisinage de plusieurs pixels avec pondération à la fin. Le résultat final sera de l'ordre de la valeur des pixels, en revanche il est d'abord nécessaire de calculer sur une dynamique suffisamment grande pour pouvoir faire l'accumulation des valeurs de tous les pixels. Des méthodes comme l'arithmétique d'intervalle [74] ou l'arithmétique affine [75] existent pour borner les calculs. Cela peut parfois permettre de réduire la taille des données utilisées en s'assurant l'impossibilité d'un dépassement de dynamique. Ces méthodes ne sont cependant pas applicables à tous les types d'algorithmes et il est parfois très difficile voire impossible de borner finement certaines méthodes numériques.

2.2 Transformations algorithmiques

La performance des applications est en général limitée par deux facteurs différents :

— La vitesse du processeur : le temps de traitement total est essentiellement limité par la vitesse de calcul du processeur. L'augmentation des performances du processeur ou la réduction du nombre de calculs suffisent à réduire le temps de traitement. En français on dit que le traitement est subordonné au temps de calcul [76], mais le terme de *compute bound* est généralement employé.

— La vitesse ou la quantité de mémoire : dans ce cas c'est essentiellement le nombre d'accès mémoire qui limite les performances du traitement. L'augmentation de la quantité de mémoire ou de la bande passante suffisent à accélérer le programme. On dit que l'application est *memory bound*.

Un accès en mémoire externe est plus lent qu'une opération de calcul sur le CPU. C'est pourquoi beaucoup d'applications ne sont pas *compute bound* mais *memory bound*. Le grand nombre d'accès mémoire requiert une bande passante importante. Dans ce cas, les parallélisations SIMD ou MIMD peuvent être inefficaces. En effet, si le bus mémoire est saturé alors les unités de calcul "attendent" que les données soient disponibles. Tant que les données ne sont pas présentes, pouvoir en calculer davantage en parallèle n'a pas ou peu d'impact sur le temps de traitement total. Il devient alors essentiel d'optimiser l'utilisation de la mémoire.

La mémoire des ordinateurs est étagée. Plus une donnée est mémorisée proche du processeur, plus y accéder est rapide. Un accès en registre est donc plus rapide qu'un accès en mémoire cache qui lui même est plus rapide qu'un accès en mémoire RAM. Les techniques d'optimisations présentées dans cette section s'appuient sur cette propriété afin d'accélérer le traitement. La figure 2.2 permet de se rendre compte de la différence de vitesse entre les accès en mémoire cache et ceux en mémoire externe. Nous y indiquons les résultats de l'ensemble des micro-tests de *STREAM* [77] pour l'une des plateformes que nous utilisons dans ces travaux. *STREAM* est un ensemble de micro-tests permettant d'évaluer la bande passante d'un processeur. En faisant varier la taille des données utilisées, nous pouvons déterminer la bande passante des différents niveaux de mémoire.

Les différents micro-tests de *STREAM* sont :

— ***Read*** **:** Mesure la bande passante en lecture.
— ***Write*** **:** Mesure la bande passante en écriture.
— ***Copy*** **:** Mesure la bande passante en recopie, opération de type $a(i) = b(i)$.
— ***Scale*** **:** Mesure la bande passante de l'opération de type $a(i) = q \times b(i)$.
— ***Add*** **:** Mesure la bande passante de l'opération de type $a(i) = b(i) + c(i)$.
— ***Triad*** **:** Mesure la bande passante de l'opération de type $a(i) = b(i) + q \times c(i)$.

Selon le type de données que l'on souhaite évaluer, il est possible d'utiliser différents formats de calcul pour ces tests. En l'occurrence, nous considérons les résultats pour des flottants 32 bits. Parmi l'ensemble des micro-tests de *STREAM*, *triad* est généralement le résultat le plus utilisé. C'est celui qui est le plus représentatif du maximum de ce que l'on peut espérer, dans le cas d'une application réelle.

Dans la figure 2.2, nous présentons les résultats obtenus pour le CPU de la plateforme Nvidia Jetson AGX. Il s'agit d'un CPU ARM possédant 8 cœurs Carmel et 3 niveaux de cache. On observe que, pour le micro-test *triad*, la bande passante mesurée est d'environ 800 Go/s en cache L1, 200 Go/s en L2, 130Go/s en L3 et 60 Go/s en mémoire externe. Soit une bande passante entre 2,2 et plus de 13 fois plus rapide lorsque les données sont en cache. On voit bien ici l'intérêt majeur de maximiser la persistance des données en cache afin d'accélérer les temps de traitement.

Les algorithmes requièrent généralement des résultats intermédiaires qui sont réutilisés plus tard avant de devenir inutiles. La sauvegarde de ces résultats augmente l'empreinte mémoire de l'application. Si un résultat intermédiaire n'est pas réutilisé rapidement après avoir été produit, il est probable que la donnée soit éjectée du cache. Cela entraîne un ralentissement important lorsque l'on souhaite y accéder de nouveau. Pour éviter cela au maximum nous utilisons deux transformations algorithmiques différentes :

— La fusion d'opérateurs : les résultats intermédiaires sont directement réutilisés sans être sauvegardés en mémoire (accès en registre).
— Le pipeline d'opérateurs : les résultats intermédiaires sont écrits en mémoire mais réutilisés le plus tôt possible pour réduire au maximum la probabilité d'un défaut de cache.

Les applications complexes sont constituées d'une ou plusieurs suite d'opérateurs interdépendants. De nombreux résultats intermédiaires sont donc utilisés. Un nombre important de transformations portant sur l'optimisation de la composition des opérateurs peut alors être apporté. Inversement, il est plus difficile d'effectuer de nombreuses optimisations sur des algorithmes plus simples qui utilisent peu d'opérateurs différents.

Dans la suite de cette section, nous détaillons les principes de la fusion et du pipeline d'opérateurs.

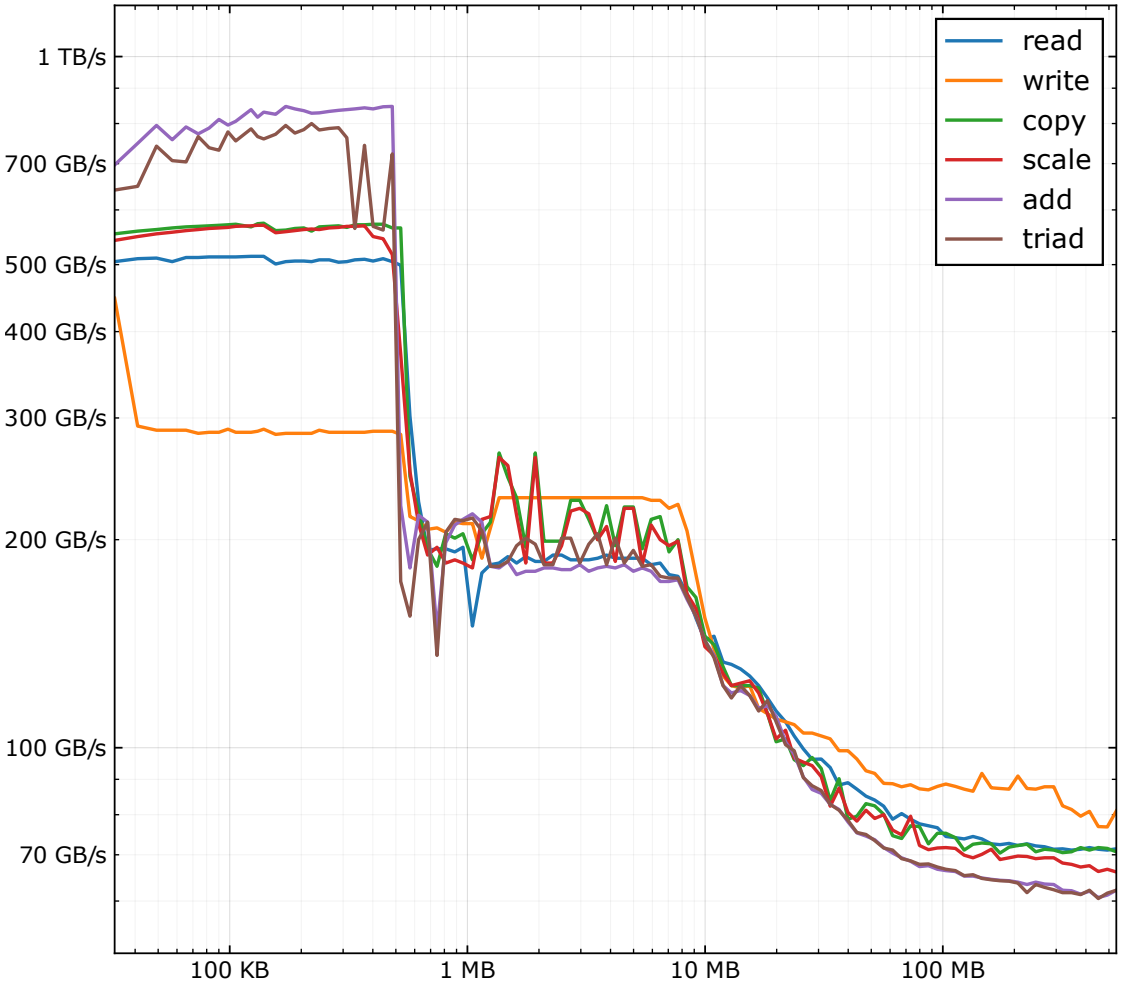


FIGURE 2.2 – Bande passante mesurée pour l'ensemble des micro-tests de STREAM en fonction de la taille des données, sur le CPU de la plateforme Nvidia Jeton AGX.

2.2.1 Fusion d'opérateurs

La fusion d'opérateurs a pour but d'éviter au maximum les accès mémoire pour écrire puis lire les résultats intermédiaires. L'objectif est de minimiser le nombre d'accès en mémoire cache ou externe et de maximiser la persistance des valeurs en registres. Le grand nombre d'accès étant un problème majeur pour les performances de notre application, la fusion d'opérateurs peut s'avérer particulièrement bénéfique. Il est important ici de tenir compte du fait que, dans l'ensemble de ces travaux, nous utilisons le langage de programmation C. La "suppression" des accès en mémoire n'est pas assurée étant donné que c'est le compilateur qui gère les accès en dernier. La fusion permet de présenter les choses au compilateur de façon à favoriser l'optimisation en registre mais sans garantie absolue.

Dans certains cas, la fusion d'opérateurs est évidente et s'applique facilement mais il peut arriver que sa mise en œuvre nécessite de profondes transformations du code car les différents opérateurs sont particulièrement espacés dans l'algorithme initial.

Prenons l'exemple du cas simple d'une addition suivie d'une multiplication. On définit les opérateurs suivants :

— **Addition :** F_1 tel que $F_1(x, y) = x + y$.
— **Multiplication :** F_2 tel que $F_2(x, y) = x \times y$.

On a ainsi, $F_2(F_1(A, B), C) = (A + B) \times C$.

La figure 2.3 illustre le principe de la fusion d'opérateurs appliquée à $F_2 \circ F_1$. Ici, nous utilisons une représentation de type *producteur-consommateur* des opérateurs comme présentée dans [44]. Chaque case représente une donnée sauvegardée en mémoire. Les données à gauche de l'opérateur sont les données utilisées en entrée : données consommées. Tandis que les données à droite sont les données produites.

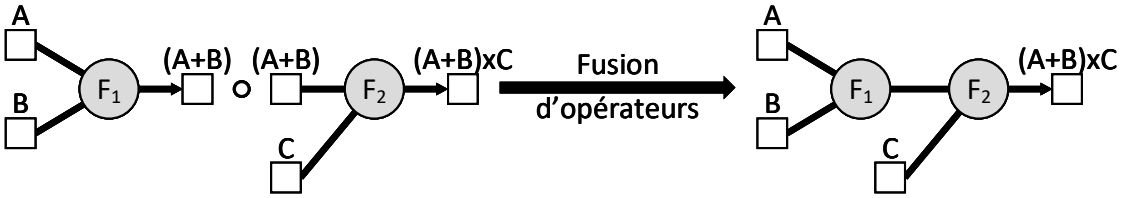


FIGURE 2.3 – Fusion d'opérateurs : addition puis multiplication.

Dans le premier cas, le résultat intermédiaire produit par F_1 est écrit en mémoire avant d'être chargé dans l'opérateur F_2. Avec la fusion d'opérateurs, la séquence d'accès mémoire, écriture puis lecture, est évitée.

Soient F et G, deux opérateurs de type opérateur ponctuel ou opérateur à voisinage (convolution de type *stencil*). Leur composition génère 4 cas différents :

— **cas 1 :** F est un opérateur point à point, G est un opérateur point à point. (Cas de l'exemple)
— **cas 2 :** F est un opérateur n vers 1, G est un opérateur 1 vers 1.
— **cas 3 :** F est un opérateur point à point, G est un opérateur n vers 1.
— **cas 4 :** F est un opérateur n vers 1, G est un opérateur n vers 1.

Des règles différentes pour mettre en place une fusion existent en fonction des différents cas. Ces règles sont présentées en figure 2.4. L'impact de ces transformations sur le nombre d'accès mémoire et le nombre d'opérations est indiqué dans le tableau 2.2.

L'impact de la fusion pour ces 4 cas est étudié dans [44] pour le cas applicatif de la détection de points d'intérêt de Harris [42]. Il y est montré que la fusion complète n'est pas toujours la solution la plus rapide. Les performances varient d'une architecture CPU à une autre. La taille de cache, la taille des données, le nombre de calculs par opérateur, la bande passante externe ou encore le rapport du temps d'exécution entre les calculs et les accès mémoire sont autant de facteurs qui vont influer sur l'efficacité de la fusion. On

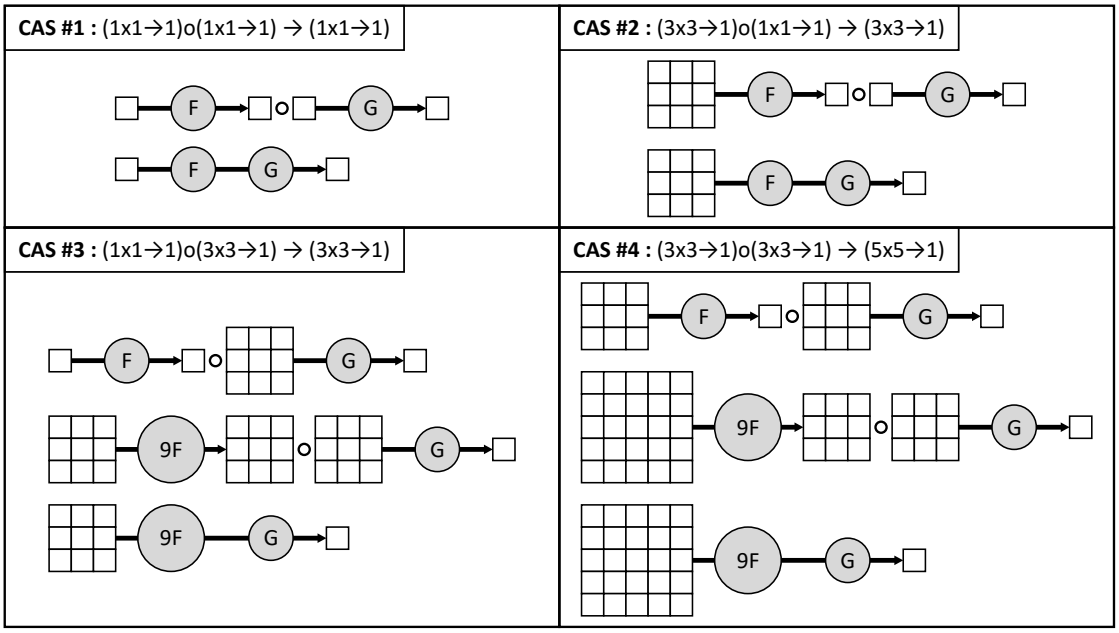


FIGURE 2.4 – Fusions pour la composition d'opérateurs ponctuels et à voisinage (stencil).

Cas	F	G	Avant fusion		Après fusion	
			Mem	OP	Mem	OP
#1	$(1 \times 1) \to 1$	$(1 \times 1) \to 1$	4	2	2	2
#2	$(3 \times 3) \to 1$	$(1 \times 1) \to 1$	12	2	10	2
#3	$(1 \times 1) \to 1$	$(3 \times 3) \to 1$	12	2	10	10
#4	$(3 \times 3) \to 1$	$(3 \times 3) \to 1$	20	2	25	10

TABLE 2.2 – Impact de la fusion d'opérateurs en fonction des 4 cas présentés en figure 2.4 sur le nombre d'accès mémoire et d'opérations. Valeurs données pour des opérateurs à voisinage 3×3.

peut tout de même d'ores et déjà remarquer l'inefficacité de la fusion dans le cas numéro 4. En effet, dans ce cas le nombre d'accès mémoire par pixel augmente de 20 à 25 tandis que le nombre d'opérations augmente de 2 à 10 par pixel. Une autre transformation doit alors être envisagée.

2.2.2 Pipeline d'opérateurs

Le pipeline d'opérateurs constitue une alternative à la fusion. L'objectif de cette transformation n'est pas de réduire le nombre d'accès mémoire mais d'en améliorer la localité.

Dans le cas d'opérateurs à voisinage, la fusion nécessite une rétro-propagation complète des contraintes d'entrée et de sortie. L'enchaînement de nombreux opérateurs multi-

points rend donc complexe la mise en place de la fusion. De plus, le nombre de registres nécessaires devient plus important. La distance entre les opérateurs peut également être un obstacle à la mise en place d'une fusion. Pour ces raisons, si l'on considère que la fusion est trop coûteuse, trop complexe ou inefficace on peut alors choisir de réaliser un pipeline à la place.

Contrairement à la fusion, le pipeline ne supprime pas les écritures et lectures en mémoire mais améliore la localité des accès. Les données intermédiaires sont toujours écrites en mémoire mais sont réutilisées au plus tôt dans l'algorithme. Cela permet d'augmenter la probabilité d'accéder aux données lorsque celles-ci sont encore présentes en mémoire cache.

Dans l'exemple suivant, nous expliquons le déroulé du pipeline d'opérateurs pour un cas simple. Supposons que l'on souhaite calculer un gradient sur le carré des valeurs d'une image A. Pour plus de simplicité, nous considérons un gradient avant à une dimension. Soit $A(i, j)$ le pixel de la ligne i et de la colonne j de l'image A. On définit les opérateurs suivants :

— **Carré :** F_3 tel que $F_3(A(i, j)) = A(i, j)^2$.
— **Gradient :** F_4 tel que $F_4(A)(i, j) = A(i, j + 1) - A(i, j)$.

La composition $F_4(F_3(A))$ effectue bien l'opération souhaitée. La figure 2.5 illustre cette composition dans le cas classique d'écriture et de lecture des données intermédiaires puis dans le cas d'un pipeline d'opérateurs. On remarque que, dans ce cas, la fusion des opérateurs est possible mais plus complexe que dans le cas présenté en figure 2.3. En effet, il est nécessaire de calculer $F_3(A(i, j))$ puis $F_3(A(i, j + 1))$ pour pouvoir calculer $F_4(A(i, j)^2)$. La fusion reste possible en utilisant les règles présentées en figure 2.4.

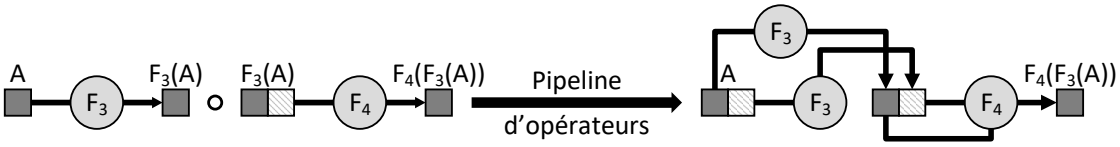


FIGURE 2.5 – Calcul de $F_4(F_3(A))$ avec sauvegarde du résultat intermédiaire $A(i, j)^2$ sans puis avec pipeline d'opérateurs.

Considérons cependant que, dans ce cas, la fusion est trop complexe ou trop lourde pour être implémentée. S'il n'est donc pas envisageable de supprimer l'écriture des données intermédiaires par la fusion, le pipeline d'opérateur permet de rapprocher au maximum l'accès en écriture du résultat intermédiaire de son accès en lecture. Ainsi, la probabilité que la donnée soit encore présente en mémoire cache est accrue. L'accès en mémoire cache étant bien plus rapide qu'un accès en mémoire externe, cette transformation peut permettre d'accélérer l'exécution du traitement de façon significative. En outre, les accès en mémoire externe entrainent également une consommation énergétique plus importante que les accès en mémoire cache. Il est donc particulièrement pertinent de les éviter pour des applications embarquées.

Pour plus de clarté, nous détaillons le déroulement des calculs du pipeline présenté en figure 2.5 dans la figure 2.6. Les opérations sont effectuées dans l'ordre de $t = 1$ à $t = 5$. Dès que cela est possible une donnée finale est produite par F_4. L'étape contenant l'opération 1 constitue le prologue du pipeline. L'étape contenant les opérations 2 et 3, constitue la première étape du régime permanent. En régime permanent, chaque étape produit une donnée par F_3 et une par F_4. Ces deux données sont spatialement décalées d'un pixel.

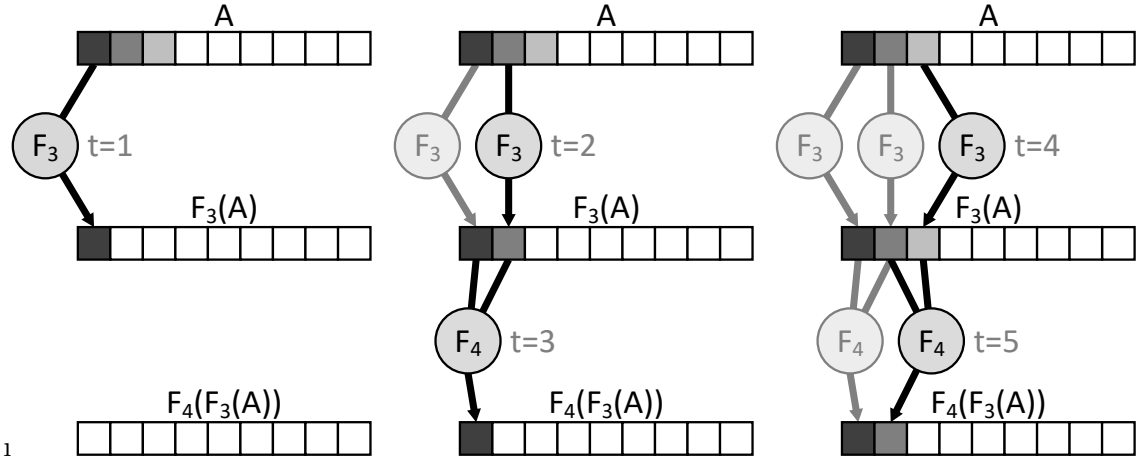


FIGURE 2.6 – Pipeline d'opérateurs pour le calcul de Grad(A^2).

Dans notre exemple, l'élément de synchronisation est de 2 pixels : il faut produire 2 pixels avec F_3 pour pouvoir produire 1 pixel avec F_4. La taille des éléments de synchronisation peut varier selon l'enchaînement et la nature des opérateurs.

- Élément de synchronisation de la taille de l'image : optimisations de composition d'opérateurs compromises.
- Élément de synchronisation de une ou plusieurs lignes : pipeline d'opérateurs possible améliorant la localité mémoire.
- Élément de synchronisation de quelques pixels : pipeline d'opérateurs possible améliorant la localité mémoire. Fusion d'opérateurs également possible permettant la suppression de certains accès.

La taille de l'élément de synchronisation détermine la profondeur du pipeline. Dans la pratique le pipeline est généralement appliqué sur des éléments de synchronisation d'une ou plusieurs lignes et la fusion sur quelques pixels.

2.3 Vecteurs de Iliffe et allocation modulaire

2.3.1 Vecteurs de Iliffe

Avant d'introduire la notion d'allocation modulaire, il est important de comprendre comment nous allouons la mémoire. Dans notre chaîne de traitement, les tableaux mul-

tidimensionnels sont alloués en vecteurs de Iliffe [66]. En C, les tableaux multidimensionnels sont traditionnellement alloués comme des tableaux à 1 dimension (1D) et les adresses sont linéarisées. Prenons l'exemple d'un tableau 2D A, de taille $N \times M$. Avec une allocation classique, A sera alloué comme un tableau 1D comprenant $N \times M$ éléments. Accéder à l'élément $A(i, j)$ se fera de la sorte : $A[i \times M + j]$. Il est possible d'allouer les tableaux différemment en utilisant les vecteurs de Iliffe : n tableau de pointeurs sur les lignes est alloué en plus du tableau 1D précédent. Ainsi, l'élément $A(i, j)$ est accédé par $A[i][j]$. La figure 2.7 illustre l'agencement mémoire d'une matrice de taille 3×3 en appliquant le principe des vecteurs de Iliffe.

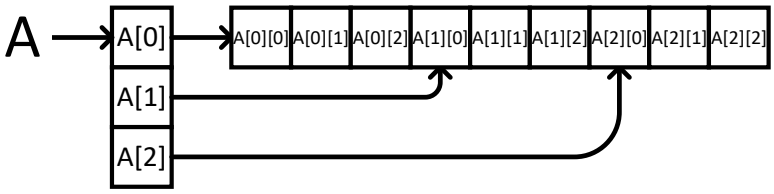


FIGURE 2.7 – Vecteur de Iliffe pour une matrice 3×3.

Les vecteurs de Iliffe peuvent être utilisés pour un nombre quelconque de dimensions [51]. De part l'ajout d'un pointeur de lignes, un surcoût est présent lors de la phase d'allocation. Cependant, pour notre application nous utilisons uniquement des tableaux pré-alloués ce qui nous permet de ne pas prendre en compte les phases d'allocation dans les mesures de temps d'exécution.

2.3.2 Allocation modulaire

La contraction de tableaux (*array contraction* [64]) est un ensemble de méthodes permettant la réduction de l'empreinte mémoire ainsi que l'amélioration de la localité des accès [78, 79, 80, 81]. L'allocation modulaire est l'une de ces méthodes. Dans la littérature, il peut également y être fait référence sous les termes de technique de Lefebvre-Feautrier, de modulo successif ou encore de modèle polyédrique [62, 82, 83]. L'allocation modulaire permet, pour un tableau intermédiaire, de réduire la quantité de mémoire allouée à seulement quelques lignes.

Grâce au pipeline, il est possible d'utiliser l'allocation modulaire pour améliorer encore davantage la localité des données. Le pipeline permet d'accéder aux données temporaires le plus tôt possible après leur création. L'idéal serait de pipeliner points par points mais cela peut s'avérer particulièrement complexe. La localité des accès est donc améliorée une première fois grâce au pipeline. Les risques de défaut de cache sont alors réduits. Cependant, il n'est pas garanti que les données accédées soient effectivement présentes en mémoire cache. L'allocation modulaire permet d'augmenter la probabilité qu'une donnée, issue d'un calcul intermédiaire, soit toujours présente dans le cache lorsque l'on y accède de nouveau.

Grâce au pipeline par ligne, un résultat intermédiaire est réutilisé au plus tôt puis devient inutile. Si un résultat intermédiaire n'est plus utile, il n'est pas nécessaire de

le conserver en mémoire. L'allocation modulaire permet de profiter de cet aspect en réutilisant les données devenues inutiles et en les remplaçant par de nouvelles.

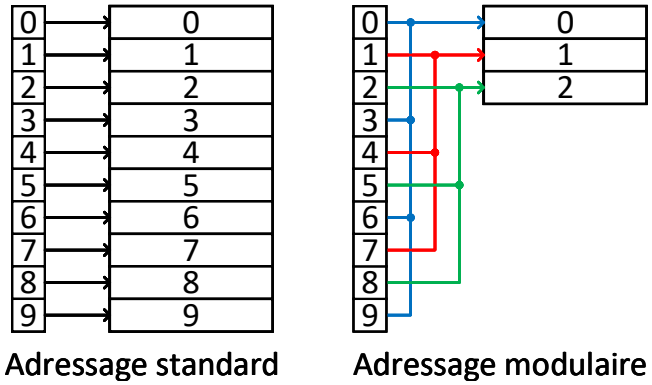


FIGURE 2.8 – Allocation standard *vs* allocation modulaire.

Comme illustré en figure 2.8, l'utilité éphémère des résultats intermédiaires permet de limiter la quantité de mémoire allouée à seulement quelques lignes. Dans le tableau de pointeurs de lignes, plusieurs pointeurs pointent sur une même ligne. Le nombre réel de lignes allouées dépend de la profondeur du pipeline et de la dépendance des données entre les opérateurs. Dans la figure 2.8, nous considérons un modulo sur 3 lignes. Cela signifie que lorsque le résultat intermédiaire de la ligne 3 est produit, celui de la ligne 0 n'est plus utile. Par conséquent il n'est pas gênant d'écraser le résultat de la ligne 0 pour le remplacer par celui de la ligne 3. C'est pourquoi ces deux lignes partagent le même emplacement, de même que les résultats la ligne 6 écrasent ceux de la 3, ceux de la 9 ceux de la 6 etc. Le même raisonnement décalé respectivement d'une et de deux lignes est appliqué aux lignes d'indices $(3k+1)$ et $(3k+2)$.

La faible quantité de mémoire réellement allouée augmente la probabilité que la totalité du tableau tienne en mémoire cache. Grâce à son adressage spécifique en vecteurs de Iliffe, l'utilisation de l'allocation modulaire est transparente pour l'utilisateur. Ce dernier aspect est important pour pouvoir éviter de limiter son utilisation uniquement par des experts.

Pour des raisons pédagogiques, nous présentons et détaillons un exemple de pipeline avec allocation modulaire. Considérons l'image I_{in}. On souhaite lui appliquer deux fois de suite l'opérateur F_s : un filtre sommateur de taille 3×3.

La figure 2.9 illustre l'exemple considéré sans pipeline ni allocation modulaire. $S_1 = F_s(I_{in})$ est le résultat intermédiaire de la première application de F_s. $S_2 = F_s(S_1) = F_s(F_s(I_{in}))$ est le résultat final de la seconde application de F_s. Ici, la composition d'opérateurs est faite de façon standard : avec sauvegarde du résultat intermédiaire et sans pipeline ni allocation modulaire. Pour simplifier l'exemple, la gestion des bords n'est pas prise en compte. Pour la même raison, aucune optimisation particulière n'est appliquée. Dans la suite de l'exemple, et toujours dans un soucis de simplification, le pipeline et l'allocation modulaire sont les seules optimisations apportées.

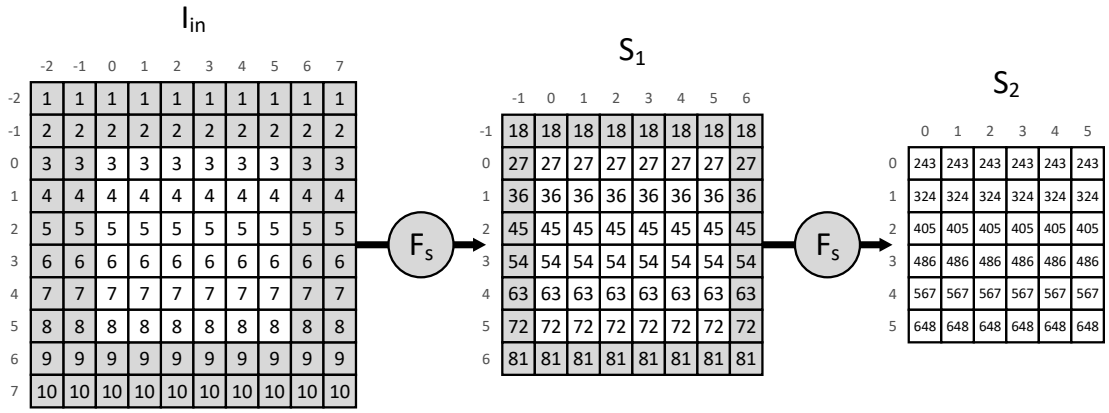


FIGURE 2.9 – Application du filtre sommateur à I_{in}.

On effectue le même calcul mais pipeliné et avec S_1 alloué de façon modulaire. F_s étant de taille 3×3, nous pouvons mettre en place un pipeline de lignes, de profondeur 3. Par conséquent la taille réelle allouée à S_1 sera de 3 lignes.

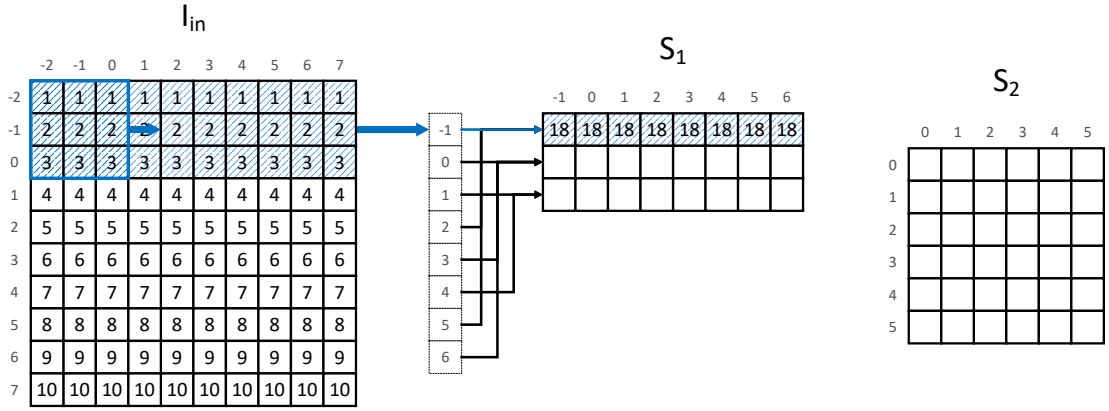


FIGURE 2.10 – Pipeline et allocation modulaire : étape 1.

On calcule tout d'abord la ligne (-1) de S_1 (figure 2.10).

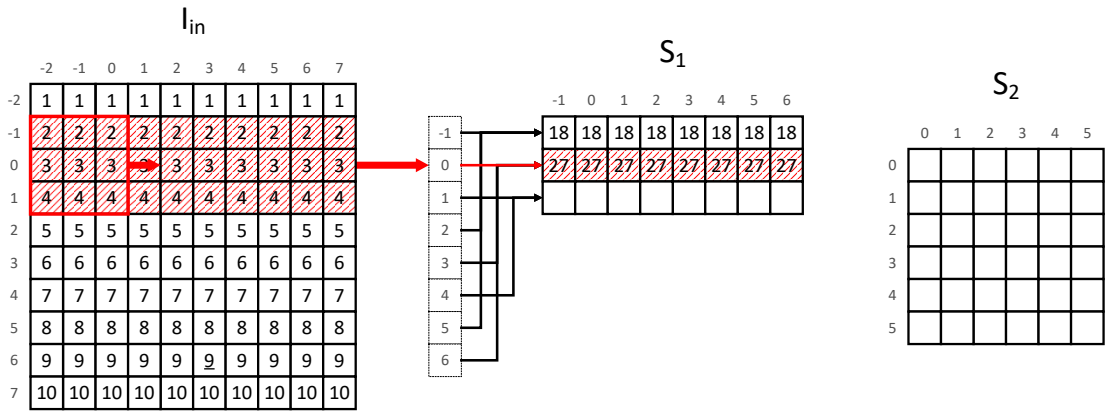


FIGURE 2.11 – Pipeline et allocation modulaire : étape 2.

On calcule ensuite la ligne (0) de S_1 (figure 2.11). Jusque ici les calculs s'effectuent de façon classique.

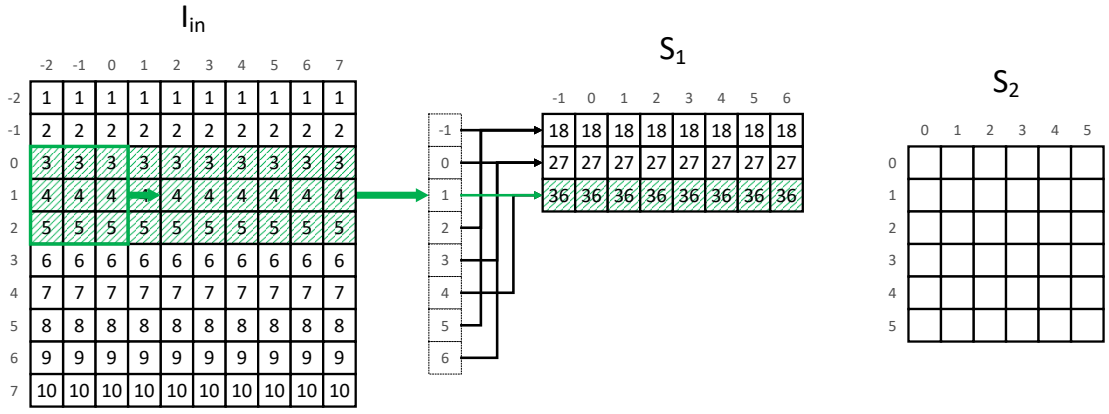


FIGURE 2.12 – Pipeline et allocation modulaire : étape 3a.

À la troisième étape, on commence par calculer la ligne (1) de S_1 (figure 2.12).

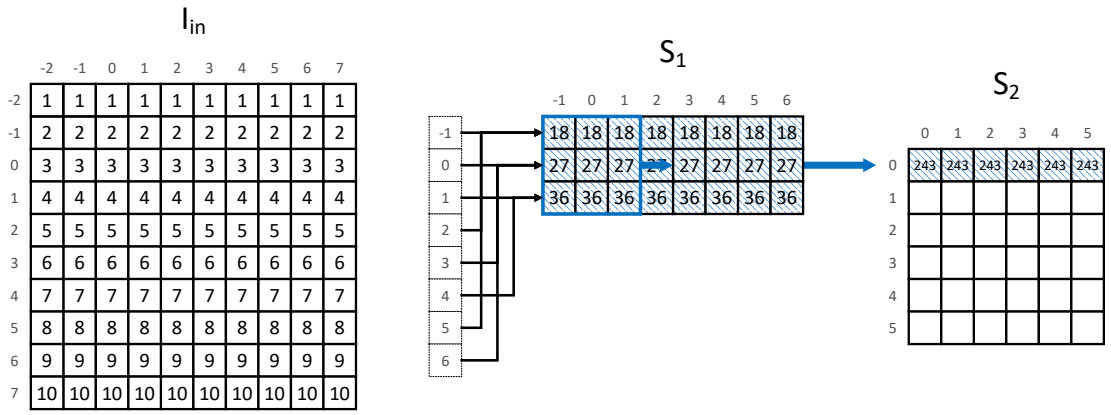


FIGURE 2.13 – Pipeline et allocation modulaire : étape 3b.

Les 3 lignes calculées de S_1 permettent alors d'effectuer les premiers calculs en cascade du pipeline et de calculer la ligne (0) de S_2 (figure 2.13).

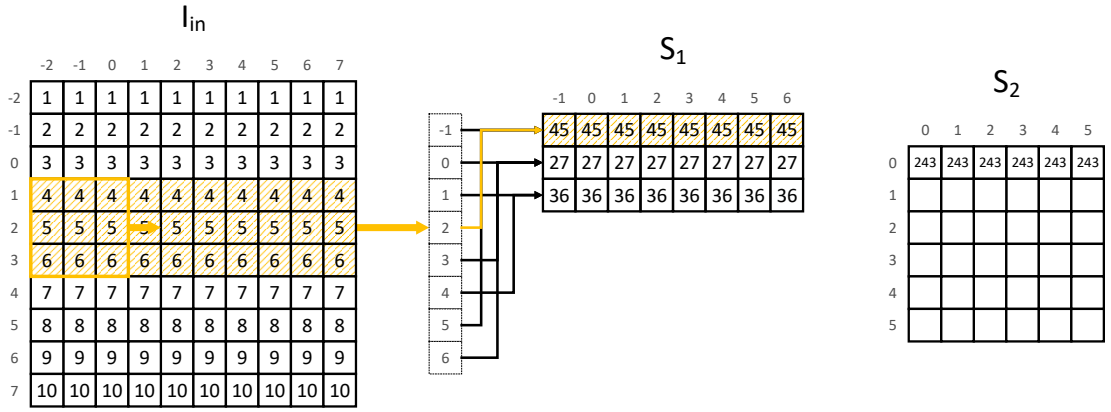


FIGURE 2.14 – Pipeline et allocation modulaire : étape 4a.

C'est à partir de la quatrième étape que l'allocation modulaire de S_1 a un effet sur les données sauvegardées en mémoire (figure 2.14). En effet les lignes (-1) et (2) de S_1 partagent le même espace mémoire. Lors du calcul de la ligne (2) les résultats de la ligne (-1) sont donc écrasés.

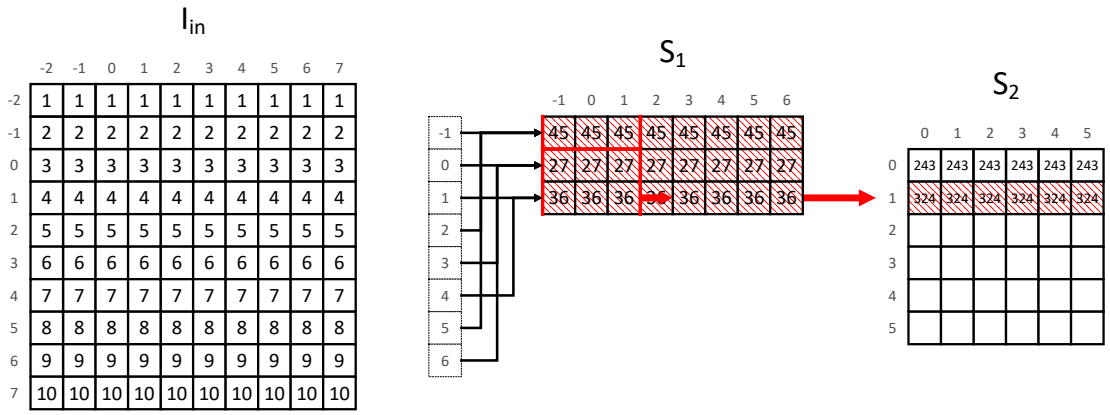


FIGURE 2.15 – Pipeline et allocation modulaire : étape 4b.

Cela n'est pas gênant puisque la ligne (-1) de S_1 n'est plus utile à ce moment. Les lignes (0), (1) et (2) de S_1 permettent de produire la ligne (1) de S_2 (figure 2.15).

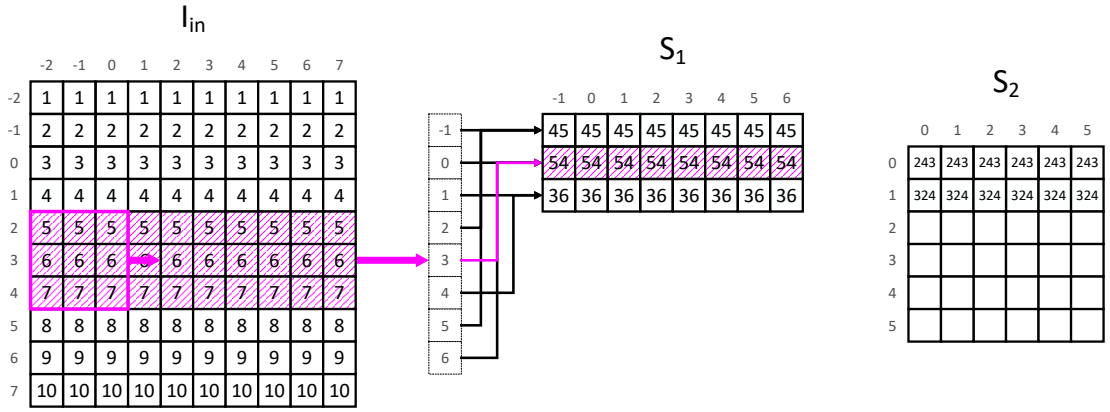


FIGURE 2.16 – Pipeline et allocation modulaire : étape 5a.

De façon analogue à l'étape précédente, lors de l'étape 5, le calcul de la ligne (3) de S_1 écrase les résultats de la ligne (0) devenue inutile (figure 2.16).

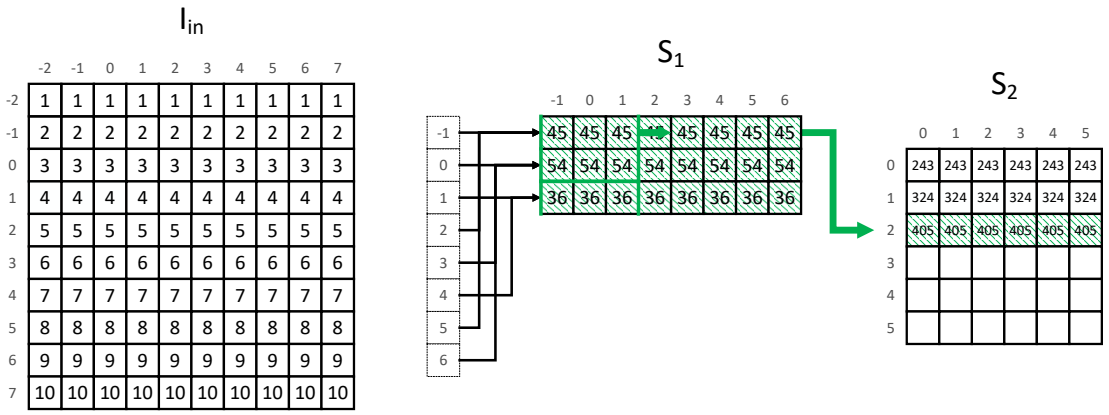


FIGURE 2.17 – Pipeline et allocation modulaire : étape 5b.

Les résultats présents dans S_1 permettent de calculer la ligne (2) de S_2 (figure 2.17).

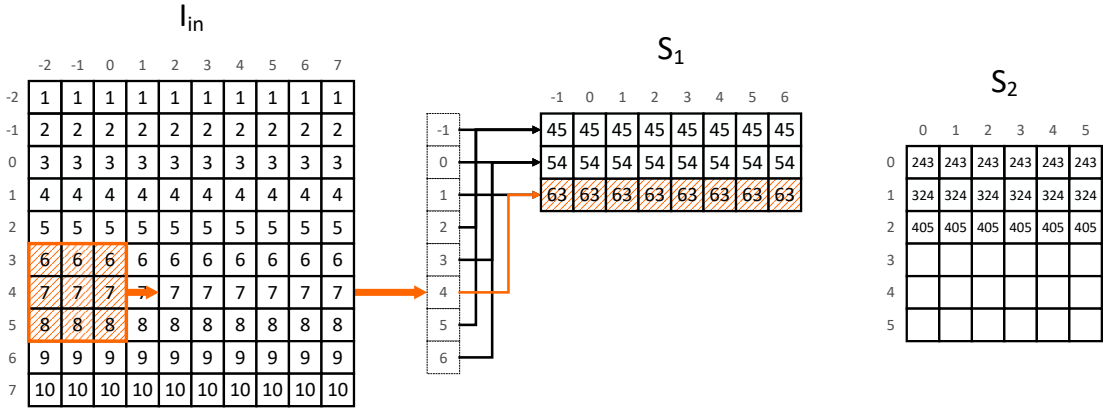


FIGURE 2.18 – Pipeline et allocation modulaire : étape 6a.

Lors de la sixième étape ce sont les résultats de la ligne (1) de S_1 qui sont écrasés par ceux de la ligne (4)(figure 2.18).

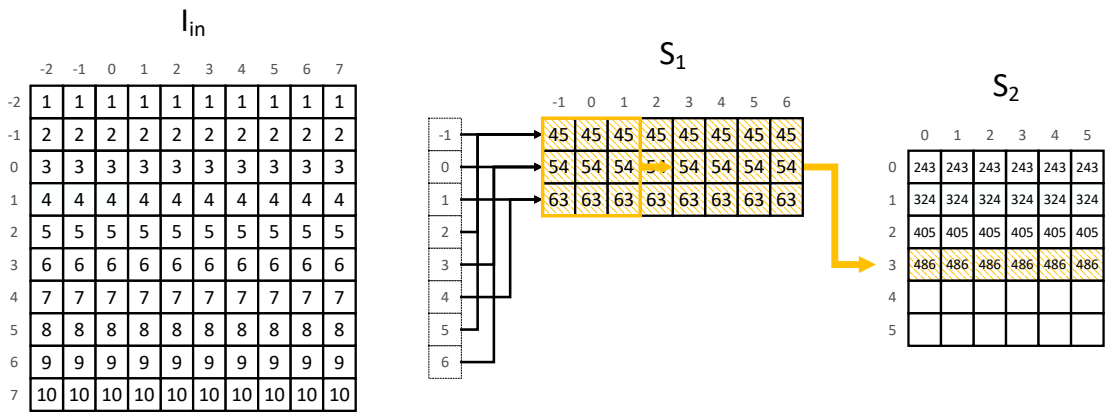


FIGURE 2.19 – Pipeline et allocation modulaire : étape 6b.

On calcule ensuite la ligne (3) de S_2 (figure 2.19). A cette étape, toutes les lignes de S_1 ont été écrasées au moins une fois sans affecter les résultats de S_2. Ce cycle va se répéter pour toutes les lignes suivantes.

2.4 Synthèse

Dans ce chapitre, nous avons abordés différentes méthodes d'optimisation. Dans un premier temps, nous avons présenté les méthodes de parallélisation de code grâce au parallélisme de tâches (MIMD) puis au parallélisme de données (SIMD). Nous avons également abordé l'impact de la taille des données sur le parallélisme SIMD, sur l'utilisation de la bande passante mais également sur le résultat qualitatif final.

Dans un second temps, nous avons présenté deux techniques de composition d'opérateurs qui ont pour but de minimiser l'impact des calculs intermédiaires sur le temps d'exécution. La fusion tout d'abord, permet de conserver les résultats intermédiaires dans des registres et d'éviter des étapes d'écriture puis de lecture en mémoire. Le pipeline, quant à lui, permet d'améliorer la localité mémoire et favorise les accès en cache plutôt qu'en mémoire externe.

Enfin, nous avons détaillé la stratégie d'allocation modulaire rendue possible grâce aux vecteurs de Iliffe et au pipeline d'opérateurs. L'allocation modulaire profite de l'utilité éphémère des résultats intermédiaires pour pouvoir réduire l'empreinte mémoire totale et maximiser la persistance des données en mémoire cache.

Les optimisations présentées dans ce chapitre ont été appliquées à l'ensemble de la chaîne de traitement présentée dans ces travaux. D'autres optimisations, plus spécifiques à chaque maillon, ont également été appliquées et seront détaillées lors de la présentation du maillon en question.

Chapitre 3

Chaîne de traitement : RTE-VD

3.1 Présentation

Dans ce chapitre nous présentons l'ensemble des différents maillons de la chaîne de traitement. L'algorithme complet de la chaîne de débruitage est composé de plusieurs algorithmes existants, qui ont été modifiés et optimisés afin de s'exécuter en temps réel pour différents formats de vidéo. La chaîne de traitement est nommée RTE-VD pour *Real Time Embedded Video Denoising*.

La difficulté majeure dans le débruitage de vidéo est de filtrer le bruit sans provoquer un flou de bougé. Par rapport au traitement d'une image seule, la vidéo apporte une dimension temporelle supplémentaire. On peut filtrer la valeur d'un pixel avec ses valeurs passées et futures. Le bruit étant un événement transitoire, il pourra ainsi être atténué. C'est le filtrage temporel. On peut également filtrer la valeur d'un pixel avec ses voisins comme pour une image seule. On appelle cela le filtrage spatial. OUn filtrage dit Spatio-temporel (ou 3D) peut être appliqué pour tirer parti des deux méthodes.

Des filtres spatio-temporels sont actuellement employés dans les caméras LHERITIER mais présentent un problème majeur : ils sont efficaces pour des scènes statiques mais pas pour des scènes dynamiques. En effet, pour le filtrage temporel, une simple pondération est appliquée entre les images. En cas de déplacement d'un objet au sein de la vidéo, son mouvement est donc pondéré temporellement. Cela a pour effet d'entraîner un important flou de bougé.

Pour éviter ce problème, il faut procéder à un recalage des pixels en fonction du mouvement apparent de la scène. Ce mouvement est appelé flot optique. C'est grâce à ce recalage que l'on peut procéder correctement au filtrage temporel. Les figures 3.1 à 3.6 illustrent ce phénomène. On peut voir avec les figures 3.1, à 3.3 que lors d'une scène statique le débruitage temporel sans compensation de mouvements est presque aussi efficace qu'avec. En revanche, les figures 3.4, à 3.6 montrent qu'en cas de mouvements dans la scène, l'absence de compensation génère un flou de bougé important. Certes la quantité de bruit est amoindrie, mais l'image n'est pas exploitable car trop floue. Grâce au recalage des pixels par flot optique, l'image est plus nette et le débruitage est efficace.

FIGURE 3.1 – Scène statique bruitée.

FIGURE 3.2 – Scène statique filtrée temporellement.

FIGURE 3.3 – Scène statique filtrée temporellement avec recalage.

Comme indiqué, le flot optique traduit le mouvement apparent entre deux images consécutives d'une même scène. De nombreuses méthodes permettent de l'estimer. Sans se référer aux méthodes numériques de résolution du calcul de flot optique, il est possible de calculer 3 types de flots différents :

- **Flot optique dense** : un déplacement spécifique est calculé pour chaque pixel.
- **Flot optique creux** : le déplacement n'est calculé que pour un nombre fixe de pixels répartis soit de façon régulière dans l'image soit selon des points d'intérêts choisis.
- **Flot optique global** : seul le déplacement moyen entre les deux images est calculé. On chercher à inverser une transformation géométrique globale (translation, rotation...). Cela permet d'estimer le déplacement de la caméra.

Pour pouvoir recaler chaque pixel, la chaîne de traitement requiert donc une estimation de flot dense. Les méthodes numériques d'estimation de flot optique dense sont cependant coûteuses et nécessitent d'importantes ressources de calcul. Il peut donc être intéressant d'effectuer une phase de stabilisation globale au préalable. Cela permet d'alléger le calcul de flot dense par la suite, en limitant l'amplitude des déplacements. Après une compensation de mouvement en fonction du flot optique calculé, il est alors possible

FIGURE 3.4 – Scène dynamique bruitée.

FIGURE 3.5 – Scène dynamique filtrée temporellement.

FIGURE 3.6 – Scène dynamique filtrée temporellement avec recalage.

d'appliquer une étape de filtrage pour atténuer le bruit de la vidéo.

En définitive, RTE-VD se décompose en 3 étapes principales :

— **La stabilisation globale** : permet de compenser le mouvement de la caméra.
— **L'estimation dense du flot optique** : permet d'attribuer un vecteur vitesse à chaque pixel en fonction du mouvement apparent de la scène.
— **Le filtrage spatio-temporel** : applique la pondération cohérente entre les pixels.

Dans la suite de ce chapitre, nous présentons séparément les étapes de stabilisation, d'estimation de flot optique dense et de filtrage spatio-temporel. Pour chaque étape, nous expliquons tout d'abord le choix de la méthode utilisée. Dans un deuxième temps, nous détaillons son expression mathématique et algorithmique. Dans un troisième temps nous présentons les optimisations spécifiquement apportées. Enfin, nous évaluons l'impact des différentes transformations apportées. Pour l'ensemble des résultats présentés dans cette thèse, nous utilisons le compilateur GCC 7.4.0 [84] avec l'option "-O3". La vectorisation n'est jamais désactivée.

3.2 Stabilisation

L'étape de stabilisation permet de compenser le mouvement de la caméra. Cela permet par la suite de faciliter l'estimation du mouvement de chaque pixel lors de l'étape de calcul de flot optique dense.

3.2.1 Motivations et choix de la méthode

Pour une observation à longue distance, les mouvements de caméra peuvent rapidement devenir importants. Prenons l'exemple d'une caméra LHERITIER CatEye [85]. Cette dernière possède un champ de vision de 2.3° en horizontal et de 1.3° en vertical et peut permettre la détection d'un humain à 6 km, sa reconnaissance à 4 km et son identification à 2 km. Dans ces conditions, et même en utilisant un trépied, le vent ou bien les vibrations de pas sur le sol entraînent des déplacements de plusieurs pixels dans l'image. Il est alors facile de comprendre l'ampleur que ces mouvements peuvent avoir avec un tel dispositif embarqué à bord d'un véhicule ou tenu à main levée.

La méthode de calcul de flot optique dense utilisée, TV-L^1 [41], est pyramidale et l'amplitude maximale du déplacement estimable par cette méthode est directement liée au nombre d'étages de cette pyramide et au facteur de taille entre chacun. Calculer de grands déplacements entraîne un plus grand nombre d'étages et donc plus de calculs par TV-L^1. De plus, cette méthode est particulièrement lourde et la pénalité en termes de temps d'exécution est importante. C'est pourquoi, une première étape de stabilisation est mise en place afin de compenser les mouvements de la caméra. Ces mouvements sont en général les plus importants. Cette approche permet donc de réduire le nombre d'échelles de TV-L^1 et de se concentrer sur les mouvements plus faibles qui constituent l'action de la scène.

La méthode de stabilisation utilisée est une approche globale similaire au calcul de flot optique de Lucas et Kanade [40]. Cette méthode a été introduite comme méthode dense. Ici, nous proposons une méthode globale pour la stabilisation. Nous l'appelons StabLK. Il est important de noter que la méthode proposée ici se rapproche de la méthode de Lucas et Kanade au niveau de la résolution finale. L'équation de départ et le cheminement ne sont toutefois pas les mêmes que ceux de la solution originale présentée par Lucas et Kanade. Le choix de cette méthode s'est fait assez naturellement puisqu'elle est déjà utilisée et implémentée de façon efficace dans les FPGA de certains produits Lheritier. StabLK peut être déclinée en différentes versions selon le type de stabilisation souhaitée. Les éléments suivants peuvent être associés entre eux :

— Stabilisation en translation (horizontale et verticale),
— Stabilisation en rotation,
— Stabilisation de la luminosité.

La stabilisation de la luminosité peut s'avérer particulièrement intéressante si l'on considère un éclairage ambiant artificiel. En effet, l'éclairage artificiel n'est ni stable ni synchrone avec la fréquence d'acquisition de la caméra, ce qui crée un effet de scintillement.

Même sans éclairage artificiel, cet effet de scintillement peut être causé par une variation du gain asservi de la caméra. Cela peut totalement perturber le calcul du flot optique dense. Comme beaucoup d'autres méthodes [86], l'estimation du flot optique par TV-L^1 repose sur des calcul de gradients au sein des images et entre elles. Bien que TV-L^1 soit relativement robuste au bruit, le postulat de luminosité constante y est donc tout de même posé. Un changement brutal de luminosité sera interprété comme un mouvement et risque d'entraîner des déformations de l'image lors de la reconstruction. Cependant, si l'on considère une application extérieure en condition de BNL, il est peu probable d'être en présence d'un éclairage ambiant artificiel. En outre, l'ajout d'une nouvelle composante à stabiliser complexifie les calculs et rend moins stable la méthode. L'ajout de la stabilisation en luminosité n'est donc pas forcément pertinent pour toutes les applications.

De la même façon, la stabilisation en rotation peut faire l'objet d'un choix applicatif. Sur trépied ou même à main levée, les mouvements en rotation ne sont pas significatifs. En revanche à bord d'une embarcation maritime par exemple, il devient indispensable de les compenser.

3.2.2 Description de l'algorithme

Comme indiqué en section 3.2.1, la méthode de stabilisation est inspirée de l'estimation du flot optique de Lucas et Kanade. Il est possible d'estimer les translations, les rotations et les variations de luminosité. Dans la suite, nous commencerons par détailler le cas le plus simple, c'est à dire l'estimation des translations seules. Dans un second temps, le cas le plus complexe (Translations + Rotation + Luminosité) sera abordé. Son développement est analogue au cas des translations seules.

Cas des translations seules

Soient 2 images consécutives I_{t+dt} et I_t. On note Ω le sous espace d'intérêt dans les images. Nous souhaitons déterminer d_x et d_y respectivement les déplacements horizontal et vertical de I_{t+dt} par rapport à I_t. Pour se faire, nous cherchons à minimiser l'équation 3.1.

$$\sum_{i,j\in\Omega} \left(I_{t+dt}(i+dy, j+dx) - I_t(i,j)\right)^2 = \varepsilon \tag{3.1}$$

On note $s = (i, j)$ et $ds = (dy, dx)$. En dérivant les paramètres d_x et d_y nous obtenons l'expression 3.2, qui est nulle pour un minimum de ε.

$$\sum_{i,j\in\Omega} 2\left(\overrightarrow{\nabla} I_{t+dt}\right)(s+ds) \times \left(I_{t+dt}(s+ds) - I_t(s)\right) = 0 = \begin{bmatrix} F_y \\ F_x \end{bmatrix} = F \tag{3.2}$$

La solution de cette équation peut être approchée par le développement limité - ou équation de Newton à plusieurs paramètres - présenté dans la formule 3.3.

$$\begin{bmatrix} dy \\ dx \end{bmatrix} \approx -H^{-1}F \tag{3.3}$$

H étant la matrice Hessienne de la fonction de l'équation 3.1. H est définie dans l'équation 3.4 et simplifiée dans l'équation 3.5. On rappelle que l'opérateur de produit scalaire " $\cdot$ " correspond à la somme des produit point à point de deux matrices.

$$H = \begin{bmatrix} H_{00} & H_{01} \\ H_{10} & H_{11} \end{bmatrix} = \begin{bmatrix} \frac{\partial^2 \varepsilon}{\partial y^2} & \frac{\partial^2 \varepsilon}{\partial y \partial x} \\ \frac{\partial^2 \varepsilon}{\partial x \partial y} & \frac{\partial^2 \varepsilon}{\partial x^2} \end{bmatrix} \tag{3.4}$$

$$H \approx \begin{bmatrix} (\nabla_y I_{t+dt}) \cdot (\nabla_y I_{t+dt}) & (\nabla_y I_{t+dt}) \cdot (\nabla_x I_{t+dt}) \\ (\nabla_x I_{t+dt}) \cdot (\nabla_y I_{t+dt}) & (\nabla_x I_{t+dt}) \cdot (\nabla_x I_{t+dt}) \end{bmatrix} \tag{3.5}$$

Afin que la solution de l'équation 3.3 puisse converger, la fonction de l'équation 3.1 doit être rendue convexe. Pour cela, I_{t+dt} et I_t sont tout d'abord convoluées par une fonction Π de type porte de taille $N \times N$. Rigoureusement, la fonction porte est la fonction indicatrice de l'ensemble $\left[-\frac{1}{2}, \frac{1}{2}\right]$. C'est à dire que :

$$\forall x \in \mathbb{R} \quad \Pi(x) = \begin{cases} 1, \text{ si } x \in \left[-\frac{1}{2}, \frac{1}{2}\right] \\ 0 \text{ sinon} \end{cases} \tag{3.6}$$

En traitement d'images ou de vidéos, ce concept peut s'étendre aux deux dimensions d'une image. De plus, nous ne limitons généralement pas à l'ensemble $\left[-\frac{1}{2}, \frac{1}{2}\right]$ mais à $\left[-\frac{N}{2}, \frac{N}{2}\right]$. Dans le cas d'une convolution, cela se traduit par l'utilisation d'un noyau de convolution de taille $N \times N$ dont tous les coefficients sont fixés à 1. On note alors :

$$\begin{cases} \tilde{I}_{t+dt} = I_{t+dt} * \Pi \\ \tilde{I}_t = I_t * \Pi \end{cases} \tag{3.7}$$

L'équation 3.1 s'exprime alors comme dans l'équation 3.8. L'amplitude des mouvements calculables dépend de la taille de Π. Typiquement N est 2 fois supérieur à la valeur maximale de $|ds|$.

$$\sum_{s \in \Omega} \left(\tilde{I}_{t+dt}(s + ds) - \tilde{I}_t(s) \right)^2 = \varepsilon \tag{3.8}$$

Il est possible d'améliorer la précision du calcul de ds en remplaçant l'équation 3.8 par l'équation 3.9.

$$\sum_{s\in\Omega}\left(\tilde{I}_{t+dt}\left(s+\frac{ds}{2}\right)-\tilde{I}_t\left(s-\frac{ds}{2}\right)\right)^2=\varepsilon \tag{3.9}$$

Pour alléger les expressions, on définit $s^+ = \left(s+\frac{ds}{2}\right)$ et $s^- = \left(s-\frac{ds}{2}\right)$. L'équation 3.2 s'exprime alors comme dans l'équation 3.10.

$$\sum_{s\in\Omega}\begin{bmatrix}\left(\nabla_y\tilde{I}_{t+dt}(s^+)+\nabla_y\tilde{I}_t(s^+)\right)\times\left(\tilde{I}_{t+dt}(s^+)-\tilde{I}_t(s^-)\right)\\ \left(\nabla_x\tilde{I}_{t+dt}(s^+)+\nabla_x\tilde{I}_t(s^+)\right)\times\left(\tilde{I}_{t+dt}(s^+)-\tilde{I}_t(s^-)\right)\end{bmatrix}=\begin{bmatrix}0\\0\end{bmatrix}=\begin{bmatrix}F_y\\F_x\end{bmatrix}=F \tag{3.10}$$

On note $G_x = \nabla_x\tilde{I}_{t+dt}(s^+)+\nabla_x\tilde{I}_t(s^-)$ et $G_y = \nabla_y\tilde{I}_{t+dt}(s^+)+\nabla_y\tilde{I}_t(s^-)$. La matrice H est désormais exprimée par l'équation 3.11

$$H=\sum_{s\in\Omega}\frac{1}{2}\begin{bmatrix}{G_y}^2 & G_y\times G_x\\ G_y\times G_x & {G_x}^2\end{bmatrix} \tag{3.11}$$

Finalement l'algorithme permettant d'estimer les mouvements de translation de la caméra est détaillé dans l'algorithme 1.

Algorithm 1: Estimation des translations horizontale et verticale par la méthode StabLK

Input: $I0$ et $I1$: Deux images consécutives, Π une fonction porte, h et w Respectivement la hauteur et la largeur des images

Output: X et Y : Déplacements horizontal et vertical entre les deux images

// Convolutions et gradients

$\tilde{I0} \leftarrow \Pi * I0$

$\tilde{I1} \leftarrow \Pi * I1$

$(\tilde{I0}_x, \tilde{I0}_y) \leftarrow \nabla \tilde{I0}$

$(\tilde{I1}_x, \tilde{I1}_y) \leftarrow \nabla \tilde{I1}$

// Calculs de F point à point

for $i = 0$ **to** $(h-1)$ **do**

  for $j = 0$ **to** $(w-1)$ **do**

$$F(i,j) \leftarrow \begin{bmatrix} (\tilde{I0}_x + \tilde{I1}_x)(i,j) \times (\tilde{I1} - \tilde{I0})(i,j) \\ (\tilde{I0}_y + \tilde{I1}_y)(i,j) \times (\tilde{I1} - \tilde{I0})(i,j) \end{bmatrix}$$

// Calculs de H point à point

// On note $\tilde{I}_x \leftarrow \tilde{I0}_x + \tilde{I1}_x$ et $\tilde{I}_y \leftarrow \tilde{I0}_y + \tilde{I1}_y$

for $i = 0$ **to** $(h-1)$ **do**

  for $j = 0$ **to** $(w-1)$ **do**

$$H(i,j) \leftarrow \frac{1}{2} \begin{bmatrix} \left(\tilde{I}_x(i,j)\right)^2 & \left(\tilde{I}_x(i,j)\right) \times \left(\tilde{I}_y(i,j)\right) \\ \left(\tilde{I}_x(i,j)\right) \times \left(\tilde{I}_y(i,j)\right) & \left(\tilde{I}_y(i,j)\right)^2 \end{bmatrix}$$

// Calculs de H global

$$H_s \leftarrow \sum_{\substack{0 \leq i < h \\ 0 \leq j < w}} H(i,j)$$

// Calculs de F global

$$F_s \leftarrow \sum_{\substack{0 \leq i < h \\ 0 \leq j < w}} F(i,j)$$

// Résolution du système

$$\begin{bmatrix} X \\ Y \end{bmatrix} \leftarrow -\left[H_s\right]^{-1} \left[F_s\right]$$

L'extension de cette méthode à d'autres paramètres se fait assez naturellement en ajoutant la prise en compte de paramètres souhaités dans l'équation 3.1.

Extension à la rotation et à la variation d'intensité.

Nous cherchons désormais à étendre la méthode à un plus grand nombre de paramètres : translation, rotation et variation de luminosité. Pour cela, nous définissons tout d'abord les notations suivantes :

— A : le rapport d'intensité entre les images I_{t+dt} et I_t
— α : l'angle de la rotation.
— $C = \begin{bmatrix} C_y \\ C_x \end{bmatrix}$: le centre de rotation
— $s = \begin{bmatrix} i \\ j \end{bmatrix}$: les coordonnées du pixel considéré.
— $\Phi(\alpha) = \begin{bmatrix} cos(\alpha) & -sin(\alpha) \\ sin(\alpha) & cos(\alpha) \end{bmatrix}$: la matrice de rotation
— $ds = \begin{bmatrix} dy \\ dx \end{bmatrix}$: le vecteur de translation.
— $\Phi'(\alpha) = \begin{bmatrix} -sin(\alpha) & -cos(\alpha) \\ cos(\alpha) & -sin(\alpha) \end{bmatrix}$: la dérivée de Φ selon α

La valeur que l'on cherche à minimiser dans l'équation 3.1 s'exprime alors comme dans l'équation 3.12. Il suffit ensuite d'appliquer le même type de transformations qu'en équations 3.2 et 3.3 pour pouvoir résoudre le système.

$$\sum_{s\in\Omega} \left(AI_{t+dt}(\Phi(\alpha)(s-C)+C+ds) - I_t(s)\right)^2 = \varepsilon \tag{3.12}$$

On note $\overline{s} = \Phi(\alpha)(s-C)+C$. En dérivant l'équation 3.12 on obtient l'équation 3.13.

$$\sum_{s\in\Omega} 2\left(AI_{t+dt}(\overline{s}+ds) - I_t(s)\right) \times \left(\overrightarrow{\nabla}\left(AI_{t+dt}(\overline{s}+ds) - I_t(s)\right)\right) = \overrightarrow{0} = F \tag{3.13}$$

L'équation 3.13 peut également s'écrire sous la forme présentée en équation 3.14

$$\sum_{s\in\Omega} \begin{bmatrix} 2\left(A\nabla_y I_{t+dt}(\overline{s}+ds)\right) \times \left(AI_{t+dt}(\overline{s}+ds) - I_t(s)\right) \\ 2\left(A\nabla_x I_{t+dt}(\overline{s}+ds)\right) \times \left(AI_{t+dt}(\overline{s}+ds) - I_t(s)\right) \\ 2\left(\Phi'(\alpha)(s-C)\right)\left(A\overrightarrow{\nabla}_\Phi I_{t+dt}(\overline{s}+ds)\right)\left(AI_{t+dt}(\overline{s}+ds) - I_t(s)\right) \\ 2\left(I_{t+dt}(\overline{s}+ds)\right)\left(AI_{t+dt}(\overline{s}+ds) - I_t(s)\right) \end{bmatrix} = \overrightarrow{0} = F \tag{3.14}$$

En considérant α proche de 0 on peut écrire :

$$\Phi'(\alpha) \approx \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}$$

Comme précédemment, les images I_{t+dt} et I_t sont convoluées par la fonction porte Π. En considérant A proche de 1 et en appliquant le même type de transformation que dans l'équation 3.8, la résolution du système se fait de façon similaire à celle proposée en équation 3.3 :

$$\begin{bmatrix} dy \\ dx \\ \alpha \\ A-1 \end{bmatrix} \approx -H^{-1}F \tag{3.15}$$

avec :

$$F = \begin{bmatrix} G_y \cdot \delta \\ G_x \cdot \delta \\ G_\alpha \cdot \delta \\ 2G_A \cdot \delta \end{bmatrix} \text{ et } H = \begin{bmatrix} \frac{1}{2}G_y \cdot G_y & \frac{1}{2}G_y \cdot G_x & \frac{1}{2}G_y \cdot G_\alpha & G_y \cdot G_A \\ \frac{1}{2}G_x \cdot G_y & \frac{1}{2}G_x \cdot G_x & \frac{1}{2}G_x \cdot G_\alpha & G_x \cdot G_A \\ \frac{1}{2}G_\alpha \cdot G_y & \frac{1}{2}G_\alpha \cdot G_x & \frac{1}{2}G_\alpha \cdot G_\alpha & G_\alpha \cdot G_A \\ G_A \cdot G_y & G_A \cdot G_x & G_A \cdot G_\alpha & 2G_A \cdot G_A \end{bmatrix} \tag{3.16}$$

tels que :

$$\begin{cases} G_y = \nabla_y \tilde{I}_{t+dt} + \nabla_y \tilde{I}_t \\ G_x = \nabla_x \tilde{I}_{t+dt} + \nabla_x \tilde{I}_t \\ G_\alpha(i,j) = (i - Cy) \times G_x(i,j) - (j - Cx) \times G_y(i,j) \\ G_A = \tilde{I}_{t+dt} \\ \delta = \tilde{I}_{t+dt} - \tilde{I}_t \end{cases} \tag{3.17}$$

Finalement l'algorithme complet à 4 paramètres est présenté dans l'algorithme 2.

Algorithm 2: Estimation des translations, rotation et variation de luminosité par la méthode StabLK

Input: $I0$ et $I1$: 2 images consécutives, Π fonction porte, (C_y, C_x) cordonnées du centre de rotation (centre de l'image), h et w Hauteur et largeur des images

Output: x, y, α et A : Respectivement les déplacements horizontal et vertical, angle de rotation et variation d'intensité lumineuse

// Convolutions et gradients

$\tilde{I0} \leftarrow \Pi * I0$

$\tilde{I1} \leftarrow \Pi * I1$

$(\tilde{I1}_x, \tilde{I1}_y) \leftarrow \nabla \tilde{I0}$

$(I1_{fx}, I1_{fy}) \leftarrow \nabla \tilde{I1}$

// Calculs de G_x, G_y, Ga, G_A et δ

$G_x \leftarrow \tilde{I1}_x + I1_{fx}$

$G_y \leftarrow \tilde{I1}_y + I1_{fy}$

for $i = 0$ **to** $(h-1)$ **do**

  for $j = 0$ **to** $(w-1)$ **do**

    $G_\alpha \leftarrow (i - Cy)G_x(i,j) - (j - Cx)G_y(i,j)$

$G_A \leftarrow \tilde{I1}$

$\delta \leftarrow \tilde{I1} - \tilde{I0}$

// Calculs de F point à point

for $i = 0$ **to** $(h-1)$ **do**

  for $j = 0$ **to** $(w-1)$ **do**

    $$F(i,j) \leftarrow \begin{bmatrix} (\delta(i,j) \times G_x(i,j)) \\ (\delta(i,j) \times G_y(i,j)) \\ (\delta(i,j) \times G_\alpha(i,j)) \\ (2\delta(i,j) \times G_A(i,j)) \end{bmatrix}$$

// Calculs de H point à point

for $i = 0$ **to** $(h-1)$ **do**

  for $j = 0$ **to** $(w-1)$ **do**

    $H(i,j) \leftarrow$

$$\begin{bmatrix} \frac{1}{2}G_x(i,j) \times G_x(i,j) & \frac{1}{2}G_x(i,j) \times G_y(i,j) & \frac{1}{2}G_x(i,j) \times G_\alpha(i,j) & G_x(i,j) \times G_A(i,j) \\ \frac{1}{2}G_y(i,j) \times G_x(i,j) & \frac{1}{2}G_y(i,j) \times G_y(i,j) & \frac{1}{2}G_y(i,j) \times G_\alpha(i,j) & G_y(i,j) \times G_A(i,j) \\ \frac{1}{2}G_\alpha(i,j) \times G_x(i,j) & \frac{1}{2}G_\alpha(i,j) \times G_y(i,j) & \frac{1}{2}G_\alpha(i,j) \times G_\alpha(i,j) & G_\alpha(i,j) \times G_A(i,j) \\ G_A(i,j) \times G_x(i,j) & G_A(i,j) \times G_y(i,j) & G_A(i,j) \times G_\alpha(i,j) & 2G_A(i,j) \times G_A(i,j) \end{bmatrix}$$

// Calculs de H global

$$H_s \leftarrow \sum_{\substack{0 \le i < h \\ 0 \le j < w}} H(i,j)$$

// Calculs de F global

$$F_s \leftarrow \sum_{\substack{0 \le i < h \\ 0 \le j < w}} F(i,j)$$

// Résolution du système

$$\begin{bmatrix} x \\ y \\ \alpha \\ A - 1 \end{bmatrix} \leftarrow -[H_s]^{-1} \times [F_s]$$

3.2.3 Implémentation et optimisations

Dans cette partie, nous présentons les différentes optimisations algorithmiques et d'implémentation apportées à la méthode StabLK afin de minimiser le plus possible le temps de traitement. L'impact de ces optimisations est discuté dans la section suivante.

La principale optimisation apportée à StabLK porte sur le calcul des convolutions des deux images d'entrée avec la fonction porte Π de taille $N \times N$ (avec $N = 2r + 1, r \in \mathbb{N}^*$). Comme précisé en section 3.2.2 N est à priori grand puisque sa valeur est environ 2 fois supérieure à l'amplitude maximale des mouvements que l'on cherche à pouvoir compenser. Selon les applications, cette amplitude peut varier d'une dizaine de pixels à plusieurs centaines de pixels. On voit bien que, même dans le cas le plus favorable, le noyau de convolution de Π sera supérieur à 20×20 pixels.

Ce dernier point est problématique. En effet, le nombre d'accès mémoire croît de façon quadratique avec le diamètre du noyau de convolution et cela est donc particulièrement pénalisant avec des noyaux de grande taille telle que ceux que nous considérons ici pour notre application.

La convolution par une porte est une sommation sur un voisinage de chacun des pixels de l'image. La figure 3.7 illustre le cas de la somme sur un voisinage 3×3. La production d'un pixel requiert $3 \times 3 = 9$ accès mémoire en lecture et 1 en écriture pour le résultat. Pour un voisinage quelconque on aura $N^2 + 1$ accès mémoire.

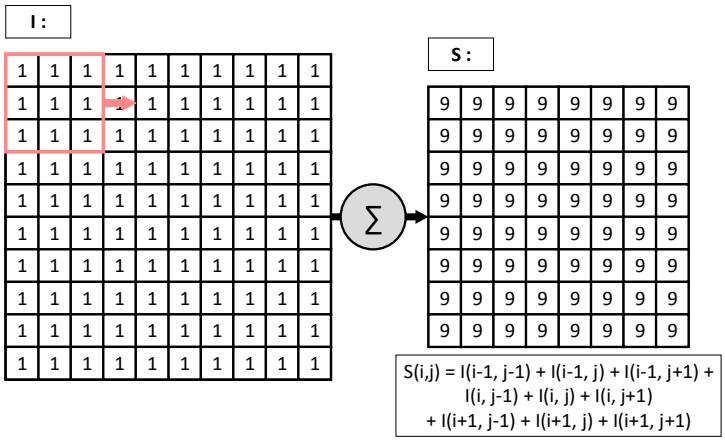


FIGURE 3.7 – Convolution $N \times N$ standard : $N^2 + 1$ accès mémoire par pixel.

Cependant, la convolution par une porte est séparable. Cela permet une optimisation très connue et possible pour de nombreux autres filtres. La séparation du filtre permet de rendre l'accroissement du nombre d'accès linéaire et non plus quadratique. La figure 3.8 illustre la convolution séparée pour un voisinage 3×3. A partir de l'image d'entrée I on calcule un premier résultat temporaire R_t qui applique la transformation dans une seule dimension (1×3). Le résultat final R est ensuite calculé à partir de R_t en

appliquant la transformation dans la seconde dimension (3×1). De cette manière, dans l'exemple de la figure 3.8, 6 accès mémoire en lecture et 2 en écriture sont nécessaires pour produire chaque pixel. En étendant à un noyau de taille $N \times N$ quelconque, on a bien une évolution linéaire du nombre d'accès avec $2N + 2$ accès. A partir de $N = 3$ cette solution est avantageuse par rapport à la convolution standard.

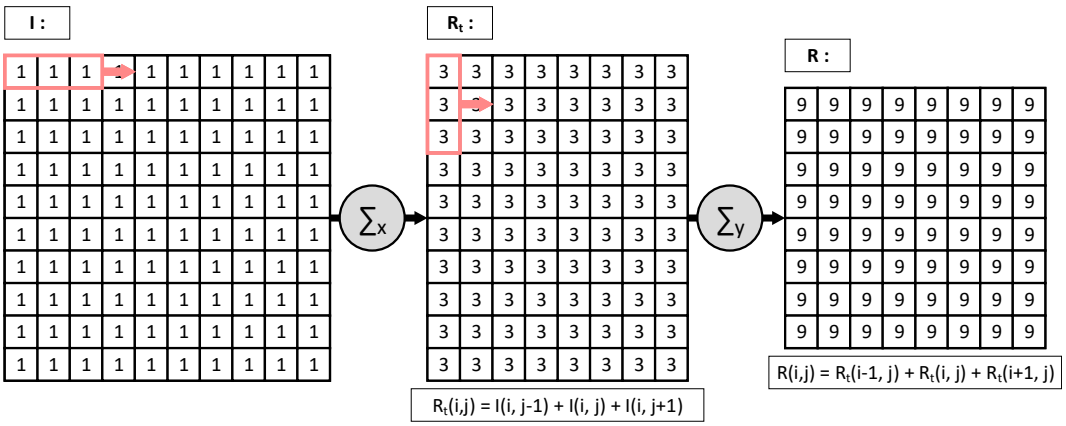


FIGURE 3.8 – Convolution $N \times N$ séparée : $2N + 2$ accès mémoire par pixel.

Si la convolution séparée permet de réduire de façon conséquente le nombre d'accès mémoire, sa complexité reste dépendante de la taille du noyau de convolution. Cela reste un problème et son calcul reste long pour de grands voisinages. Une autre méthode existe pour calculer efficacement la somme sur de grands voisinages. Il s'agit du calcul par image intégrale [87].

Le terme d'image intégrale est utilisé en français et en anglais mais le terme *Summed Area Table* peut aussi être utilisé en anglais. Soit I, une image de taille $h \times w$, l'image intégrale S de I est l'image de taille $h \times w$ telle que : chaque pixel a pour valeur la somme des pixels de I situés au dessus et à sa gauche. Plus rigoureusement : $S(i,j) = \sum_{\substack{k \leq i \\ l \leq j}} I(k,l)$.

Comme illustré dans la figure 3.9, le calcul de l'image intégrale peut se faire de façon incrémentale et nécessite alors 4 lectures et 1 écriture en mémoire par pixel. On peut noter que, par construction, le dernier pixel de S (en bas à droite) a pour valeur la somme de tous les pixels de I.

La figure 3.10 montre comment calculer le résultat R de la somme sur un voisinage d'une image I à partir de son image intégrale correspondante S. La somme sur un voisinage peut être considérée comme la valeur de la surface que représente ce voisinage, pondérée par la valeur des pixels à l'intérieur de la région. L'image intégrale permet d'obtenir la valeur de n'importe quelle surface rectangulaire englobant le premier pixel de I. En manipulant les surfaces comme décrit dans la figure 3.10, il devient possible de retrouver la somme de n'importe quel voisinage rectangulaire de I.

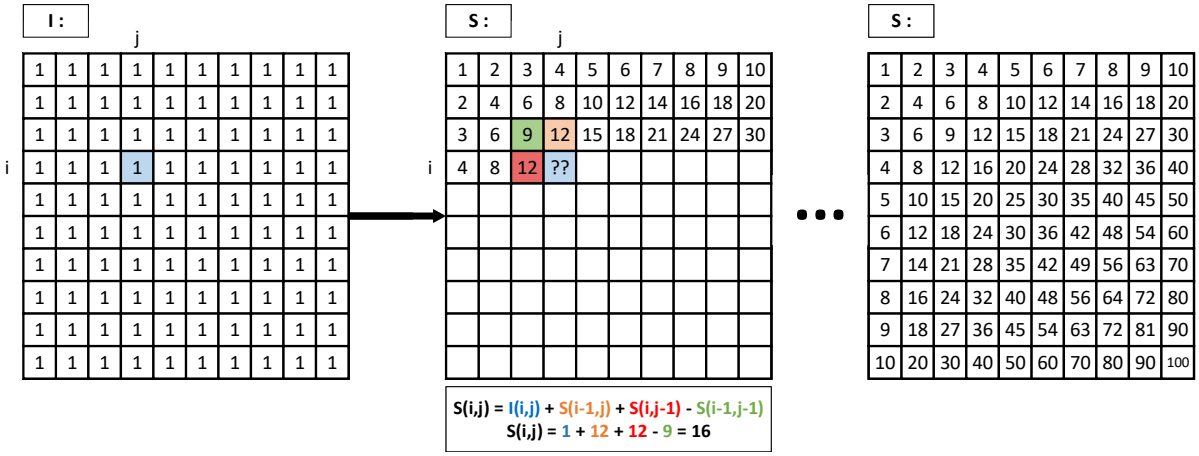


FIGURE 3.9 – Calcul de l'image intégrale S à partir de l'image d'entrée I.

Dans notre application, nous ne considérons que des voisinages carrés de taille $N \times N$ avec $N = 2r + 1$. La figure 3.10 montre que, dans ce cas, le résultat de la somme peut être obtenu par la formule :

$$R(i,j) = S(i+r, j+r) - S(i-(r+1), j+r) - S(i+r, j-(r+1)) + S(i-(r+1), j-(r+1))$$

On remarque que le nombre d'opérations et d'accès mémoire par pixel est constant et indépendant de la taille du noyau de convolution. La convolution par image intégrale nécessite donc pour chaque pixel : 4 lectures et 1 écriture pour le calcul de l'image intégrale puis à nouveau 4 lectures et 1 écriture pour le calcul du résultat final. Soit un total de 8 lectures et 2 écritures par pixels. À partir de $N = 5$, le recours à l'image intégrale requiert donc moins d'accès mémoire que la convolution séparée qui en requiert $2N + 1$. Dans notre cas, nous considérons que N est strictement supérieur à 20 et peut même être de l'ordre de 100. L'utilisation de cette technique s'avère donc pertinente pour notre application.

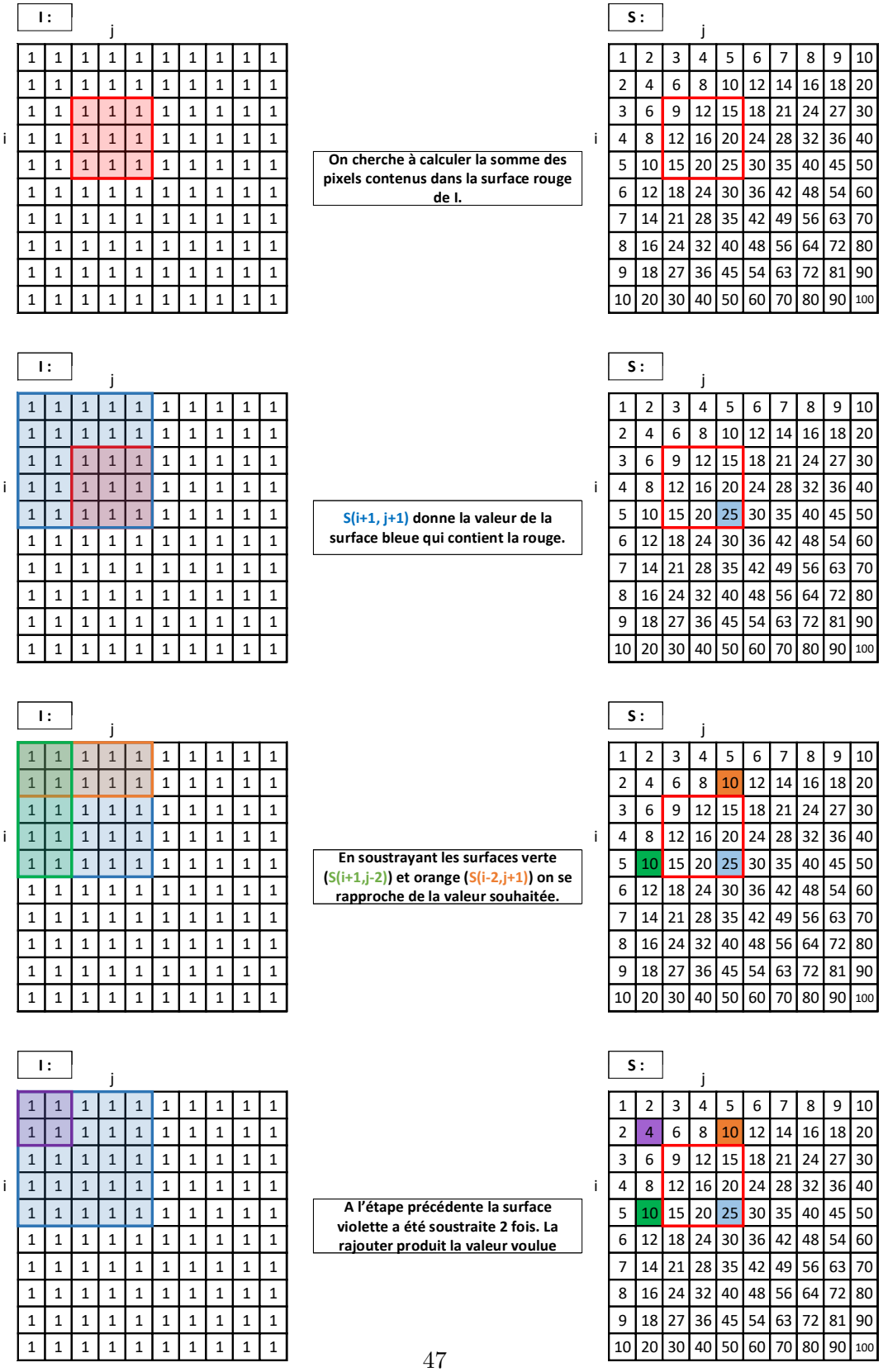
Le calcul d'image intégrale peut cependant poser plusieurs problèmes. Tout d'abord l'accumulation de tous les pixels d'une image risque de provoquer des dépassements de valeur appelés *overflows* si les calculs sont effectués en entier ou des absorptions si les calculs sont effectués en flottants. Plusieurs paramètres sont à prendre en compte pour s'assurer que cela ne puisse pas se produire :

— La définition des images : une définition plus haute augmente le nombre de pixels et donc les risques d'*overflows*.

— La dynamique du signal : une dynamique plus grande augmente la valeur maximale que peut prendre chaque pixel et donc les risques d'*overflows*.

— Le format de calcul utilisé : effectuer les calculs sur un nombre de bits plus important permet d'éloigner la limite d'*overflow* mais augmente l'empreinte mémoire et la bande passante nécessaire en plus de réduire le parallélisme SIMD maximal.

Dans la suite de ce texte, il est important de ne pas confondre le format d'une image avec le format de calcul. Le format d'une image détermine sa largeur et sa hauteur.



Finalement on a : R(i,j) = S(i+1,j+1) - S(i-2,j+1) - S(i+1,j-2) + S(i-2, j-2) = 25 - 10 - 10 + 4 = 9
Généralisation à la convolution de taille NxN avec N=2r+1 : R(i,j) = S(i+r,j+r) - S(i-r-1,j+r) - S(i+r,j-r-1) + S(i-r-1, j-r-1)

FIGURE 3.10 – Calcul de somme sur un voisinage à partir de l'image intégrale.

Exemple : une image au format Full HD fait 1920 pixels de large et 1080 pixels de haut. Le format de calcul détermine la quantité de bits utilisés pour représenter les nombres lors des calculs et la façon dont ils sont interprétés. Exemple : les calculs peuvent être effectués en flottants 32 bits.

Pour donner un ordre d'idée de l'impact des différents facteurs (définition, dynamique de signal, format de calcul) , nous présentons quelques exemples de configurations typiques. Les calculs sont généralement effectués en 16, 32 ou 64 bits. Ici, nous les considérons, dans un premier temps, sous forme d'entiers non signés. De nombreux formats standards de vidéos existent mais les plus connus et utilisés de nos jours sont l'ensemble des formats dit *16:9* et le format 4K cinéma en *17:9*. Ce sont les seuls formats d'images que nous considérons ici. Enfin, la dynamique des images est souvent considérée sur 8 bits mais des capteurs plus performants permettent d'obtenir des dynamiques plus importantes sur 10, 12 ou 14 bits.

Considérons tout d'abord le format de calculs en entiers non signés sur 32 bits : la limite d'*overflow*) est fixée à $2^{32} - 1 = 4\,294\,967\,295 \approx 4,3 \times 10^9$. Les plus grands formats sans risque d'*overflow* en fonction de la dynamique du signal sont présentés dans le tableau 3.1. On remarque que pour des dynamiques sur 12 bits ou moins, le format de calcul entiers non signés 32 bits, est suffisant pour calculer une image intégrale à partir d'images en haute définition (HD). En revanche, il n'existe pas de format d'image standard suffisamment petit qui permet d'être sûr d'éviter les *overflows* avec une dynamique de 14 bits. Dans ce cas, il est donc nécessaire de coder les données sur plus de bits. Cela peut avoir d'importantes répercussions sur les performances et le temps de calcul total.

Dynamique	Format max			Valeur max image intégrale ($\times 10^9$)	Définition max (#pixels)
	Nom	Largeur	Hauteur		
8 bits	4K	4096	2160	$\approx 2,3$	$\approx$ 17Mpix
10 bits	WQHD	2560	1440	$\approx 4,1$	$\approx$ 4Mpix
12 bits	HD	1280	720	$\approx 3,8$	$\approx$ 1Mpix
14 bits	–	672	378	$\approx 4,2$	$\approx 0,3$Mpix

TABLE 3.1 – Plus grands formats vidéo standards et définitions maximales sans risque d'*overflow* pour un calcul d'image intégrale sur des entiers 32 bits non signés. Seuls les standards 16:9 et 4K sont considérés. La "valeur max image" correspond à la valeur maximale que peut prendre un pixel de l'image intégrale avec le "format max" et la "dynamique" indiqués. La "définition max" correspond au nombre maximal de pixels qui peuvent être traités sans risque de débordement indépendamment du format de l'image.

Dans le cas de calculs effectués entiers non signés 16 bits, la limite d'*overflow* est fixée à $2^{16} - 1 = 65\,535$. La résolution maximale sans *overflow* pour une image en dynamique 8 bits est alors de 256 pixels (16×16). Cette résolution est trop petite pour pouvoir envisager un calcul sur 16 bits de l'image intégrale.

Considérons désormais que nous effectuons les calculs en flottants simple précision : sur 32 bits (F32). Dans ce cas, le problème se pose moins sur le dépassement de valeurs

que sur la précision des calculs. En simple précision, le plus grand entier représentable est $(2-2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38}$. Cette valeur est bien supérieure au maximum possible en entiers 32 bits. En revanche, la dynamique de calcul en F32 est de seulement 23 bits (+1 bit implicite), les autres bits étant réservés à l'exposant et au bit de signe. Par conséquent, seul l'intervalle d'entiers $[-2^{24}; 2^{24}] = [-16777216; 16777216]$ peut être représenté de façon exacte. Pour $[2^{24}+1; 2^{25}]$ les entiers sont arrondis à un multiple de 2, puis par un multiple de 4 jusque 2^{26} puis par un multiple de 8 jusque 2^{27} et ainsi de suite.

Se pose alors le problème d'absorption. Ce phénomène peut apparaître lorsque l'on additionne (ou soustrait) une très grande valeur avec une très petite valeur. Dans ce cas, il peut arriver que la petite valeur soit "absorbée" au moment du calcul, c'est à dire que le résultat de l'addition (ou de la soustraction) est arrondi à la même valeur que la grande valeur initiale. L'absorption est particulièrement gênante dans le cas d'une accumulation d'un grand nombre de petites valeurs comme pour le calcul de l'image intégrale. Dans ce cas, lorsque la valeur de l'accumulation devient grande par rapport aux valeurs à accumuler, on risque d'être confronté à une absorption successive des valeurs qui restent encore à accumuler. On comprend que cela est problématique si la somme des valeurs restantes, est significative par rapport la valeur de l'accumulation. Il est alors important de prendre en compte l'ordre des calculs afin de réduire les risques d'absorption.

Comme présenté en figure 3.9, pour calculer une nouvelle valeur de l'image intégrale S, issue de l'image I, on applique l'opération :

$$S(i,j) = I(i,j) + S(i-1,j) + S(i,j-1) - S(i-1,j-1)$$

Au cours de l'accumulation, $I(i,j)$ devient de plus en plus petit par rapport à $S(i-1,j)$ et $S(i,j-1)$ est risque d'être absorbé. Afin de limiter les risques d'absorption, une première approche directe est de changer l'ordre des calcul comme suit :

$$S(i,j) = (((S(i,j-1) - S(i-1,j-1)) + I(i,j)) + S(i-1,j))$$

Ainsi $I(i,j)$ est d'abord cumulé au pixels précédents de la même ligne, avant d'être cumulé au reste de l'image intégrale $S(i-1,j)$. Cette solution peut ne pas suffire si :

$$(S(i,j-1) - S(i-1,j-1)) + I(i,j) << S(i-1,j)$$

Dans ce cas, la ligne en cours est absorbée au niveau de $S(i,j)$. Cette absorption peut se propager sur le reste de la ligne et aux lignes suivantes. Pour éviter cela, nous nous inspirons de la méthode présentée dans [88]. Un accumulateur de ligne permet d'éviter que l'absorption puisse se propager sur toute une ligne.

Dans ces travaux, nous présentons les résultats obtenus pour des calculs effectués en flottants 32 bits. Nous utilisons un accumulateur de ligne afin de réduire les risques d'absorption. De plus, la parallélisation du calcul "en bandes", permet également de réduire les risques d'absorption. La méthode de parallélisation, détaillée dans la suite de ce chapitre, permet de n'effectuer le calcul d'image intégrale que sur une portion de

l'image complète. Par exemple, pour une parallélisation sur 4 cœurs, le nombre de pixels accumulés est divisé par 4. Pour une image Full HD en dynamique 8 bits et des calculs en F32, la valeur maximale l'accumulation est :

$$Max = \frac{1920 \times 1080 \times 255}{4} = 132192000 < 2^{27}$$

En dessous de 2^{27} les entiers naturels sont, au pire, arrondis à un multiple de 8. Dans le cas le plus défavorable, une ligne sera donc absorbée si la somme de ses pixels est inférieure à 4.

Une implémentation avec une accumulation en entiers non signés sur 32 bits, associée à une réduction supplémentaire de la taille de la fenêtre d'accumulation, est en cours de développement mais n'est pas présentée ici.

Un autre problème rencontré lors du calcul de l'image intégrale est sa parallélisation. En effet, de part sa nature incrémentielle, l'algorithme de calcul de l'image intégrale n'est intrinsèquement pas parallèle. Plusieurs travaux ont cependant permis de mettre en place des techniques de parallélisation efficace. La majeure partie de ces travaux reposent sur la mise en place d'une architecture matérielle spécifique[89, 90]. En 2011, une approche sur processeur généraliste est proposée par Wu [91]. L'ensemble de ces méthodes reposent sur une approche similaire. L'image d'origine est subdivisée en plusieurs sous-images. A partir de ces sous-images, des images intégrales partielles sont calculées en parallèle et de façon indépendante. Une étape de fusion est ensuite appliquée pour obtenir l'image intégrale complète.

Pour notre application, nous proposons une nouvelle approche qui, à notre connaissance, n'a encore jamais été proposée. Cela est certainement dû au fait que, dans la littérature, les optimisations de calcul d'image intégrale sont proposées pour un cas général : le calcul d'image intégrale est une fin en soit et n'est pas partie intégrante d'une application précise. Dans notre cas cependant, il est possible de ne jamais calculer d'image intégrale complète. En effet, l'image intégrale n'est ici qu'un moyen de calculer une seule convolution. Il s'agit d'un résultat intermédiaire, s'il est possible de ne calculer le produit de convolution qu'à partir des images intégrales partielles, il n'est pas nécessaire d'effectuer les opérations de fusion qui cassent le parallélisme. C'est cette idée qui est au cœur de notre méthode.

Soit N_{th} le nombre de threads exécutés en parallèle. L'image d'origine I de taille $h \times w$ est subdivisée en N_{th} bandes de hauteur $h_{bande} = \frac{h}{N_{th}}$. Pour chaque thread t, une image intégrale S_t est calculée à partir d'une bande. Le résultat R de la somme est calculé pour chaque bande d'image à partir de l'image intégrale partielle associée. Un problème se pose alors à chaque frontière entre 2 bandes. Soit i_{debut} et i_{fin} les indices des première et dernière lignes d'une bande quelconque et $N = 2r + 1$ le diamètre du noyau de convolution. On considère un pixel de coordonnées (i, j). On a :

$$R(i,j) = S_t(i+r, j+r) - S_t(i-r-1, j+r) - S_t(i+r, j-r-1) + S_t(i-r-1, j-r-1)$$

On remarque que si $i \leq i_{debut} + r$ ou $i \geq i_{fin} - r$, une partie du voisinage du pixel n'est pas disponible car présent uniquement dans la bande voisine. Le calcul de la somme n'est donc pas possible en tous points.

Pour contourner ce problème, on ajoute à chaque bande un bord de r lignes au dessus et au dessous appartenant aux bandes voisines. Les bandes ainsi créées se chevauchent les unes les autres. Les images intégrales partielles sont donc en réalité calculées à partir de ces bandes nouvellement définies. De cette façon chaque image intégrale partielle de hauteur $h_{bande} + 2r$ permet de calculer une portion du résultat final R d'une hauteur de h_{bande}. La figure 3.11 illustre la méthode de calcul parallèle de la convolution avec images intégrales partielles.

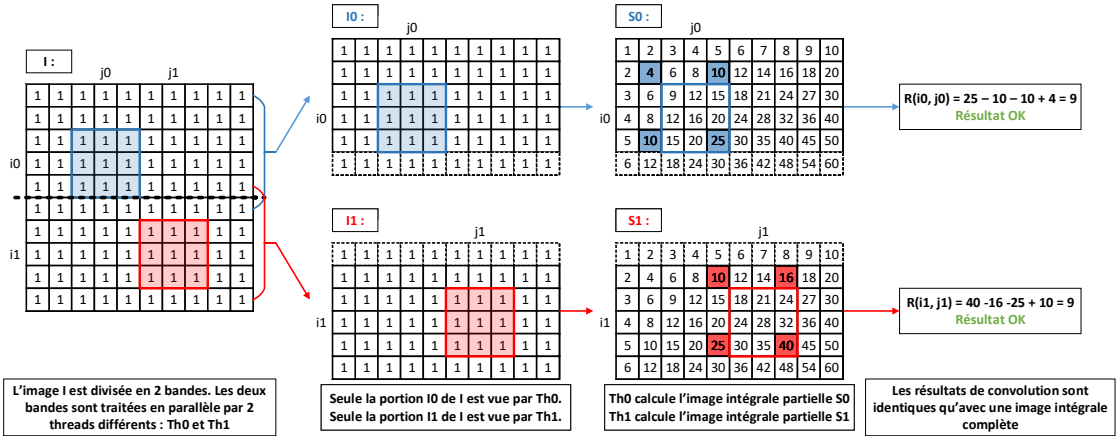


FIGURE 3.11 – Calcul parallèle de somme sur un voisinage à partir d'images intégrales partielles.

La méthode de calcul parallèle proposée ici, entraîne une redondance des calculs au niveau des frontières des différentes bandes. En revanche cela permet d'éviter les phases de fusion qui cassent le parallélisme. En outre, en utilisant cette méthode il est possible de pipeliner les calculs de l'image intégrale et de la convolution dans lequel les images intégrales sont allouées de façon modulaire (voir chapitre 2 pour plus détails sur le pipeline d'opérateurs et l'allocation modulaire).

Une autre optimisation possible pour StabLK est d'éviter une redondance dans le calcul de la matrice hessienne H en remarquant que cette dernière est symétrique. En définitive, nous proposons une implémentation de StabLK utilisant les optimisations présentées dans cette section et en chapitre 2. Ses performances sont discutées en section suivante.

3.2.4 Résultats et performances de StabLK

Dans cette section, nous étudions l'impact des différentes optimisations apportées à StabLK. La plateforme utilisée pour fournir les résultat est la carte Nvidia Jetson AGX Xavier [92] (AGX). L'implémentation proposée n'utilise pas le GPU qui ne sera donc pas considéré pour l'instant. Le CPU embarqué de l'AGX possède 8 cœurs Carmel ARM V8.2 cadencés jusqu'à 2,3GHz.

Différentes implémentations de StabLK sont comparées afin de déterminer l'impact des transformations sur le temps d'exécution :

— **Base** : Implémentation non optimisée de StabLK.
— **II** : Implémentation avec Image Intégrale.
— **Pipe** : Pipeline de l'image intégrale et de la convolution.
— **AM** : Pipe + Allocation Modulaire de l'image intégrale.
— **SIMD** : AM + SIMDisation. (Registres SIMD 128 bits soit $4 \times F32$)
— **OMPx** : SIMD + Parallélisation sur x threads grâce au calcul d'images intégrales partielles.

Pour les mesures, la fréquence du CPU est fixée à 2,3GHz. Chaque implémentation est exécutée sur 1 thread (sauf pour les versions OMPx qui sont exécutées sur x threads). Toute application autre que StabLK est coupée avant de démarrer les mesures de temps. Le système de refroidissement actif est au maximum. Les images utilisées sont au format fullHD (1920×1080 pixels). Le rayon du noyau utilisé pour la convolution porte est de 12 pixels, soit un noyau de taille 25×25.

Le tableau 3.2 indique le temps d'exécution de StabLK pour les configurations considérées. Pour les implémentations mono-thread, les accélérations sont indiquées par rapport à l'implémentation directement précédente. Pour les implémentations multi-thread, les accélérations sont indiquées par rapport à l'implémentation mono-thread la plus rapide. Enfin, l'accélération totale entre le version la plus lente et la plus rapide est donnée.

Version	Base	II	Pipe	AM	SIMD	OMP2	OMP4	OMP8	**Total**
Temps (ms)	1742	56,0	33,5	29,0	9,2	5,5	2,8	1,6	–
Accélération	×1	×31	×1,75	×1,2	×3,1	×1,7	×3,3	×5,9	×1120

TABLE 3.2 – Impact des optimisations de StabLK pour des images Full HD sur le CPU de l'AGX.

On constate que, sur 8 threads, l'ensemble des optimisations apportées permet un gain en vitesse de ×1120 par rapport à l'implémentation de base non parallélisée. L'accélération la plus importante est, de loin, apportée par l'utilisation de l'image intégrale : ×31. Ici, le rayon du noyau de convolution est de 12 pixels. Comme expliqué précédemment, ce rayon peut être bien plus grand. Contrairement à une convolution classique ou séparée, le temps de calcul de la convolution par image intégrale ne dépend pas de la taille du noyau de convolution. C'est pourquoi, le gain apporté par l'image intégrale sera d'autant plus important que le noyau est grand.

Par rapport au maximum théorique, la parallélisation multi-tâche a une efficacité entre 74% et 85%. Plus le nombre de *threads* augmente, moins la parallélisation est efficace. Sur 2 cœurs l'accélération est de $\times 1,7$ soit 85% du maximum de $\times 2$ théorique. Sur 4 cœurs l'accélération est de $\times 3,3$ soit 83% d'efficacité. Enfin, sur 8 cœurs l'accélération est de $\times 5,9$ soit 75% d'efficacité. Cela peut s'expliquer par deux points principaux. Tout d'abord, la redondance des calculs est plus importante avec l'augmentation du parallélisme. En effet, les images sont découpées en bandes d'autant plus fines qu'il y a de threads disponibles. Hors, le bord supplémentaire ajouté à ces bandes dépend de la taille du noyau et non du nombre de threads. La surface "utile" de chaque bande est donc proportionnellement moins importante. De plus, par construction, StabLK possède une portion non parallèle : Il est nécessaire à la fin d'accumuler l'ensemble des résultats calculés en parallèle pour pouvoir calculer le résultat final. En accord avec la loi d'Amdhal [93] cela induit une limite quant aux gains dûs à la parallélisation multi-threads.

Les calculs sont effectués en F32 et les registres SIMD NEON utilisés sont sur 128 bits. Les registres ont donc un cardinal de 4. La parallélisation SIMD permet une accélération de $\times 3,1$ soit une efficacité de 78%.

3.2.5 Synthèse : StabLK

Dans cette partie, nous avons présenté une méthode de stabilisation appelée StabLK. Cette méthode est comparable à l'algorithme de calcul de flot optique introduit par Lucas et Kanade en 1981 [40]. L'approche cependant est ici globale et non point à point. Les mouvements de translation, de rotation et la variation d'intensité lumineuse peuvent être estimés grâce à cette méthode.

Une première phase de stabilisation dans la chaîne de traitement permet de faciliter le calcul de flot optique dense par la suite. Cette étape peu coûteuse en termes de temps de calcul, permet d'alléger la charge de l'étape suivante plus complexe de TV-L^1. La méthode StabLK étant maîtrisée et ayant montré sont efficacité au sein de Lheritier, c'est cette méthode qui a naturellement été retenue pour l'étape de stabilisation. En outre, les implémentations optimisées de StabLK se font d'ordinaire sur les FPGA des caméras Lheritier. L'implémentation optimisée sur processeur généraliste permet d'étendre la maîtrise de cette méthode à une nouvelle architecture.

L'implémentation de StabLK bénéficie de plusieurs optimisations dont la convolution par image intégrale, le pipeline d'opérateurs, les parallélisations SIMD et de tâches. L'accélération la plus importante (de $\times 31$) est obtenue grâce à l'utilisation des images intégrales. La parallélisation de l'algorithme est rendue possible grâce au calcul d'images intégrales partielles indépendantes. Cette méthode est, à notre connaissance, la seule permettant un calcul de convolution par image intégrale sans rupture du parallélisme. Cela permet sur 8 cœurs d'être $\times 5,9$ plus rapide que sur un seul, soit une efficacité de 75%. La troisième plus importante accélération est apportée par la parallélisation SIMD qui permet d'être 3,3 fois plus rapide avec des données de 32 bits et un jeu d'instruction de 128 bits (cardinal de 4).

3.3 Recalage temporel par calcul de flot optique dense

L'étape de recalage temporel a pour but de compenser les mouvements de chaque pixel entre deux images. Cela permet d'éviter l'apparition d'un flou de bougé lors du filtrage temporel. Le mouvement apparent entre deux images consécutives s'appelle le flot optique. L'estimation du flot optique permet d'obtenir un vecteur vitesse différent en tout point. Ce type d'estimation est appelé dense, en opposition aux méthodes creuses (en anglais *sparse*) qui n'estiment le mouvement que pour quelques points. Dans RTE-VD, nous estimons le flot optique par la méthode TV-L[1] [41]. A partir de cette estimation, le recalage temporel est effectué par une interpolation bicubique.

3.3.1 Motivations et choix de la méthode

Le recalage temporel peut s'effectuer de 2 façons différentes :
— En utilisant un algorithme de calcul de flot optique dense.
— En effectuant une mise en correspondance de blocs. Le terme de *block matching* est généralement employé.

Une majorité d'algorithmes de débruitage utilisent le *block matching* [27, 29, 94, 25, 26]. Si ce dernier permet un débruitage efficace, il requiert cependant d'importantes ressources et est difficile à accélérer.

Soient 2 images consécutives I_0 et I_1. Le *block matching* permet d'estimer le mouvement entre ces deux images en faisant correspondre un voisinage de pixels (ou bloc) de I_0, avec un voisinage de I_1. Pour cela, pour chaque pixel p de I_0, on considère son voisinage qui constitue le bloc de pixels B. On recherche dans I_1, le bloc de pixel $\tilde{B}$ le plus proche de B selon un critère donné : par exemple le bloc dont la différence avec les pixels de B est la plus petite. On considère alors que l'objet représenté par B dans I_0 et représenté par $\tilde{B}$ dans I_1. On en déduit le déplacement de l'objet représenté par p entre I_0 et I_1. En général, la recherche de $\tilde{B}$ ne s'effectue pas sur l'ensemble de I_1 mais est limitée à une portion d'image autour des coordonnées de p. Cette fenêtre de recherche est plus ou moins grande en fonction de l'amplitude maximale du mouvement que l'on souhaite pouvoir mesurer. Usuellement ce n'est pas un mais plusieurs blocs similaires qui sont associés à B.

Lorsque l'on utilise une méthode de *block matching*, dans le cadre du débruitage, il n'est pas nécessaire de produire un champ de vecteur correspondant au flot optique mesuré. En effet, le champ de vecteurs permet d'effectuer le recalage temporel des pixels. Ici, par construction, le recalage temporel est fait avant l'estimation du mouvement : on estime le mouvement à partir du recalage et non l'inverse. Il n'y a donc pas de phase de calcul de flot optique. Il est cependant possible de guider la recherche de blocs avec un calcul préalable du flot optique [28].

Intrinsèquement, la compensation par *block matching* requiert de très nombreux accès mémoire lors de la recherche de blocs similaires. La recherche de blocs se fait sur un grand nombre de pixels. Un moyen de réduire cette zone de recherche est d'effectuer

une estimation préalable du flot optique. Cela entraîne le besoin d'avoir une bande passante importante et le modèle d'accès mémoire est susceptible de générer beaucoup de défauts de cache. Les temps de traitement annoncés pour ce type de méthodes sont d'ailleurs particulièrement grands (de quelques secondes à quelques minutes [10, 29, 26, 25] Il apparaît donc difficile d'accélérer suffisamment le temps de traitement d'une telle méthode pour envisager une implémentation en temps réel sur système embarqué. C'est pour cette raison que les méthodes de débruitage par *block matching* n'ont pas été retenues pour notre application.

Si, dans le cadre de notre application, il n'est pas envisageable d'utiliser du *block matching* ; il demeure indispensable de conserver une cohérence temporelle du filtrage grâce au recalage temporel. Nous choisissons donc d'utiliser un algorithme d'estimation de flot optique dense.

Bien que moins lourd qu'une approche par *block matching*, estimer le flot optique en temps réel reste difficile pour un système embarqué. Cela requiert en effet, beaucoup de puissance de calcul alors que la consommation d'énergie doit rester maîtrisée. Le processeur utilisé doit alors faire un compromis sur le rapport entre la performance et la consommation. Il est connu que ce rapport est meilleur pour les petits cœurs de calculs [95]. Les architectures de type CPU SIMD ou GPU apparaissent comme de bons candidats pour obtenir les performances requises dans les contraintes de consommation imposées. Dans ces travaux, nous nous concentrons essentiellement sur les architectures SIMD. Bien que abordées ici, les architectures GPU feront l'objet d'un travail plus approfondit dans un second temps.

Plus d'une centaine d'algorithmes de flot optique existent. Chacun possède ses propres caractéristiques que ce soit en termes de complexité, de vitesse d'exécution, de précision des résultats ou encore de la gestion de situations complexes telles que les occlusions. Une analyse qualitative [96] substantielle et à jour, est disponible sur le site de Middleburry [86]. Des codes sources y sont également proposés. Le site internet IPOL [97] (*Image Processing On Line*), propose également différentes implémentations d'algorithmes récents.

Les algorithmes de calcul de flot optique sont sensibles aux transformations de code. Leur optimisation a d'ailleurs fait l'objet de nombreux travaux, particulièrement sur FPGA [98, 99, 100, 101] et sur GPU [102, 103, 104, 105, 106, 107] mais assez peu sur CPU [106, 108]. Le récent gain en popularité dans la communauté scientifique des méthodes d'estimation de flot optique par *machine learning*, est également notable [109, 110, 30].

Le grand nombre d'algorithmes existants nous empêche de proposer une étude exhaustive permettant de choisir la meilleure méthode pour notre application. En revanche, plusieurs critères peuvent orienter notre choix. D'après [33], il est important que l'estimation du flot soit robuste au bruit pour pouvoir apporter un débruitage efficace. En outre, le flot doit être dense et fournir une information de déplacement pour chaque pixel afin de conserver un maximum de détails. La gestion des discontinuités dans le champ

de vecteurs est également un point à prendre en compte. Il est en effet important de pouvoir rester efficace lorsqu'une forte discontinuité apparaît dans la scène : véhicules roulant en sens inverse et se croisant, jambes d'un individu en mouvement... En raison de nos besoins d'embarquabilité, la complexité algorithmique de la méthode employée est également un critère déterminant. Pour ces raisons, TV-L^1 apparaît comme une méthode pertinente pour notre application.

TV-L^1 a été introduit en 2007 par Zach, Pock et Bischof dans [41]. Il s'agit d'un algorithme de référence bien connu pour le calcul de flot optique dense. Parmi les différentes méthodes envisagées, TV-L^1 semble apporter un bon compromis entre qualité du flot optique, robustesse au bruit et complexité algorithmique. En comparaison avec d'autres méthodes de référence comme Lucas et Kanade [40] ou encore Horn et Schunck [111], TV-L^1 apporte une réelle amélioration en termes de qualité de flot optique estimé. En revanche, le temps de calcul est plus important. D'autres méthodes avec une estimation plus précise existent. Cependant, elles sont généralement plus lentes. Plusieurs de ces méthodes plus précises ont une base de TV-L^1 à laquelle sont ajoutées des étapes supplémentaires [112, 113, 114]. Dans l'optique éventuelle d'utiliser de telles méthodes il est donc pertinent dans un premier temps d'implémenter et d'optimiser TV-L^1 dans sa version standard.

D'autres méthodes semblent également envisageables mais n'ont pas été retenues :

En 2016 Adarve a présenté une méthode de calcul de flot optique dense temps réel présentant de bonnes propriétés [105]. Cette méthode n'a cependant pas été retenue pour notre application en raison de la latence nécessaire pour obtenir un flot dense. En effet, dans cette méthode, le flot est densifié au fur et à mesure de la séquence vidéo. Le flot optique ne devient dense qu'à partir d'environ 300 images. Dans certaines circonstances, on peut considérer que cela n'est pas gênant : une fois la phase d'initialisation passée, le calcul de flot est efficace et le temps de traitement faible. Cependant, en cas de changement brutal de la scène une nouvelle phase d'initialisation est nécessaire.

Les méthodes d'estimation de flot optique utilisant des réseaux de neurones connaissent également un essor important dans la littérature récente [115, 110, 116, 117]. La méthode TOFlow (*Task Oriented Flow*), présentée en 2017 [30], est d'ailleurs utilisée, entre autres, pour effectuer du débruitage vidéo. Bien que les résultats qualitatifs soient parmi les meilleurs de l'état de l'art, ces méthodes restent pour l'heure trop lourdes pour envisager une implémentation embarquée temps réel. En outre, il est important de tenir compte de l'utilité des systèmes dans lesquels RTE-VD est susceptible d'être implémenté. L'observation à longue distance, en conditions de faible luminosité, est principalement utile pour des applications militaires ou de maintient de l'ordre. Il est donc pertinent, pour le choix de notre méthode, de prendre en compte l'avis des opérationnels, c'est à dire des personnes amenées à utiliser des systèmes intégrant la chaîne de traitement. Hors, beaucoup d'opérationnels sont encore méfiants envers les systèmes de visions à base d'intelligence artificielle. Si les méthodes à base de réseaux de neurones montrent en effet de très bon résultats académiques, il est difficile de prévoir leur comportement en conditions opérationnelles. Les bases de données représentatives de ce genre de situa-

tion ne sont en effet ni faciles d'accès ni suffisamment grandes pour permettre un bon apprentissage de la part du réseau.

Un test préliminaire interne à LHERITIER permet de conforter notre choix d'utiliser TV-L^1 en le comparant à deux autres méthodes plus simples : Lucas & Kanade (LK) [40] et Horn & Schunck (HS) [111]. De la plus rapide à la plus lente, les 3 méthodes sont classées comme suit : LK, HS, TV-L^1. Pour ce test, nous appliquons un même filtrage spatio-temporel à partir d'un recalage temporel effectué grâce à LK, HS ou TV-L^1. Les résultats sont visuellement comparés pour déterminer l'algorithme à implémenter. Les résultats obtenus avec LK montrent un problème de cohérence temporelle. Ce phénomène est difficile à illustrer avec une image fixe mais au moment de la visualisation de la vidéo on constate un effet gênant de turbulence. Avec HS, on constate une moins bonne gestion des détails qu'avec TV-L^1. Un exemple est donné en figure 3.12. On peut voir que la patte du dindon est mieux restituée avec TV-L^1 qu'avec HS. Les résultats de ce test tendent à confirmer notre idée que le choix de TV-L^1 est pertinent pour notre application.

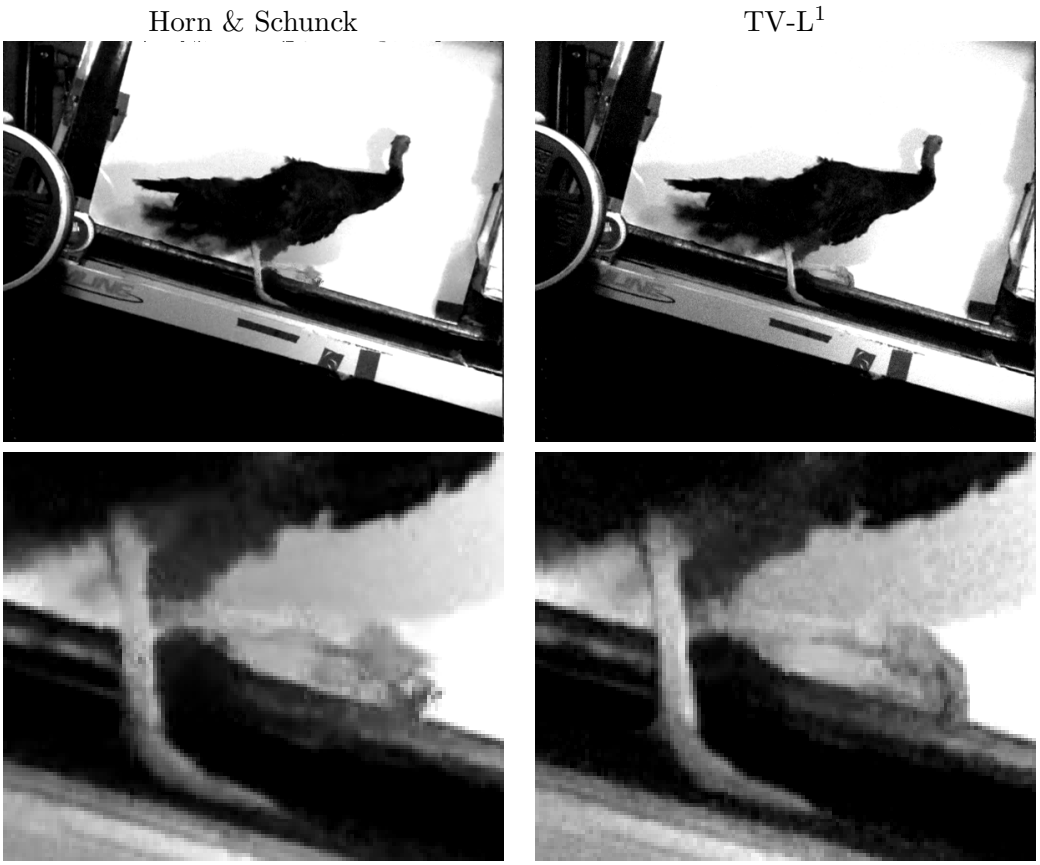


FIGURE 3.12 – Extraits des résultats du test préliminaire pour le choix de l'algorithme utilisé pour le recalage temporel. Résultats du débruitage obtenu avec un recalage temporel par HS (à gauche) puis par TV-L^1 (à droite).

Un dernier élément conforte également notre choix d'utiliser TV-L^1 : cette méthode est utilisée dans de nombreuses autres chaînes de débruitage vidéo, notamment pour orienter la zone de recherche de bloc en cas de *block matching* [27, 94, 33, 28].

3.3.2 Description de l'algorithme

L'algorithme de TV-L^1 présenté par Zach, Poch et Bischof en 2007 dans [41] a été repris en 2013 par Sanchez *et al.* et publié sur le site internet IPOL [118]. Dans cette nouvelle publication, des détails supplémentaires sur l'implémentation ainsi qu'un code source sont fournis.

Résolution mathématique du problème

Dans cette partie nous détaillons l'estimation du flot optique par TV-L^1 en reprenant notamment les explications fournies dans [41] et [118].

TV-L^1 estime le flot optique grâce à une méthode variationnelle. La méthode est basée sur la minimisation de la fonctionnelle d'énergie avec une régularisation de la variation totale en norme L^1 (*Total Variation - L^1 norm*). Pour résoudre cette minimisation, on pose l'hypothèse d'une luminosité constante. Cela signifie que si $I(x, y, t)$ est une séquence vidéo et $(x(t), y(t))$ est la trajectoire d'un point donné de l'image plane, alors $I(x(t), y(t), t)$ est constant, donc :

$$\frac{d}{dt} I(x(t), y(t), t) = 0 \tag{3.18}$$

Soit :

$$\nabla I \cdot (\dot{x}, \dot{y}) + \frac{\partial}{\partial t} I = 0 \tag{3.19}$$

On définit $\mathbf{u}(x, y) = (u_1(x, y), u_2(x, y))$ le champ de vecteurs de déplacement de tous les points du domaine de l'image. Ce champ de vecteur satisfait en tous points l'équation de flot optique suivante [111] :

$$\nabla I \cdot \mathbf{u} + \frac{\partial}{\partial t} I = 0 \tag{3.20}$$

En chaque point de l'image, la condition 3.20 est une équation linéaire à deux variables (nombre de composantes de $\mathbf{u}$). Dès lors, il y a deux fois plus de variables que d'équations et le système est mal posé. Une manière usuelle de résoudre un système sous-déterminé, est d'ajouter une condition de lissage pour régulariser $\mathbf{u}$. La solution, apportée par les pionniers Horn et Schunck [111], était de choisir $\mathbf{u}$ de façon à minimiser la fonctionnelle suivante :

$$E_{HS}(\mathbf{u}) = \int_\Omega (\nabla I \cdot \mathbf{u} + \frac{\partial}{\partial t} I)^2 + \alpha(|\nabla u_1|^2 + |\nabla u_2|^2) \tag{3.21}$$

Cette minimisation est assez simple à résoudre et permet d'estimer le flot de façon correcte dans beaucoup de situations. Cependant, le terme de régularisation en norme L^2 ne permet pas d'obtenir une solution avec de forts gradients et donc une forte discontinuité. L'équation 3.20 est valable dans le cas d'un signal continu, hors ce n'est pas le cas pour une vidéo : chaque image est capturée à intervalles de temps réguliers. L'équation 3.20 devient alors $I_1(\mathbf{x}+\mathbf{u}) - I_0(\mathbf{x}) = 0$. En linéarisant cette équation par le développement de Taylor, on obtient :

$$\rho(\mathbf{u}) = \nabla I_1(\mathbf{x}+\mathbf{u}^0) \cdot (\mathbf{u}-\mathbf{u}^0) + I_1(\mathbf{x}+\mathbf{u}^0) - I_0(\mathbf{x}) = 0 \tag{3.22}$$

Où $\mathbf{u}^0$ est proche de $\mathbf{u}$. La fonctionnelle de Horn et Schunck peut être modifiée pour favoriser les discontinuités dans le champ de vecteurs en changeant les facteurs quadratiques. On effectue alors une minimisation de la fonctionnelle d'énergie suivante, qui est la somme de la variation totale de $\mathbf{u}$ en norme L^1 :

$$E(\mathbf{u}) = \int_\Omega |\nabla u_1| + |\nabla u_2| + \lambda|\rho(\mathbf{u})| \tag{3.23}$$

Un façon de minimiser cette fonctionnelle, consiste à introduire un terme de relaxation convexe $\mathbf{v}$, tel que :

$$E_\theta(\mathbf{u},\mathbf{v}) = \int_\Omega |\nabla u_1| + |\nabla u_2| + \frac{1}{2\theta}|\mathbf{u}-\mathbf{v}|^2 + \lambda|\rho(\mathbf{v})| \tag{3.24}$$

En fixant θ très petit, E_θ est minimisée lorsque $\mathbf{u}$ et $\mathbf{v}$ sont très proches. En considérant $\mathbf{u} = \mathbf{v}$, on retrouve l'équation 3.23. Grâce à cette relaxation, E_θ peut être minimisée en fixant alternativement $\mathbf{u}$ puis $\mathbf{v}$ et en appliquant la résolution sur le variable non fixée.

— En fixant $\mathbf{v}$, on résout :

$$\min_{\mathbf{u}} \int_\Omega |\nabla u_1| + |\nabla u_2| + \frac{1}{2\theta}|\mathbf{u}-\mathbf{v}|^2 \tag{3.25}$$

— En fixant $\mathbf{u}$, on résout :

$$\min_{\mathbf{v}} \int_\Omega \frac{1}{2\theta}|\mathbf{u}-\mathbf{v}|^2 + \lambda|\rho(\mathbf{v})| \tag{3.26}$$

L'équation 3.25 correspond au modèle de variation totale de Rudin-Osher-Fatemi [119] qui peut être résolu par l'algorithme de Chambolle [120]. L'équation 3.26 quant à elle, ne dépend pas des dérivées partielles de $\mathbf{v}$ et peut donc être résolue point à point par une fonction de seuillage.

La première minimisation (équation 3.25) peut être résolue en calculant le point fixe de la suite récurrente sur le double champ de vecteur $\mathbf{P} = (\mathbf{p}_1, \mathbf{p}_2)$ tel que, pour toute itération k, on a :

$$\mathbf{p}_d^{k+1} := \frac{\mathbf{p}_d^k + \frac{\tau}{\theta} \nabla \left(v_d^{k+1} + \theta \operatorname{div}(\mathbf{p}_d^k) \right)}{1 + \frac{\tau}{\theta} \left| \nabla \left(v_d^{k+1} + \theta \operatorname{div}(\mathbf{p}_d^k) \right) \right|}, d \in \{1, 2\} \tag{3.27}$$

On en déduit $\mathbf{u}$ tel que :

$$u_d^{k+1} := v_d^{k+1} + \theta \operatorname{div}(\mathbf{p}_d^k), d \in \{1, 2\} \tag{3.28}$$

La seconde minimisation (équation 3.26) se résout par la suite récurrente suivante :

$$\mathbf{v}^{k+1} := \mathbf{u}^k + TH(\mathbf{u}^k, \mathbf{u}^0) \tag{3.29}$$

Où TH est la fonction de seuillage suivante :

$$TH(\mathbf{u}, \mathbf{u}^0) := \begin{cases} \lambda\theta \nabla I_1(\mathbf{x} + \mathbf{u^0}) & si : \rho(\mathbf{u}, \mathbf{u^0}) < -\lambda\theta |\nabla I_1(\mathbf{x} + \mathbf{u^0})|^2 \\ -\lambda\theta \nabla I_1(\mathbf{x} + \mathbf{u^0}) & si : \rho(\mathbf{u}, \mathbf{u^0}) > \lambda\theta |\nabla I_1(\mathbf{x} + \mathbf{u^0})|^2 \\ -\rho(\mathbf{u}, \mathbf{u^0}) \frac{\nabla I_1(\mathbf{x}+\mathbf{u^0})}{|\nabla I_1(\mathbf{x}+\mathbf{u^0})|^2} & si : |\rho(\mathbf{u}, \mathbf{u^0})| \leq \lambda\theta |\nabla I_1(\mathbf{x} + \mathbf{u^0})|^2 \end{cases} \tag{3.30}$$

Pour que le développement limité défini dans l'équation 3.22 soit juste, il faut que le champ de vecteurs $\mathbf{u}^0$ soit proche de $\mathbf{u}$. C'est pourquoi, nous utilisons une approche multi-échelle (ou pyramidale). Cela signifie que l'on sous échantillonne plusieurs fois les images d'origine, afin d'exécuter l'approximation du flot sur des images de plus en plus grandes. Pour les images les plus sous échantillonnées, l'amplitude des mouvements en termes de nombre de pixels est réduite. En considérant un nombre d'échelles suffisamment important et des mouvements suffisamment petits, il est possible d'initialiser $\mathbf{u}^0$ comme nul pour la première échelle.

Pseudo code et choix d'implémentation

Dans cette partie nous détaillons certaines subtilités d'implémentation de TV-L^1 ainsi que les différents paramètres nécessaires.

Nous considérons une estimation du flot optique $\mathbf{u} = (u_1, u_2)$ entre deux images consécutives I_0 et I_1. Comme dans [118] nous choisissons les opérateurs différentiels tels que :

- Le gradient appliqué sur l'image I_1 est un gradient centré.
- Le gradient appliqué sur les différentes composantes du flot $\mathbf{u}$ est un gradient avant.
- La divergence appliquée sur les différentes composantes de $\mathbf{P}$ est une divergence arrière.

Avec $\mathbf{G_c}$ l'opérateur de gradient centré tel que, pour tout pixel $I(i, j)$ d'une image I de coordonnées (i, j) on a :

$$\mathbf{G_c}(I(i,j)) = \frac{1}{2} \times \begin{pmatrix} I[i][j+1] - I[i][j-1] \\ I[i+1][j] - I[i-1][j] \end{pmatrix} \tag{3.31}$$

L'opérateur $\mathbf{G_f}$ de gradient avant est définit tel que :

$$\mathbf{G_f}(I(i,j)) = \begin{pmatrix} I[i][j+1] - I[i][j] \\ I[i+1][j] - I[i][j] \end{pmatrix} \tag{3.32}$$

Et **Div** l'opérateur de divergence arrière associé à un champ de vecteur $\mathbf{v} = (v_1, v_2)$ tel que :

$$\mathbf{Div}(\mathbf{v}(i,j)) = v_1[i][j] - v_1[i][j-1] + v_2[i][j] - v_2[i-1][j] \tag{3.33}$$

Plusieurs opérateurs impliquent des calculs d'interpolation. Le choix de la méthode d'interpolation est subjectif et peut varier selon la précision souhaitée. Dans notre cas les choix suivants ont été faits :

— *Warps* : interpolation bicubique. Un *warp* est appliqué sur I_1 et permet d'évaluer $I_1(\mathbf{x} + \mathbf{u}^0(\mathbf{x}))$. Un ou plusieurs *warps* peuvent être appliqués à chaque échelle d'image entre les itérations.

— Fonction *ZoomOut* : interpolation bicubique. Cette fonction s'applique sur une image I. A partir du facteur de zoom spécifié en paramètre, elle permet de sous échantillonner I. *ZoomOut* est utilisée afin de créer les différents étages des pyramides à partir de I_0 et de I_1.

— Fonction *ZoomIn* : interpolation bilinéaire. Cette fonction produit l'effet inverse de *ZoomOut* et permet d'extrapoler un tableau 2D à partir d'un facteur de zoom donné. Elle est appliquée sur les différentes composantes de $\mathbf{u}$ pour pouvoir remonter la pyramide.

D'autres paramètres sont à prendre en compte :

— N_{scales} : Nombre d'échelles (ou d'étages) de la pyramide. Plus le nombre d'échelles est important plus il est possible d'estimer de grands mouvements mais l'ajout d'échelles supplémentaires augmente la quantité de calculs.

— Z_{factor} : Facteur de zoom entre les échelles. Plus ce facteur est petit, plus le raffinement entre les échelles est précis. Cependant, un petit facteur de zoom requiert un plus grand nombre d'échelles pour pouvoir estimer une amplitude de mouvement donnée. Typiquement, Z_{factor} est compris entre 1,5 et 2.

— N_{warps} : Nombre de *warps* par échelle. Les *warps* permettent de stabiliser la méthode et apportent des résultats nettement meilleurs mais il s'agit d'une des étapes les plus coûteuses.

— N_{iter} : Nombre d'itérations par *warp*. Le choix du nombre d'itérations est directement lié au rapport précision/vitesse souhaité. N_{iter} est en général une limite maximale du nombre d'itérations et les itérations sont stoppées à l'aide d'un critère de convergence [118]. Ici cependant, dans le but d'avoir un temps d'exécution constant et indépendant des données, nous fixons le nombre d'itérations.

— τ : Échantillonnage temporel. Chambolle [120] prouve que le modèle de résolution converge pour $\tau \leq 1/8$ mais précise que la convergence apparaît comme optimale pour $\tau = 1/4$.

— λ Terme d'attachement. Ce paramètre à valeur réelle, a un impact significatif sur le résultat final. Il agit sur la continuité du flot optique estimé. Plus sa valeur est petite plus le flot calculé est lisse.

— θ Terme de précision. Ce paramètre sert de lien entre les termes d'attachement et de régularisation. Il doit être petit pour maintenir une correspondance entre les deux [118].

Un filtre gaussien est appliqué en premier lieu sur chacune des images d'entrée. Cela permet localement une légère réduction du bruit, ainsi qu'un adoucissement des transitions internes de chaque image. De cette façon nous améliorons localement la continuité du signal ce qui améliore la robustesse de l'estimation du flot optique.

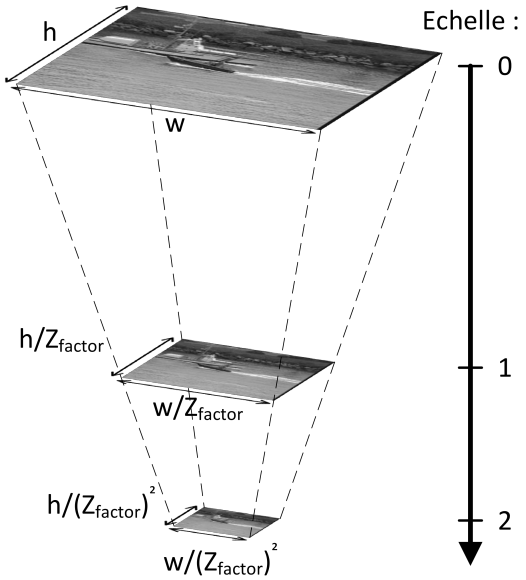


FIGURE 3.13 – Représentation de l'approche pyramidale pour 3 échelles.

Avant de considérer les pseudo codes de TV-L^1, il est important de connaître également certaines variables élémentaires au bon déroulé de l'algorithme :

— I_{S0} et I_{S1} : Pyramides des images I_0 et I_1. Chaque étage de la pyramide contient une image plus ou moins sous-échantillonnée de l'image d'origine. Les étages $I_{S0}[0]$ et $I_{S1}[0]$ contiennent les images pleine définition tandis que les étages du niveau $N_{scales} - 1$ contiennent les images les plus sous-échantillonnées. La figure 3.13 présente les différents étages de la pyramide d'une image pour 3 niveaux d'échantillonnage.

— $\mathbf{u}_S = (u_{S1}, u_{S0})$: Pyramide de flot optique. $\mathbf{u}_S$ est construite de façon analogue à I_{S0} et I_{S1} mais contient les informations du flot optique calculé.

Le pseudo code 3 décrit la structure du calcul de TV-L^1 dans son approche multi-échelles tandis que l'algorithme 4 de la fonction $TVL1\_MonoScale$, décrit le calcul du flot optique pour une échelle donnée.

Algorithm 3: TV-L^1 Structure multi-échelle

Input: $I_0, I_1, N_{scales}, Z_{factor}, N_{warps}, N_{iter}, \tau, \lambda, \theta$
Output: $\mathbf{u} = (u_1, u_2)$

// Initialisation : filtre gaussien
$(I_{G0}, I_{G1}) \leftarrow Gauss(I_0, I_1)$

// Création des échelles
$I_{S0}[0] \leftarrow I_{G0}$
$I_{S1}[0] \leftarrow I_{G1}$
for $s = 1$ **to** $N_{scales} - 1$ **do**
    $(I_{S0}[s], I_{S1}[s]) \leftarrow ZoomOut(I_{S0}[s-1], I_{S1}[s-1], Z_{factor})$

// Initialisation à 0 du flot optique pour la plus petite échelle
$u_{S1}[N_{scales} - 1] \leftarrow u_{S2}[N_{scales} - 1] \leftarrow \mathbf{0}$

// Calcul du flot
for $s = N_{scales} - 1$ **to** 0 **do**
    $\mathbf{u}_S[s] \leftarrow TVL1\_MonoScale(I_{S0}[s], I_{S1}[s], \mathbf{u}_S[s], N_{warps}, N_{iter}, \tau, \lambda, \theta)$
    if *(s = 0)* **then**
        break
    $\mathbf{u}_S[s-1] \leftarrow ZoomIn(\mathbf{u}_S[s], Z_{factor})$

// Sauvegarde du résultat final
$\mathbf{u} \leftarrow \mathbf{u}_S[0]$
return u

Algorithm 4: TV-L1 Algorithme Mono-échelle : Fonction TVL1_MonoScale

Input: $I_0, I_1, N_{warps}, N_{iter}, \tau, \lambda, \theta$
Output: $\mathbf{u} = (u_1, u_2)$

// Initialisation de $\mathbf{P} = (\mathbf{p}_1, \mathbf{p}_1)$
$\mathbf{p}_1 \leftarrow (0, 0)$
$\mathbf{p}_2 \leftarrow (0, 0)$

// Calcul du gradient centré de I_1
$(I_{1x}, I_{1y}) \leftarrow \nabla I_1$
for $w = 0$ **to** $N_{warps} - 1$ **do**
    // Calcul des *warps*
    $I_{1w} \leftarrow Warp(I_1, u_1, u_2)$
    $I_{1xw} \leftarrow Warp(I_1 x, u_1, u_2)$
    $I_{1yw} \leftarrow Warp(I_1 y, u_1, u_2)$
    // Calcul des itérations sur $\mathbf{v}$, $\mathbf{u}$ puis $\mathbf{P}$
    for $n = 0$ **to** $N_{iter} - 1$ **do**
        $\mathbf{v} \leftarrow TH(\mathbf{u}, \mathbf{u}^0)$
        $\mathbf{u} \leftarrow \mathbf{v} + \theta \operatorname{div}(\mathbf{P})$ // divergences arrières
        $\mathbf{P} \leftarrow \frac{\mathbf{P} + (\tau/\theta)\nabla \mathbf{u}}{1 + (\tau/\theta)|\nabla \mathbf{u}|}$ // gradients avants

Dans cette section, nous avons vu comment résoudre le problème de flot optique par la méthode TV-L^1. Cette méthode est itérative et, du nombre d'itérations, dépendent fortement la précision et le temps de calcul. Le nombre de *warps* et d'échelles jouent également un rôle important sur le temps d'exécution. Dans la section suivante, nous cherchons à déterminer les valeurs des différents paramètres, afin d'obtenir le meilleur compromis vitesse précision pour notre application.

3.3.3 Paramétrage de TV-L^1

Dans la section précédente, nous avons pu voir que la résolution du flot optique par TV-L^1 nécessite de nombreux paramètres. Ces paramètres ont un impact plus ou moins important sur la qualité du résultat et sur le temps d'exécution. En outre, la complexité de TV-L^1 est telle, qu'il est difficile d'obtenir une implémentation en temps réel. Cela n'est pas vrai uniquement pour de petits systèmes embarqués mais également pour des systèmes plus puissants. Certains paramètres comme le nombre d'itérations ou de *warps* sont en effet souvent élevés : plusieurs centaines d'itérations et de nombreux *warps*. Dans cette section, nous chercherons à déterminer les valeurs optimales des différents paramètres afin d'obtenir le meilleur rapport possible entre qualité et temps d'exécution.

Certains paramètres comme τ, λ ou θ affectent la qualité du résultat sans avoir d'impact direct sur le temps d'exécution. En revanche, d'autres paramètres comme N_{iter}, N_{warps} ou N_{scales} influent de façon significative sur l'ensemble des performances de l'algorithme. Un premier travail préliminaire de cette thèse a été de fixer l'ensemble des paramètres pour avoir une configuration de référence sur laquelle travailler. Cette configuration de référence permet d'effectuer un premier compromis entre précision et vitesse pour pouvoir envisager une implémentation temps réelle par la suite.

Si l'on considère les paramètres utilisés dans [118], il n'est pas possible d'envisager une exécution en temps réel du calcul de flot optique. Dans [118], le nombre d'échelles est de 5 avec 5 *warps* par échelle et jusque 300 itérations par *warp* (le nombre d'itérations étant variable car fixé par un critère d'arrêt sur la convergence). En plus d'engendrer un temps de calcul dépendant des données, ce paramétrage nécessite un traitement de plusieurs secondes pour des images de 640×580 pixels sur un processeur Intel Xeon E5-1607 v3 4C@3.1GHz. Une vidéo étant considérée comme fluide à partir de 25 images par seconde, le temps maximal pour un traitement en direct, ne doit donc pas excéder 40 millisecondes. Ces performances préliminaires sont trop loin de ce seuil pour pouvoir espérer une implémentation temps réel embarquée en ne s'appuyant que sur des optimisations architecturales et des transformations algorithmiques. Il est donc indispensable de chercher un compromis sur le réglage de ces paramètres.

Pour déterminer les différents paramètres de TV-L^{1}1, on choisit deux séquences vidéos pour lesquelles on étudie la convergence de l'algorithme en fonction de la valeur du paramètre évalué. Un extrait de chaque séquence est présenté en figure 3.14) :

— La première vidéo est la séquence *Coastguard* issue de la base de donnée Derf's Test Media Collection [121]. Cette base de données est souvent utilisée dans la littérature pour évaluer toutes sortes de traitements de vidéo. La séquence *Coastguard* suit un bateau jusqu'à ce qu'il en croise un second en sens inverse. Au moment du croisement il y a un sursaut de caméra avant que cette dernière ne suive le second bateau. Cette scène est particulièrement intéressante pour l'étude du flot optique : elle comporte une première phase relativement simple avec un seul objet mobile et un traveling de camera. Lorsque que les 2 embarcations se rencontrent cela crée une zone de discontinuité dans le flot ainsi qu'une occlusion. Le sursaut de la caméra permet également d'évaluer les performances de l'algorithme sur des mouvements de plus grande ampleur. Enfin, l'arrière plan est riche en textures avec de l'eau, de l'herbe, des arbres ainsi que des rochers. Cela permet d'évaluer le comportement de l'algorithme sur des textures plus ou moins complexes. Cette séquence est de dimensions 352 × 288 pixels.

— La seconde vidéo, *Cateye*, est une séquence LHERITIER prise à l'aide de la caméra *CatEye* [85]. Dans cette scène, la marche de piéton est la principale difficulté pour l'estimation du flot optique. En effet, on peut y voir un individu de profil marchant de la droite vers la gauche du champ de la caméra. Cette dernière est fixée sur un trépied. Cette vidéo est au format Full HD : 1920 × 1080 pixels.

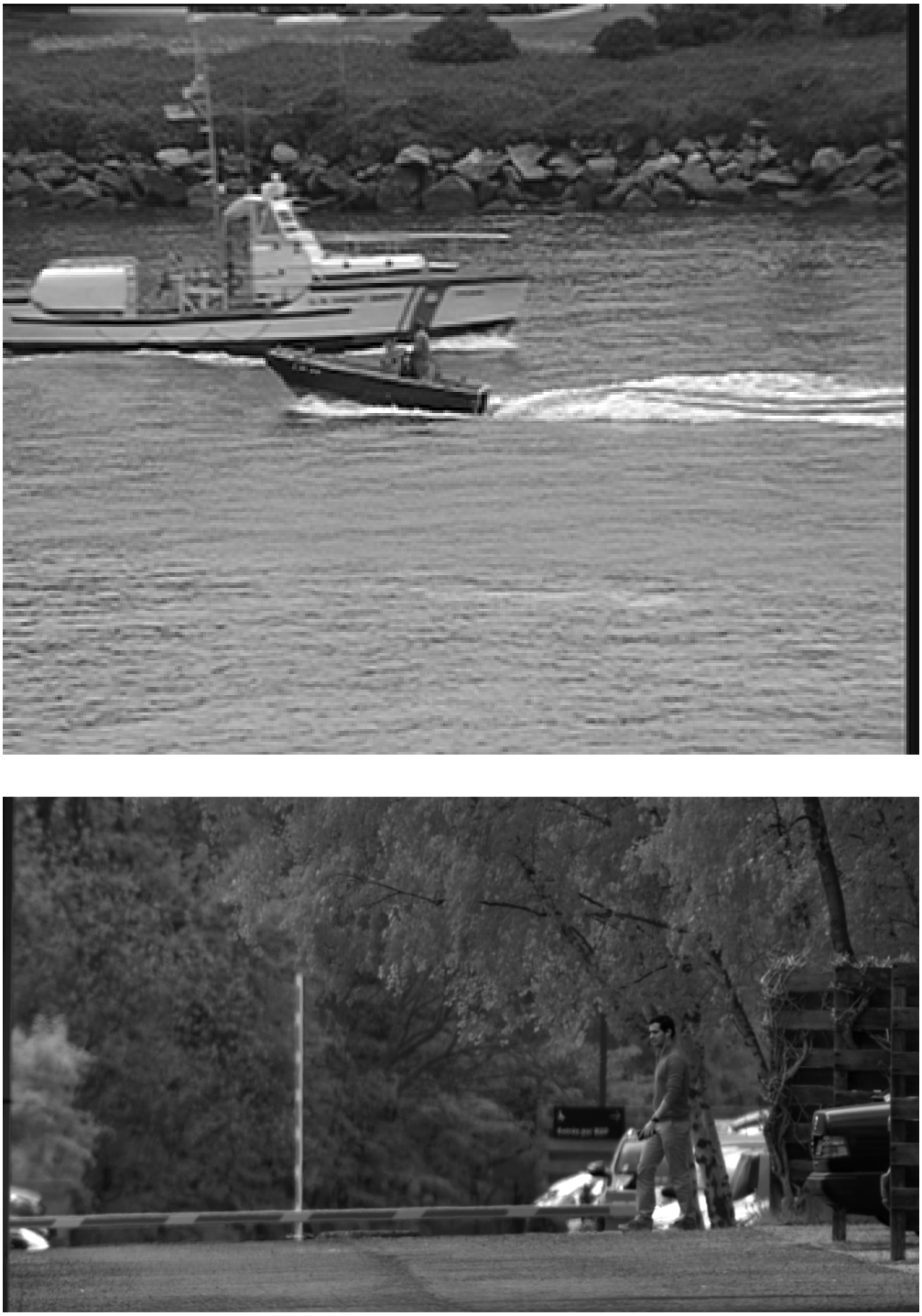


FIGURE 3.14 – Extraits des séquences Coastguard (haut) et Cateye (bas).

Afin de déterminer la qualité du flot optique calculé, nous mesurons l'erreur quadratique moyenne de reconstruction (MSE : *Mean Square Error*). On estime le flot optique $\mathbf{u}$ entre les deux images consécutives I_0 et I_1. On en déduit $\tilde{I_1}$ l'image interpolée depuis I_1 en fonction de $\mathbf{u}$ pour correspondre à I_0. Si N est le nombre de pixels dans une image, la MSE indiquée se calcule par :

$$MSE(\tilde{I_1}) = \frac{1}{N} \sum_{p<N} (\tilde{I_1}(p) - I_0(p))^2$$

Plus la MSE est faible plus le flot optique mesuré est considéré précis.

La MSE est indicative et ne reflète pas complètement la qualité de la reconstruction. Il est possible par exemple, d'obtenir une image globalement bien reconstruite, avec une MSE faible, mais contenant des artefacts ponctuels importants. À l'inverse, il est possible d'avoir une image mieux reconstruite, sans artefacts, mais légèrement décalée en raison d'un biais constant dans l'estimation du flot optique. Cela peut entraîner une MSE plus importante avec pourtant une reconstruction visuellement de meilleure qualité. Une faible différence de MSE n'est donc pas nécessairement significative. Ici, l'objectif est d'apporter une réflexion pour obtenir une configuration initiale d'un jeu de paramètres pertinents, dans le cadre d'une implémentation temps réel embarquée. Cette configuration sert de base de travail pour ensuite optimiser notre implémentation. Les paramètres définis ne sont donc pas immuables et pourrons éventuellement être modifiés lors de l'introduction de TV-L^1 dans la chaîne complète de traitement. C'est pourquoi, nous considérons que la mesure seule de la MSE sur les deux séquences vidéo considérées est suffisante pour nos besoins.

Détermination du nombre d'échelles et du facteur de zoom pour l'approche pyramidale

La figure 3.15, montre l'évolution de la MSE en fonction du nombre d'échelles pour une image donnée de la séquence *Coastguard*. Cette courbe est représentative de l'évolution de la MSE pour la majeure partie de la séquence. Ici, le maximum de convergence est presque atteint au bout de 2 échelles. En revanche, en présence de mouvements plus amples, comme présenté en figure 3.16, il est nécessaire d'effectuer les calculs sur un plus grand nombre d'échelles. Ces résultats s'expliquent par la nature même de TV-L^1. En effet, pour une échelle donnée, les mouvements estimés par une méthode différentielle comme TV-L^1, ne peuvent excéder un pixel d'amplitude. C'est l'approche pyramidale qui permet justement d'estimer de plus grands mouvements. En sous-échantillonnant les images, les mouvements apparents deviennent moins grands. Il est donc possible d'estimer les mouvements de moins de 1 pixel dans la nouvelle échelle. Les résultats obtenus sont alors utilisés et raffinés dans l'échelle suivante de plus grande définition.

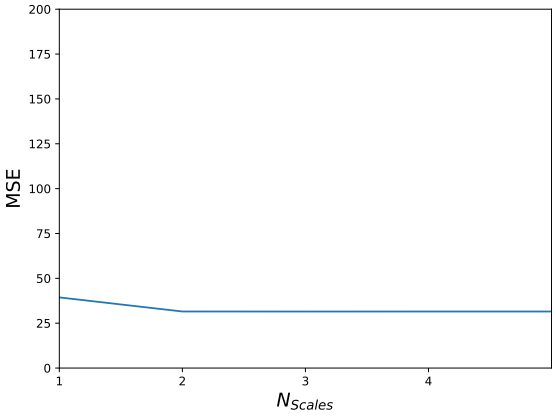


FIGURE 3.15 – Évolution de la MSE en fonction du nombre d'échelles de TV-L^1. Résultats pour une image donnée de la sequence *Coastguard*. Mouvements de faible amplitude

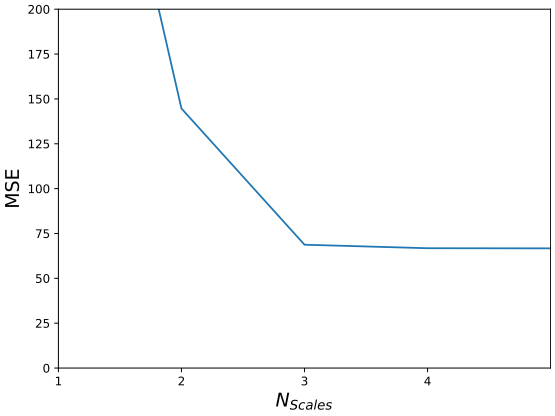


FIGURE 3.16 – Évolution de la MSE en fonction du nombre d'échelles de TV-L^1. Résultats pour une image donnée de la sequence *Coastguard*. Mouvements de forte amplitude

Dans notre cas, nous choisissons un facteur de zoom (Z_{factor}) de 2. Ainsi il est possible de couvrir l'ensemble des valeurs possibles en 0 et l'amplitude maximale estimable. En outre, le choix d'un facteur de zoom de 2 nous permet de simplifier significativement les interpolations de mise à l'échelle. Avec $Z_{factor} = 2$, il nous est possible d'estimer l'amplitude maximale estimable en fonction du nombre d'échelles : soit N_{Scales} le nombre d'échelles utilisées, l'amplitude maximale estimable des mouvements est $2^{N_{Scales}} - 1$.

L'impact du nombre d'échelles sur la MSE est similaire pour la séquence *Cateye* : les grands mouvements nécessitent plus d'échelles pour s'approcher du maximum de convergence. Il est cependant important de prendre en compte la définition des images d'entrée avant de définir le nombre de sous-échantillonnages. En effet, pour une scène identique, les mouvements vont être plus importants en termes de pixels pour une image de plus grande définition. Comme expliqué précédemment, c'est ce phénomène qui est utilisé dans l'approche pyramidale. C'est pourquoi, plus la vidéo d'entrée est d'une définition élevée, plus il peut être nécessaire d'augmenter le nombre d'échelles.

Nous pouvons considérer que, pour une paire d'images donnée, la MSE converge pour un grand nombre d'échelles. On note MAX_{Scales} cette valeur. Pour la scène *Coastguard* en faible définition nous prenons $MAX_{Scales} = 5$ et pour la scène *Cateye* en haute définition nous prenons $MAX_{Scales} = 10$. On définit pour chaque image reconstruite la différence proportionnelle de MSE $\%\Delta MSE(N_{Scales})$ tel que :

$$\%\Delta MSE(N_{Scales}) = \frac{MSE(N_{Scales}) - MSE(MAX_{Scales})}{MSE(MAX_{Scales})} \tag{3.34}$$

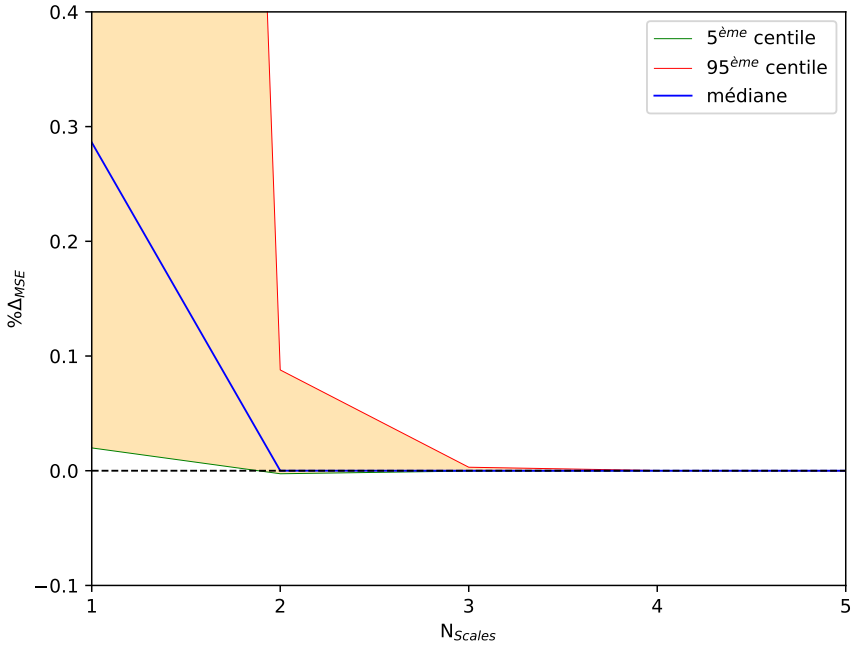


FIGURE 3.17 – Différence proportionnelle de MSE en fonction du nombres d'échelles pour la séquence *Coastguard*.

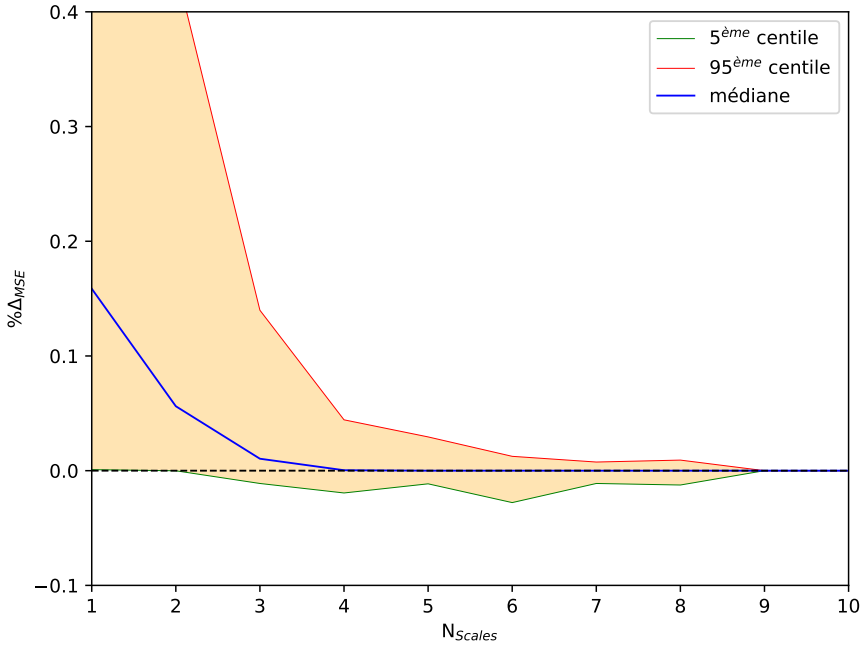


FIGURE 3.18 – Différence proportionnelle de MSE en fonction du nombres d'échelles pour la séquence *Cateye*.

Les figures 3.17 et 3.18 représentent l'évolution de $\%\Delta MSE(N_{Scales})$ pour l'ensemble des séquences *Coastguard* et *Cateye*. Plus la valeur de $\%\Delta MSE$ est faible plus cela signifie que la MSE équivalente est basse aussi. Si $\%\Delta MSE(N_{Scales}) < 0$ alors $MSE(N_{Scales}) < MSE(MAX_{Scales})$. Cela signifie que la valeur de la MSE vers laquelle converge la méthode n'est pas la meilleure possible. Comme expliqué précédemment cela ne signifie cependant pas que la qualité de reconstruction devient réellement moins bonne surtout si les écarts entre les différentes MSE sont faibles.

La figure 3.17 montre que sur la séquence *Coastguard*, pour 3 échelles ou plus, au moins 95% des images sont reconstruites au mieux selon le critère de la MSE. Sur la figure 3.18, on peut voir que, pour la scène *Cateye*, la valeur de la MSE vers laquelle converge la méthode n'est pas toujours la plus basse. Pour cette scène, Il faut 4 échelles ou plus pour que $\%\Delta MSE$ soit inférieur à 5% pour au moins 95% des images. Il apparaît donc qu'il est nécessaire d'utiliser plus d'échelles que pour *Coastguard* pour atteindre la valeur de convergence. Cela s'explique par le fait que la vidéo *Cateye* est de plus grande définition : les mouvements apparents sont donc de plus grande ampleur et il faut plus d'étages dans la pyramide d'images. Comme nous travaillons dans un premier temps avec des vidéos de définitions inférieures à celles de *Cateye*, nous choisissons de fixer le nombre d'échelles à 3.

Détermination du nombre de *warps*

Nous cherchons désormais à étudier l'influence du nombre de *warps* effectués par échelle. L'utilisation de *warps* est particulièrement pénalisant pour le temps d'exécution. C'est pourquoi le réglage de ce paramètre est crucial. En théorie, l'utilisation de plus d'un *warp* par échelle n'est pas nécessaire mais dans les faits, cela permet d'améliorer la stabilité de convergence de la méthode, au prix d'une interpolation en fonction du champ de vecteurs jusque ici estimé [118]. Cette interpolation est particulièrement lente en raison de son attachement aux vecteurs vitesse. Les accès en mémoire sont dépendants de la norme et de la direction de ces vecteurs. Ils peuvent donc être considérés comme aléatoires. Il est alors difficile de proposer des solutions permettant de régulariser et d'optimiser les accès mémoire. Pour ces raisons, la réduction du nombre de *warps* apparaît comme particulièrement pertinente pour notre application, à condition que la baisse de qualité ressentie ne soit pas trop importante.

De façon analogue à précédemment on définit la différence proportionnelle de MSE en fonction du nombre de *warps* $\%\Delta MSE(N_{Warps})$ tel que :

$$\%\Delta MSE(N_{Warps}) = \frac{MSE(N_{Warps}) - MSE(MAX_{Warps})}{MSE(MAX_{Warps})} \tag{3.35}$$

Avec $MAX_{Warps} = 5$.

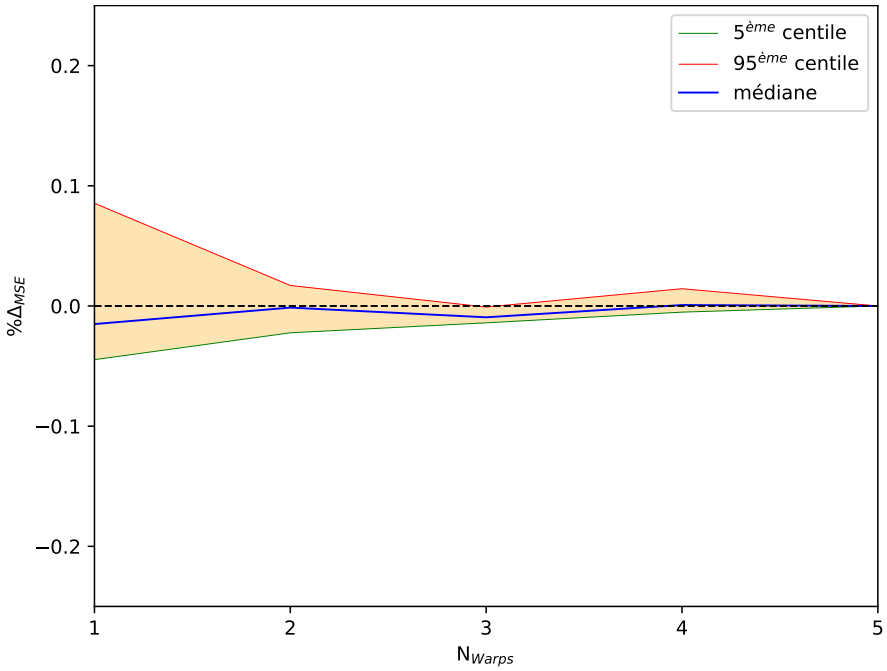


FIGURE 3.19 – Différence proportionnelle de MSE en fonction du nombres de *warps* pour la séquence *Coastguard*.

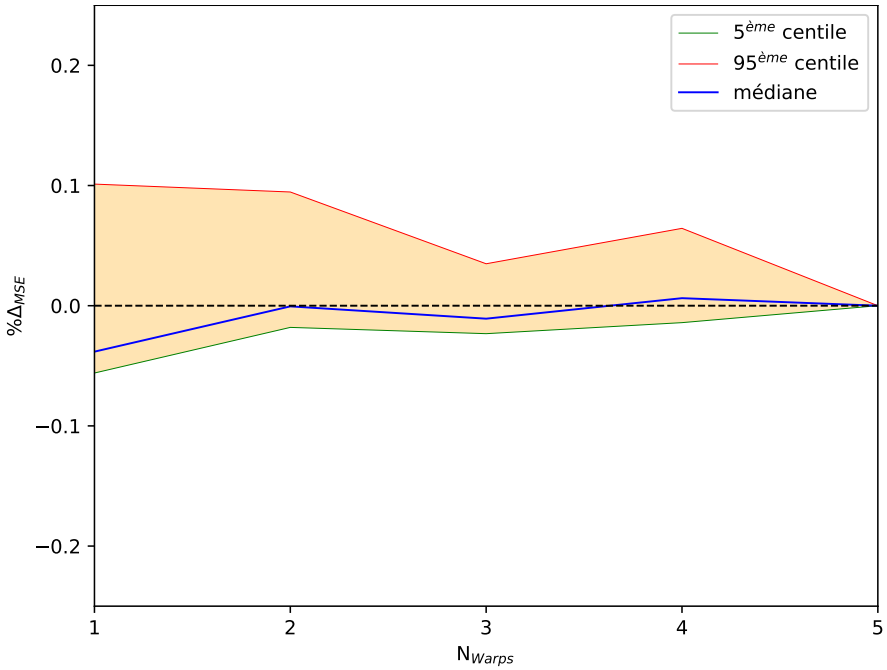


FIGURE 3.20 – Différence proportionnelle de MSE en fonction du nombres de *warps* pour la séquence *Cateye*.

Les figures 3.19 et 3.20 représentent l'évolution de cette différence proportionnelle en fonction du nombre de *warps* pour 100 itérations par *warp* et par échelle, sur les séquences *Coastguard* et *Cateye*. Pour Cateye, l'augmentation du nombre de *warps* ne permet pas d'améliorer de façon significative la MSE. Pour Coastguard, deux *warps* suffisent pour que le minimum de convergence soit presque atteint pour au moins 95% des images de la séquence. Pour plus de la moitié des images, ce minimum est atteint (voire dépassé) dès le premier *warp*. Dans tous les cas et pour les deux séquences, $\%\Delta MSE(N_{Warps})$ est inférieur à 10% peu importe le nombre de *warps* utilisés. On peut voir que pour la moitié des images de chaque séquence, l'erreur différentielle de MSE est proche de zéro voir négative. Cela confirme que, en termes de MSE, la reconstruction n'est pas forcément meilleure en utilisant plus de *warps*. En revanche, on constate bien une stabilisation des résultats avec l'augmentation du nombre de *warps*. Cette stabilisation est particulièrement marquée pour la scène *Coastguard* (figure 3.19).

Étant donné son faible impact sur la qualité de reconstruction mais son impact important sur le temps de calcul, nous choisissons de fixer le nombre de *warps* à 1 par échelle.

Détermination du nombre d'itérations

Toujours dans l'optique d'obtenir une configuration qui servira de base de départ pour notre application, nous cherchons pour finir à déterminer le nombre d'itérations pour chaque échelles de TV-L[1]. Ce dernier paramètre est le plus susceptible de varier de façon significative étant donné qu'il est au cœur même de l'algorithme d'estimation du flot optique. C'est pourquoi, nous choisirons un intervalle de valeurs plutôt qu'un nombre unique d'itérations.

Comme pour N_{Scales} et N_{Warps}, on définit la différence proportionnelle de MSE en fonction du nombre d'itérations $\%\Delta MSE(N_{iter})$ tel que :

$$\%\Delta MSE(N_{iter}) = \frac{MSE(N_{iter}) - MSE(MAX_{iter})}{MSE(MAX_{iter})} \tag{3.36}$$

Avec $MAX_{iter} = 100$.

Les figures 3.21 et 3.22 représentent la différence proportionnelle de MSE en fonction du nombre d'itérations pour les séquences *Coastguard* et *Cateye*. Dans ce cas, on observe que, pour un faible nombre d'itérations (typiquement de 3 à 5), la reconstruction est souvent meilleure en termes de MSE que pour un nombre infini d'itérations (100). Un comportement différent est également notable entre les deux séquences.

Pour la séquence *Coastguard*, la convergence de la MSE est très marquée, et, à partir de 3 itérations, la différence de MSE est inférieure à 10% de la MSE à l'infini pour au moins 95% des images de la séquence. La convergence est stable et l'écart type de $\%\Delta MSE(N_{iter})$ tend vers 0.

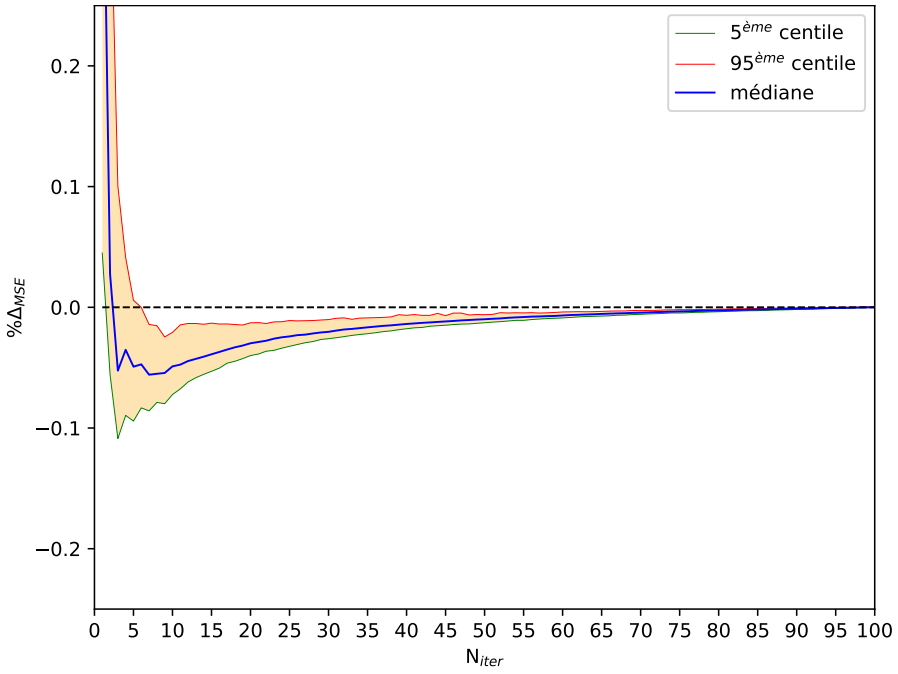


FIGURE 3.21 – Différence proportionnelle de MSE en fonction du nombres d'itérations pour la séquence *Coastguard*.

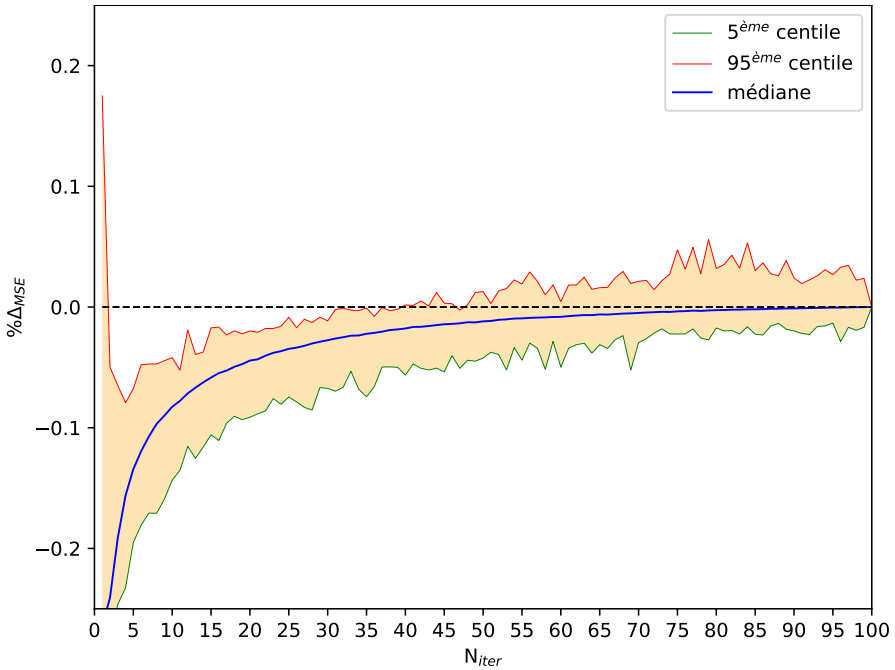


FIGURE 3.22 – Différence proportionnelle de MSE en fonction du nombres d'itérations pour la séquence *Cateye*.

Pour la séquence *Cateye*, il semble que l'augmentation du nombre d'itérations pénalise la qualité de la reconstruction pour une grande majorité des images. En outre, l'écart type entre les MSE ne semble pas diminuer significativement avec le nombre croissant d'itérations. Pour comprendre ce phénomène, il faut prendre en compte la nature de la scène. Pour rappel, la séquence met en scène un piéton seul, marchant dans un environnement immobile. La vidéo est capturée sur un trépied et en haute définition (1920×1080 pixels). Hormis le piéton, très peu d'éléments sont donc mobiles. L'individu n'occupant pas une grande place dans le cadre de la vidéo, la majorité des pixels représente des éléments immobiles. C'est pourquoi, dans ce cas, une reconstruction avec peu d'itérations est meilleure en termes de MSE qu'avec un plus grand nombre. L'erreur quadratique, bien que localement grande au niveau du piéton, reste globalement meilleure avec peu d'itérations. La mise en évidence de ce comportement peut nous amener à considérer une technique permettant de ne pas prendre en compte les mouvements estimés de très faible amplitudes au moment de la reconstruction. Cette considération faite, elle ne sera pas appliquée dans un premier temps étant donné la difficulté que représente l'implémentation en temps réel de TV-L^1. L'ajout d'une étape supplémentaire ne serait pas pertinent à ce stade.

Une interrogation demeure cependant. Est-il réellement plus intéressant de n'effectuer que quelques itérations ? Il est vrai que pour les deux séquences, la MSE est en général meilleure pour un faible nombre d'itérations que pour un grand nombre d'itérations. Les résultats visuels semblent pourtant meilleurs pour un nombre d'itérations plus important. Les évaluations visuelles montrent qu'en réalité, nous retrouvons la situation décrite en début de section : pour moins d'itérations les images sont globalement bien reconstruites mais présentent d'importants artefacts. Avec un nombre d'itérations plus élevé, l'image est reconstruite avec moins d'artefacts bien que la MSE soit globalement légèrement supérieure. On voit bien ici les limites de cette métrique.

A partir de ces résultats, nous choisissons une configuration de départ entre 3 et 10 itérations. 3 itérations permettent pour la majorité des images de minimiser la MSE tandis qu'une dizaine d'itérations permettent de réduire les artefacts.

Synthèse, paramétrage initial de TV-L^1

In fine, nous avons déterminé une configuration de base de TV-L^1 permettant un premier compromis entre précision et temps de calcul. Dans cette configuration, nous exécutons une version pyramidale de TV-L^1. La pyramide est composée de 3 étages (ou échelles). Le facteur de zoom entre les échelles est fixé à 2. Un unique *warp* est utilisé pour chaque échelle. Entre 3 et 10 itérations sont exécutées à chaque échelle.

3.3.4 Optimisation d'implémentation de TV-L^1

Nous avons détaillé dans un premier temps les différents choix effectués dans le cadre d'un compromis "précision/vitesse". Nous nous concentrons désormais sur les différentes

optimisations et transformations appliquées pour accélérer l'exécution d'une configuration donnée de TV-L^1. L'impact de ces transformations est présenté dans la section suivante.

TV-L^1 étant un algorithme itératif, les données sont fortement dépendantes entre elles. La solution d'implémentation la plus simple est d'effectuer les itérations les unes après les autres. Les résultats intermédiaires de chaque itération sont entièrement calculés avant d'exécuter l'itération suivante. Dans ce cas, la réutilisation des données n'est pas optimale : entre la production d'une donnée à l'itération k et sa réutilisation à l'itération $k+1$ il est fort probable que cette dernière ait été éjectée hors du cache. Il est toutefois possible de s'inspirer des optimisations présentées en chapitre 2, pour fusionner ou pipeliner non pas uniquement les opérateurs mais également les itérations.

Afin de faciliter la compréhension des différentes optimisations, nous rappelons les principaux aspects de TV-L^1. En section 3.3.2, nous avons vu que le calcul d'une itération de TV-L^1 se déroule en trois étapes principales :

Dans un premier temps, le terme de relaxation $\mathbf{v}$ est calculé :

$$\mathbf{v}^{k+1} := \mathbf{u}^k + TH(\mathbf{u}^k, \mathbf{u}^0) \tag{3.37}$$

Suit, dans un deuxième temps, le calcul du flot optique $\mathbf{u}$:

$$u_d^{k+1} := v_d^{k+1} + \theta \operatorname{div}(\mathbf{p}_d^k), d \in \{1,2\} \tag{3.38}$$

Enfin, le double champ de vecteur $\mathbf{P}$ est mis à jour. A noter que cette dernière étape n'est pas nécessaire lors de la dernière itération puisque c'est le résultat du flot optique $\mathbf{u}$ qui nous intéresse en définitive :

$$\mathbf{p}_d^{k+1} := \frac{\mathbf{p}_d^k + \frac{\tau}{\theta} \nabla \left(v_d^{k+1} + \theta \operatorname{div}(\mathbf{p}_d^k) \right)}{1 + \frac{\tau}{\theta} \left| \nabla u_d^{k+1} \right|}, d \in \{1,2\} \tag{3.39}$$

Également en section 3.3.2, nous avons précisé notre choix d'utiliser une divergence arrière pour les calculs de divergences de $\mathbf{P}$, ainsi que des gradients avants pour les calculs de gradients de $\mathbf{u}$. Pour une itération donnée de TV-L^1, nous pouvons donc considérer les 5 opérateurs principaux suivants :

— $\mathbf{V}$: permet de calculer le terme de relaxation $\mathbf{v} = \{v_1, v_2\}$.
— $\mathbf{Div}$: calcule les divergences arrières de $\mathbf{P} = \{\mathbf{p_1}, \mathbf{p_2}\} = \{\{p_{1x}, p_{1y}\}, \{p_{2x}, p_{2y}\}\}$.
— $\mathbf{U}$: permet de calculer le flot optique $\mathbf{u} = \{u_1, u_2\}$.
— $\mathbf{G_f}$: permet de calculer les gradients avants de $\mathbf{u}$: u_{1_x}, u_{1_y}, u_{2_x} et u_{2_y}.
— $\mathbf{P}$: qui permet de calculer le double champ de vecteurs $\mathbf{P}$.

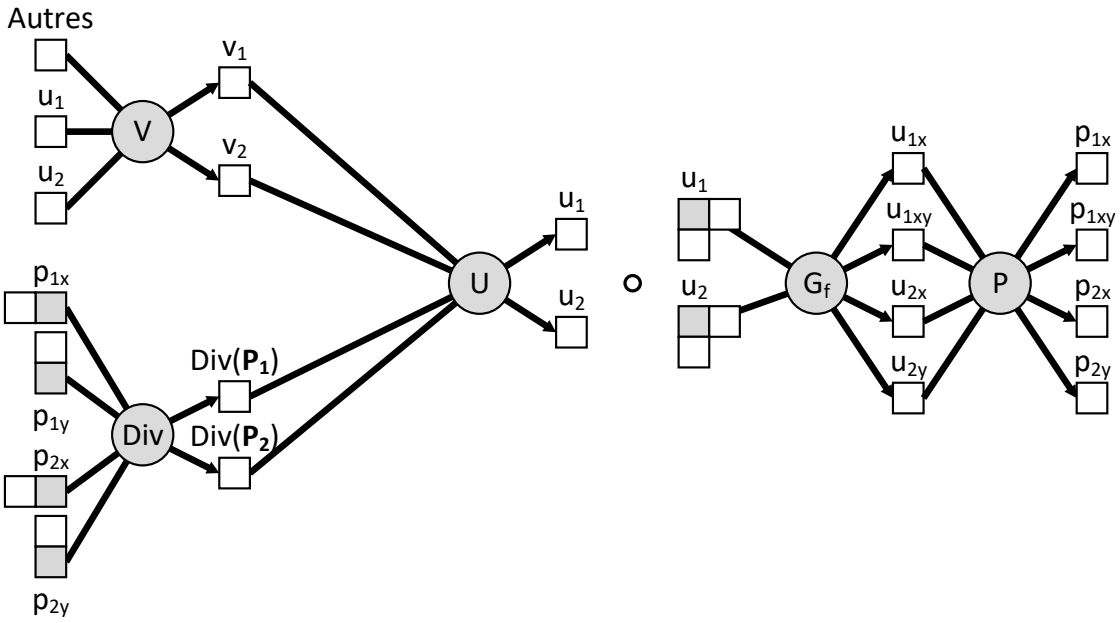


FIGURE 3.23 – Déroulé d'une itération de TV-L^1 sans optimisation.

Ces opérateurs définis, le déroulé d'une itération est schématisé en figure 3.23. On remarque qu'il est aisé de fusionner l'opérateur **U** avec les opérateurs **V** et **Div** ainsi que l'opérateur **P** avec $\mathbf{G_f}$. Cela permet d'éviter l'écriture en mémoire des résultats intermédiaires de **V**, **Div** et $\mathbf{G_f}$. En revanche, les dépendances de données entre **U** et **P** rendent la fusion entre ces deux derniers opérateurs plus complexe. La figure 3.24 illustre le déroulé d'une itération de TV-L^1 avec fusion. Dans un soucis de lisibilité certains tableaux ont été regroupés dans le modèle "producteur-consommateur".

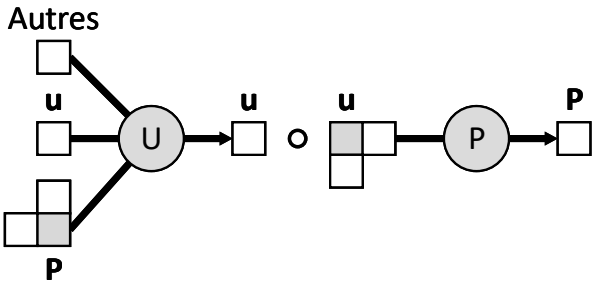


FIGURE 3.24 – Déroulé d'une itération de TV-L^1 avec fusions : {**V**, **Div**, **U**} →**U** et {$\mathbf{G_f}$, **P**} →**P**.

Les dépendances de données entre **U** et **P** ainsi que le caractère itératif de TV-L^1, rendent particulièrement difficile la fusion de ces deux opérateurs. En outre, il est nécessaire de conserver dans leur entièreté les résultats **u** et **P** d'une itération à l'autre. Cela réduit significativement l'impact bénéfique potentiel de la fusion puisque les accès mémoire intermédiaires entre ces deux opérateurs ne sont pas supprimés.

Si la fusion de **U** et de **P** reste potentiellement intéressante pour une itération de TV-L^1, l'augmentation du nombre d'itérations rend cette technique complètement caduque. En effet, plus le nombre d'itérations est élevé, plus les dépendances de données se propagent entre **u** et **P**.

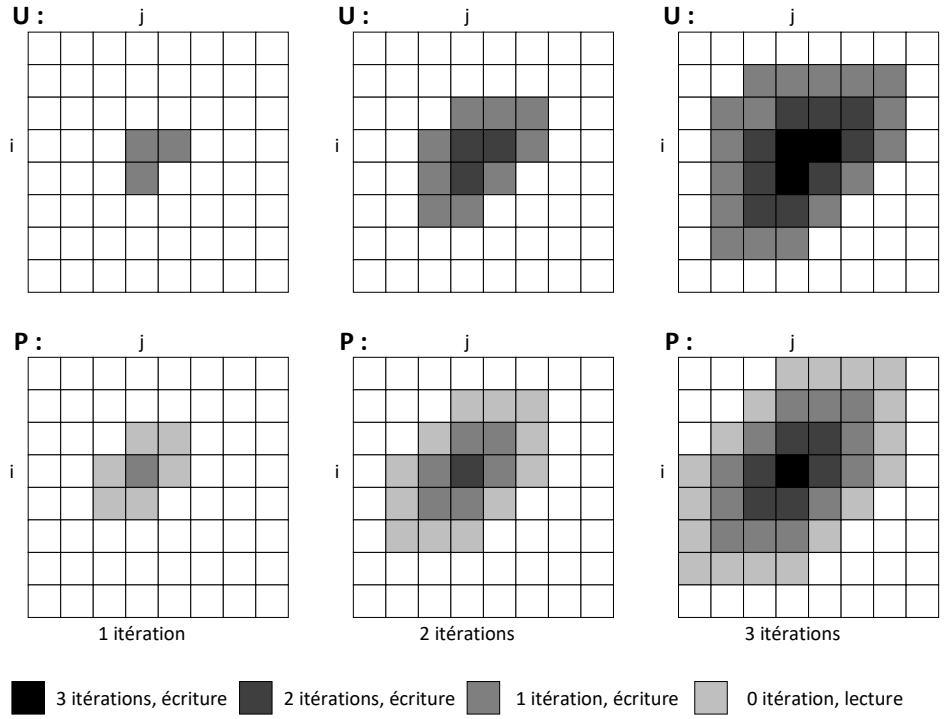


FIGURE 3.25 – Propagation des dépendances de données entre **u** et **P** en fonction du nombre d'itérations complètes de TV-L^1. Dépendances pour un pixel quelconque de cordonnées (i, j).

La figure 3.25 illustre cette propagation de dépendances pour 1, 2 et 3 itérations complètes de TV-L^1. On s'aperçoit ici qu'il est difficile d'envisager une fusion des itérations étant donné l'augmentation du nombre d'accès mémoire nécessaire pour la production d'un point final. La fusion des itérations entraîne également un fort accroissement de la redondance des calculs et des accès.

Si la fusion des itérations n'est pas une solution qui paraît pertinente, il en va différemment du pipeline d'itérations. Dans ce cas, le but est d'améliorer la localité mémoire. Pour cela, une ligne de **u** ou de **P** est réitérée dès que possible. Ici, nous choisissons l'élément de synchronisation du pipeline comme étant de la taille d'une ligne. La figure 3.26 illustre le déroulé d'un pipeline d'itérations pour pouvoir itérer 3 fois sur 3 lignes de **u**. Une phase de prologue est nécessaire avant d'entrer dans un régime permanent qu'on appelle corps de boucle.

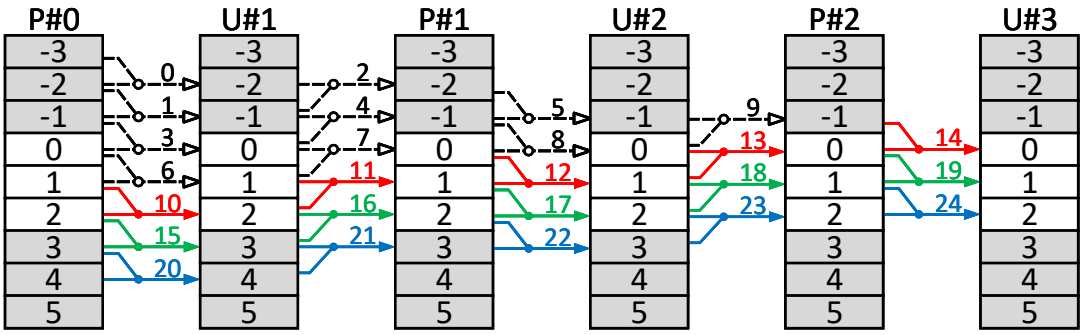


FIGURE 3.26 – Ordre des calculs sur **u** et **P** pour 3 itérations sur **u**. Les étapes en pointillés correspondent aux étapes de prologue, celles en traits pleins au corps de boucle. Chaque couleur représente un passage dans le corps de boucle. $A\#x$ représente le tableau A à l'itération numéro x. Chaque case de tableau représente une ligne dont l'indice est spécifié dans la case correspondante.

Dans le cas du pipeline la localité des accès mémoire est améliorée et il n'y a pas de redondance significative dans les calculs ou une augmentation du nombre d'accès mémoire. En revanche, on peut voir dans la figure 3.26, que des tableaux intermédiaires ont dû être créés pour chaque itération. Cela entraîne une augmentation de l'empreinte mémoire qui peut être péjorative pour le maintien des données en cache. En première approche on peut cependant remarquer que ces tableaux intermédiaires peuvent être alloués de façon modulaire. Cela a pour effet de minimiser leur impact sur l'empreinte mémoire.

Les figures 3.27 à 3.31 illustrent le déroulement du pipeline avec une allocation modulaire des tableaux intermédiaires. Tous les tableaux intermédiaires peuvent être alloués avec un modulo de 2 lignes, peu importe le nombre d'itérations considéré. C'est le nombre de tableaux qui varie en fonction du nombre d'itérations. Entre les figures 3.27 et 3.31, le corps de boucle a été exécuté 2 fois et les lignes des tableaux modulaires ont toutes été réécrites au moins 1 fois. Dans ces conditions, aucun conflit d'écriture ou de lecture n'a eu lieu. La première étape de régime permanent est donc correcte (étape 4, figure 3.30).

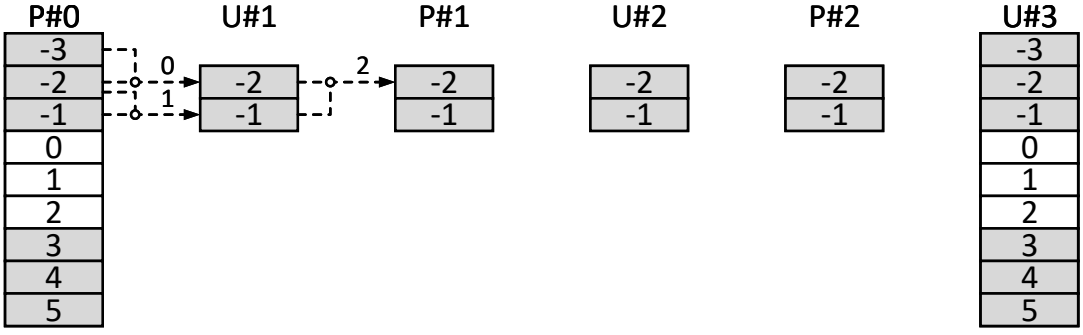


FIGURE 3.27 – Pipeline d'itérations avec allocation modulaire des tableaux intermédiaires, étape 1.

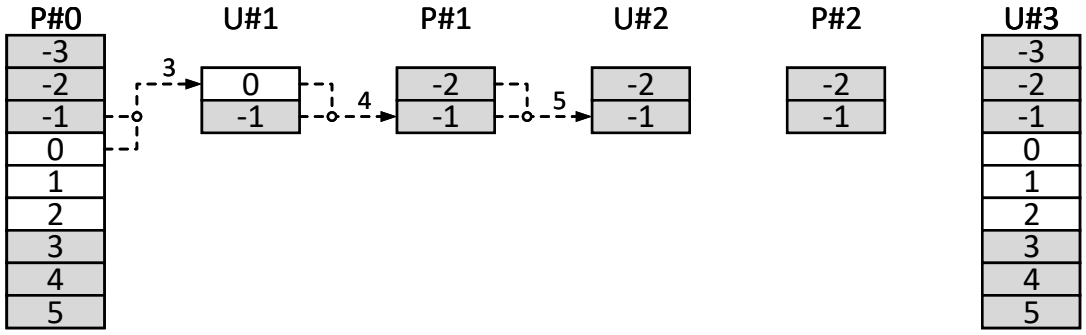


FIGURE 3.28 – Pipeline d'itérations avec allocation modulaire des tableaux intermédiaires, étape 2.

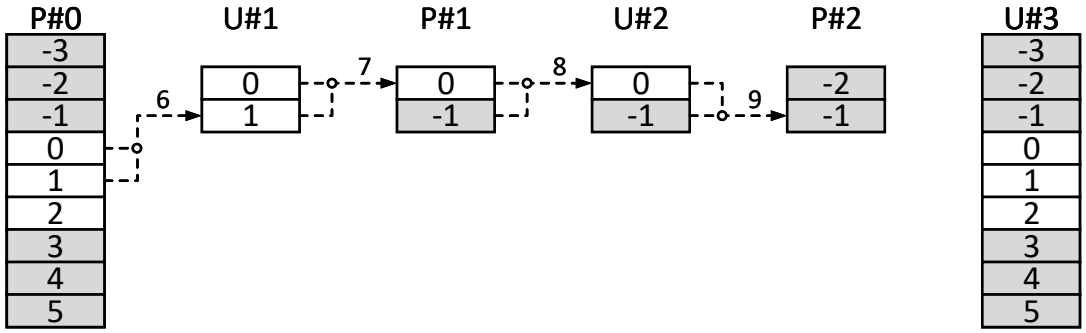


FIGURE 3.29 – Pipeline d'itérations avec allocation modulaire des tableaux intermédiaires, étape 3.

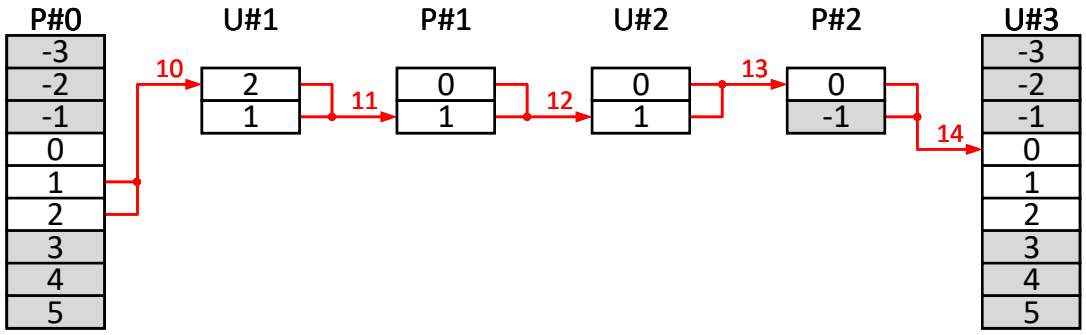


FIGURE 3.30 – Pipeline d'itérations avec allocation modulaire des tableaux intermédiaires, étape 4.

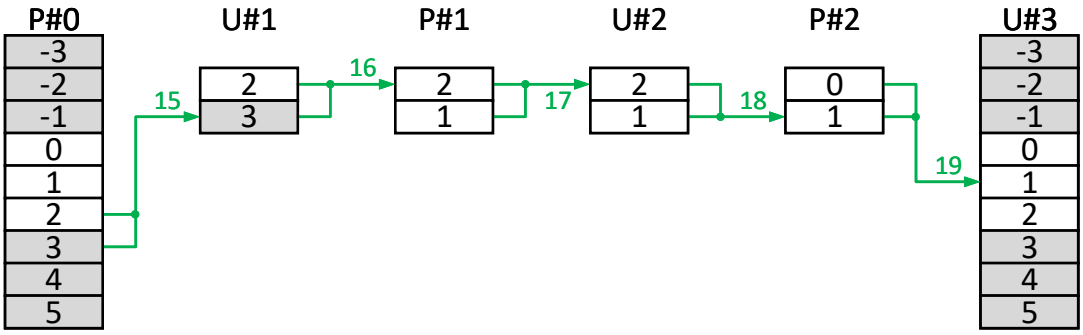


FIGURE 3.31 – Pipeline d'itérations avec allocation modulaire des tableaux intermédiaires, étape 5.

On montre par récurrence grâce à l'étape suivante (étape 5, figure 3.31) que la méthode est correcte pour l'ensemble des autres lignes.

Si l'allocation modulaire permet de réduire l'impact de l'ajout de tableaux intermédiaires, il est possible d'aller plus loin. En effet, ces derniers peuvent être supprimés complètement. Avec le pipeline d'itérations, les différents accès en mémoire ne sont pas concurrents entre eux. Dans ce cas, un seul tableau est créé pour chaque composante 2D de $\mathbf{u}$ et de $\mathbf{P}$. On appelle cette optimisation le pipeline mono-buffer.

La figure 3.32 montre le déroulé du pipeline d'itérations dans le cas d'un pipeline mono-buffer. Nous pouvons constater l'absence de conflit en cas de suppression des tableaux intermédiaires. Un problème survient cependant si l'on cherche à paralléliser cette optimisation. Pour rendre le pipeline parallèle, nous subdivisons horizontalement les images en autant de bandes qu'il y a de fils d'exécution. Chaque bande est alors traitée en parallèle de façon indépendante. Les tableaux initiaux et finaux sont partagés entre les différents fils d'exécution. Dans le cas général, en dehors des frontières, chaque fil d'exécution accède à ces tableaux à des endroits différents des autres. Le fait que les tableaux soient partagés n'engendre donc pas de conflit. En revanche, au niveaux des frontières entre 2 bandes, des conflits apparaissent. En effet, dans le cas du pipeline mono-buffer, le prologue et les derniers passages du corps de boucle entraînent un "débordement" des calculs aux frontières de la bande. Si l'on se trouve aux bords du tableau cela ne pose pas de réel problème puisque les valeurs manquantes sont évaluées en fonction de la condition aux limites choisie. Entre deux bandes, il n'est pas pertinent de fixer de conditions aux limites puisque nous disposons des vraies valeurs au delà des frontières. La concurrence des mises à jour de ces tableaux au frontières entraîne un comportement faux et non déterministe s'il n'est pas pris en compte.

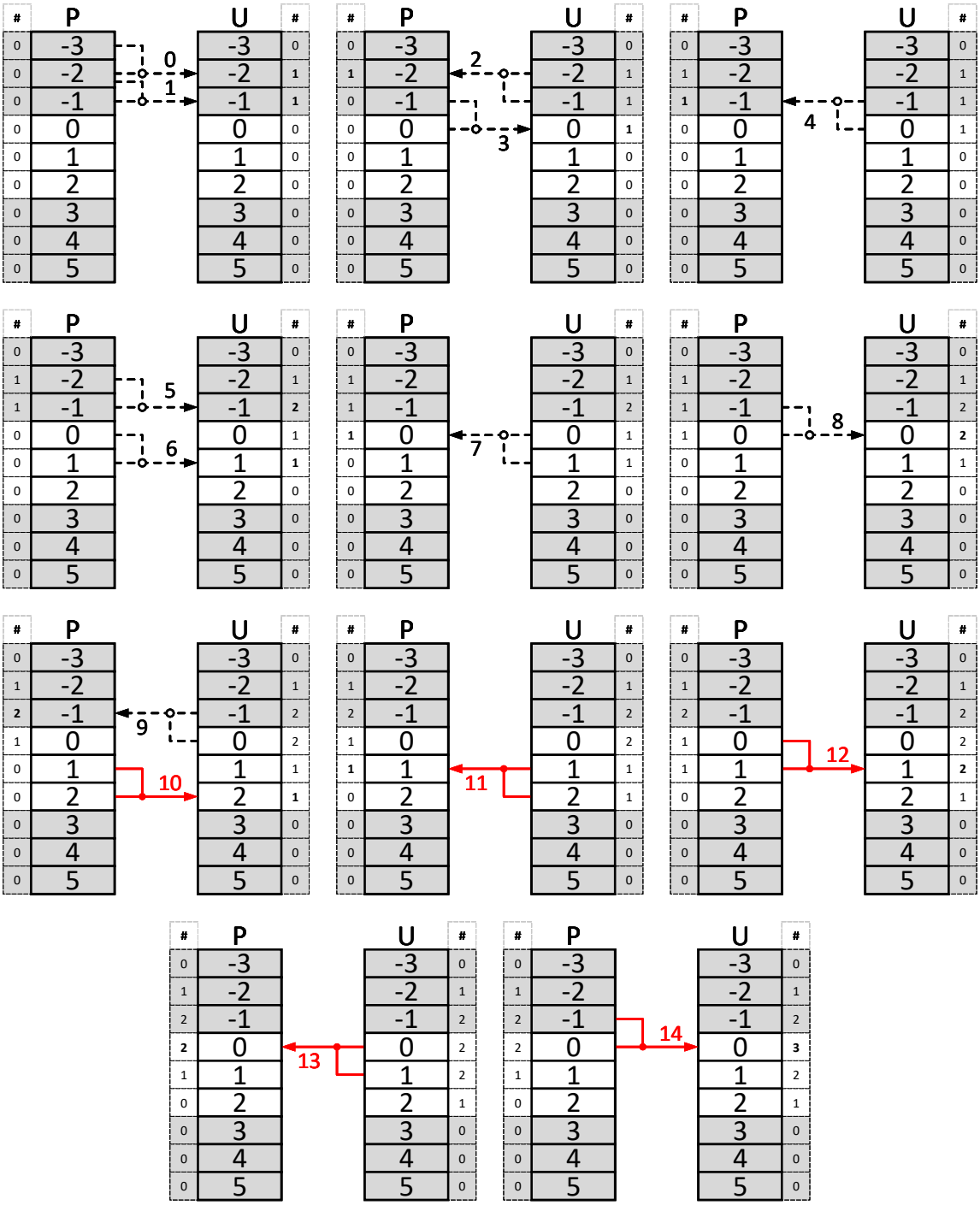


FIGURE 3.32 – Déroulé du pipeline d'itérations mono-buffer. Le numéro de l'itération est indiqué pour chaque ligne à chaque étape dans la colonne #. Un numéro d'itération en gras signifie que la ligne a été mise à jour à l'étape considérée.

Pour remédier à ce problème deux solutions peuvent être envisagées. Tout d'abord, plutôt que de partager les tableaux pleine taille entre les fils d'exécution, il est possible de les subdiviser en plusieurs sous tableaux indépendants avec les bords nécessaires pour le traitement complet d'une bande. Chaque sous tableau n'est alors vu que par un seul fil d'exécution. Cette solution résout les problèmes de conflits mais entraîne à nouveau un surplus de mémoire allouée puisque les bords sont alloués plusieurs fois. Aussi, le résultat final n'est plus disponible dans un seul tableau mais subdivisé en plusieurs sous tableaux indépendants. Ce dernier point est gênant car l'ensemble de la chaîne de traitement n'est pas parallélisé de façon similaire. Il est donc nécessaire de recopier le résultat final dans un seul et unique tableau. L'étape de recopie engendre forcément un surcoût mémoire.

La seconde solution, plus pertinente, consiste a placer une barrière de synchronisation entre les fils d'exécution juste avant l'apparition des conflits aux frontières. A partir de cette barrière un épilogue différent du corps de boucle est exécuté.

La figure 3.33 représente l'état de **u** avant la synchronisation et après l'épilogue dans le cas de trois itérations parallélisées sur 2 fils d'exécution. Cet épilogue prend en compte les mises à jour effectuées auparavant par le prologue de la bande inférieure. Puisque les fils d'exécution sont synchronisés, il est certain que pour n'importe quelle bande donnée, le prologue de la bande inférieure a été exécuté avant le franchissement de la barrière de synchronisation. Les dernières lignes de la bande sont donc partiellement itérées au moment de l'épilogue. Le rôle de l'épilogue est alors de ne calculer que les itérations manquantes pour compléter la bande traitée par le fil d'exécution considéré.

Avec cette solution, les conflits et le non déterminisme sont évités. En outre, contrairement à la solution précédente, aucun surcoût d'allocation mémoire n'est requis. De plus, l'épilogue permet d'éviter une partie de la redondance des calculs aux frontières par rapport au pipeline classique. Nous retiendrons donc cette solution aux dépens de la première.

Pour simplifier les explications, la dépendance associée à la donnée "Autres" visible en figure 3.24, n'a pas été abordée jusque à maintenant. Cette donnée correspond en réalité à l'ensemble des données connues avant la première itération et immuables au cours des itérations. Notamment, les images d'entrée et les images des *warps* I_{1w}, I_{1xw} I_{1yw}. Pour rappel, ces *warps* sont calculés juste avant les itérations (cf algorithme 4). Il est donc possible de pipeliner le calcul des *warps* avec celui des itérations. Ainsi, une ligne de *warp* n'est calculée qu'au moment de la première itération de cette ligne. Cela permet encore une fois d'utiliser au plus tôt les données calculées. Lorsque la dernière itération a été calculée, les *warps* ne sont plus utiles pour la ligne considérée. On peut donc allouer les *warp* de façon modulaire. Comme on peut le voir en figure 3.26, la profondeur du pipeline est égale au nombre d'itérations. Le modulo d'allocation des *warps* sera donc égal au nombre d'itérations.

En plus du pipeline, il est possible d'effectuer des calculs SIMD en flottants 16 bits (F16) plutôt que 32 bits (F32). Cela permet de réduire la taille des données et donc d'augmenter le parallélisme SIMD tout en diminuant la bande passante nécessaire. Cette

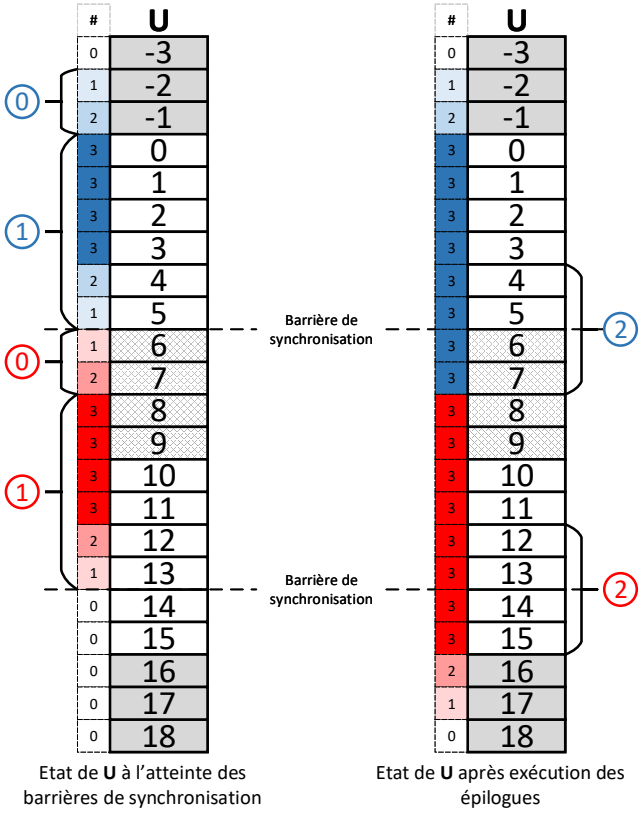


FIGURE 3.33 – États de **u** pour 3 itérations pipelinées en mono-buffer et parallélisées sur 2 fils d'exécution th0 et th1. Le premier état correspond à l'état de **u** avant le franchissement de la barrière de synchronisation. Le second état correspond à l'état final après l'exécution des épilogues. Les lignes quadrillées correspondent à la zone conflictuelle en l'absence de synchronisation et d'épilogue. Les itérations en bleu sont traitées par th0, celles en rouge sont traitées par th1. Les zones 0 correspondent aux prologues, les zones 1 au régime permanent et les zones 2 aux épilogues.

option peut permettre d'accélérer de façon significative l'application mais pose toutefois le problème de la précision des calculs. En effet, en 16 bits, la dynamique des calculs est réduite et la précision amoindrie. Le risque de dépassement de valeur devient également important. En fin de section suivante nous discuterons de la pertinence de l'utilisation des F16 pour notre application.

Dans cette section, nous avons présenté les principales optimisations apportées à TV-L^1 pour minimiser le temps de traitement. Le caractère itératif de TV-L^1 et la particularité des dépendances de données rend la fusion entre les itérations particulièrement difficile et peu pertinente à mettre en place. En revanche, le pipeline d'itérations apparaît comme une solution viable. D'autres techniques plus génériques, comme la parallélisation SIMD, ont également été appliquées. Dans la section suivante, nous présentons l'impact de ces transformations sur les performances de TV-L^1 en termes de vitesse et de consommation d'énergie.

3.3.5 Résultats et performances de TV-L^1

Dans cette section, nous discutons des performances de TV-L^1 en termes de vitesse d'exécution et de consommation d'énergie. Dans un soucis de reproductibilité nous détaillons notre protocole de mesure avant de présenter les résultats. TV-L^1 est une méthode connue et ses performances en termes de qualité de flot estimé ont déjà été analysées dans différents travaux. Une synthèse et comparaison avec l'état de l'art est notamment disponible sur le site internet de Middlebury [86]. En revanche, aucune étude n'évalue de façon approfondie les performances de la méthode en termes de vitesse et de consommation. C'est pourquoi, dans cette section nous ne considérons pas les résultats qualitatifs de la méthode. L'impact des transformations apportées est au cœur des résultats présentés ici.

Protocole expérimental

L'exécution complète de TV-L^1 en multi-échelles se décompose en plusieurs étapes. Parmi celles-ci, seule l'étape du calcul des itérations est réellement spécifique à TV-L^1. Le choix des méthodes pour les autres étapes sont l'objet d'un choix subjectif fait par le développeur. Pour le sous-échantillonnage et le calcul des *warps* une interpolation est nécessaire. Ces interpolations peuvent être bicubiques, bilinéaires ou autres et la méthode choisie a un impact majeur sur le temps de calcul total. En outre, le temps de calcul des interpolations de *warps* est dépendant du contenu des données et peut donc varier d'une paire d'images à l'autre. C'est pourquoi, dans un premier temps nous évaluons uniquement les performances de TV-L^1 dans la configuration suivante : trois itérations, une seule échelle et pas de *warp*. Cela permet de reproduire aisément les tests et de s'affranchir de nombreuses considérations subjectives.

Nous cherchons à évaluer les performances CPU de TV-L^1 pour différentes plateformes embarquées. Les architectures utilisées sont les 3 plus récentes cartes Nvidia jetson : TX2, AGX et NANO possédant également des GPU embarqués. Dans un premier

temps cependant, nous ne considérons pas l'utilisation des GPU. Les caractéristiques techniques de ces plateformes sont détaillées dans le tableau 3.3. L'algorithme est évalué pour différentes tailles d'images et à différentes fréquences de fonctionnement.

Plateforme	Gravure	CPU	Fmax (GHz)	GPU	Fmax (MHz)
TX2	16 nm	4×A57 + 2×Denver2	2.00	256c Pascal	1300
AGX	12 nm	8×Carmel	2,27	512c Volta	1377
NANO	12 nm	4×A57	1,43	128c Maxwell	921

TABLE 3.3 – Spécifications techniques des plateformes utilisées.

Dans le but de fournir des résultats facilement reproductibles, nous mesurons la consommation d'énergie de tout le système considéré. Une carte spécifiquement développée à cet effet a été conçue [122]. Cette dernière est insérée entre le système et son alimentation. La tension d'alimentation et le courant sont mesurés puis envoyés à un ordinateur hôte à une fréquence de 5 kHz. à 5000 échantillons par seconde. La carte de mesures reçoit des informations de la plateforme testée grâce à des GPIOs. Cela permet d'associer correctement les mesures avec la configuration testée. Les alimentations utilisées sont des alimentations stabilisées de laboratoire. Pour les plateformes possédant un système de refroidissement actif, celui-ci est à son maximum lors des mesures. Les outils et cartes développés durant cette thèse ont aussi été utilisés dans le cadre du projet Météorix, pour la conception d'un nano-satellite embarquant une charge utilise servant à la détection en temps réel de météors et de débris spatiaux [123, 124, 125, 126].

De nombreuses versions de TV-L^1 plus ou moins optimisées ont été testées. Pour des raisons de lisibilité, seules les plus pertinentes sont présentées dans la suite de cette section. De même, dans un soucis d'homogénéité, les résultats principalement mis en avant ici, sont ceux obtenus sur la plateforme AGX. Les résultats obtenus pour les autres plateformes sont disponibles en annexe A

Impact des transformations

Comme nous avons pu le voir dans la section précédente, les transformations ont principalement pour objectif de minimiser l'impact et le nombre d'accès mémoire. Le grand nombre d'accès mémoire par rapport au nombre de calculs (intensité arithmétique faible) nécessaires pour TV-L^1 est en effet un goulet d'étranglement majeur pour la vitesse d'exécution. Le tableau 3.4 indique l'intensité arithmétique pour le calcul d'une itération de TV-L^1 avant et après optimisation. Initialement, le nombre d'accès est quasiment identique au nombre d'opérations. Après optimisations, l'intensité arithmétique est de 1,9 et a à peine doublée. Le nombre d'accès mémoire par pixel demeure important et on comprend que la localité des accès sera également primordiale pour accélérer davantage l'exécution de l'algorithme.

Algorithme	OP/pix	MEM/pix	intensité arithmétique
TV-L$^1_{base}$	54	50	1,1
TV-L$^1_{opt}$	54	28	1,9

TABLE 3.4 – Intensité arithmétique pour une itération de TV-L^1 avant et après optimisation.

La figure 3.34 montre les évolutions du temps de calcul en cycles par pixel (CPP) et de l'énergie consommée en nanojoules par pixels (nJ/pix) pour trois itérations de TV-L^1, en fonction du niveau d'optimisation et de la taille des images traitées. Ces résultats sont donnés pour la plateforme AGX cadencée à 2,3 GHz mais sont représentatifs de l'ensemble des résultats obtenus. Toutefois, l'amplitude des gains varie en fonction de la fréquence et des plateformes.

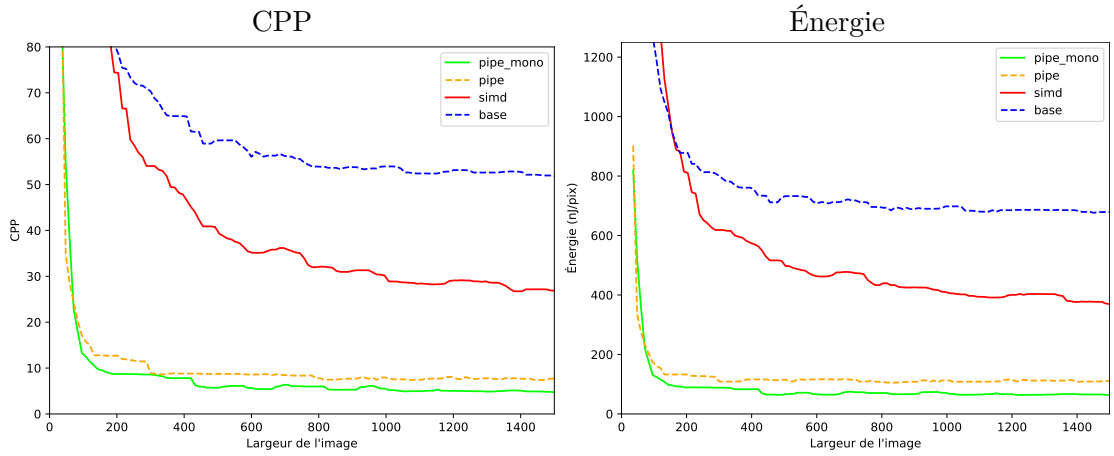


FIGURE 3.34 – Évolution du nombre de cycles par pixel (CPP) et de l'énergie consommée en nano-joules par pixel en fonction de la taille des images. Mesures pour le CPU de l'AGX à 2,3GHz.

Les implémentations de TV-L^1 présentées en figure 3.34 sont :
— La configuration *base* : implémentation standard non optimisée.
— La configuration *simd* : *base* + SIMDisation.
— La configuration *pipe* : *simd* + pipeline d'itérations.
— La configuration *pipe_mono* : *pipe* + pipeline mono-buffer.
Toutes les versions sont parallélisées sur l'ensemble des cœurs du CPU.

Les résultats présentés en figure 3.34 montrent que, en régime permanent, l'implémentation *base* permet un traitement avec un CPP de 51,9. L'implémentation *simd* permet un CPP de 27,0, *pipe* un CPP de 7,5 et *pipe_mono* en CPP de 4,8. Au niveau de la consommation énergétique l'implémentation *base* entraîne une consommation de 698 nJ par pixel, *simd* 402 nJ par pixel, *pipe* 104 nJ par pixel et *pipe_mono* 64 nJ par pixel.

L'ensemble de optimisations apportées permet donc, en régime permanent et pour la configuration présentée, une accélération de $\times 10,8$ et un gain de consommation de $\times 10,9$. La vitesse maximale est atteinte plus rapidement avec les versions *pipe* et *pipe_mono* qu'avec les deux autres versions moins optimisées. De plus, lorsqu'il est possible de visualiser une sortie de cache (cf annexe A TX2 et NANO), on constate que cette dernière se fait de façon plus tardive avec la version *pipe_mono*. Cela peut s'expliquer par une meilleure localité des accès mémoire qui permet de tenir plus longtemps en mémoire cache.

Bien que cela ne soit pas vraiment significatif dans les résultats numériques annoncés au paragraphe précédent, on observe que le gain énergétique est légèrement plus important que le gain en vitesse. Cela implique que les transformations que nous avons apportées ont un impact sur la puissance consommée.

La figure 3.35, montre l'évolution de la consommation de puissance en fonction du niveau d'optimisation, de la taille des images traitées et de la fréquence de fonctionnement. On constate que la consommation instantanée de l'implémentation *pipe_mono* est effectivement généralement la plus faible. Le cas le plus désavantageux correspond à celui présenté précédemment : grande taille d'image et fréquence élevée. Dans ce cas la puissance consommée par *pipe_mono* devient très proche de celle consommée par *base*. Les résultats présentés en annexe A montrent une tendance similaire pour les différentes plateformes : dans la plupart des cas *pipe_mono* consomme autant ou moins que *base*.

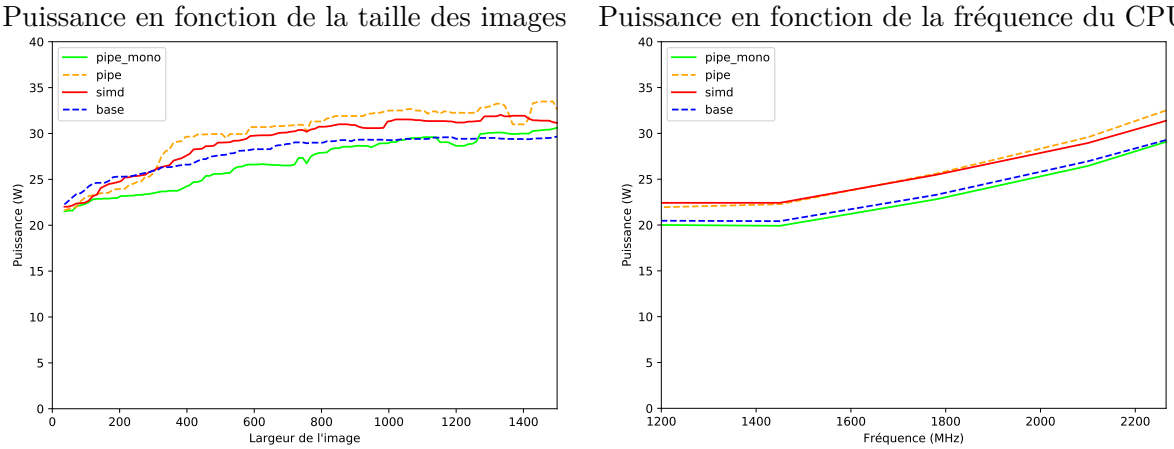


FIGURE 3.35 – Puissance consommée par l'AGX en fonction du niveau d'optimisation. Gauche : puissance consommée en fonction de la taille des images traitées pour une fréquence de 2,3GHz. Droite : puissance consommée en fonction de la fréquence pour des images de taille 1008×1008.

Cette diminution de consommation peut s'expliquer par une meilleure utilisation de la mémoire. Avoir une meilleure persistance des données en cache ou en registres permet en effet d'éviter de faire transiter les données en dehors du processeur. Le coût énergétique des accès à la mémoire externe est alors évité. Les différentes optimisations

apportées supposent cependant une utilisation plus intensive du CPU. Nous pouvons donc nous attendre également à une augmentation de la puissance CPU consommée. Puisque la consommation instantanée diminue dans notre cas, il apparaît que c'est le premier effet qui prédomine pour la majeure partie des configurations. TV-L[1] requiert en effet de nombreux accès mémoire et son temps de calcul est essentiellement subordonné à la bande passante. In fine, les transformations apportées permettent un calcul plus rapide avec une *consommation de puissance* plus faible apportant un gain encore plus important sur l'énergie totale consommée.

Les différents gains varient d'une plateforme à l'autre mais la tendance reste inchangée entre les plateformes. Il est en outre pertinent d'évaluer l'impact de la fréquence du CPU pour une plateforme donnée.

La figure 3.36 montre l'évolution du CPP pour des fréquences de 1450 MHz et 2265 MHz sur l'AGX. À basse fréquence, le rapport de vitesse entre les versions *base* et *pipe_mono* est plus faible qu'à haute fréquence. Pour nos évaluations, nous faisons varier la fréquence du CPU mais pas celle de la mémoire qui est laissée en permanence à son maximum. Diminuer la fréquence du CPU réduit donc l'écart avec celle de la mémoire. S'il est nécessaire au CPU "d'attendre" un accès mémoire externe, celui-ci se fera à la même vitesse puisque la mémoire est à fréquence constante. En revanche le CPU "attendra" pendant plus ou moins de cycles selon s'il est à haute ou basse fréquence.

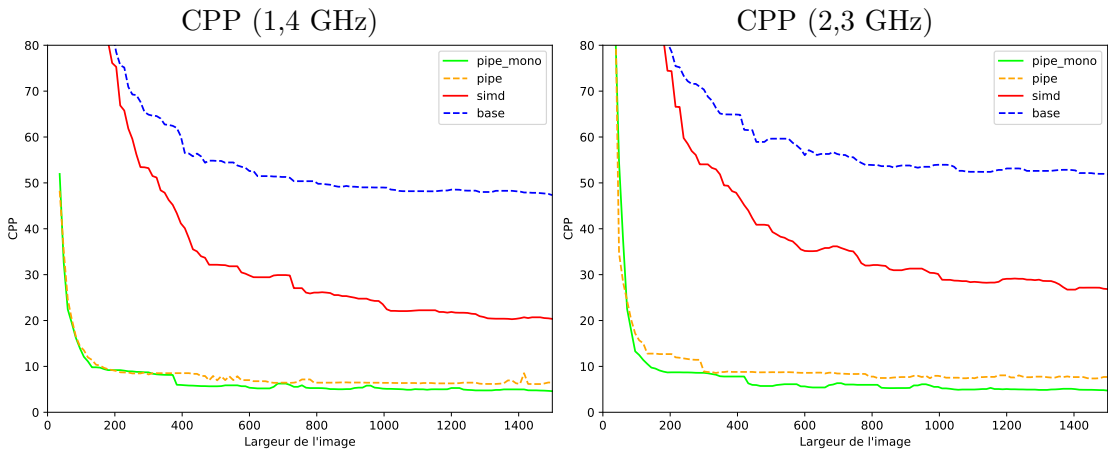


FIGURE 3.36 – Évolution du (CPP) en fonction de la taille des images sur le CPU de l'AGX à 1450 MHz et 2265 MHz.

Entre 1450 et 2265 MHz, le rapport de fréquence est légèrement inférieur à $1,6$. Le CPP de l'implémentation *pipe_mono* est de 4,6 à 1450 MHz et de 4,8 à 2265 MHz soit un rapport quasi unitaire. Le CPP de l'implémentation *base* est de 47 à 1450 MHz et de 52 à 2265 MHz soit un rapport de $1,1$. Les versions les moins optimisées de TV-L[1] sont particulièrement limitées par la bande passante. A haute fréquence, la vitesse relative de la mémoire étant plus faible, le CPP est plus élevé. Dans ce cas, il n'est pas possible

de profiter pleinement de l'augmentation de la fréquence. La version *pipe_mono* quant à elle présente un CPP similaire à 1,4GHz et à 2,3GHz. Cela montre que l'ensemble des transformations que nous avons apportées permet de mieux profiter de l'augmentation de la fréquence du CPU.

Nous avons vu que les différentes optimisations apportées permettent une nette amélioration des performances. Il est toutefois pertinent de s'interroger sur l'efficacité de notre implémentation par rapport aux performances crêtes de l'architecture considérée.

Prenons l'exemple du CPU de l'AGX. Les performances crêtes du processeur sont mesurées grâce à *Stream Triad* [77] et *OpenBLAS* [127]. Les résultats de ces micro tests sont donnés pour des calculs en flottants 32 bits dans le tableau 3.5. Nous comparons les temps d'exécution des implémentations *simd* et *pipe_mono* parallélisées sur les 8 cœurs du CPU. Nous comparons donc les implémentations non optimisée et optimisée à parallélisme équivalent (SIMD + SPMD). Pour rappel, l'intensité arithmétique de ces deux implémentations est donnée en tableau 3.4. On mesure le temps d'exécution de 3 itérations de TV-L^1 sur des images Full HD (1920×1080) et des images de taille (5000×5000). Les résultats obtenus sont présentés en tableau 3.6.

Bande passante (Go/s)				Puissance de calcul (GFlops)
Cache L1	Cache L2	Cache L3	Mémoire externe	
800	200	130	60	170

TABLE 3.5 – Performances CPU de l'AGX obtenues avec les micro tests *Stream triad* et *Openblas*. Valeurs pour des calculs en flottants 32 bits.

Format (#pixels)	Implémentation	Temps (ms)	Bande passante (Go/s)	Puissance de calcul (GFlops)
1920×1080	*pipe_mono*	4,5	155	75
	simd	21	59	16
5000×5000	*pipe_mono*	98	86	41
	simd	228	66	18

TABLE 3.6 – Performances CPU de l'AGX obtenues pour le calcul de 3 itérations de TV-L^1. Valeurs pour des calculs en flottants 32 bits.

On observe que le nombre de GFlops obtenu est bien inférieur au maximum obtenu avec *OpenBLAS*. Cela n'est pas le cas pour la bande passante. Dans le cas de la version *simd* la bande passante mesurée dans les deux cas, est proche de la bande passante maximale en mémoire externe. Dans le cas de l'implémentation *pipe_mono*, la bande passante est significativement supérieure à la bande passante maximale de la mémoire externe. C'est résultats confirment l'hypothèse que, dans le cas de TV-L^1, c'est la bande passante mémoire qui limite la vitesse de traitement.

Avec l'implémentation *simd*, la localité des accès est mal exploitée et les données sont souvent évincées du cache avant d'être utilisées à nouveau. C'est pourquoi, dans ce cas, les performances sont visiblement limitées par la bande passante en mémoire externe externe. En revanche, avec *pipe_mono*, les performances obtenues en Full HD atteignent 82% de la bande passante maximale en cache L3 mesurée avec *Stream Triad*. Cela suppose une bien meilleure utilisation de la localité des accès avec une forte persistance des données en cache.

Comparaison avec l'état de l'art

Les différents travaux existants sur TV-L^1 ne traitent généralement pas des performances de l'algorithme en termes de vitesse et de consommation. De manière plus générale, la littérature sur le calcul de flot optique se concentre très majoritairement sur les résultats qualitatifs des algorithmes et non sur leur accélération pourtant critique pour de nombreuses applications temps réel. C'est pourquoi, il nous est difficile de nous comparer à l'état de l'art. À notre connaissance il n'existe pas de travaux sur l'optimisation de TV-L^1 sur processeur généraliste. En revanche, plusieurs implémentations sont disponibles. Nous pouvons plus particulièrement citer celle du site IPOL [97] ainsi que celle de l'API OpenCV [128]. L'implémentation disponible sur IPOL constitue les fondements de la version *base* présentée précédemment. Nous avons donc d'ores et déjà montré l'efficacité de nos optimisations vis à vis de cette implémentation.

La comparaison avec OpenCV est moins évidente. Il ne nous est pas possible de désactiver les *warps* ou l'étape de construction des échelles sans apporter d'importantes transformations au code source. En outre, une étape supplémentaire est utilisée. Un filtre médian est notamment appliqué sur le flot à chaque *warp*. Ce filtre, introduit pour la première fois dans [113], permet de meilleurs résultats qualitatifs. Dans le but d'effectuer une comparaison plus juste nous avons choisi d'ôter cette étape dans OpenCV. Les méthodes d'interpolation utilisées par OpenCV sont les mêmes que celles que nous utilisons : bilinéaire pour la mise à l'échelle des champs de vecteurs et bicubique pour les *warps*. Dans ce cas, il est pertinent de comparer les deux méthodes dans leur version multi-échelles complète.

Plateforme	Configuration CPU	Temps (ms)		Gain
		OpenCV	*pipe_mono*	
TX2 @2,0GHz	4×A57	1312	434	×3,0
	4×A57 + 2×Denver	880	276	×3,2
AGX @2,27GHz	8×Carmel	379	78	×4,9

TABLE 3.7 – Comparaison du temps d'exécution des implémentations OpenCV et *pipe_mono* pour TV-L^1 avec 3 échelles, 1 *warp* par échelle et 10 itérations par *warp*. Résultats pour des images Full HD (1920 × 1080 pixels).

Le tableau 3.7 compare les temps d'exécution de notre implémentation *pipe_mono* et de celle d'OpenCV. Les temps sont donnés en millisecondes pour des images Full HD

(1920 × 1080 pixels). La configuration de TV-L^1 est : 3 échelles, 1 *warp* par échelle et 10 itérations par *warp*. Sur la TX2 notre implémentation est au moins 3 fois plus rapide que sur celle d'OpenCV. Sur l'AGX *pipe_mono* est presque 5 fois plus rapide qu'avec OpenCV. Ces résultats montrent l'efficacité des transformations que nous avons apporté par rapport aux implémentations disponibles dans la littérature.

Implémentation F16

Jusque ici, tous les résultats présentés ont été donnés pour des calculs effectués en F32. Les différentes transformations introduites ont un impact uniquement sur la vitesse d'exécution mais pas sur la précision des calculs. Comme évoqué dans la section 3.3.4, il est cependant possible d'effectuer des calculs en F16. Parmi les plateformes que nous utilisons, seule l'AGX dispose de la capacité de calculer en F16. Il a été montré que la précision des calculs en F16 est suffisante pour de nombreux autres algorithmes de traitement d'images y compris pour des méthodes de calcul de flot optique [129, 130, 131, 132, 133]. Il est en revanche nécessaire de s'interroger sur la viabilité des calculs en F16 appliqués au cas spécifique de TV-L^1.

Dans un premier temps, nous avons tenté de borner les calculs de TV-L^1 et d'interpolation. Le but était de connaître de façon certaine, la dynamique minimale de calcul pour éviter les débordements de valeur en prenant en compte la dynamique des images d'entrée. Nous avons utilisé les techniques d'arithmétique d'intervalle [74] puis d'arithmétique affine [75] sans parvenir à borner les calculs de façon suffisamment précise. En effet, TV-L^1 étant un algorithme itératif il n'existe pas, à notre connaissance, de méthode simple permettant de borner au plus près les calculs. En outre, les méthodes d'arithmétique d'intervalle et affine prennent en compte la dynamique des valeurs d'entrée mais pas l'importance des variations d'une donnée à l'autre. Les données sont en effet des images naturelles, il n'y a donc pas ou peu, de grande discontinuité d'un pixel à l'autre. Hors, les calculs de TV-L^1 sont majoritairement des calculs de gradients. Ces valeurs intermédiaires sont donc, à priori, petites devant la dynamique des images d'entrée. Ce problème de dépendance des données les unes par rapport aux autres est un sujet complexe que nous n'étions pas en mesure de traiter dans nos travaux.

N'étant pas en mesure de prouver formellement la viabilité de l'utilisation des F16, nous proposons une comparaison préliminaire entre F32 et F16. Pour cela, nous considérons à nouveau les séquences Cateye et Coastguard. Pour ces 2 séquences, nous comparons les temps de calcul et l'évolution de la MSE de reconstruction en F32 et en F6. Cette étude est une première approche et nécessitera un approfondissement dans la suite de nos travaux.

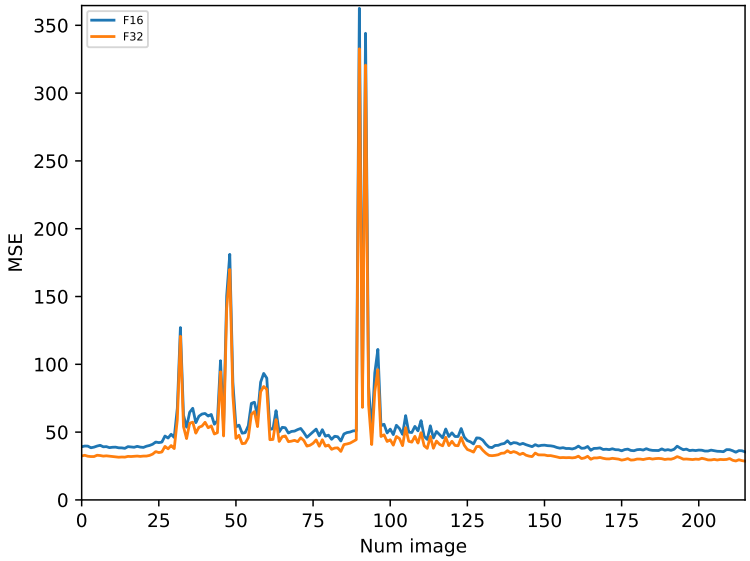


FIGURE 3.37 – Évolution de la MSE de reconstruction pour des calculs en F32 et en F16 au cours de la séquence Cateye.

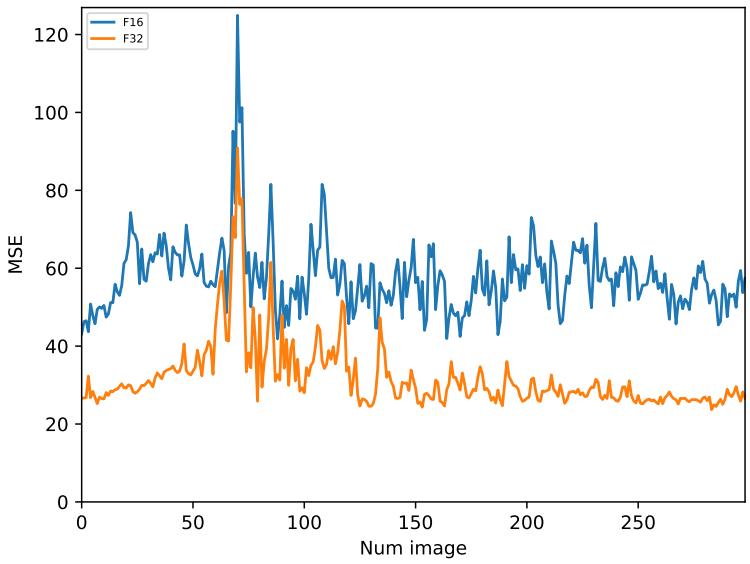


FIGURE 3.38 – Évolution de la MSE de reconstruction pour des calculs en F32 et en F16 au cours de la séquence Coastguard.

Pour l'ensemble des comparaisons entre F32 et F16, la configuration de TV-L^1 utilisée est la suivante : 3 échelles, 1 *warp* par échelle et 10 itérations par *warp*. La figure 3.37 montre l'évolution de la MSE au cours de la séquence Cateye, pour des calculs en F32 et en F6. La figure 3.38 montre les résultats de cette même comparaison pour la séquence Coastguard. On constate que l'utilisation de F16 entraîne une perte de qualité de reconstruction générale dans les deux cas. On peut voir sur la figure 3.39, que la pénalité due à l'utilisation de F16 n'excède pas les 25% pour la séquence Cateye. En revanche, sur la séquence Coastguard, la figure 3.40 montre que l'utilisation des F16 peut engendrer une erreur de MSE de plus de 150% avec une moyenne de 87%. Cette différence d'impact entre les deux séquences s'explique surtout par la faible définition de la séquence Coastguard. En effet, si la séquence Cateye est en Full HD (1920×1080), Coastguard est d'une définition bien plus faible : 352×288 pixels. Une erreur de même amplitude sera donc plus significative avec Coastguard qu'avec Cateye.

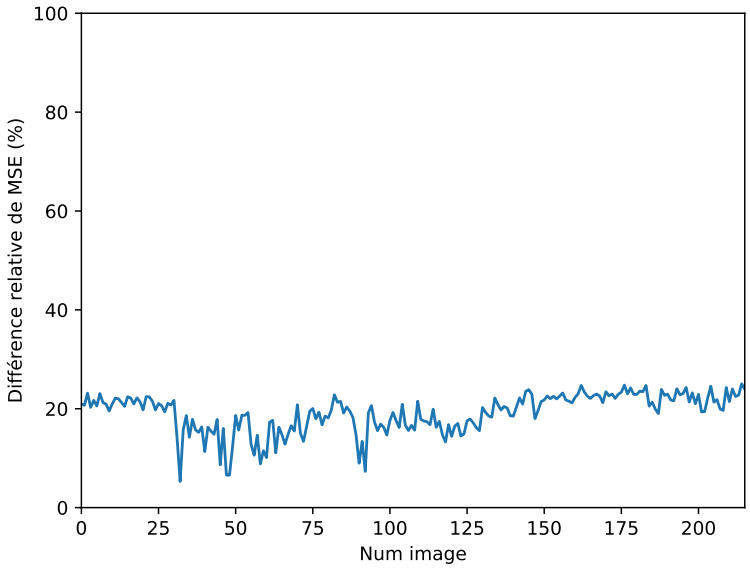


FIGURE 3.39 – Erreur relative de MSE entre F16 et F32 pour la séquence Cateye.

On comprend ici que, selon les cas, l'utilisation des F16 peut être limitante vis à vis de la qualité de reconstruction. Les comparaisons visuelles entre les résultats obtenus ne montrent pas de bien meilleures estimations du flot en F32 qu'en F16. Les images semblent être recalées de façon similaire avec les deux formats de calcul. En revanche, la réduction de la précision due aux F16 est nettement visible sur les images reconstruites. Si les images sont globalement recalées de la même façon, un phénomène d'escalier apparaît sur les images reconstruites à partir des calculs effectués en F16. Ce phénomène est illustré en figure 3.41 pour un extrait de la séquence Cateye.

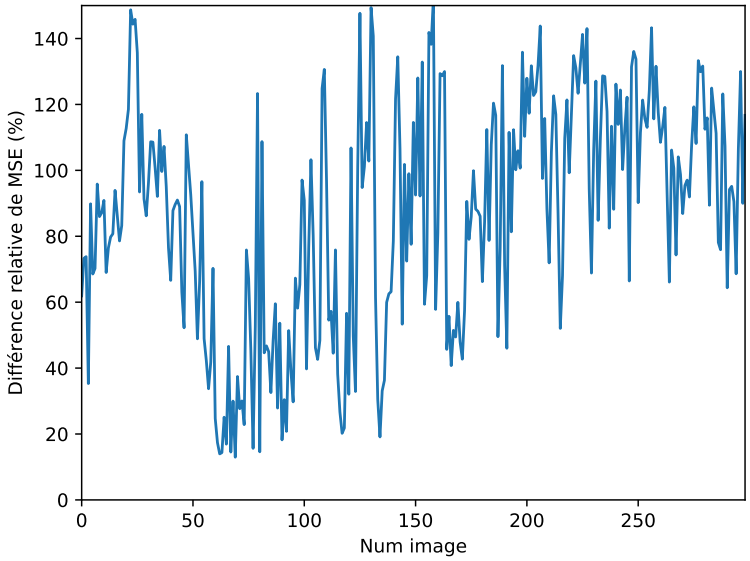


FIGURE 3.40 – Erreur relative de MSE entre F16 et F32 pour la séquence Coastguard.

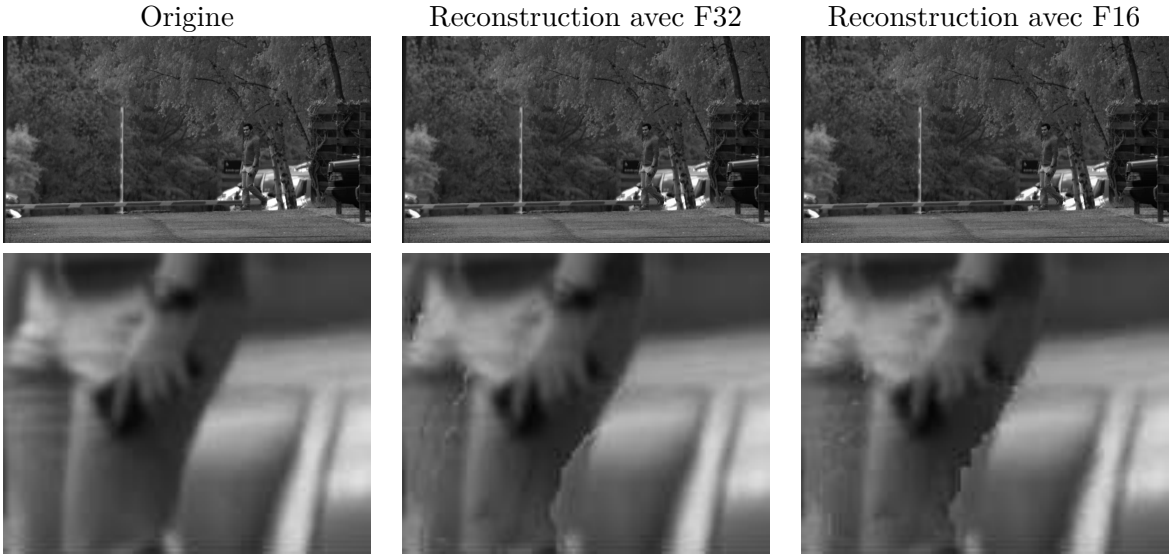


FIGURE 3.41 – Comparaison visuelle de la qualité de reconstruction entre des calculs en F32 et des calculs en F16. Image extraite de la séquence Cateye

Cet effet d'escalier est clairement visible notamment au niveau des contours de l'image. On peut le voir par exemple, dans l'extrait de la figure 3.41, au niveau de l'arrière de la cuisse gauche de l'individu présent dans la scène. La figure 3.42 indique pour le même extrait, l'image de la norme du flot optique estimé en F16 et en F32.

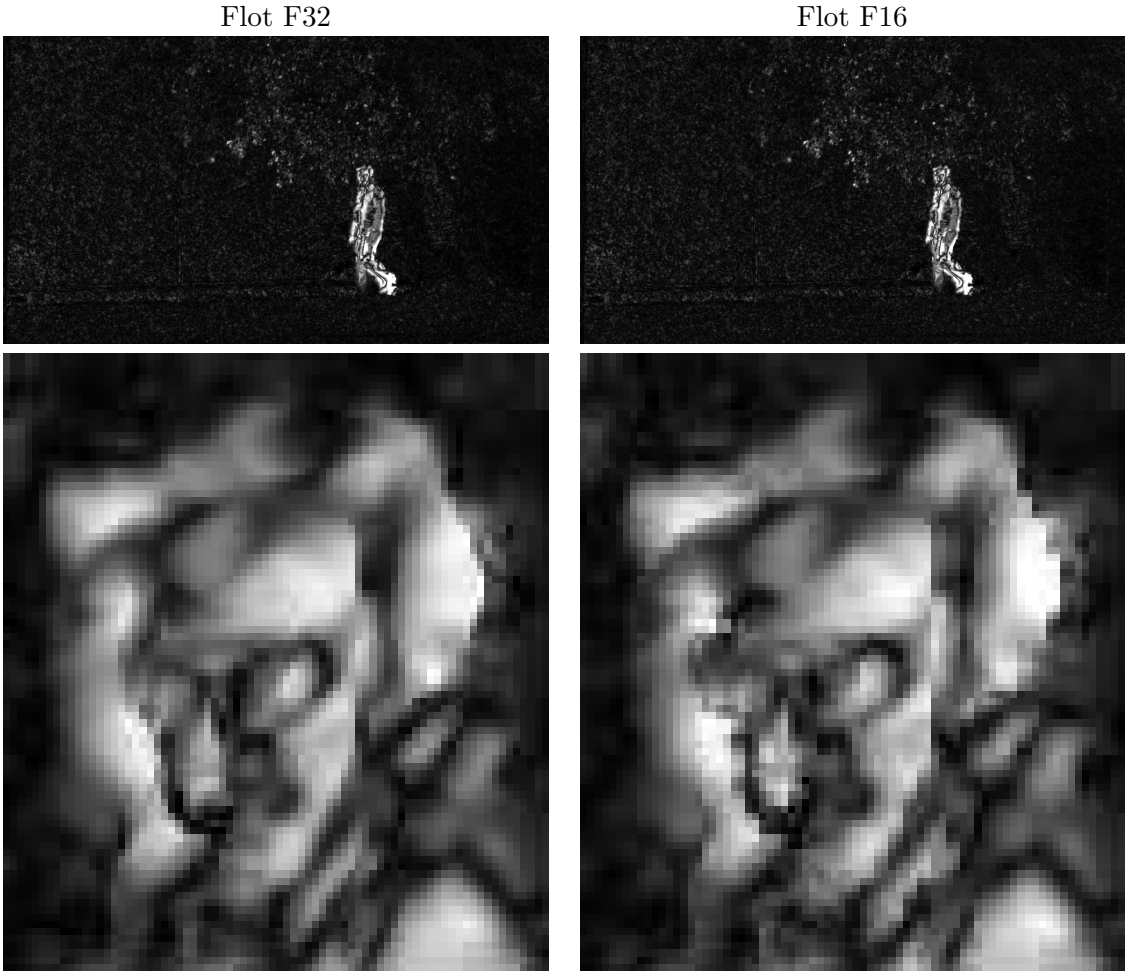


FIGURE 3.42 – Comparaison du flot optique estimé en F32 et en F16 : image de la norme du flot optique estimé. Extrait de la séquence Cateye : un crénelage apparaît en F16

Cela nous permet de mettre en évidence que la perte de précision apparaît bien lors de l'estimation du flot. Cette perte de précision s'explique par la baisse de quantification due à l'utilisation de flottants 16 bits. Il est important cependant de relativiser l'impact visuel de l'utilisation de F16. Les effets, bien que sensibles, sont particulièrement mis en avant ici en sélectionnant les zones de l'image les plus remarquables et en appliquant un zoom important.

Pour notre configuration, le temps de traitement en F32 pour la séquence Cateye est de 78 millisecondes par image contre 49 millisecondes en F16. Soit une accélération de $\times 1,5$ en F16. Étant donné que le parallélisme SIMD est doublé en F16, cela peut paraître faible par rapport au $\times 2$ potentiel. Ce manque à gagner de performances est dû au calcul des *warps*. En effet, si l'on considère seulement le calcul de 10 itération de TV-L^1, les calculs en F16 sont bien 2 fois plus rapides que ceux en F32. Le manque

d'efficacité des F16 pour les *warps* vient de la nature de l'interpolation bicubique. Le jeu d'instruction Neon utilisé, permet de manipuler des registres de 128 bits. Le cardinal SIMD en F16 est donc de 8. Hors, l'interpolation bicubique d'un point requiert le chargement de 4 paquets de seulement 4 points adjacents. Il n'est donc pas possible de profiter au maximum de toute la largeur du jeu d'instructions : 4 points sont inutiles. Pour minimiser la perte d'efficacité, nous utilisons des paires de registres 64 bits pour calculer 2 points en même temps.

Pour conclure, l'utilisation des F16 permet une accélération de $\times 1,5$ du temps de traitement. Cela se fait au prix d'une perte de précision dans l'estimation du flot optique. Les images semblent être globalement bien reconstruites par rapport à une implémentation en F32 mais la diminution de la quantification est tout de même visible. Elle se manifeste par un effet d'escalier. Pour l'heure, seul le calcul de TV-L^1 a fait l'objet d'une implémentation F16 et seuls des calculs en F32 ont été considérés pour l'exécution de la chaîne complète. C'est pourquoi il est difficile de savoir quel sera l'impact de la perte de précision sur la chaîne de débruitage complète. Il est possible que, dans le cas d'images bruitées, l'impact des F16 soit moins significatif et que leur utilisation demeure pertinente.

Pour la suite de nos travaux, nous envisageons une implémentation complète de la chaîne de traitement en F16. L'impact sur la qualité du débruitage restant un critère majeur pour décider de son effective utilisation ou non. La comparaison effectuée ici sera appliquée à un plus grand nombre de séquences afin d'évaluer un meilleur panel de situations. Par ailleurs, certains maillons de RTE-VD risquent de ne pas pouvoir supporter la baisse de dynamique, en particulier l'étape de stabilisation à cause du calcul d'image intégrale et de son accumulation d'un grand nombre de valeurs. Dans ce cas, nous prévoyons une implémentation en arithmétique hybride, dans laquelle certains pans du traitement seraient calculés en F16 et d'autres en F32. En outre, l'utilisation de nombres codés en virgule fixe plutôt que flottante est également un aspect que nous projetons d'étudier. Cette représentation est particulièrement utilisée pour accélérer des applications embarquées [134, 135, 136] mais pose également le problème de la précision.

AGX et difficultés de mesures

Au cours de nos mesures, nous avons pu constater certains résultats instables ou inattendus. Ces difficultés sont principalement apparues lors des mesures de TV-L^1 en F32 avec le CPU de la plateforme AGX. Dans cette section, nous revenons plus particulièrement sur la présence de pics de défaut de cache.

La figure 3.43 indique le temps de calcul pour 10 itérations de TV-L^1 en fonction de la taille des images considérées. Le temps y est donné en nanosecondes par pixel pour des calculs en F32 et en F16. On constate qu'en F32, la vitesse de traitement est bien moins stable qu'en F16. On observe un motif répétitif de la chute des performances. Ce motif, constitué d'un pic suivi d'un plateau puis de trois pics, se répète exactement tous les 512 pixels de largeur. Ce phénomène peut être expliqué par une taille des données causant un motif d'accès à la mémoire défavorable. Cela se traduit par une éviction régulière des

données en cache alors qu'elles vont être réutilisées, et donc un nombre élevé de défaut de cache (pic de défaut de cache).

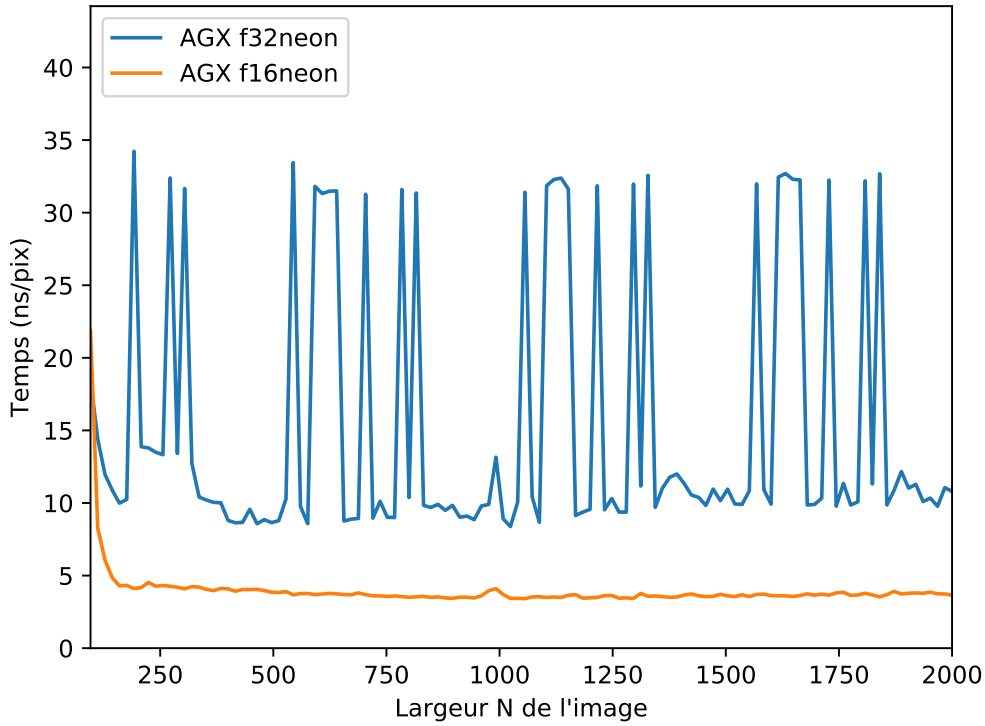


FIGURE 3.43 – Temps de calcul pour 10 itérations de TV-L[1] en nanosecondes par pixel en fonction de la taille des images. (La taille des images est incrémentée par pas de 16.)

Pour un cache non associatif, l'éviction régulière de cache peut survenir lorsque plusieurs données sont placées sur la même ligne de cache et sont accédées plusieurs fois et alternativement. La ligne de cache attribuée à une donnée est calculée à partir de l'adresse de la donnée. La taille du cache étant limitée, plusieurs données peuvent donc se voir attribuer la même ligne de cache. Si deux données D_1 et D_2 ont une adresse qui engendre le même index de ligne de cache, alors l'accès à D_1 provoquera l'éviction de D_2 et inversement.

De nos jours, la très grande majorité des processeurs utilise un cache associatif afin d'être robuste à ce type de conflit. Dans ce cas, une même adresse peut se voir attribuer au choix plusieurs lignes de cache différentes. Le nombre de lignes possibles dépend du degré d'associativité K. On dit qu'un cache est K-associatif lorsqu'une ligne associée à une adresse donnée peut être copiée dans K lignes différentes du cache.

Soit un cache 2-associatif, et D_1, D_2 et D_3, trois données qui ont le même index de ligne de cache. Les lignes de cache attribuées à ces données peuvent être soit a soit b. On utilise une politique d'éviction appelée LRU (*Less Recently Used*) : en cas de conflit, la donnée la moins récemment utilisée est éjectée du cache. On accède à D_1 puis à D_2 puis à D_3. D_1 est dans un premier temps chargée par exemple en a. D_2 est ensuite chargée en b car a est occupée par D_1. Lors de l'accès à D_3, D_1 est éjectée du cache et D_3 est chargée en a. Si l'on souhaite accéder à nouveau à D_1, la donnée ne sera donc pas présente en cache et le temps d'accès en sera pénalisé.

Nous cherchons à montrer que les pertes de performance observées en figure 3.43 peuvent effectivement être dues à une éviction régulière de cache. Dans TV-L^1, douze tableaux de même taille sont accédés de façon régulière et parcourus à la même vitesse à chaque itération. Le cache L1 de l'AGX fait 64Ko et est 4-associatif. Pour chaque tableau considéré, nous avons l'index accédé pour une itération donnée (les index avancent également à la même vitesse). Si ces index sont identiques, il peut y avoir un conflit dans le cache. Au delà de 4 index identiques, l'éviction de cache est très probable.

Dans les cas problématiques (présence de pics de baisse de performance), Nous avons observé que, parmi les 12 tableaux considérés, 10 ont le même index. Cela est bien supérieur au 4 "autorisés" par l'associativité du cache et permet d'expliquer le grand nombre de défauts de cache. Pour résoudre ce problème, nous avons pu observer qu'une phase de désallocation et de réallocation entraîne un désalignement des tableaux qui permet d'éviter cette situation. C'est la solution que nous utilisons pour contourner ce problème. Si sur l'AGX la perte de performance est flagrante sans phase de réallocation, cela n'est pas le cas pour les autres plateformes testées qui ne présentent pas ou très peu de pics de défaut de cache. En revanche, la réallocation a permis d'accélérer le traitement pour l'ensemble des plateformes testées. Pour résoudre ce problème de manière systématique et indépendamment de l'allocateur (ici, *posix_memalign*), nous prévoyons de forcer le désalignement au moment de l'allocation de chaque tableau pour s'assurer de l'absence de conflit lors des chargements en cache.

Résultats préliminaires sur GPU et prévisions

Lors de précédents travaux [124], nous avions montré que, dans le cas du calcul de flot optique par Horn & Schunck [111], l'utilisation de GPU permet d'être à la fois plus rapide et plus économe en énergie que sur CPU. Dans le cas spécifique de TV-L^1, cette *suprématie* du GPU ne semble pas aussi évidente. Le développement et l'évaluation d'une version optimisée de TV-L^1 sur GPU est actuellement en cours de développement et nous ne disposons pas encore de l'ensemble des résultats pour permettre une comparaison complète CPU vs. GPU. Les travaux sur l'implémentation GPU de TV-L^1 font l'objet d'une seconde thèse "LIP6/LHERITIER" menée par Thomas Romera. Dans cette section, nous présentons toutefois des résultats préliminaires afin de donner une visibilité sur ce que l'on peut espérer obtenir.

Si pour l'algorithme de Horn & Schunck nous avons été capable de mettre en place le

pipeline d'itérations sur GPU, cette approche n'est pas viable pour TV-L^1. L'intérêt du pipeline est d'accélérer la réutilisations des données. Un moyen efficace de faire cela sur les GPU Nvidia est d'utiliser la *shared memory* qui permet de charger en mémoire rapide (SRAM) certaines données partagées au sein d'un bloc de calcul. La *shared memory* remplace d'une certaine façon le cache du CPU mais possède une capacité bien plus faible. La quantité de données nécessaires pour le pipeline de TV-L^1 est plus importante que celle nécessaire pour le calcul de Horn & Schunck et est trop importante pour pouvoir être chargé en *shared memory*. C'est pourquoi, pour l'heure, nous n'avons pas réussi à pipeliner TV-L^1 sur les GPU utilisés.

Pour Horn & Schunck, nous avons observé, que la puissance maximale consommée par le GPU est environ 2 fois supérieure à celle du CPU pour une plateforme Nvidia Jetson TX1 ou TX2. Si le GPU permet d'accélérer le temps de traitement de TV-L^1 par 2 ou plus, cela devrait permettre une meilleure efficacité à la fois en termes de vitesse et de consommation du GPU par rapport au CPU. Le tableau 3.8 indique les temps de traitement pour 3 itérations de TV-L^1 sur CPU et sur GPU à fréquences maximales pour les plateformes TX2 et AGX. On observe que pour la TX2, le GPU 2,2 fois plus rapide que le CPU. Étant données les considérations précédentes, il est probable que, dans ce cas, le GPU soit plus efficace en tous points que le CPU. En revanche sur l'AGX, le CPU est plus rapide que le GPU. La consommation instantanée du GPU étant assurément plus élevée que celle du CPU, il est certain que le CPU sera plus efficace que le GPU dans ce cas.

Plateforme	Architecture	# cœurs	Fréquence (GHz)	Temps (ms)	Ratio
TX2	CPU	4×A57 + 2×Denver	2,00	35	$\times 2,2$
	GPU	256×Pascal	1,30	16	
AGX	CPU	8×Carmel	2,27	4,5	$\times 0,82$
	GPU	512×Volta	1,38	5,5	

TABLE 3.8 – Comparaison du temps d'exécution des implémentations CPU et GPU pour 3 itérations de TV-L^1. Résultats des calculs en F32 sur des images Full HD (1920×1080 pixels).

Ces résultats concernent le calcul de 3 itérations seules de TV-L^1. Au niveau applicatif, il faut considérer le temps de traitement multi-échelle et avec *warps*. Cela implique des calculs d'interpolation supplémentaires. Sur ce point, le GPU est nettement plus rapide que le CPU pour toutes les plateformes considérées. Cela permet au GPU de redevenir globalement plus rapide dans le cas d'une estimation complète du flot optique. Il est alors pertinent d'envisager un calcul hybride CPU/GPU de TV-L^1 afin de tirer parti au maximum de chaque architecture.

La maîtrise des deux architectures apporte un avantage stratégique lors de la conception. Si le GPU semble permettre un traitement plus rapide, il semble que certaines parties de TV-L^1 gagnent à être traitées sur CPU. La maîtrise du CPU possède un avantage supplémentaire : dans le cas des systèmes embarqués, l'encombrement est un

point souvent critique. Hors, un CPU peut être utilisé seul quand un GPU est forcément un accélérateur externe. L'utilisation du GPU implique donc également l'utilisation d'un CPU en plus. Cela entraîne nécessairement un surplus de surface et de consommation.

3.3.6 Synthèse : TV-L^1

Dans cette section, nous avons présenté la méthode de calcul de flot optique TV-L^1, qui permet d'effectuer le recalage temporel nécessaire entre les images afin de pouvoir appliquer un filtrage temporel de la vidéo sans créer de flou de bougé. Afin de d'obtenir une implémentation qui puisse être embarquée en temps réel, nous avons apporté d'importantes transformation à cette méthode.

Dans un premier temps, des compromis entre la précision et la vitesse ont été faits. Ces compromis sont déterminés par le paramétrage initial de l'algorithme : le nombre d'échelles, d'itérations et de *warps*. Des transformations algorithmiques ont également été apportées afin d'accélérer au maximum le temps de traitement. Ces modifications permettent de tirer parti des architectures CPU multicœurs SIMD. Le pipeline d'itérations permet notamment une meilleure persistance des données en mémoire cache.

Les différentes optimisations ont permis une réduction du temps de traitement et un gain énergétique substantiels sur CPU. À parallélisme SPMD équivalent, le calcul des itérations de TV-L^1 est permet un traitement environ 11 fois plus rapide qu'une implémentation non optimisée sur le CPU de l'AGX. Dans sons approche multi-échelle avec 3 échelles, 1 *warp* par échelle et 10 itérations par *warp*, notre implémentation est presque 5 fois plus rapide que l'implémentation d'OpenCV qui constitue, à notre connaissance, l'implémentation sur CPU la plus rapide de l'état de l'art. Une évaluation préliminaire a permis de constater que l'utilisation de flottants 16 bits pour le calcul du flot optique permet une accélération du calcul d'un facteur de 1,5 en multi-échelles par rapport aux flottants 32 bits. Cette accélération se fait au prix d'une perte de la précision des calculs qui se perçoit visuellement par un effet d'escalier sur des régions de l'image. Enfin, des travaux sont menés pour fournir une implémentation optimisée de TV-L^1 sur GPU. Cela permettra à terme de développer une implémentation hybride CPU/GPU pour tirer au maximum partie de ces deux architectures.

Si l'on considère la configuration multi-échelles du paragraphe précédent, notre implémentation permet d'estimer le flot optique en environ 38 nanosecondes par pixel en F32 sur le CPU embarqué de l'AGX. Il est donc possible de calculer en temps réel (en moins de 40ms) des images de définition $1025 \times 1025 \approx 1Mpix$. Cette définition est légèrement supérieur à celle du format HD : $1280 \times 720 \approx 0,9Mpix$.

Dans cette section, plusieurs façons d'évaluer TV-L^1 ont été présentées (itérations seules ou multi-échelles). Le tableau 3.9 permet une vision plus synthétique des différentes comparaisons effectuées.

Configuration			Format données	Archi	Temps (ms)	Ratio CPU/GPU
# échelles	# *warp*/échelle	# iter/*warp*				
1	0	3	F32	CPU	5,0	0,90
				GPU	5,5	
			F16	CPU	2,4	0,57
				GPU	4,2	
3	1	10	F32	CPU	78	2,0
				GPU	39	
			F16	CPU	47	1,8
				GPU	26	

TABLE 3.9 – Synthèse des temps de calcul de TV-L[1] pour la plateforme AGX pour des images Full HD. Le CPU est cadencé à 2,27 GHz et le GPU à 1,38 GHz.

3.4 Filtrage spatio-temporel

Dans cette section nous détaillons l'étape finale de filtrage de la chaîne de débruitage RTE-VD. Cette étape est celle qui permet effectivement de réduire la quantité de bruit dans la vidéo. Puisque nous traitons des vidéos, le filtre appliqué peut être spatio-temporel. Les filtres de débruitage partent du postulat suivant : le bruit est un évènement ponctuel et aléatoire spatialement et temporellement. Il est donc possible de pondérer la valeur d'un pixel avec celle de ses voisins et avec lui même au cours du temps. Dans RTE-VD, nous utilisons un filtre trilatéral de dimension 3, basé sur une approche de filtre bilatéral avec décorrélation temporelle.

3.4.1 Motivations et choix de la méthode

Une fois le recalage temporel effectué, de nombreuses méthodes de filtrage peuvent être envisagées. L'importante littérature sur le débruitage d'image seule est majoritairement adaptable en 3 dimensions. Les filtres peuvent être de type point à point [1, 2, 3, 137, 5] ou basés sur la recherche de patchs [10, 16]. Des méthodes à base de réseaux de neurones convolutionels (CNN) existent également [23, 24, 22].

Pour notre application, certaines contraintes doivent être respectées. La nécessité de débruiter les vidéos en direct apporte une contrainte forte sur le temps de traitement maximal. Dans les standards, une vidéo est considérée comme fluide à partir de 25 images par seconde, soit environ 40 millisecondes par image. Étant donné le temps de calcul nécessaire à l'estimation du flot optique, il est important que cette dernière étape de filtrage soit la plus rapide possible. Un filtrage trop long impliquerait de réduire la définition des images traitées de façon significative alors que celle-ci est déjà limitée en raison du calcul de TV-L^1. Dans la section 3.3.1 nous avons pu voir que les méthodes de recherche de patchs ou de CNN sont trop lentes pour être considérées ici.

En outre, le filtre appliqué doit posséder certaines propriétés assurant la stabilité de notre méthode. En cas de changement majeur et brutal au sein de la scène filmée, il est important que l'évènement soit transmis à l'opérateur. Dans ce cas, le filtrage doit être minimisé pour pouvoir mettre en avant le changement. Cette considération est aussi vraie spatialement que temporellement. On dit que le filtrage doit respecter les transitions. Cette propriété permet également de minimiser l'impact d'éventuels artefacts dus au recalage par flot optique.

Le filtrage bilatéral [5] est un filtre point à point connu pour respecter les transitions. C'est à partir de cet algorithme que nous avons mis en place notre filtre spatio-temporel. D'autres filtres comme le filtre guidé [138, 139] permettent également un respect des transitions avec une complexité faible. S'il n'est pas présenté ici, il pourra être pertinent, à l'avenir, de considérer également le filtre guidé pour notre application.

3.4.2 Description de l'algorithme

Le filtre bilatéral applique, à chaque pixel d'un voisinage, une double pondération. La bilatéralité est apportée par la décorrélation de ces deux pondérations : la première est dépendante de la distance tandis que la seconde dépend de la différence de valeur entre les pixels considérés. Le filtre bilatéral dans sa forme générale est exprimé dans l'équation 3.40.

$$I_F(x) = \frac{1}{W_p} \sum_{x_i \in \Omega} I(x_i) \times f(||I(x_i) - I(x)||) \times g(||x_i - x||) \tag{3.40}$$

Où W_p est le terme de normalisation tel que :

$$W_p = \sum_{x_i \in \Omega} f(||I(x_i) - I(x)||) \times g(||x_i - x||) \tag{3.41}$$

Et :
— I est l'image d'origine,
— I_F est l'image filtrée,
— x les coordonnées du pixel courant,
— Ω la fenêtre de filtrage centrée en x,
— f la fonction de pondération en fonction de la différence d'intensité,
— g la fonction de pondération en fonction de la distance.

Le plus souvent f et g sont des fonctions gaussiennes. Le filtre s'exprime alors comme dans l'équation 3.42 :

$$I_F(x) = \frac{1}{W_p} \sum_{x_i \in \Omega} I(x_i) e^{\frac{-[I(x_i)-I(x)]^2}{2\,\sigma_i^2}} \times e^{\frac{-[x_i-x]^2}{2\,\sigma_d^2}} \tag{3.42}$$

Où σ_i et σ_d sont les termes d'adoucissement associés respectivement à la différence d'intensité et à la distance entre les pixels. Ce filtre initialement 2D peut être directement transcrit en 3D pour ajouter la prise en compte de la dimension temporelle. Pour cela, il suffit de considérer Ω comme un voisinage 3D composé de plusieurs images consécutives.

Le voisinage 3D a cet inconvénient qu'il augmente de façon significative le nombre de pixels pris en compte. Cela est particulièrement pénalisant pour le temps d'exécution. En outre, il est intéressant de chercher à décorréler l'intensité du filtrage temporel de celle du filtrage spatial. Pour ces raisons nous avons choisi de séparer le filtre temporel du filtre spatial et d'utiliser une troisième fonction de pondération sur la dimension temporelle. L'image filtrée temporellement est notée I_t. On peut considérer que le filtre

utilisé n'est donc pas un filtre bilatéral mais plutôt un filtre trilatéral de dimension 3. L'expression complète du filtre est donnée dans les équations 3.43 et 3.44.

$$I_F(x) = \frac{1}{W_p} \sum_{x_i \in \Omega} I_t(x_i) e^{\frac{-[I_t(x_i) - I_t(x)]^2}{2\,\sigma_i^2}} \times e^{\frac{-[x_i - x]^2}{2\,\sigma_d^2}} \tag{3.43}$$

Où :

$$I_t(x) = I_p(x) e^{\frac{-[I_p(x) - I(x)]^2}{2\,\sigma_t^2}} + I(x) \left(1 - e^{\frac{-[I_p(x) - I(x)]^2}{2\,\sigma_t^2}} \right) \tag{3.44}$$

Avec I_p l'image précédente filtrée et compensée, et σ_t le facteur d'adoucissement du filtrage temporel. Plus les valeurs respectives de σ_i, σ_d et σ_t sont élevées, plus le filtrage sera fort sur la composante associée.

3.4.3 Implémentation et optimisations

Le filtre bilatéral n'est mathématiquement pas séparable. L'augmentation de la fenêtre de filtrage entraîne donc une augmentation quadratique du nombre de données nécessaires au traitement. Même en considérant un voisinage assez petit (5×5 ou 3×3) le temps de calcul est déjà problématique. Il a cependant été montré dans [140] que la séparation du filtre bilatéral est une bonne approximation qui respecte les transitions dans l'ensemble des directions. Cela reste vrai tant que la taille du voisinage considéré n'est pas très grande [141]. Dans notre cas nous considérons uniquement des noyaux de petite taille et c'est pourquoi nous pouvons nous permettre de séparer spatialement notre filtre. Cette séparation constitue la principale optimisation spécifique à cette dernière étape de débruitage.

Sans refaire le travail de [140], on peut comprendre intuitivement pourquoi la séparation du filtre bilatéral est une bonne approximation. En reprenant l'équation 3.42 on constate que :

— Si les valeurs des pixels considérés sont proches, alors : $-[I(x_i) - I(x)]^2 \to 0$.
Donc $e^{\frac{-[I(x_i) - I(x)]^2}{2\,\sigma_i^2}} \to 1$.
Dans ces conditions le filtre s'apparente donc à un filtre gaussien qui lui est séparable.

— Dans le cas d'une transition forte alors : $[I(x_i) - I(x)]^2$ est grand.
Donc $e^{\frac{-[I(x_i) - I(x)]^2}{2\,\sigma_i^2}}$ est proche de 0.
Dans ce cas, la valeur du pixel courant est prépondérante sur le résultat du filtrage. Si une transition est présente dans une dimension alors la valeur du pixel courant filtré reste très proche de celle d'origine. Ainsi, si la transition est également

présente dans la seconde direction alors la valeur finale du pixel restera encore une fois proche de la valeur de départ. En revanche s'il n'y a pas de transition dans la seconde dimension on retombe dans une situation de filtrage pseudo-gaussien.

Une seconde optimisation peut être appliquée, en constatant que l'étape de filtrage repose essentiellement sur le calcul de plusieurs exponentielles. Cette fonction est particulièrement lente et implique un mauvais pipeline d'instructions. Nous avons donc cherché à accélérer la fonction exponentielle. Les résultats des exponentielles sont compris entre 0 et 1 et servent à pondérer l'importance de la valeur d'un pixel dans le résultat final. Étant donné le caractère approximatif de la méthode, on comprend qu'il n'est pas nécessaire d'obtenir des résultats très précis. Une approximation à 2 ou 3 chiffres significatifs est largement suffisante pour notre application. Dès lors, il est possible d'utiliser une LUT contenant des résultats d'exponentiel pré-calculés. Selon la prise en compte ou non du facteur d'adoucissement lors de la génération de la LUT, cette dernière peut nécessiter une régénération en cas de changement du paramètre. La LUT étant de petite taille cela peut rester avantageux par rapport à un calcul systématique d'exponentielles.

Avec une parallélisation SIMD du filtre, l'utilisation d'une LUT devient caduque : Il est nécessaire de casser le parallélisme pour accéder aux résultats associés à chaque élément du registre SIMD. C'est pourquoi, nous avons implémenté une approximation rapide de la fonction exponentielle. Cette approximation rapide a été introduite dans [142] et étendue dans [143]. Une description détaillée de la méthode incluant une implémentation SIMD est donnée dans [144]. À partir du constat que : $e^x = 2^{xlog_2(e)} = 2^{x_i+x_f}$, avec x_i la partie entière de x et x_f sa partie fractionnaire, l'idée est de manipuler la représentation binaire des flottants du standard IEEE-754 pour calculer rapidement les puissance de 2. Bien que cela ne soit pas nécessaire pour notre application, il est possible d'améliorer la précision de la méthode en lui associant une une interpolation polynomiale de la partie fractionnaire 2^{x_f} [145].

La séparation du filtre bilatéral et l'accélération du calcul d'exponentielles sont les principales optimisations apportées au filtrage spatio-temporel. Dans la section suivante nous détaillons l'impact de l'ensemble des optimisations apportées à ce maillon de la chaîne de débruitage RTE-VD.

3.4.4 Résultats

Dans cette section nous étudions l'impact des transformations apportées au filtrage spatio-temporel sur le temps d'exécution. Les plateformes utilisées pour fournir les résultats sont les cartes Nvidia Jetson AGX (AGX) et Jetson NANO (NANO). Pour rappel, le CPU de L'AGX possède 8 cœurs ARM Carmel et le celui de la NANO possède 4 cœurs ARM A57.

Différentes implémentations du filtre sont comparées afin de déterminer l'impact de chaque transformation sur le temps d'exécution :

- **Base :** Implémentation non optimisée du filtre.
- **Sépar :** Implémentation avec séparation du filtre bilatéral pour les composantes spatiales.
- **IKJ :** Sépar + fusions de calculs, scalarisation et modification de boucles i,j,k en boucles i,k,j.
- **fExp :** IKJ + approximation rapide de la fonction exponentielle.
- **SIMD :** fExp + SIMDisation.
- **OMPx :** SIMD + Parallélisation sur x cœurs.

Pour les mesures, la fréquence du CPU est fixée à 2,3GHz sur l'AGX et 1,4GHz sur la NANO. Chaque implémentation est exécutée sur un fil d'exécution sauf pour la version OMPx qui est exécutée sur x fils d'exécution. Toute application autre que le filtrage est coupée avant de démarrer les mesures de temps. Le système de refroidissement actif est au maximum. Les images utilisées sont au format Full HD (1920×1080 pixels). Le rayon du voisinage de pondération pour le filtrage spatial est de 2 soit une fenêtre de taille 5×5 pixels.

Version	Base	Sépar	IKJ	fExp	SIMD	OMP4	**Mono-cœur**	**Total**
Temps (ms)	1395	594	570	383	71	18	–	–
Accélération	$\times 1$	$\times 2,3$	$\times 1,04$	$\times 1,5$	$\times 5,4$	$\times 3,9$	$\times 20$	$\times 76$

TABLE 3.10 – Impact des optimisations du filtre 3D pour des images Full HD sur le CPU de la NANO. Les accélérations sont indiquées par rapport à l'implémentation directement précédente.

Version	Base	Sépar	IKJ	fExp	SIMD	OMP8	**Mono-cœur**	**Total**
Temps (ms)	316	134	108	58	17	2,4	–	–
Accélération	$\times 1$	$\times 2,4$	$\times 1,2$	$\times 1,9$	$\times 3,4$	$\times 7,1$	$\times 19$	$\times 132$

TABLE 3.11 – Impact des optimisations du filtre 3D pour des images Full HD sur le CPU de l'AGX. Les accélérations sont indiquées par rapport à l'implémentation directement précédente.

Les tableaux 3.10 et 3.11 indiquent les temps de filtrage pour les implémentations considérées pour les CPU de la NANO et de l'AGX. Le parallélisme SPMD étant particulièrement connu et utilisé dans la littérature, nous indiquons le gain total obtenu sans parallélisme SPMD (Mono-cœur). Cela nous permet de montrer de façon plus équitable, l'apport de nos optimisations par rapport à une implémentation standard. Enfin, l'accélération totale entre la version de base et la version la plus rapide est donnée.

On constate que l'ensemble des optimisations apportées permet un gain total d'environ $\times 20$ en mono-cœur de $\times 76$ en 4 cœurs sur la NANO et de $\times 132$ en 8 cœurs sur l'AGX. Les accélérations les plus importantes sont apportées par les parallélisations

multi-cœur et SIMD. On observe une efficacité de la parallélistaion SPMD de 98% sur 4 cœurs et de 89% sur 8 cœurs. La séparation du filtre bilatéral permet une accélération supérieure à $\times 2$ pour une fenêtre de taille 5×5. Pour des voisinages plus grands, la complexité du filtre augmente de façon linéaire avec un filtre séparé. Si le filtre n'était pas séparé cette augmentation se ferait de façon quadratique. Le gain apporté par la séparation du filtre sera donc proportionnellement plus important avec une fenêtre de voisinage plus grande. Enfin, l'approximation des exponentielles par manipulation du standard IEEE-754, permet un calcul presque 2 fois plus rapide.

3.4.5 Synthèse : Filtrage spatio-temporel

Dans cette partie nous avons abordé la dernière étape de la chaîne de débruitage RTE-VD. C'est à cette étape que le bruit est effectivement filtré. Étant donné le temps de traitement important requis par le calcul du flot optique à l'étape précédente, nous avons choisi un filtre rapide et peu complexe. Nous avons choisi une méthode qui s'apparente à un filtrage bilatéral en 3 dimensions avec une décorrélation du filtrage temporel par rapport au filtrage spatial. Cette méthode permet de respecter les transitions spatiales et temporelles. Ainsi, les évènements notables de la scène filmée sont bien retransmis à l'opérateur. Cet aspect assure, en outre, une plus grande robustesse aux éventuelles erreurs et artefacts dus à une mauvaise estimation du flot optique.

Des transformations ont été apportées afin de minimiser le temps de calcul du filtrage. La séparation du filtre bilatéral ainsi que l'approximation rapide des exponentielles ont notamment apporté des gains substantiels respectivement de $\times 2,67$ et $\times 1,92$. L'ensemble des optimisations appliquées entraîne une accélération totale de $\times 20$ en mono-cœur. La bonne efficacité de la parallélisation SPMD permet une accélération de $\times 3,9$ sur 4 cœurs et de $\times 7,1$ sur 8 cœurs. *In fine*, l'accélération totale obtenue est de $\times 76$ sur la NANO et de $\times 132$ sur l'AGX. La NANO permet de traiter des images Full HD en 18 ms soit à une cadence de plus de 55 images par secondes. L'AGX traite une image Full HD en 2,4 ms soit une cadence de plus de 416 images par secondes.

3.5 Synthèse : Chaîne de traitement RTE-VD

Dans ce chapitre, nous avons présenté et détaillé l'ensemble des maillons de la chaîne complète de débruitage mise au point au cours de cette thèse. La chaine de traitement, nommée RTE-VD (*Real-Time Embedded Video Denoising*), est constituée de 3 étapes distinctes. Une première étape de stabilisation permet de compenser le mouvement de la caméra afin de faciliter l'étape suivante. Une fois la stabilisation effectuée, la deuxième étape estime le flot optique dense entre deux images consécutives. Cela permet de recaler les images entre elles point à point. Ainsi, il est possible d'appliquer une pondération dans le temps en conservant la cohérence temporelle. La troisième et dernière étape est l'application d'un filtre spatio-temporel qui permet, in fine, de réduire le bruit de la vidéo.

La première étape de stabilisation est nommée StabLK et est basée sur une approche globale de l'estimation de flot de Lucas et Kanade. La seconde étape utilise l'algorithme d'estimation dense du flot optique TV-L^1. Cette étape est la plus critique en termes de temps de calcul et a fait l'objet d'une étude plus approfondie. Enfin, l'étape de filtrage utilise un filtre trilatéral séparé avec une décorrélation temporelle du filtre. Chaque étape a été optimisée afin de minimiser le temps de calcul pour permettre un débruitage en temps réel (25 images par secondes). Tous les maillons de la chaîne de traitement sont parallélisés en SIMD et en SPMD. La fusion et le pipeline d'opérateurs ainsi que d'autres transformations classiques d'optimisation (rotation de registre, déroulage de boucle...) sont appliqués dès que cela est possible. Des transformations plus spécifiques à chaque maillon ont été détaillées dans ce chapitre.

Pour StabLK, la principale optimisation repose sur la l'utilisation et la parallélisation du calcul de convolution par image intégrale. Pour TV-L^1, un compromis entre le temps de calcul et la précision a dû être fait. Le pipeline d'itérations, une nouvelle façon d'itérer l'algorithme, a été mis en place afin d'améliorer la localité mémoire au maximum. Pour le filtrage spatio-temporel, nous avons appliqué une séparation du filtre ainsi qu'une approximation rapide SIMD de la fonction exponentielle.

L'ensemble des transformations apportées ont permis un gain important sur le temps d'exécution de tous les maillons de la chaîne de traitement. À parallélisme SPMD équivalent, les optimisations apportées à StabLK permettent une accélération d'environ $\times 190$ par rapport à l'implémentation de base avec un gain majeur de $\times 31$ apporté par l'utilisation d'images intégrales pour les calculs de convolution. Les optimisations apportées à TV-L^1 quant à elles, permettent une itération presque 11 fois plus rapide qu'avec la version de base. En comparaison avec l'implémentation d'OpenCV, notre implémentation multi-échelles est jusque 4,2 fois plus rapide sur l'AGX. Enfin, Notre implémentation optimisée du filtre spatio-temporel est 20 fois plus rapide que son implémentation de base à parallélisme SPMD équivalent.

Chapitre 4

Mise en place de RTE-VD et résultats

Après avoir introduit et détaillé les différents éléments de la chaîne de traitement dans le chapitre 3, nous abordons ici les performances de RTE-VD dans son intégralité. Dans un premier temps, nous expliquons les différents paramètres choisis pour exécuter RTE-VD. Nous discutons ensuite de l'efficacité de notre méthode en termes de débruitage et de temps d'exécution. Pour ce faire, nous comparons RTE-VD à trois méthodes de référence dans la littérature : VBM3D [25], VBM4D [26] et STMKF [35]. Nous proposons également une étude du temps d'exécution et de la consommation énergétique de notre chaîne de traitement pour différentes architectures embarquées. Enfin, nous présentons le prototype VIRTANS de débruitage temps réel embarqué pour les caméras LHERITIER.

4.1 Paramétrage de RTE-VD

Avant d'évaluer les performances de RTE-VD, nous présentons les différents paramètres et la configuration finale de la méthode. Le paramétrage de TV-L^1 est particulièrement critique pour les performances de la méthode que ce soit vis-à-vis de la qualité du débruitage ou du temps de calcul. La force du filtrage a également un impact important sur les résultats qualitatifs de notre méthode.

4.1.1 Configuration de TV-L^1

Comme nous avons pu l'évoquer dans la section 3.3, le choix du nombre d'itérations de TV-L^1 est en partie arbitraire et il est difficile de déterminer un nombre optimal d'itérations. L'évolution de la MSE en fonction du nombre d'itérations donne peu d'indications sur la qualité réelle du flot obtenu. En outre, il est important de considérer le calcul du flot dans son application. Si peu d'itérations suffisent à obtenir une bonne reconstruction dans la plupart des cas simples, il est important d'avoir une certaine robustesse dans l'estimation du flot. Cela est d'autant plus vrai si l'on considère des images bruitées.

Dans la section 3 du chapitre 3, nous avons déterminé une configuration de base de TV-L^1 avec 3 échelles, 1 *warp* par échelle et 10 itérations par *warp*. On appelle cette configuration la configuration 10-10-10. Cette dénomination met en valeur le nombre d'itérations spécifique à chaque échelle qui est primordial pour la différenciation des autres configurations considérées dans la suite de ce chapitre. Dans le chapitre 3, nous avons également noté qu'augmenter le nombre d'itérations n'a pas d'impact significatif sur la valeur de la MSE. Visuellement, il semble tout de même qu'augmenter le nombre d'itérations soit légèrement plus robuste et permet de générer moins d'artefacts. Au regard du temps de traitement nécessaire pour 10 itérations par échelles, il n'est cependant pas envisageable d'augmenter de façon significative le nombre d'itérations pour l'ensemble des échelles. En considérant un nombre d'échelles suffisamment grand par rapport à l'amplitude des mouvements, nous avons également pu observer que les artefacts interviennent principalement au niveau des mouvements les plus importants.

À partir de ces observations, nous avons cherché à augmenter le nombre total d'itérations de TV-L^1 sans augmenter la durée du traitement et en privilégiant les mouvements de plus grande ampleur. Pour ce faire, nous avons procédé à une nouvelle répartition du nombre d'itérations entre les différentes échelles. L'objectif est de répartir les itérations entre les échelles en gardant un nombre total d'opérations équivalent à celui de la configuration 10-10-10. Le facteur d'échantillonnage entre les échelles est à prendre en compte. Dans notre cas, les dimensions des images sont divisées par 2 d'une échelle à l'autre, soit une réduction du nombre de pixels d'un facteur 4.

Nous faisons le choix d'utiliser 3 échelles dans la pyramide de TV-L^1 : E_0, E_1 et E_2. E_0 est l'échelle pleine taille non échantillonnée, E_1 est la seconde échelle sous-échantillonnée 1 fois et donc 4 fois plus petite que E_0. E_2 est l'échelle la plus petite, sous-échantillonnée 2 fois et donc 16 fois plus petite que E_0. On définit une opération comme étant l'itération d'un pixel quelconque. On note K le nombre total d'opérations effectuées et D la définition des images d'entrée. Pour la configuration 10-10-10 on a :

$$K = 10D + 10\frac{D}{4} + 10\frac{D}{16} = \frac{210}{16}D \tag{4.1}$$

Dans un premier temps, nous cherchons à répartir le nombre d'opérations de façon équitable entre les échelles. Cela a pour effet de réduire le nombre d'itérations effectuées sur l'échelle la plus grande et de l'augmenter pour les échelles les plus petites. On a alors :

$$K = \frac{70}{16}D + \frac{70}{16}D + \frac{70}{16}D = \left(4D + \frac{6}{16}D\right) + \frac{70}{16}D + \frac{70}{16}D \tag{4.2}$$

Nous "basculons" les opérations en surplus de E_0 sur l'échelle directement inférieure E_1. K devient :

$$K = 4D + \frac{76}{16}D + \frac{70}{16}D = 4D + 19\frac{D}{4} + 70\frac{D}{16} \tag{4.3}$$

Ainsi, en exécutant 4 itérations sur E_0, 19 sur E_1 et 70 sur E_2, nous pouvons nous attendre à un temps de traitement similaire à celui de la configuration 10-10-10. Bien que cette configuration soit strictement équivalente en termes de nombre d'opérations, nous faisons le choix de privilégier encore davantage les mouvements de plus grande amplitude en répartissant l'équivalent d'une itération d'E_0 sur E_2 puis sur E_1. On obtient :

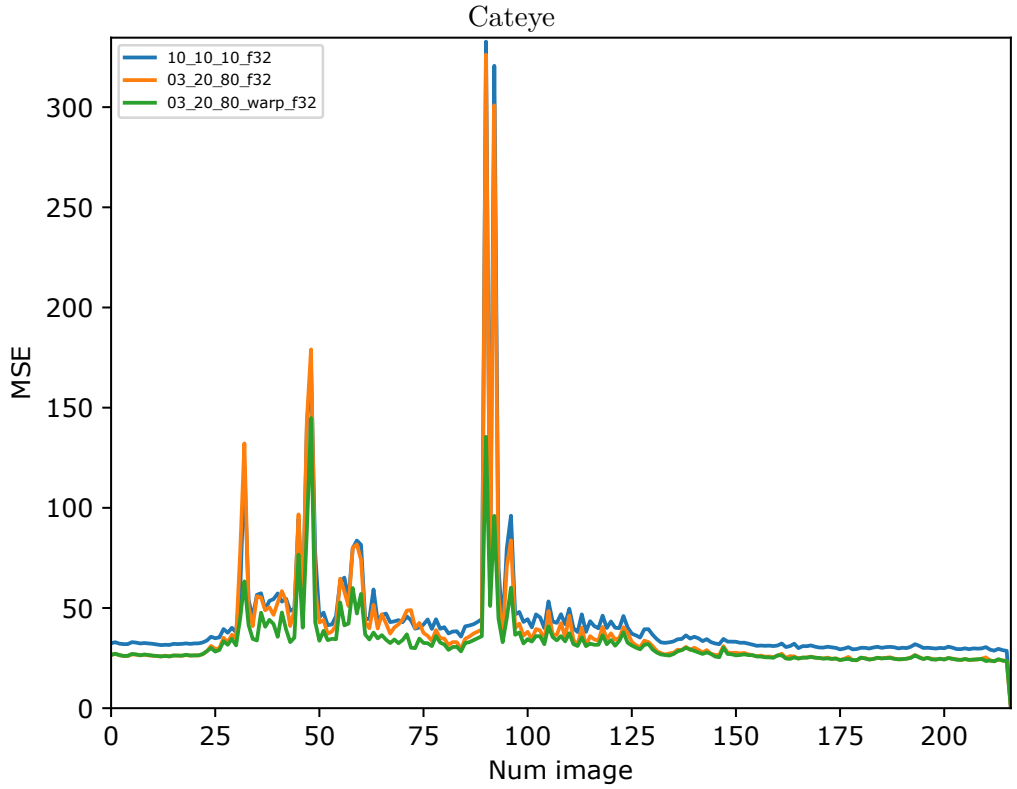
$$K = \left(3 + \frac{16}{16}\right) D + \frac{76}{16}D + \frac{70}{16}D = 3D + \frac{76+6}{16}D + \frac{70+10}{16}D \tag{4.4}$$

$$K = 3D + 20,5\frac{D}{4} + 80\frac{D}{16} \approx 3D + 20\frac{D}{4}D + 80\frac{D}{16}D \tag{4.5}$$

On considère donc finalement la configuration 03-20-80 de TV-L[1], qui effectue 80 itérations sur E_2, 20 itérations sur E_1 puis 3 itérations sur E_0. Dans cette configuration le nombre d'opérations est toujours similaire à celui de la configuration initiale 10-10-10. En revanche, un nombre important d'itérations est appliqué aux plus petites échelles. Comme nous l'avons fait remarqué en section 3.3, il peut être intéressant qualitativement, d'exécuter des *warps* supplémentaires dans le cas d'un grand nombre d'itérations. C'est pourquoi, dans le but d'améliorer encore la robustesse de l'estimation du flot nous ajoutons des calculs de *warps* dans la configuration 03-20-80 pour obtenir : 1 *warp* et 3 itérations par *warp* sur E_0, 2 *warps* et 10 itérations par *warp* sur E_1 et 4 *warps* et 20 itérations par *warp* sur E_2. Soit une configuration (1×3)-(2×10)-(4×20). Pour simplifier, on appelle cette configuration 03-20-80$_{warp}$.

La configuration 03-20-80$_{warp}$ sera assurément plus lente, puisque l'on effectue plus de *warps* pour le même nombre d'itérations. En revanche, l'impact de ces *warps* peut s'avérer relativement faible étant donné qu'ils sont effectués pour des images de taille petite par rapport à la définition d'entrée. Cette solution peut éventuellement permettre d'améliorer significativement la qualité du flot optique, en n'augmentant que légèrement le temps de traitement. Nous comparons le comportement de ces 3 configurations à la fois en termes de qualité de reconstruction et de temps de calcul.

La figure 4.1 indique l'évolution de la MSE de reconstruction pour les scènes Cateye et Coastguard pour les configurations 10-10-10, 03-20-80 et 03-20-80$_{warp}$. On constate que globalement, la configuration 03-20-80 produit de meilleurs résultats que la configuration 10-10-10. En revanche, on ne constate pas une bien meilleure stabilité sur les images les plus difficiles. La configuration 03-20-80$_{warp}$ quant à elle produit des résultats du même ordre que ceux de la configuration 03-20-80 mais avec une robustesse significativement meilleure dans les cas difficiles. Les résultats visuels confirment cette tendance avec



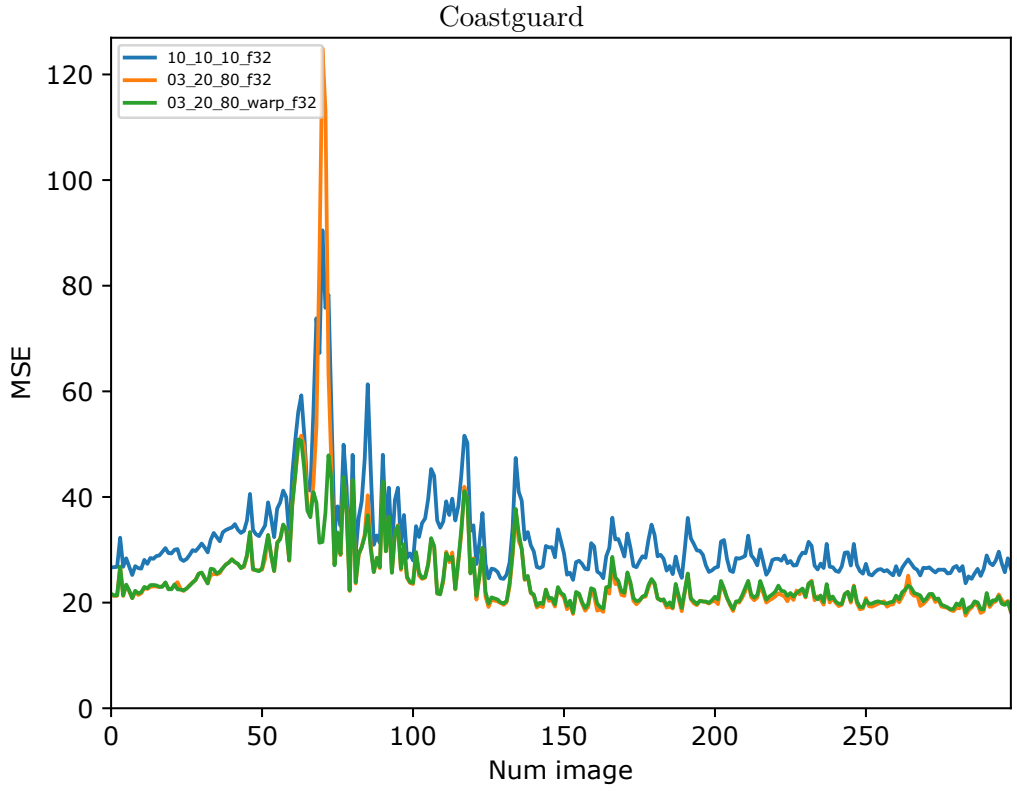


FIGURE 4.1 – Évolution de la MSE en fonction de la configuration de TV-L[1] pour les séquences Cateye et Coastguard.

notamment une réduction sensible des artefacts de reconstruction pour la version 03-20-80$_{warp}$. Un exemple de comparaison visuelle illustre ces résultats dans la figure 4.2.

FIGURE 4.2 – Comparaison de la qualité visuelle de reconstruction entre les configurations 03-20-80$_{warp}$ et 10-10-10 sur une image extraite de la séquence Cateye.

À présent, nous évaluons les temps de traitement sur les séquences Cateye et Coastguard mais également sur une version sous échantillonnée en qHD (960 × 540 pixels) de la séquence Cateye. Cela nous permet d'évaluer une séquence de taille intermédiaire. Les temps de traitement sont stables d'une image à l'autre de chaque séquence. Dans le tableau 4.1, nous présentons le temps de traitement moyen en fonction de la séquence et de la configuration de TV-L^1 considérée pour les CPU de l'AGX et de la TX2.

On constate que pour les séquences Cateye et Cateye_qHD, la configuration 03-20-80 a un temps d'exécution similaire voire plus court qu'avec la configuration 10-10-10. Cette légère accélération peut s'expliquer par un temps plus long passé dans les échelles les plus petites ce qui permet de tirer davantage partie de la mémoire cache. En revanche, pour la séquence Coastguard c'est l'effet inverse que l'on observe. La définition des images de Costguard étant très faible (352 × 288), ses sous échelles sont trop petites pour pouvoir profiter au maximum des optimisations et plus particulièrement de la parallélisation SPMD. On assiste donc à une baisse d'efficacité dans les plus petites échelles qui se fait plus ressentir avec la configuration 03-20-80 car on y passe plus de temps.

Plateforme	Séquence	Dimensions	Configuration	temps (ms)
TX2	Cateye	1920×1080	10-10-10	491
			03-20-80	421
			03-20-80$_{warp}$	497
	Cateye_qHD	960×540	10-10-10	102
			03-20-80	88
			03-20-80$_{warp}$	103
	Coastguard	352×288	10-10-10	16
			03-20-80	15
			03-20-80$_{warp}$	18
AGX$_{F32}$	Cateye	1920×1080	10-10-10	78
			03-20-80	73
			03-20-80$_{warp}$	90
	Cateye_qHD	960×540	10-10-10	20
			03-20-80	19
			03-20-80$_{warp}$	24
	Coastguard	352×288	10-10-10	4,7
			03-20-80	4,9
			03-20-80$_{warp}$	6,4
AGX$_{F16}$	Cateye	1920×1080	10-10-10	47
			03-20-80	46
			03-20-80$_{warp}$	58
	Cateye_qHD	960×540	10-10-10	13
			03-20-80	13
			03-20-80$_{warp}$	16
	Coastguard	352×288	10-10-10	3,2
			03-20-80	4,1
			03-20-80$_{warp}$	5,2

TABLE 4.1 – Temps d'exécution de TV-L[1] en fonction de la plateforme, de la séquence et de la configuration considérée.

Pour l'ensemble des mesures, la configuration 03-20-80$_{warp}$ est moins de 30% plus lente que les configurations 10-10-10 et 03-20-80. Étant donné son impact significatif sur la stabilité et la qualité du flot optique obtenu, ainsi que sa pénalisation réduite sur le temps de traitement, nous choisissons d'intégrer la configuration 03-20-80$_{warp}$ à notre chaîne de traitement finale.

Les choix qui ont été faits pour passer de la configuration 10-10-10 à la configuration 03-20-80$_{warp}$ suivent un raisonnement partiellement intuitif basé sur les différentes observations que nous avons pu faire. Il est clair que cela ne remplace pas une étude exhaustive de l'ensemble des configurations possibles. Il est tout à fait envisageable par exemple, que 80 itérations sur la plus petite échelle soit excessif. Dans ce cas, il serait possible de trouver une configuration au moins équivalente en termes de qualité mais plus rapide car contenant moins d'itérations sur E_2. De même, il peut être envisagé de réduire le nombre d'itérations dans 03-20-80$_{warp}$ pour compenser l'ajout de *warps* supplémentaires.

D'autres configurations plus rapides ou de meilleure qualité, voire les deux à la fois existent donc peut-être. Cependant la recherche de l'ensemble des configurations est un processus particulièrement long à mettre en place. En outre, nous pouvons penser que l'apport d'une nouvelle configuration sera probablement assez faible. En effet, les changements les plus importants se feront à priori au niveau du nombre d'itérations sur E_2 voir E_1. Le temps de calcul d'une ou quelques itérations sur ces échelles est faible par rapport à celui d'une itération sur E_0. L'impact d'une nouvelle configuration qualitativement équivalente mais plus rapide devrait donc être limité.

Pour ces raisons, nous faisons le choix de conserver la configuration 03-20-80$_{warp}$ et de nous concentrer, dans un premier temps, sur la mise en place d'une chaîne de traitement fonctionnelle. L'exploration de l'espace des configurations fera l'objet de futurs travaux.

4.1.2 Paramétrage du filtrage

La force du filtrage spatio-temporel a, par nature, un impact majeur sur le résultat du débruitage. Les trois paramètres σ_i, σ_d et σ_t introduits dans les équations 3.43 et 3.44 sont tout particulièrement à prendre en compte. Nous rappelons ci-dessous l'influence de chacun de ces paramètres :

— σ_i : Intensité du filtrage en fonction de la différence d'intensité entre les pixels d'un même voisinage spatial.
— σ_d : Intensité du filtrage en fonction de la distance entre les pixels d'un même voisinage spatial.
— σ_t : Intensité du filtrage temporel.

Pour déterminer les valeurs optimales de σ_i, σ_d et σ_t, nous utilisons des séquences différentes de celles étudiées jusqu'à présent. Les séquences utilisées sont issues de la base de donnée *Derf's Test Media Collection* [121]. Parmi cette base de données, 7 séquences ont été sélectionnées afin de pouvoir reproduire par la suite les conditions de la comparaison faite dans [29]. Les vidéos sont initialement en Full HD couleur (1920×1080)

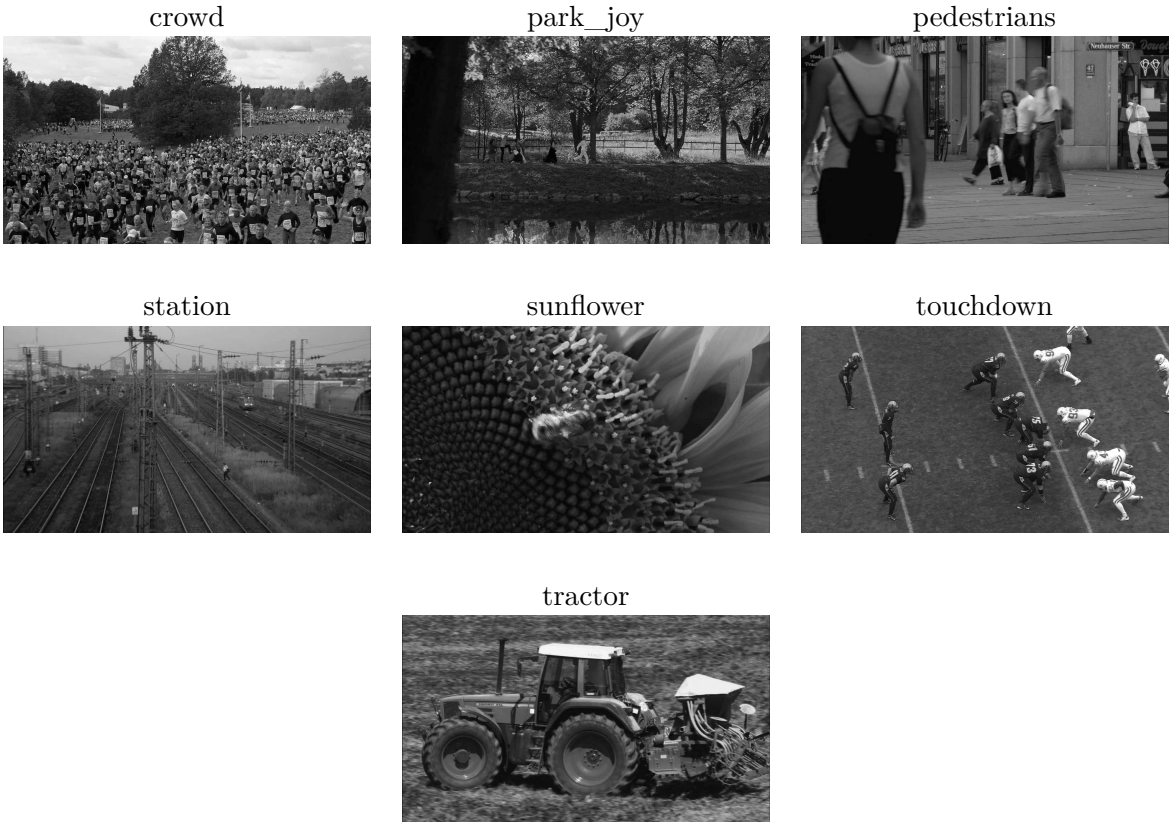


FIGURE 4.3 – Extraits des différentes séquences considérées.

et sont converties en nuances de gris qHD (960×540 pixels). Les séquences choisies sont :

- *crowd* : Foule au départ d'une course pédestre vue de haut.
- *park_joy* : Traveling sur un groupe de personnes gambadant dans un parc. De nombreux arbres passent dans le champ de vision en créant des occlusions.
- *pedestrians* : Caméra fixe dans une rue piétonne, vue à mi-hauteur d'homme.
- *station* : Zoom arrière d'une gare ferroviaire.
- *sunflower* : Bourdon butinant un tournesol.
- *touchdown* : Action de football américain.
- *tractor* : Suivi d'un tracteur dans un champ.

Un extrait de chaque séquence est donné en figure 4.3. Les séquences *crowd*, *park_joy* et *touchdown* sont particulièrement difficiles à traiter pour notre méthode en raison de très fortes discontinuités dans le flot optique, d'occlusions majeures ou encore de mouvements de très grande amplitude.

Les images sont bruitées artificiellement avec un bruit additionnel gaussien. Deux niveaux d'écart type sont utilisés : 20 et 40. Pour déterminer les valeurs optimales des

paramètres, nous exécutons la chaîne complète de débruitage RTE-VD en faisant varier σ_i, σ_d et σ_t. σ_i varie de 5 à 95 par pas de 5. σ_d varie de 0,1 à 0,9 par pas de 0,1. Et σ_t varie de 10 à 95 par pas de 5.

Les figures 4.4 et 4.5 indiquent l'évolution du PSNR (*Peak Signal to Noise Ratio*) en fonction de σ_i, σ_d et σ_t. Le PSNR est calculé de la façon suivante :

$$PSNR = 10 \times \log_{10}\left(\frac{d^2}{MSE}\right) \tag{4.6}$$

Où d correspond à la valeur maximale du signal.

Dans le cas des séquences considérées on a $d = 255$, car les images sont codées sur 8 bits. Le PSNR est exprimé en décibels (dB) et plus sa valeur est élevée mieux c'est. Le PSNR indiqué correspond au PSNR moyen de l'ensemble des images testées sur la séquence, c'est à dire au PSNR de la moyenne de MSE sur la séquence d'images. Le PSNR étant une fonction logarithmique cette valeur est différente de la moyenne des PSNR pour chaque image. L'évaluation est donnée pour les deux niveaux de bruit gaussiens considérés dans le paragraphe précédent. Dans les figures 4.4 et 4.5, plus un points est foncé, plus la valeur du PSNR est élevée. Afin de ne mettre en évidence que les meilleures valeurs de PSNR, l'échelle de couleur est saturée dans le blanc pour toute configuration inférieure à un point de PSNR au dessous de la valeur maximale atteinte pour la séquence considérée.

Il est possible d'observer une certaine tendance générale des paramètres optimaux en fonction du niveau de bruit. Pour un niveau de bruit faible (écart type de 20) les meilleurs résultats semblent être obtenus avec un filtrage spatial moyen à fort et un filtrage temporel faible. Pour un niveau de bruit plus élevé (écart type de 40), la force des filtrages spatial et temporel doit être plus importante. La plage de variation des paramètres a été fixée grâce à des tests visuels effectués au préalable. Il semble toutefois sur les figures 4.4 et 4.5, que dans certains cas la plage de valeur testée ne soit pas suffisamment grande. Cela est particulièrement vrai pour σ_d. Une exploration plus étendue est envisagée pour s'assurer d'obtenir les meilleures performances possibles. Renouveler l'exploration des paramètres est particulièrement long et c'est pourquoi cela n'a pas été encore refait. Il important de noter cependant, qu'étant donné le rôle de σ_d, sa valeur ne doit pas être trop élevée. Un filtrage spatial trop fort peut certes améliorer le PSNR mais cela risque également d'engendrer une perte de détails. Les comparaisons visuelles nous incitent à ne pas augmenter sa valeur de façon significative.

À partir de l'exploration de l'espace des paramètres, nous avons cherché à fixer au mieux ces derniers pour l'ensemble des séquences en fonction des deux niveaux de bruit considérés. Pour cela, deux stratégies ont été envisagées :

— Minimiser la Différence Maximale de PSNR entre les différentes séquences : MDMP.
— Minimiser la MSE Moyenne entre les séquences : MMM.

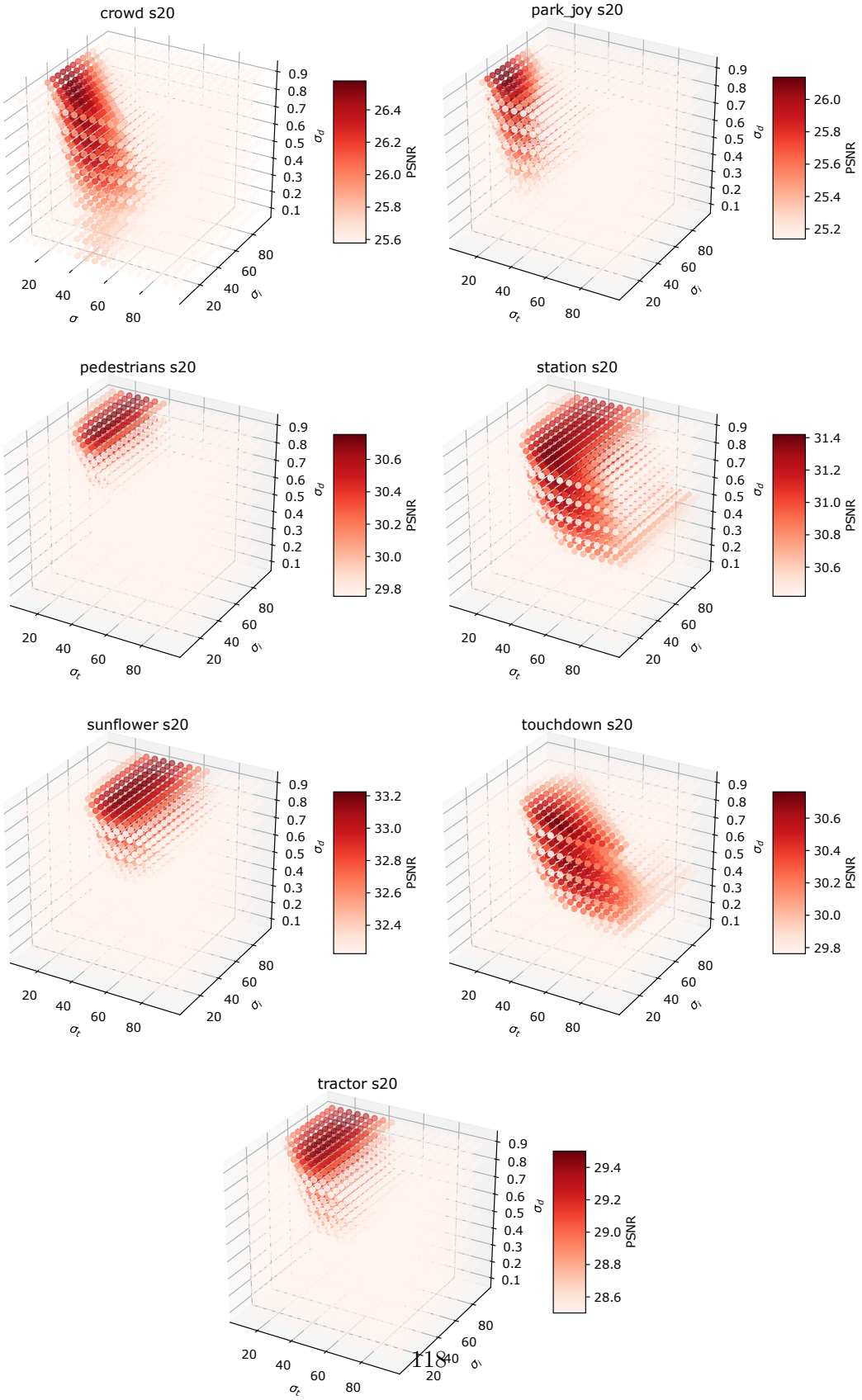


FIGURE 4.4 – Évolution du PSNR en fonction de σ_i, σ_d et σ_t pour les 7 séquences considérées et pour un bruit gaussien additionnel d'écart type égale à 20.

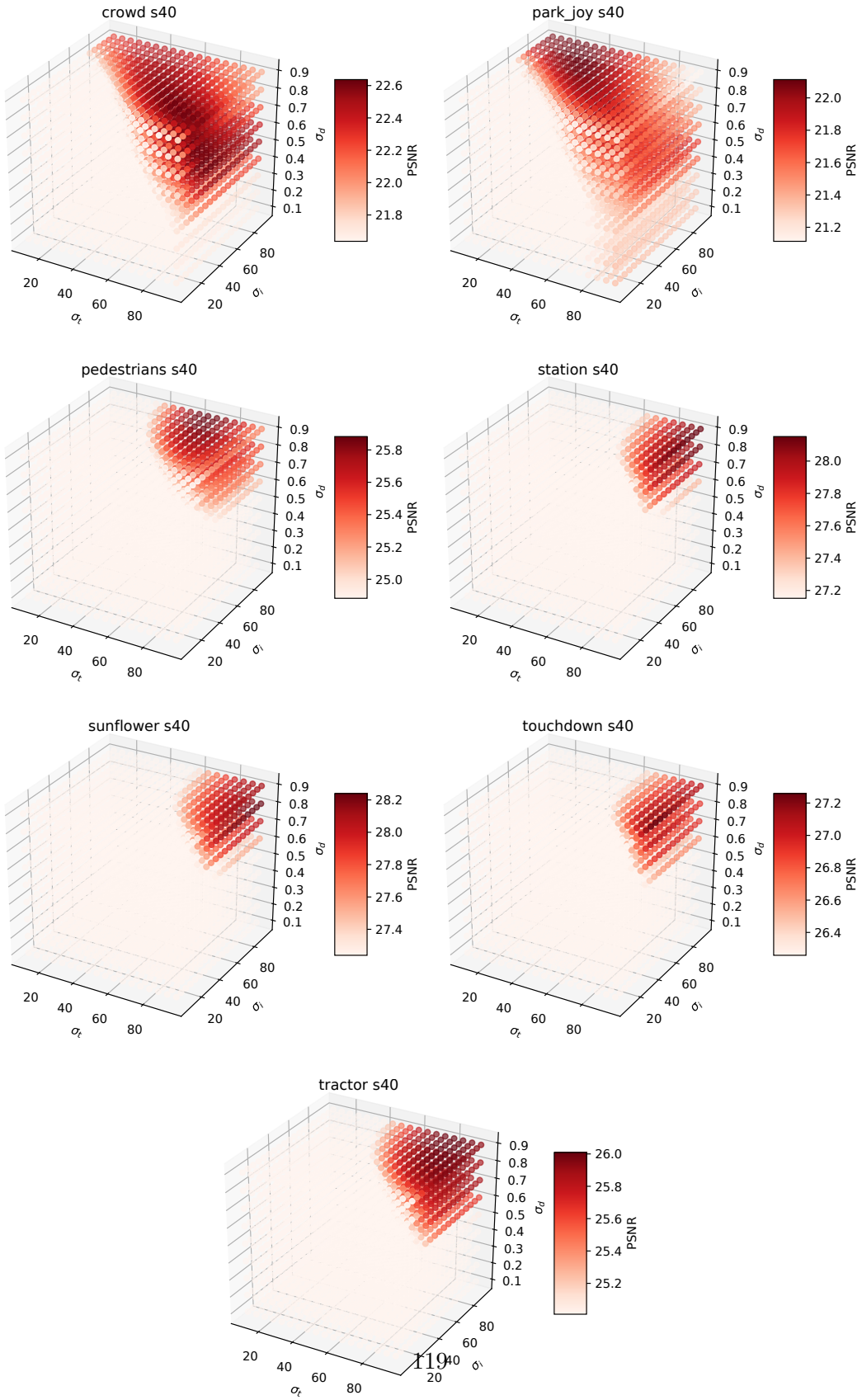


FIGURE 4.5 – Évolution du PSNR en fonction de σ_i, σ_d et σ_t pour les 7 séquences considérées et pour un bruit gaussien additionnel d'écart type égale à 40.

		PSNR		
Séquence	Écart type du bruit	SP	MDMP	MMM
crowd	20	26.60	26.35	26.38
park_joy	20	26.08	25.49	25.65
pedestrians	20	30.78	30.44	30.58
station	20	31.53	31.20	30.98
sunflower	20	33.36	32.80	32.51
touchdown	20	30.87	30.41	30.17
tractor	20	29.52	29.29	29.38
crowd	40	22.68	22.48	22.55
park_joy	40	22.09	21.69	21.64
pedestrians	40	26.03	25.70	25.72
station	40	28.22	27.80	27.76
sunflower	40	28.33	27.96	27.87
touchdown	40	27.33	26.99	27.05
tractor	40	26.12	25.92	25.99

TABLE 4.2 – PSNR obtenu pour chaque séquence en fonction de la stratégie de paramétrage appliquée. SP = *Sur-Paramétrage*, MDMP = *Minimisation de la Différence Maximale de PSNR*, MMM = *Minimisation de la MSE Moyenne*.

Pour chaque séquence, le PSNR obtenu avec les stratégies MDMP et MMM est comparé au maximum de PSNR qu'il est possible d'atteindre avec un paramétrage optimal pour la séquence en question. La configuration optimale spécifique à une séquence est appelée Sur-Paramétrage (SP).

Les deux stratégies d'optimisations des paramètres permettent d'obtenir des valeur de PSNR très proches et pour cause, les paramètres déterminés avec MDMP ou MMM sont identiques au pas de variation près. Pour un écart type de 20, les paramètres sont fixés tels que : $\sigma_t = 30$, $\sigma_i = 35$ et $\sigma_d = 0.9$. Pour un écart type de 40, les paramètres sont fixés tels que : $\sigma_t = 85$, $\sigma_i = 45$ et $\sigma_d = 0.9$. Ce sont ces paramètres qui seront utilisés pour l'évaluation des performances de RTE-VD dans la suite de ce chapitre.

4.2 Performance de RTE-VD : comparaison avec l'état de l'art

Maintenant que les différents paramètres de RTE-VD ont été établis, nous présentons dans cette section les performances de notre chaîne de traitement en termes de qualité de débruitage et de vitesse d'exécution. Afin de pouvoir évaluer ces performances, nous comparons également RTE-VD à d'autres méthodes qui font office de références dans la littérature. Les méthodes auxquelles nous nous comparons sont VBM3D [25], VBM4D [26] et STMFK [35].

VBM3D et VBM4D sont des références en termes de qualité visuelle et d'efficacité de débruitage. D'autres méthodes récentes avec un meilleur rendu existent[28, 29, 21, 30]. Cependant, ces méthodes sont en général moins connues et plus lentes que VBM3D/4D qui ne sont déjà pas des méthodes temps réel. En outre, beaucoup de ces méthodes et plus particulièrement les plus récentes, utilisent des réseaux de neurones convolutifs. Leur implémentation est donc généralement lente sur CPU mais nécessite une architecture différente de celle que nous utilisons ici, typiquement un GPU.

STMKF est une méthode de débruitage temps réel adaptée aux systèmes embarqués. Cette méthode fait office de référence en termes de débruitage temps réel. Une autre méthode temps réel présentée par Ehmann *et al* [37] semble également présenter de bons résultats. Nous n'avons cependant pas pu nous y comparer car aucun code source n'est disponible et les scènes utilisées dans l'article ne sont pas des scènes issues de base de données connues.

4.2.1 Efficacité du débruitage

Dans un premier temps nous évaluons RTE-VD en termes d'efficacité du débruitage. Pour ce faire, nous reconstituons les conditions de la comparaison effectuée dans [29]. Pour les 7 séquences issues de la *Derf's Test Media Collection*, nous comparons le PSNR moyen sur les 100 premières images en fonction de la méthode de débruitage utilisée. Le tableau 4.3 indique les PSNR obtenus avec RTE-VD, STMKF, VBM3D et VBM4D.

Bruit	Méthode	crowd	park_j	pedestr	station	sunflo	touchd	tractor	global
$\sigma = 20$	STMKF	26.25	25.59	28.34	26.66	26.97	28.87	25.37	26.70
	RTE-VD	26.38	25.65	30.58	30.98	32.51	30.17	29.38	**28.73**
	VBM3D	28.75	27.89	35.49	34.19	35.48	32.85	31.44	31.34
	VBM4D	28.43	27.11	35.91	35.00	35.97	32.73	31.65	31.11
$\sigma = 40$	STMKF	20.80	20.75	20.70	20.41	20.70	20.86	19.80	20.56
	RTE-VD	22.55	21.64	25.72	27.76	27.87	27.05	25.99	**24.85**
	VBM3D	24.81	23.78	30.65	30.62	30.88	30.21	27.82	27.43
	VBM4D	24.65	23.22	31.32	31.53	31.39	30.09	28.09	27.35

TABLE 4.3 – Comparaison du PSNR obtenu avec des méthodes de l'état de l'art et notre chaîne de traitement RTE-VD pour les 7 séquences considérées. Résultats pour 2 niveaux de bruit additionnel gaussien d'écart type σ.

Pour un bruit d'écart type de 20, le PSNR obtenu avec RTE-VD est en moyenne 2,5 dB sous celui obtenu avec VBM3D ou VBM4D. RTE-VD permet également d'obtenir un PSNR d'environ 2 dB supérieur à celui obtenu avec STMKF. Pour un bruit plus important, avec un écart type de 40, l'écart entre RTE-VD et VBM3D/4D reste aux environs de 2,5 dB tandis que l'écart avec STMKF se fait plus grand. Dans ce cas, RTE-VD permet d'obtenir un PSNR d'en moyenne 4 dB supérieur à celui de STMKF, avec un maximum de +7 dB obtenu sur les séquences *station*, *sunflofower* et *touchdown*. Ces résultats indiquent, qu'en termes d'efficacité de débruitage, RTE-VD se positionne entre

VBM3D/4D et son concurrent temps réel STMKF. La comparaison visuelle permet de confirmer ces résultats numériques.

La figure 4.6 fournit une comparaison visuelle pour un extrait de la séquence *pedestrians* et pour un écart type du bruit de 40. Dans ce cas, on peut voir que VBM3D et VBM4D produisent effectivement un débruitage de meilleure qualité que RTE-VD. En revanche, l'image produite par STMKF est plus bruitée que celle produite par RTE-VD. L'efficacité de RTE-VD par rapport à STMKF est particulièrement sensible pour les éléments en mouvement. Une perte sensible de contraste, peut en revanche être observée sur les éléments statiques de la scène comme ici sur le panneau indiquant le nom de la rue. Cela semble venir du manque de précision dans l'estimation du flot optique pour les éléments statiques. Bien que les vecteurs vitesse calculés soient de très faible magnitude, ils n'en sont par pour autant nuls. Ce phénomène est susceptible d'engendrer un effet trouble sur les zones statiques de la vidéo. Nous pensons qu'un filtrage du flot pour réduire davantage la valeur des petits vecteurs pourra résoudre ce problème.

Les observations faites pour la séquence *pedestrians* sont assez représentatives des différentes observations que l'on peut faire sur des images arrêtées, pour l'ensemble des 7 séquences de la *Derf's Test Media Collection*. Lors des comparaisons visuelles, il est également important de considérer l'observation des séquences vidéos dans leur ensemble. Dans ce cas, on observe toujours que les performances de débruitage de RTE-VD sont situées entre celles de VBM3D/4D et celles de STMKF. On peut voir également que pour certaines séquences, comme *crowd* ou *park_joy*, RTE-VD montre une baisse de ses performances. Cela est principalement dû aux mouvements de grande ampleur et aux fortes discontinuités présents dans ces scènes. En effet, les fortes discontinuités et les occlusions sont connues pour être un problème majeur lors de l'estimation du flot optique. De plus, dans cette configuration de RTE-VD, nous utilisons 3 échelles pour le calcul de flot. Les plus grands déplacements estimables sont donc de l'ordre de $2^3 - 1 = 7$ pixels.

L'ensemble des séquences vidéo étudiées est disponible en téléchargement (≈2Go) à l'adresse suivante : `http://www-soc.lip6.fr/~petreto/Download.html`.

Dans le cadre des comparaisons visuelles, nous avons également considéré d'autres séquences que celles de la *Derf's Test Media Collection*. Ces nouvelles séquences sont capturées à l'aide d'une caméra LHERITIER appelée Aether. L'idée est d'utiliser un niveau de bruit plus réaliste qu'un bruit gaussien additionnel. En effet, avec des capteurs haute performance, le bruit de lecture est si faible qu'il est possible d'obtenir des images exploitables avec uniquement quelques photons. Cependant, dans ces conditions, le bruit électrique du capteur devient visuellement significatif. Ce bruit engendré suit un modèle fixe ou répétitif. Cet aspect peut être particulièrement gênant pour le débruitage qui peut avoir tendance à accentuer le modèle du bruit. Le modèle du bruit peut se manifester sous différentes formes selon le capteur utilisé. Pour Aether, le bruit de capteur se manifeste sous la forme de ce que l'on appelle un bruit de lignes : d'une image à l'autre, le gain d'une ligne de pixels varie de façon périodique. La connaissance de ce phénomène nous permet de le prendre en compte et de l'intégrer à RTE-VD.

FIGURE 4.6 – Comparaison visuelle des résultats pour la séquence *pedestrians* (bruit : $\sigma = 40$).

Les séquences supplémentaires capturées avec Aether sont de 2 types différents. Dans le premier cas, il s'agit d'une séquence prise de jour à laquelle a été ajouté un bruit proche de celui visible sur Aether grâce à un générateur de bruit déjà existant au sein de LHERITIER. Le générateur permet d'estimer l'importance du bruit à partir d'une luminosité générale choisie. Ici, nous avons considéré des éclairages ambiants de 150, 50 et 15 millilux. À titre de comparaison, on considère que la luminosité moyenne pendant une nuit de pleine lune est de 500 millilux. Cette approche permet de pouvoir comparer les résultats réalistes obtenus avec une vidéo d'origine non bruitée. Dans le second cas nous utilisons des vidéos réellement bruitées. Outre le fait de pouvoir obtenir ce type de résultats en se plaçant volontairement dans des conditions d'enregistrement défavorables, il est possible de recréer, de jour, des conditions de nuit grâce à l'utilisation de filtres de densité neutre (ND) et à la réduction du temps de pose. L'utilisation de cette dernière méthode permet d'obtenir des images réellement bruitées tout en gardant un temps de pose relativement court. Cela permet d'éviter le flou de bougé généralement présent dans les prises de vues de nuit avec des temps de pose longs. Décorréler le résultat du filtrage du flou de bougé permet d'éviter d'attribuer un effet de flou au filtrage qui serait en réalité causé par une pose longue.

La séquence prise de jour non bruitée est appelée *simu_xxml* où *xx* est le nombre de millilux choisis pour générer le bruit de la séquence. Les séquences capturées à l'aide de filtres ND sont appelées *fausse_nuit0* et *fausse_nuit1*. Ces deux séquences représentent une action similaire mais *fausse_nuit0* est capturée dans des conditions plus difficiles que *fausse_nuit1*. L'ensemble de ces séquences est également disponible en téléchargement dans le lien fourni précédemment.

La figure 4.7 illustre les résultats obtenus pour la séquence *simu_50ml*. Le débruitage de RTE-VD apparaît encore une fois meilleur que celui de STMKF mais plus faible que celui de VBM3D/4D. La visualisation des vidéos permet cependant de mettre en avant l'influence du bruit de lignes. Ce dernier entraîne un effet d'ondulation particulièrement pénalisant pour VBM3D et surtout VBM4D. Visuellement, cet effet d'ondulation ressemble à ce que l'on pourrait observer en cas de perturbations atmosphériques. La prise en compte de ce bruit de capteur dans RTE-VD permet de limiter sont impact. La figure 4.8 présente un extrait des résultats obtenus dans le cas d'un bruit réel pour la séquence *fausse_nuit1*. Les observations faites avec la génération de bruit sont ici confirmées dans le cas d'un bruit réel. L'effet de turbulence engendré par VBM3D et VBM4D entraîne une perte de détails au niveau de l'individu qui est moins pénalisant avec RTE-VD.

En résumé, les résultats montrent que RTE-VD permet un débruitage significativement plus efficace que son concurrent temps réel STMKF mais reste en dessous des références de qualité que sont VBM3D et VBM4D. STMKF s'avère particulièrement inefficace dans les situations très bruités. Cela est dû à la nature de l'algorithme qui tend à diminuer la force du filtrage temporel en cas de variation trop importante d'une image à l'autre. Les méthodes VBM3D et VBM4D fournissent un débruitage nettement plus efficace que RTE-VD mais au prix d'un temps de calcul grandement supérieur.

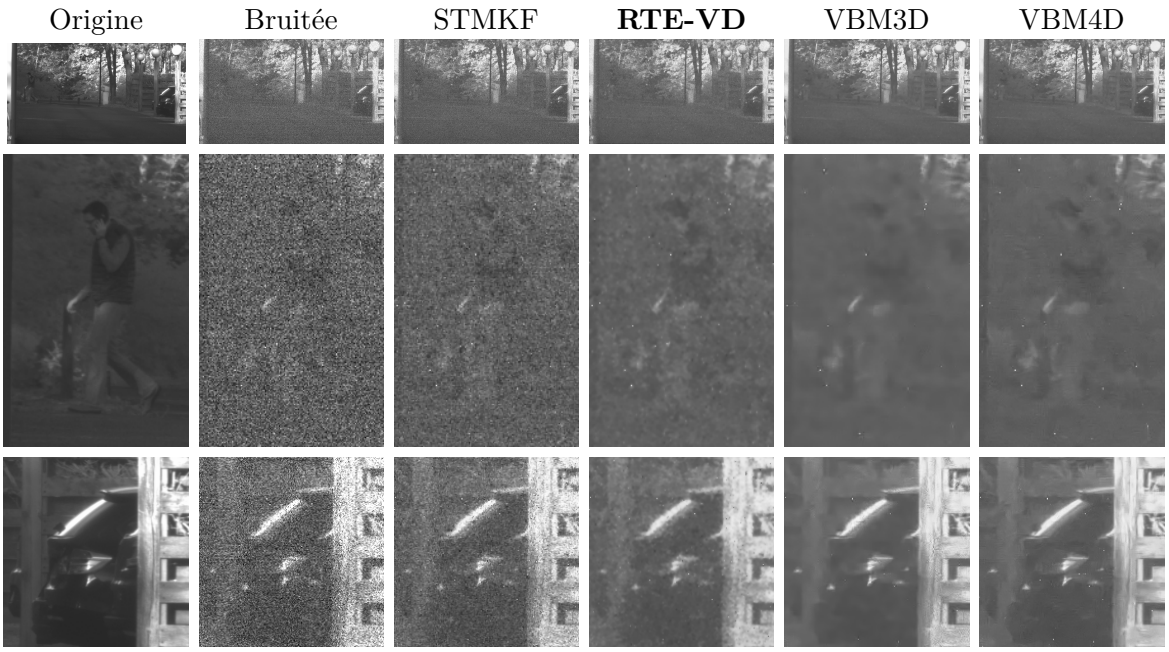


FIGURE 4.7 – Comparaison visuelle des résultats pour la séquence *simu_50ml*.

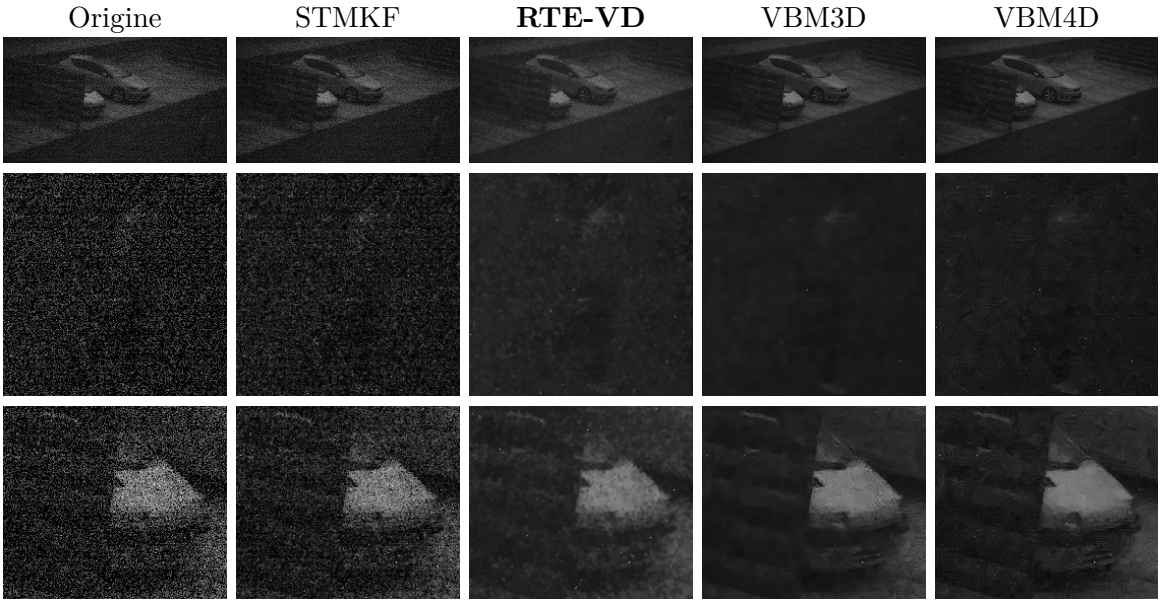


FIGURE 4.8 – Comparaison visuelle des résultats pour la séquence *fausse_nuit1*.

Afin de mieux évaluer le positionnement de RTE-VD par rapport à ces méthodes, nous comparons dans la section suivante, les différents temps de traitement nécessaires au débruitage.

4.2.2 Comparaison du temps de traitement

L'objectif de ces travaux étant de développer une chaîne de débruitage temps réel, le temps de traitement est un aspect important pour pouvoir positionner RTE-VD par rapport aux autres méthodes de l'état de l'art. Dans cette section, nous analysons le temps de traitement total de RTE-VD et le comparons à ceux de STMKF, VBM3D et VBM4D pour une même architecture. Deux processeurs différents sont utilisés dans cette section. Afin de pouvoir exécuter VBM3D et VBM4D dans des temps raisonnables, nous utilisons un processeur Intel Xeon Silver 4114 2×10C/20T cadencé à 2,20GHz. Pour évaluer et STMKF et RTE-VD sur une même plateforme embarquée, nous utilisons également le CPU de l'AGX composé de 8 cœurs ARM Carmels cadencés à 2,27GHz.

Les codes sources utilisés pour la comparaison sont ceux fournis par les auteurs sur leur page web respective. S'il est possible d'affirmer que l'implémentation de STMKF a fait l'objet d'une optimisation pour minimiser le temps de calcul, il est difficile de définir le niveau d'optimisation apporté aux implémentations de VBM3D et VBM4D. En effet, les auteurs fournissent uniquement une "boîte noire" inclue dans un environnement Matlab. Il nous est impossible de dire si l'implémentation fournie fait appel à des fonctions optimisées ou non. Pour cette raison, nous limitons à 4 fils d'exécution, le parallélisme de STMKF et RTE-VD sur le Xeon. Les temps de traitement de chacune des méthodes pour des images qHD sont indiqués dans le tableau 4.4.

Algorithme	Temps (s)	Plateforme
STMKF	0.004 5	Xeon
RTE-VD	0.009 7	Xeon
VBM3D	2.0	Xeon
VBM4D	45	Xeon
STMKF	0.015	AGX
RTE-VD	0.033	AGX

Table 4.4 – Temps de traitement en fonction de la méthode et de la plateforme utilisées pour des images qHD (960×540 pixels).

Pour une même plateforme, on constate que notre chaîne de traitement RTE-VD est 2,2 fois plus lente que STMKF. Ce rapport de vitesse entre les deux méthodes se constate à la fois sur le Xeon et sur l'AGX. Par rapport aux références de qualité que sont VBM3D et VBM4D, notre méthode est respectivement plus de 200 fois et plus de 4600 fois plus rapide. Bien que plus lent que STMKF, RTE-VD parvient à traiter en temps réel des vidéos au format qHD avec le CPU embarqué de l'AGX. Dans ces dimensions,

le temps de traitement est de 33 millisecondes par image, soit une cadence de 30 images par seconde.

La décomposition du temps de calcul du tableau 4.5 permet de se rendre compte de l'impact de chaque maillon de la chaîne de traitement sur le temps de traitement. Les mesures confirment que, comme annoncé dans les chapitres précédents, l'estimation du flot optique est la partie la plus coûteuse de RTE-VD. Le calcul de TV-L^1 prend 87% du temps total du calcul et 97% si on lui ajoute l'interpolation du recalage temporel qui en découle.

Algorithme	StabLK	TV-L^1	Recalage	Filtrage	Total
Temps (ms)	0,6	**28,7**	3,1	0,4	32,8
Proportion (%)	1,8	**87,2**	9,5	1,2	100

TABLE 4.5 – Décomposition du temps de traitement de RTE-VD pour des vidéos qHD.

4.2.3 Comparaison : synthèse

Dans cette section nous avons opposé notre chaîne de traitement RTE-VD à 3 autres méthodes de l'état de l'art. STMKF est une méthode de débruitage de vidéo temps réel adaptée aux systèmes embarqués. VBM3D et VBM4D sont des références en termes de qualité de débruitage mais ont un coup de traitement bien supérieur. Les résultats montrent que RTE-VD permet de proposer un débruitage significativement meilleur par rapport à STMKF, en particulier pour les forts niveaux de bruit pour un temps de traitement 2,2 fois supérieur. Si visuellement RTE-VD ne parvient pas à concurrencer VBM3D et VBM4D, son temps de traitement est bien inférieur puisque RTE-VD est respectivement 200 et 4600 fois plus rapide. Malgré un temps de traitement plus important que celui de STMKF, il nous est possible de traiter des vidéos au format qHD à une fréquence de 30 images par seconde avec le CPU de la plateforme embarquée AGX.

Contrairement à STMKF et aux autres algorithmes de débruitage vidéo temps réel [37], RTE-VD est conçu pour rester efficace avec des niveaux de bruit importants. Aux vues des résultats nous pensons que RTE-VD se positionne de façon intéressante sur le spectre des différentes méthodes de débruitage existantes. RTE-VD nous permet de proposer un nouveau compromis "vitesse/précision" qui nous paraît pertinent pour le type d'application visée.

4.3 Temps de calcul et consommation énergétique

Notre objectif est de pouvoir intégrer RTE-VD dans des systèmes embarqués. Il est donc pertinent de s'interroger sur la consommation énergétique de notre méthode. Dans cette section, nous étudions le rapport entre la vitesse d'exécution et l'énergie consommée par RTE-VD pour différentes plateformes de systèmes embarqués. L'objectif ici, est de

donner un ordre d'idée des performances que l'on peut obtenir en exécutant RTE-VD sur des plateformes embarquées de différente puissance.

Plateforme	Techno	CPU	Fmax (GHz)	Consommation Idle (W)
TX2	16 nm	4×A57 + 2×Denver2	2.00	2.0
AGX	12 nm	8×Carmel	2.27	6.3
NANO	12 nm	4×A57	1.43	1.2

TABLE 4.6 – Caractéristiques techniques des plateformes utilisées.

Nous exécutons RTE-VD à différentes fréquences sur 3 plateformes embarquées distinctes. Ces plateformes sont la TX2, l'AGX et la NANO, dont les caractéristiques sont rappelées dans le tableau 4.6. Encore une fois, nous ne considérons pas les GPU dans cette étude. Le protocole expérimental pour les mesures de temps et d'énergie est le même que celui décrit dans la section 3.3.5.

Les différentes fréquences utilisées ont été sélectionnées parmi les fréquences disponibles pour chaque plateforme et la fréquence de la mémoire externe est toujours fixée à sont maximum. Pour l'AGX et la NANO, nous avons exécuté une version parallèle de RTE-VD utilisant l'ensemble des cœurs du processeur : respectivement 8 et 4. Le CPU de la TX2 ayant une architecture hétérogène, 3 configurations différentes de RTE-VD y ont été testées :

— Version parallélisée uniquement sur les 2 cœurs Denver.
— Version parallélisée uniquement sur les 4 cœurs A57.
— Version parallélisée sur l'ensemble des cœurs.

Nous définissons trois types de consommation énergétique :
— E_0 : l'énergie consommée au repos par le système. Dans notre cas, il s'agit de la consommation du système lorsque ce dernier est allumé et que seul l'autogestion du système par le système d'exploitation fonctionne.
— E_c : le surplus d'énergie consommée lors de l'exécution du traitement.
— E_t : l'énergie totale consommée lors l'exécution du traitement. On a : $E_t = E_0 + E_c$.

La figure 4.9 indique le temps de traitement en nanosecondes par pixel (ns/pix) et E_t en nanojoules par pixel (nj/pix) en fonction de la fréquence et de la plateforme utilisées.

Les points de fonctionnement *AGX_8C_xxGHz* correspondent aux exécutions de RTE-VD parallélisées sur les 8 cœurs Carmel de l'AGX à la fréquence *xx*. Les points de fonctionnement *TX2_6C_xxGHz* correspondent aux exécutions de RTE-VD parallélisées sur les 6 cœurs A57 et Denver de la TX2 à la fréquence *xx*. Les points de fonctionnement *TX2_4C_xxGHz* correspondent aux exécutions de RTE-VD parallélisées sur les 4 cœurs A57 de la TX2 à la fréquence *xx*. Les points de fonctionnement *TX2_2C_xxGHz*

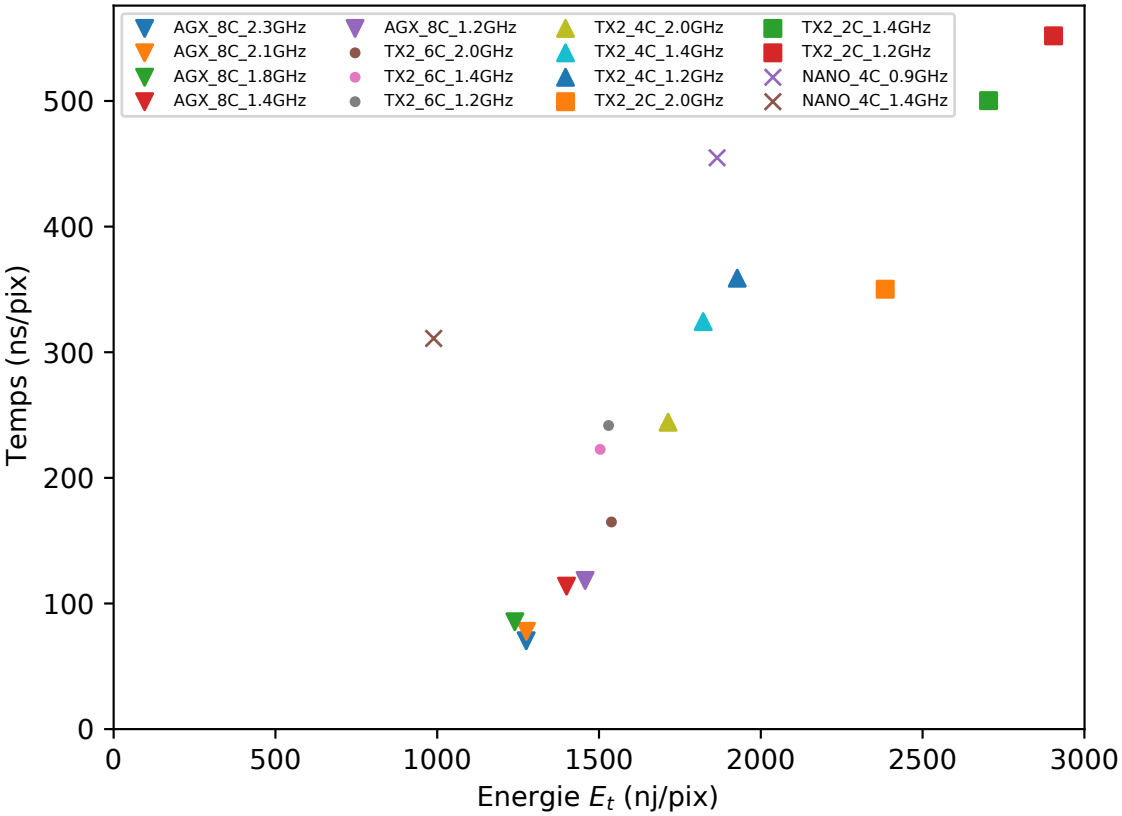


FIGURE 4.9 – Temps de traitement en ns par pixel et énergie dynamique consommée en nJ par pixel pour l'exécution de RTE-VD en fonction de la fréquence et de la plateforme utilisées.

correspondent aux exécutions de RTE-VD parallélisées sur les 2 cœurs Denver de la TX2 à la fréquence *xx*. Les points de fonctionnement *NANO_4C_xxGHz* correspondent aux exécutions de RTE-VD parallélisées sur les 4 cœurs A57 de la NANO à la fréquence *xx*.

Cette représentation permet surtout de comparer les différentes plateformes entre elles. On constate que l'AGX apparaît nettement comme la plateforme la plus rapide avec un temps de traitement minimal de 70 ns/pix pour une consommation de 1275 nj/pix. La NANO quant à elle, ressort comme étant la plateforme la plus économe en énergie avec une consommation minimale de 989 nj/pix pour un temps de traitement de 311 ns/pix.

La TX2 apparaît comme globalement moins efficace que les deux autres plateformes avec un rapport vitesse/consommation moins bon. Cela s'explique par une finesse de gravure plus grande : 16 nm contre 12 nm. Pour le prouver, nous considérerons les points de fonctionnement *TX2_4C_1.4GHz* et *NANO_4C_1.4GHz*. Ces deux configurations sont en effet essentiellement différenciées par la finesse de gravure du processeur. Dans ces deux cas, RTE-VD est exécuté sur 4 cœurs A57 cadencés à 1,4GHz. Les temps de traitement associés à chacun des points sont donc logiquement similaires mais la NANO consomme 1,7 fois moins d'énergie que la TX2.

Étant données ces observations et le fait que la TX2 peut être cadencée plus haut que la NANO, il serait particulièrement intéressant d'évaluer une architecture proche de celle de la TX2 mais gravée avec la même finesse. On peut penser que cela viendrait combler l'écart important existant entre la NANO et l'AGX. La sortie prochaine de l'AGX NX annoncée par Nvidia devrait permettre de proposer des performances similaires voire plus rapides que celles de la TX2 avec une consommation plus faible. Nous prévoyons donc une future comparaison intégrant cette nouvelle plateforme lorsque celle-ci sera disponible à la vente.

E_0 étant un biais constant pour une plateforme donnée, l'évaluation de E_c permet d'évaluer l'efficacité des différentes fréquences au sein d'une même plateforme. La figure 4.10 indique le temps de traitement en nanosecondes par pixel (ns/pix) et E_c en nanojoules par pixel (nj/pix) en fonction de la fréquence et de la plateforme utilisée. Les résultats montrent qu'il est possible de déterminer une frontière d'efficience parmi les différentes fréquences testées. Pour chaque plateforme, il n'est pas possible de trouver un point de fonctionnement à la fois plus rapide et avec une consommation plus faible. En fonction de la plateforme visée, un compromis plus ou moins important devra être fait entre consommation maximale et taille des images traitées. La diminution de la fréquence permet généralement de réduire la consommation, au prix d'un temps de calcul plus élevé. Cela n'est plus vrai pour les fréquences les plus faibles. En dessous d'un certain point, la réduction de la consommation instantanée n'est plus significative par rapport à la réduction du temps de calcul pour que l'énergie par pixel soit réduite. Les fréquences les plus faibles sont donc à éviter étant donné qu'elles sont à la fois plus lentes et plus énergivores que d'autres fréquences plus élevées.

Cette représentation permet essentiellement de déterminer une fréquence de fonctionnement optimale en fonction d'une plateforme donnée et des contraintes de l'application. Par exemple : nous souhaitons exécuter RTE-VD sur l'AGX pour des images de 1000×1000 pixels, à une cadence de 5 images par secondes (soit 200 ms par image). Toutes le fréquences testées permettent d'effectuer le traitement dans le temps imparti, en revanche le point de fonctionnement à 1,4 GHz est celui qui entraîne la plus faible énergie consommée. Dans ce cas nous choisirons donc d'exécuter notre application à une fréquence de 1,4 GHz.

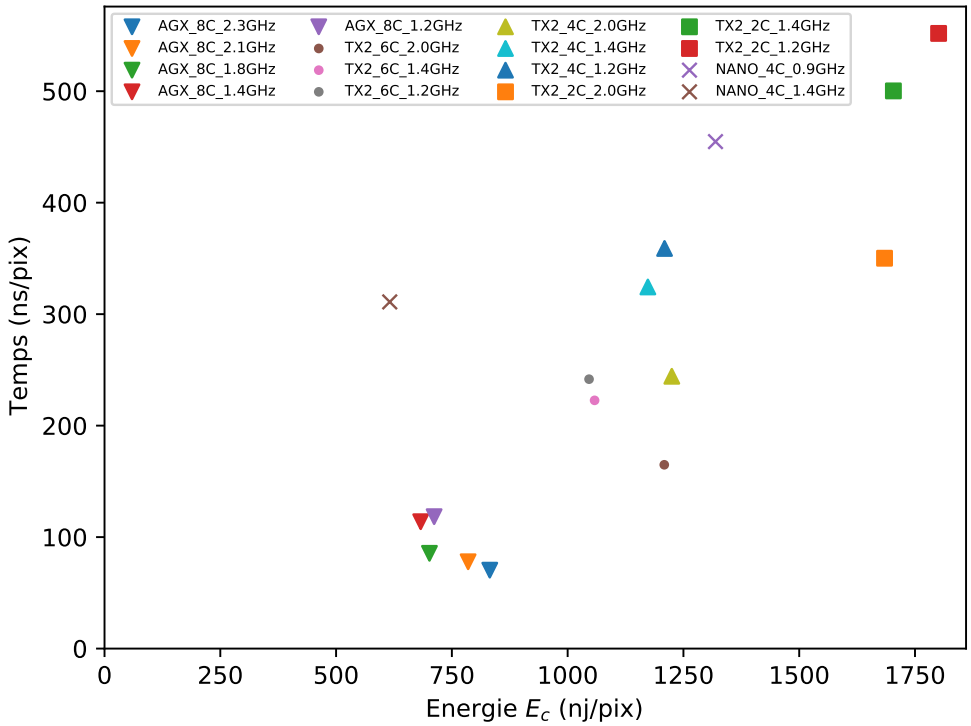


FIGURE 4.10 – Temps de traitement en ns par pixel et énergie dynamique consommée en nJ par pixel pour l'exécution de RTE-VD en fonction de la fréquence et de la plateforme utilisée.

4.4 Projet VIRTANS

Au cours de ces travaux de thèse, nous avons initié le projet VIRTA et son premier prototype le VIRTANS. Dans cette section nous les présentons succinctement.

Le projet VIRTA, pour *VIdeo Real-Time Algorithm* a pour objectif de développer une nouvelle plateforme LHERITIER qui permet d'intégrer des traitements de vidéos lourds sur des systèmes embarqués. Le but est de proposer une solution pouvant être adaptée à différents traitements comme le débruitage ou la réjection de perturbations atmosphériques. Dans un premier temps, nous nous somme concentrés sur le débruitage et avons mis au point un premier prototype appelé VIRTANS (pour *VIdeo Real-Time Algorithm : Noise Suppression*). Un aperçu de VIRTANS est disponible en figure 4.11.

FIGURE 4.11 – Prototype VIRTANS.

VIRTANS est basé sur une architecture Jetson TX2i qui est la version industrielle de la TX2 classique. La différence majeure entre ces deux plateformes étant que le bon fonctionnement de la TX2i est garanti pour une plage de température bien plus large. Une carte mère différente de celle de la plateforme de développement est utilisée afin de réduire significativement le facteur de forme [146]. Une carte d'acquisition de flux video SDI est également utilisée [147]. Si pour l'heure, des composants du marché sont utilisés, un développement en interne de cartes sur mesure est envisagé. Une structure spécifique a été développée. Cette dernière permet une bonne dissipation de la chaleur tout en conférant un format facilement intégrable.

VIRTANS peut être connecté aux caméras LHERITIER via un connecteur SDI et il est possible de visualiser le résultat du débruitage via une sortie HDMI. La caméra peut transmettre des informations à VIRTANS par Ethernet. Le débruitage est effectué en temps réel par RTE-VD et retransmis en direct. Pour l'heure, VIRTANS est encore à un stade peu avancé de son développement. Seul les 4 cœurs A57 du CPU sont utilisés pour le débruitage et les vidéos traitées sont seulement de définition 480×270 pixels. Dans sa configuration de démonstration, VIRTANS dispose côte à côte les images traitées et non traitées afin de pouvoir évaluer l'efficacité de la méthode en direct.

4.5 Synthèse des résultats

Dans ce chapitre nous avons présenté les différents résultats obtenus pour RTE-VD. Nous avons comparé notre méthode à deux méthodes non temps réel, VBM3D et VBM4D, et une autre temps réel STMKF. Les résultats ont montré que RTE-VD permet un débruitage significativement meilleur que celui produit par STMKF. Par rapport à STMKF et aux autres méthodes temps réel, RTE-VD se différencie par sa capacité à continuer à fournir un débruitage sensible pour de forts niveaux de bruit. Les mesures de PSNR ont montré que RET-VD peut fournir des résultats jusque 7dB au dessus de ceux de STMKF pour un temps de traitement 2,2 fois plus long. Par rapport à VBM3D et VBM4D, RTE-VD ne parvient pas à proposer un débruitage aussi efficace mais permet un traitement respectivement 200 et 4600 fois plus rapide que ces méthodes.

Pour ces raisons, nous pensons que RTE-VD amène un nouveau positionnement pertinent dans le rapport vitesse/précision parmi les algorithmes de débruitage existants.

Dans un second temps, nous avons également présenté une étude du rapport entre le temps de traitement et la consommation énergétique de notre chaîne de traitement pour différentes architectures embarquées. Cette étude nous permet d'orienter nos choix de plateforme en fonction des différentes contraintes qu'il est possible de rencontrer lors du développement d'un système complet. Il nous est désormais possible de dimensionner un système en fonction de la définition d'images souhaitée ou, à l'inverse, de définir la définition maximale de traitement en fonction des contraintes de consommation imposées.

Enfin, nous avons présenté VIRTANS, un prototype de plateforme embarquée pour le débruitage de vidéos en direct. VIRTANS permet d'intégrer RTE-VD avec un encombrement et une consommation limités. Ce prototype est encore à un stade peu avancé de son développement et sera amené à évoluer en même temps que RTE-VD.

Chapitre 5

Conclusion et travaux futurs

5.1 Synthèse

Ce travail de thèse a permis la mise en place d'une chaîne de débruitage vidéo temps réel sur des systèmes embarqués. Le temps réel correspond ici à la capacité d'effectuer le traitement à une cadence minimum de 25 images par seconde. Dans un premier temps, nous nous limitons à des processeurs embarqués de type CPU avec une consommation maximale de l'ordre de 30 Watts. La méthode proposée doit rester efficace pour de forts niveaux de bruits. Étant donnée la forte complexité de la majorité des méthodes de la littérature, il nous faut trouver une nouvelle méthode pour répondre aux contraintes posées.

Nous proposons pour cela, la chaîne de traitement RTE-VD (*Real-Time Embedded Video Denoising*). Cette chaîne de traitement est composée de 3 étapes successives. Tout d'abord, une stabilisation globale de la vidéo permet de compenser le mouvement de la caméra. La deuxième étape est la plus critique en termes de temps de traitement. Il s'agit du recalage temporel de chaque pixel par calcul de flot optique dense. Enfin, la troisième étape est l'application d'un filtre spatio-temporel.

Afin de tenir les contraintes du temps réel, de nombreuses optimisations ont été appliquées à chacune des étapes. Des optimisations architecturales permettant de profiter des parallélismes de données et de tâches ont été mises en place pour l'ensemble de la chaîne de traitement. Des transformations algorithmiques ont également été utilisées pour permettre une meilleure utilisation de la mémoire lors de l'enchaînement de plusieurs opérateurs. La fusion et le pipeline d'opérateurs, associés à l'allocation modulaire de certains tableaux intermédiaires, permettent d'améliorer la localité et l'empreinte mémoire. D'autres optimisations plus spécifiques à chaque étape ont également été apportées.

Pour la stabilisation, nous utilisons une méthode appelée StabLK, similaire au calcul de flot optique de Lucas & Kanade. Ici toutefois, nous utilisons une approche permettant d'estimer le mouvement global entre deux images consécutives. La principale optimisation spécifique à cette étape porte sur la convolution par une porte des images d'entrée.

Pour ce faire, nous utilisons des images intégrales. Nous proposons une façon inédite, à notre connaissance, de paralléliser le calcul de la convolution par images intégrales partielles. Contrairement aux autres méthodes proposées dans la littérature, notre méthode évite les barrières de synchronisations pour la construction de l'image intégrale complète. L'utilisation seule de l'image intégrale permet une accélération de $\times 31$ du calcul de StabLK. Sa parallélisation multicœur est efficace à plus de 80% sur 4 cœurs et de 75% sur 8 cœurs. L'ensemble des optimisations apportent une accélération de $\times 1120$ sur 8 cœurs. Sur le CPU de la plateforme AGX, les vidéos au format FullHD peuvent être stabilisées en 1,6 ms par image.

Pour le recalage temporel par calcul de flot optique dense, nous utilisons l'algorithme TV-L^1. La spécificité de cette étape est son caractère itératif. La propagation des dépendances de données au fur et à mesure des itérations rend difficile la fusion des itérations et la réduction du nombre d'accès mémoire. C'est pourquoi, nous proposons un pipeline des itérations ligne par ligne afin d'améliorer la localité mémoire. À parallélisme SPMD équivalent, les transformations effectuées accélèrent le calcul des itérations d'un facteur 11 sur le CPU de l'AGX. À notre connaissance, notre implémentation est la plus rapide existante sur CPU. En version multi-échelle, notre implémentation est jusqu'à 5 fois plus rapide que celle proposée par OpenCV. Pour l'une des plateformes testées, l'AGX, notre implémentation CPU est même susceptible d'être compétitive par rapport au GPU en comparant le ratio vitesse/énergie.

Pour le filtrage spatio-temporel, nous appliquons un filtre bilatéral avec une décorrélation de la dimension temporelle. Le filtre bilatéral permet de respecter les transitions fortes dans la vidéo. Cet aspect permet d'être plus robuste aux changements brusques du contenu dans une scène et aux éventuels artefacts engendrés par le recalage temporel. Dans le but d'accélérer le temps de traitement, nous utilisons une approximation séparée du filtre original. En monocœur, les différentes optimisations apportées permettent une accélération d'environ $\times 20$, tandis que la parallélisation multicœur apporte une accélération supplémentaire de $\times 3,9$ sur 4 cœurs et de $\times 7,1$ sur 8 cœurs.

Afin de pouvoir évaluer la pertinence de notre méthode, nous la comparons à trois autres méthodes de la littérature : VBM3D, VBM4D et STMKF. VBM3D et VBM4D sont des références connues en termes de qualité visuelle. Ces deux méthodes permettent un débruitage efficace des vidéos au prix d'un long temps de calcul. La troisième méthode, STMKF, est une référence récente dans le spectre des méthodes de débruitage vidéo temps réel. Grâce à une faible complexité, STMKF permet un traitement rapide des images mais est limité dans ses applications. La méthode n'est pas adaptée à des mouvements de grande amplitude ou de forts niveaux de bruit.

Les résultats numériques de PSNR ainsi que les comparaisons visuelles, montrent qu'en termes de qualité de débruitage, RTE-VD surpasse STMKF mais reste inférieur à VBM3D/4D. Sur les séquences vidéo testées, RTE-VD permet d'obtenir un PSNR d'en moyenne 2 décibels supérieur à celui obtenu avec STMKF pour un bruit additionnel gaussien avec un écart type de 20. Cet écart se creuse pour un niveau de bruit supérieur et le PSNR obtenu avec RTE-VD est en moyenne 4 décibels au dessus de celui obtenu

avec STMKF pour un écart type de 40. Sur les scènes pour lesquelles l'estimation du flot optique est plus facile, on observe même des gains de +7 dB par rapport à STMKF. En revanche, que ce soit pour un écart type de 20 ou de 40, le PSNR obtenu avec RTE-VD est en moyenne 2,5 décibels inférieur à celui obtenu avec VBM3D ou VBM4D. Ces résultats montrent que, par rapport aux autres méthodes temps réel, RTE-VD reste efficace même avec de forts niveaux de bruit. Les méthodes de traitement à posteriori demeurent visuellement meilleures.

Les quatre méthodes considérées sont également comparées en termes de vitesse d'exécution. Dans ce cas, les résultats s'inversent par rapport aux résultat qualitatifs. Sur une même architecture, STMKF demeure la méthode la plus rapide et est 2,2 fois plus rapide que notre solution RTE-VD. En revanche, RTE-VD apporte un gain de temps conséquent par rapport à VBM3D et VBM4D en étant respectivement 200 et 4600 fois plus rapide. Notre implémentation optimisée de RTE-VD permet de traiter, en temps réel, des vidéos au format qHD (960×540 pixels) avec un CPU embarqué consommant environ 30 Watts (AGX).

Les résultats montrent donc, qu'en termes de qualité, RTE-VD est plus proche de VBM3D/4D que de STMKF et, qu'en termes de temps d'exécution, RTE-VD se rapproche plus de STMKF. Cela nous permet de penser que RTE-VD apporte un positionnement particulièrement pertinent par rapport aux autres chaînes de débruitage vidéo présentes dans la littérature. Le spectre des méthodes de débruitage est en effet assez fourni à ses extrémités mais vide entre les deux. De nombreuses méthodes très qualitatives mais lentes ou très rapides mais peu qualitatives existent. Entre ces deux approches, la littérature ne propose que très peu de compromis intermédiaires entre la qualité et la vitesse. Le besoin de systèmes de vision embarquée toujours plus performants allant grandissant, il est cependant nécessaire de chercher de nouvelles approches permettant d'améliorer la qualité visuelle des implémentations embarquées. Les travaux sur FastDVDnet, présentés dans [31] en 2019, montrent qu'il s'agit effectivement d'un enjeu actuel important. Nous pensons que RTE-VD amène un compromis "qualité/vitesse" intéressant pour ce genre d'application.

Dans ces travaux, nous avons également évalué les performances de notre chaîne de traitement sur différentes architectures embarquées et plus particulièrement sur les plateformes Nvidia Jetson TX2, AGX et NANO. Cette étude permet d'aider au dimensionnement d'un système dans le cadre du développement d'un système utilisant RTE-VD. Parmi les plateformes testées, l'AGX apparaît comme la plateforme la plus rapide tandis que la NANO est la plus économe en énergie. La TX2 permet de combler l'écart de vitesse entre ces deux plateformes mais est en retrait vis-à-vis de la consommation. Cette perte d'efficacité s'explique par la différence de finesse de gravure entre ces plateformes. La TX2 étant la plus ancienne des 3 plateformes, son processus de fabrication en 16 nm est moins performant que celui de l'AGX ou de la NANO en 12 nm.

En résumé, la configuration la plus rapide obtenue pour l'ensemble des plateformes est obtenue avec l'AGX à 2,3 GHz avec une vitesse de calcul de 70 ns par pixel et une consommation de 1275 nJ par pixel. Cela permet un traitement en temps de réel

de vidéos avec une définition d'environ 0,6 Mpix par image. La configuration la plus économe est obtenue avec la NANO à 1,4 GHz. Dans ce cas, la vitesse de calcul est de 311 ns par pixel pour une consommation de 989 nJ par pixel. Dans cette configuration, le temps réel est atteint pour des images légèrement supérieures à 0,1 Mpix.

Pour finir, nous présentons dans ces travaux, le prototype VIRTANS pour *VIdeo Real-Time Algorithm : Noise Suppression*. Ce dispositif, au format compact, permet de traiter en direct un flux vidéo SDI en provenance d'une caméra LHERITIER. VIRTANS est basé sur une architecture TX2i pour pouvoir fonctionner dans une large plage de température. La puissance de calcul supplémentaire ainsi apportée, nous permet d'exécuter RTE-VD et de restituer à un opérateur une vidéo débruitée en direct. VIRTANS est actuellement en cours de développement et n'utilise pas, pour l'heure, toute la puissance de calcul disponible. C'est pourquoi, la définition maximale de traitement est, dans un premier temps, limitée à 480×270 pixels par image.

5.2 Critiques et travaux futurs

La chaîne de traitement que nous proposons permet de rester efficace pour de forts niveaux de bruit, et ce, dans un temps de traitement maîtrisé. On constate cependant qu'il est encore impossible d'exécuter RTE-VD, en temps réel en haute définition sur les systèmes embarqués testés. De plus, le temps de calcul, ainsi que la qualité du débruitage obtenu, reposent essentiellement sur les performances du calcul de flot optique.

L'intégration des calculs en F16 dans la chaîne de traitement, devrait nous permettre d'accélérer de façon significative le temps de traitement. Il n'est en revanche pas nécessairement possible, ou pertinent, d'effectuer l'ensemble des étapes de RTE-VD en F16. C'est pourquoi nous envisageons une implémentation en arithmétique hybride, dans laquelle certaines étapes seront traitées en F16 et d'autres en F32. Toutefois, l'utilisation de F16 impacte la qualité des résultats. Il nous sera donc nécessaire d'évaluer la viabilité des F16 dans le cadre de notre application.

Toujours en ayant pour objectif d'accélérer les traitements, nous prévoyons de poursuivre l'implémentation de RTE-VD sur GPU embarqué. La maîtrise CPU et GPU peut permettre d'envisager une implémentation avec ces deux architectures travaillant en symbiose. Une implémentation hybride CPU/GPU peut permettre à la fois d'accélérer le temps de traitement et d'améliorer la qualité des résultats. L'équilibrage de la charge de travail entre le CPU et le GPU permettra de réduire le temps de traitement total ou d'augmenter la définition maximale de traitement. Il est également envisageable de profiter de cette hybridation des calcul pour augmenter l'importance de l'étape de filtrage par rapport au reste de la chaîne de traitement.

Si l'on prend l'algorithme de RTE-VD tel qu'il est implémenté dans ces travaux, la méthode se décompose en 3 étapes : StabLK, TV-L^1 et le filtrage bilatéral. Présentement, l'étape de TV-L^1 représente 97% du temps traitement et influe grandement sur le choix de la méthode de filtrage finale. Avec une hybridation des calculs CPU/GPU, il serait

possible de mettre en place un pipeline global du traitement afin d'effectuer un traitement plus lourd mais en maintenant la cadence de 25 images par secondes. On peut par exemple imaginer que le calcul de flot optique sera effectué sur GPU tandis que l'étape de filtrage sera effectuée sur CPU. En mettant en place un pipeline de traitement entre CPU et GPU, chaque étape pourra alors prendre 40 ms individuellement. La latence du traitement sera plus grande mais la cadence de traitement restera à 25 images par seconde. Dans ce cas, il nous est possible d'envisager une méthode de filtrage plus lourde que le filtrage bilatéral et donc d'augmenter la qualité visuelle des résultats.

Ces travaux ont donné lieu à trois publications dans des conférences internationales :

— "Energy and execution time comparison of optical flow algorithms on SIMD and GPU architectures", A. Petreto, A. Hennequin, T. Koehler, T. Romera, Y. Fargeix, B. Gaillard, M. Bouyer, Q. L. Meunier, and L. Lacassagne - *International Conference on Design and Architectures for Signal and Image Processing* (DASIP) 2018 [124].

— "A new real-time embedded video denoising algorithm", A. Petreto, T. Romera, F. Lemaitre, I. Masliah, B. Gaillard, M. Bouyer, Q. Meunier, and L. Lacassagne - *IEEE International Conference on Design and Architectures for Signal and Image Processing* (DASIP) 2019 [148]

— "Real-time embedded video denoiser prototype", A. Petreto, T. Romera, F. Lemaitre, M. Bouyer, B. Gaillard, P. Menard, Q. Meunier, and L. Lacassagne - *9th International Symposium-Optronics in Defense and Security* (OPTRO) 2020 [149].

Trois communications dans des conférences ou colloques nationaux :

— "Comparaison de la consommation énergétique et du temps d'exécution d'un algorithme de traitement d'images optimisé sur des architectures SIMD et GPU", A. Petreto, A. Hennequin, T. Koehler, T. Romera, Y. Fargeix, B. Gaillard, M. Bouyer, Q. L. Meunier, and L. Lacassagne - Colloque du Groupe de Recherche System On Chip, Systèmes embarqués et Objets Connectés (SOC^2) 2018 [150] : prix du meilleur poster.

— "Comparaison de la consommation énergétique et du temps d'exécution d'un algorithme de traitement d'images optimisé sur des architectures SIMD et GPU", A. Petreto, A. Hennequin, T. Koehler, T. Romera, Y. Fargeix, B. Gaillard, M. Bouyer, Q. L. Meunier, and L. Lacassagne - Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS) 2018 [151]

— "Débruitage temps réel embarqué pour vidéos fortement bruitées", A. Petreto, T. Romera, F. Lemaitre, I. Masliah, B. Gaillard, M. Bouyer, Q. Meunier, and L. Lacassagne - Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS) 2019 [152]

Bibliographie

[1] M. Zhang and B. K. Gunturk, "Multiresolution bilateral filtering for image denoising," *IEEE Transactions on image processing*, vol. 17, no. 12, pp. 2324–2333, 2008.

[2] T. Chen, K.-K. Ma, and L.-H. Chen, "Tri-state median filter for image denoising," *IEEE Transactions on Image processing*, vol. 8, no. 12, pp. 1834–1838, 1999.

[3] A. Buades, B. Coll, and J.-M. Morel, "A review of image denoising algorithms, with a new one," *Multiscale Modeling & Simulation*, vol. 4, no. 2, pp. 490–530, 2005.

[4] I. Pitas and A. N. Venetsanopoulos, *Nonlinear digital filters : principles and applications*. Springer Science & Business Media, 2013, vol. 84.

[5] C. Tomasi and R. Manduchi, "Bilateral filtering for gray and color images," in *Computer Vision, 1998. Sixth International Conference on.* IEEE, 1998, pp. 839–846.

[6] M. Kazubek, "Wavelet domain image denoising by thresholding and wiener filtering," *IEEE Signal Processing Letters*, vol. 10, no. 11, pp. 324–326, 2003.

[7] M. K. Mihcak, I. Kozintsev, K. Ramchandran, and P. Moulin, "Low-complexity image denoising based on statistical modeling of wavelet coefficients," *IEEE Signal Processing Letters*, vol. 6, no. 12, pp. 300–303, 1999.

[8] S. G. Chang, B. Yu, and M. Vetterli, "Adaptive wavelet thresholding for image denoising and compression," *IEEE transactions on image processing*, vol. 9, no. 9, pp. 1532–1546, 2000.

[9] K. Dabov, A. Foi, V. Katkovnik, and K. Egiazarian, "Image denoising by sparse 3-d transform-domain collaborative filtering," *IEEE Transactions on image processing*, vol. 16, no. 8, pp. 2080–2095, 2007.

[10] A. Buades, B. Coll, and J.-M. Morel, "A non-local algorithm for image denoising," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, vol. 2. IEEE, 2005, pp. 60–65.

[11] B. Goossens, H. Luong, A. Pizurica, and W. Philips, "An improved non-local denoising algorithm," in *2008 International Workshop on Local and Non-Local Approximation in Image Processing (LNLA 2008)*, 2008, pp. 143–156.

[12] A. Dauwe, B. Goossens, H. Q. Luong, and W. Philips, "A fast non-local image denoising algorithm," in *Image Processing : Algorithms and Systems VI*, vol. 6812. International Society for Optics and Photonics, 2008, p. 681210.

[13] M. Mahmoudi and G. Sapiro, "Fast image and video denoising via nonlocal means of similar neighborhoods," *IEEE signal processing letters*, vol. 12, no. 12, pp. 839–842, 2005.

[14] P. Coupé, P. Yger, and C. Barillot, "Fast non local means denoising for 3D MR images," in *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer, 2006, pp. 33–40.

[15] J. Hu, Y. Pu, X. Wu, Y. Zhang, and J. Zhou, "Improved DCT-based nonlocal means filter for MR images denoising," *Computational and mathematical methods in medicine*, vol. 2012, pp. 1–14, 2012.

[16] M. Lebrun, A. Buades, and J.-M. Morel, "A nonlocal bayesian image denoising algorithm," *SIAM Journal on Imaging Sciences*, vol. 6, no. 3, pp. 1665–1688, 2013.

[17] M. Lebrun, "An analysis and implementation of the BM3D image denoising method," *Image Processing On Line*, vol. 2, pp. 175–213, 2012.

[18] N. Ahmed, T. Natarajan, and K. R. Rao, "Discrete cosine transform," *IEEE transactions on Computers*, vol. 100, no. 1, pp. 90–93, 1974.

[19] J. Fourier, "Mémoire sur la propagation de la chaleur dans les corps solides (extrait)," *Nouveau Bulletin des Sciences, par la Société Philomathique de Paris*, vol. 1, pp. 112–16, 1808.

[20] J. L. W. Jacques Hadamard, Hans Rademacher, "Wikipedia : https://en.wikipedia.org/wiki/Hadamard_transform."

[21] S. Lefkimmiatis, "Universal denoising networks : a novel CNN architecture for image denoising," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 3204–3213.

[22] K. Zhang, W. Zuo, Y. Chen, D. Meng, and L. Zhang, "Beyond a gaussian denoiser : Residual learning of deep CNN for image denoising," *IEEE Transactions on Image Processing*, vol. 26, no. 7, pp. 3142–3155, 2017.

[23] H. C. Burger, C. J. Schuler, and S. Harmeling, "Image denoising : Can plain neural networks compete with BM3D ?" in *2012 IEEE conference on computer vision and pattern recognition*. IEEE, 2012, pp. 2392–2399.

[24] V. Jain and S. Seung, "Natural image denoising with convolutional networks," in *Advances in neural information processing systems*, 2009, pp. 769–776.

[25] K. Dabov, A. Foi, and K. Egiazarian, "Video denoising by sparse 3D transform-domain collaborative filtering," in *2007 15th European Signal Processing Conference*. IEEE, 2007, pp. 145–149.

[26] M. Maggioni, G. Boracchi, A. Foi, and K. Egiazarian, "Video denoising using separable 4D nonlocal spatiotemporal transforms," in *Image Processing : Algorithms and Systems IX*, vol. 7870. International Society for Optics and Photonics, 2011, pp. 1–11.

[27] P. Arias, G. Facciolo, and J.-M. Morel, "A comparison of patch-based models in video denoising," in *2018 IEEE 13th Image, Video, and Multidimensional Signal Processing Workshop (IVMSP)*. IEEE, 2018, pp. 1–5.

[28] A. Buades and J.-L. Lisani, "Video denoising with optical flow estimation," *Image Processing On Line*, vol. 8, pp. 142–166, 2018.

[29] A. Davy, T. Ehret, G. Facciolo, J.-M. Morel, and P. Arias, "Non-local video denoising by CNN," *arXiv preprint arXiv :1811.12758*, 2018.

[30] T. Xue, B. Chen, J. Wu, D. Wei, and W. T. Freeman, "Video enhancement with task-oriented flow," *arXiv preprint arXiv :1711.09078*, 2017.

[31] M. Tassano, J. Delon, and T. Veit, "Fastdvdnet : Towards real-time video denoising without explicit motion estimation," *arXiv preprint arXiv :1907.01361*, 2019.

[32] C. Zuo, Y. Liu, X. Tan, W. Wang, and M. Zhang, "Video denoising based on a spatiotemporal kalman-bilateral mixture model," *The Scientific World Journal*, vol. 2013, 2013.

[33] C. Liu and W. T. Freeman, "A high-quality video denoising algorithm based on reliable motion estimation," in *European Conference on Computer Vision*. Springer, 2010, pp. 706–719.

[34] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Journal of basic Engineering*, vol. 82, no. 1, pp. 35–45, 1960.

[35] S. G. Pfleger, P. D. Plentz, R. C. Rocha, A. D. Pereira, and M. Castro, "Real-time video denoising on multicores and GPUs with kalman-based and bilateral filters fusion," *Journal of Real-Time Image Processing*, pp. 1–14, 2017.

[36] P. Shukla and M. J. Srivas, "Video denoising algorithm : A review," 2016.

[37] J. Ehmann, L.-C. Chu, S.-F. Tsai, and C.-K. Liang, "Real-time video denoising on mobile phones," in *2018 25th IEEE International Conference on Image Processing (ICIP)*. IEEE, 2018, pp. 505–509.

[38] X. Tan, Y. Liu, C. Zuo, and M. Zhang, "A real-time video denoising algorithm with FPGA implementation for poisson–gaussian noise," *Journal of Real-Time Image Processing*, vol. 13, no. 2, pp. 327–343, 2014.

[39] P. Arias and J.-M. Morel, "Kalman filtering of patches for frame-recursive video denoising," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2019, pp. 1–8.

[40] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *Workshop of Image Understanding*, 1981, pp. 121–130.

[41] C. Zach, T. Pock, and H. Bischof, "A duality based approach for realtime TV-L1 optical flow," in *Joint Pattern Recognition Symposium*. Springer, 2007, pp. 214–223.

[42] C. Harris and M. Stephens, "A combined corner and edge detector." in *Alvey vision conference*, vol. 15, no. 50. Citeseer, 1988, pp. 10–5244.

[43] O. Haggui, C. Tadonki, L. Lacassagne, F. Sayadi, and B. Ouni, "Harris corner detection on a NUMA manycore," *Future Generation Computer Systems*, vol. 88, pp. 442–452, 2018.

[44] L. Lacassagne, D. Etiemble, A. Hassan-Zahraee, A. Dominguez, and P. Vezolle, "High level transforms for SIMD and low-level computer vision algorithms," in *ACM Workshop on Programming Models for SIMD/Vector Processing (PPoPP)*, 2014, pp. 49–56.

[45] C. Tadonki, L. Lacassagne, T. Saidani, J. Falcou, and K. Hamidouche, "The harris algorithm revisited on the cell processor," in *Proceedings of the 1st International Workshop on Highly Efficient Accelerators and Reconfigurable Technologies HEART 2010*, 2010.

[46] T. Saidani, J. Falcou, C. Tadonki, L. Lacassagne, and D. Etiemble, "Algorithmic skeletons within an embedded domain specific language for the cell processor," in *2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 2009, pp. 67–76.

[47] T. Saidani, L. Lacassagne, J. Falcou, C. Tadonki, and S. Bouaziz, "Parallelization schemes for memory optimization on the cell processor : a case study on the harris corner detector," in *Transactions on high-performance embedded architectures and compilers III*. Springer, 2011, pp. 177–200.

[48] T. Saidani, L. Lacassagne, S. Bouaziz, and T. M. Khan, "Parallelization strategies for the points of interests algorithm on the cell processor," in *International Symposium on Parallel and Distributed Processing and Applications*. Springer, 2007, pp. 104–112.

[49] D. Etiemble, S. Piskorski, and L. Lacassagne, "Performance evaluation of altera C2H compiler on image processing benchmarks," in *TCHA : Workshop on Tools And Compiler for Hardware Acceleration*, vol. 17, 2006.

[50] L. Lacassagne, L. Cabaret, D. Etiemble, F. Hebache, and A. Petreto, "A new SIMD iterative connected component labeling algorithm," in *Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing*, 2016, pp. 1–8.

[51] F. Lemaitre, B. Couturier, and L. Lacassagne, "Cholesky factorization on SIMD multi-core architectures," *Journal of Systems Architecture*, vol. 79, pp. 1–15, 2017.

[52] M. Gouiffes, A. R. M. y Terán, and L. Lacassagne, "Color enhanced local binary patterns in covariance matrices descriptors (ELBCM)," *Journal of Visual Communication and Image Representation*, vol. 49, pp. 447–458, 2017.

[53] A. Romero, L. Lacassagne, M. Gouiffès, and A. H. Zahraee, "Covariance tracking : architecture optimizations for embedded systems," *EURASIP Journal on Advances in Signal Processing*, vol. 2014, no. 1, p. 175, 2014.

[54] F. Laguzet, A. Romero, M. Gouiffes, L. Lacassagne, and D. Etiemble, "Color tracking with contextual switching : real-time implementation on CPU," *Journal of Real-Time Image Processing*, vol. 10, no. 2, pp. 403–422, 2015.

[55] H. Ye, L. Lacassagne, J. Falcou, D. Etiemble, L. Cabaret, and O. Florent, "High level transforms to reduce energy consumption of signal and image processing operators," in *IEEE International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2013, pp. 247–254.

[56] A. R. M. y Terán, L. Lacassagne, A. H. Zahraee, and M. Gouiffàs, "Real-time covariance tracking algorithm for embedded systems," in *2013 Conference on Design and Architectures for Signal and Image Processing*. IEEE, 2013, pp. 104–111.

[57] H. Ye, L. Lacassagne, D. Etiemble, L. Cabaret, J. Falcou, A. Romero, and O. 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, pp. 1–8.

[58] L. Lacassagne, A. Manzanera, and A. Dupret, "Motion detection : Fast and robust algorithms for embedded systems," in *2009 16th IEEE international conference on image processing (ICIP)*. IEEE, 2009, pp. 3265–3268.

[59] A. Verdant, A. Dupret, H. Mathias, P. Villard, and L. Lacassagne, "Adaptive multiresolution for low power CMOS image sensor," in *2007 IEEE International Conference on Image Processing*, vol. 5. IEEE, 2007, pp. V–185.

[60] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE transactions on computers*, vol. 100, no. 9, pp. 948–960, 1972.

[61] OpenACC Community, "OpenACC website https://www.openacc.org/."

[62] V. Lefebvre and P. Feautrier, "Storage management in parallel programs." in *Parallel, Distributed and Network-based Processing (PDP)*, 1997, pp. 181–188.

[63] OpenMP Community, "OpenMP website https://www.openmp.org/."

[64] A. Floc'h, T. Yuki, A. El-Moussawi, A. Morvan, K. Martin, M. Naullet, M. Alle, L. L'Hours, N. Simon, S. Derrien, F. Charot, C. Wolinski, and O. Sentieys, "Gecos : A framework for prototyping custom hardware design flows," in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*, Sept 2013, pp. 100–105.

[65] D. Etiemble and L. 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 (ICPP)*. IEEE, 2004, pp. 1–4.

[66] J. Iliffe, "The use of the genie system in numerical calculation," in *International Tracts in Computer Science and Technology and Their Application*. Elsevier, 1961, vol. 2, pp. 1–28.

[67] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J.-F. Nezan, and S. Aridhi, "Preesm : A dataflow-based rapid prototyping framework for simplifying multicore dsp programming," in *2014 6th european embedded design in education and research conference (EDERC)*. IEEE, 2014, pp. 36–40.

[68] S. Xu and D. Gregg, "Semi-automatic composition of data layout transformations for loop vectorization," in *IFIP International Conference on Network and Parallel Computing*. Springer, 2014, pp. 485–496.

[69] F. Gouin, C. Ancourt, and C. Guettier, "An up to date mapping methodology for GPUs," in *Connected Plant Conference (CPC)*, 2018, pp. 1–8.

[70] PIPS Community, "PIPS website https://pips4u.org/."

[71] N. Lossing, C. Ancourt, and F. Irigoin, "Automatic code generation of distributed parallel tasks," in *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*. IEEE, 2016, pp. 234–241.

[72] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "OpenACC—first experiences with real-world applications," in *European Conference on Parallel Processing*. Springer, 2012, pp. 859–870.

[73] B. Barney, "Posix threads programming https://computing.llnl.gov/tutorials/pthreads/."

[74] R. E. Moore, *Interval analysis*. Prentice-Hall Englewood Cliffs, NJ, 1966, vol. 4.

[75] L. H. De Figueiredo and J. Stolfi, "Affine arithmetic : concepts and applications," *Numerical Algorithms*, vol. 37, no. 1-4, pp. 147–158, 2004.

[76] Office Québécois de la langue française, "Fiche terminologique http://gdt.oqlf.gouv.qc.ca/ficheOqlf.aspx?Id_Fiche=8875278."

[77] J. D. McCalpin, "Stream : Sustainable memory bandwidth in high performance computers http://www.cs.virginia.edu/stream/#StandardResults."

[78] C. Alias, A. Darte, and F. Baray, "Lattice-based array contraction : From theory to practice," 2007.

[79] A. Darte, A. Isoard, and T. Yuki, "Extended lattice-based memory allocation," in *Proceedings of the 25th International Conference on Compiler Construction*, 2016, pp. 218–228.

[80] S. Kobeissi and P. Clauss, "The polyhedral model beyond loops recursion optimization and parallelization through polyhedral modeling," in *International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2019.

[81] J. Shirako and V. Sarkar, "Integrating data layout transformations with the polyhedral model," in *Proceedings of International Workshop on Polyhedral Compilation Techniques (IMPACT'19), D. Wonnacott and O. Zinenko (Eds.). Valencia, Spain. http ://impact. gforge. inria. fr/impact2019/papers/IMPACT_2019_paper_8. pdf*, 2019.

[82] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye, "Alphaz : A system for design space exploration in the polyhedral model," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2012, pp. 17–31.

[83] C. Bastoul, "Contributions to high-level program optimization," Ph.D. dissertation, Habilitation Thesis. Paris-Sud University, France, 2012.

[84] GCC Community, "GCC website https://gcc.gnu.org/."

[85] Lheritier, "cf fiche cateye en annexe https://www.lheritier-alcen.com/fr."

[86] Middlebury, "Optical flow database http://vision.middlebury.edu/flow/."

[87] F. C. Crow, "Summed-area tables for texture mapping," in *ACM SIGGRAPH computer graphics*, vol. 18, no. 3. ACM, 1984, pp. 207–212.

[88] A. R. M. y Teran, "Real-time multi-target tracking : a study on color-texture covariance matrices and descriptor/operator switching," Ph.D. dissertation, 2013.

[89] P. Ouyang, S. Yin, Y. Zhang, L. Liu, and S. Wei, "A fast integral image computing hardware architecture with high power and area efficiency," *IEEE Transactions on Circuits and Systems II : Express Briefs*, vol. 62, no. 1, pp. 75–79, 2015.

[90] S. Ehsan, A. Clark, N. Rehman, K. McDonald-Maier *et al.*, "Integral images : efficient algorithms for their computation and storage in resource-constrained embedded vision systems," *Sensors*, vol. 15, no. 7, pp. 16 804–16 830, 2015.

[91] Y.-T. Wu, Y.-T. Wu, C.-Y. Cho, S.-Y. Tseng, C.-N. Liu, and C.-T. King, "Parallel integral image generation algorithm on multi-core system," in *2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications*. IEEE, 2011, pp. 31–35.

[92] NVIDIA, "Jetson AGX specifications https://developer.nvidia.com/embedded/buy/jetson-agx-xavier-devkit."

[93] G. 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*. ACM, 1967, pp. 483–485.

[94] P. Arias and J.-M. Morel, "Towards a bayesian video denoising method," in *International Conference on Advanced Concepts for Intelligent Vision Systems*. Springer, 2015, pp. 107–117.

[95] S. Borkar, "Thousand core chips : a technology perspective," in *Proceedings of the 44th annual Design Automation Conference*. ACM, 2007, pp. 746–749.

[96] S. Baker, D. S. J. P. Lewis, S. Roth, M. J. Black, and R. Szeliski, "A database and evaluation methodology for optical flow," *International Journal of Computer Vision*, vol. 19,2, pp. 1–31, 2011.

[97] IPOL, "Image processing on line http://www.ipol.im/."

[98] G. Gultekin and A. Saranli, "An FPGA-based high performance optical flow hardware design for computer vision," *Microprocessors and Microsystems*, vol. 37, no. 1-3, pp. 270–286, 2013.

[99] L. Bako, S. Hajdu, S.-T. Brassai, F. Morgan, and C. Enachescu, "Embedded implementation of a real-time motion estimation method in video sequences," *Procedia Technology*, vol. 22, pp. 897–904, 2016.

[100] M. Kunz, A. Ostrowski, and P. Zipf, "An FPGA-optimized architecture of horn and schunck optical flow algorithm for real-time applications," in *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2014, pp. 1–4.

[101] Z. Chai, H. Zhou, Z. Wang, and D. Wu, "Using C to implement high-efficient computation of dense optical flow on FPGA-accelerated heterogeneous platforms," in *International Conference on Field-Programmable Technology (FPT)*. IEEE, 2014, pp. 260–263.

[102] O. Haggui, C. Tadonki, F. Sayadi, and O. Bouraoui, "Efficient GPU implementation of lucas-kanade through OpenACC," in *Proceedings of the 14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications, VISAPP, Prague, Czech Republic*, vol. 5, 2019, pp. 768–775.

[103] O. Haggui, C. Tadonki, F. Sayadi, and B. Ouni, "Memory efficient deployment of an optical flow algorithm on GPU using OpenMP," in *International Conference on Image Analysis and Processing*. Springer, 2019, pp. 477–487.

[104] A. Plyer, G. L. Besnerais, and F. Champagnat, "Massively parallel lucas kanade optical flow for real-time video processing applications," *Journal of Real-Time Image Processing*, vol. 11,4, pp. 713–730, 2016.

[105] J.D.Adarve and R. Mahony, "A filter formulation for computing real time optical flow," in *(IEEE Robotic and Automation Letters)*. IEEE, 2016, pp. 1192–1199.

[106] T. Kroeger, R. Timofte, D. Dai, and L. V. Gool, "Fast optical flow using dense inverse search," in *(ECCV)*, 2016.

[107] T. Senst, R. H. Evangelio, I. Keller, and T. Sikora, "Clustering motion for real-time optical flow based tracking," in *Advanced Video and Signal-Based Surveillance (AVSS), 2012 IEEE Ninth International Conference on*. IEEE, 2012, pp. 410–415.

[108] A. Garcia-Dopico, J. L. Pedraza, M. Nieto, A. Pérez, S. Rodríguez, and J. Navas, "Parallelization of the optical flow computation in sequences from moving cameras," *EURASIP Journal on Image and Video Processing*, vol. 2014, no. 1, p. 18, 2014.

[109] P. Weinzaepfel, J. Revaud, Z. Harchaoui, and C. Schmid, "Deepflow : Large displacement optical flow with deep matching," in *International Conference on Computer Vision (ICCV)*. IEEE, 2013, pp. 1385–1392.

[110] A. Ranjan and M. J. Black, "Optical flow estimation using a spatial pyramid network," in *(CVPR)*. IEEE, 2017, pp. 2720–2729.

[111] B. K. P. Horn and B. G. Schunk, "Determining optical flow," *ACM Computing Surveys (CSUR)*, vol. 17, no. 1-3, pp. 185–203, 1981.

[112] A. Wedel, T. Pock, J. Braun, U. Franke, and D. Cremers, "Duality TV-L1 flow with fundamental matrix prior," in *2008 23rd International Conference Image and Vision Computing New Zealand*. IEEE, 2008, pp. 1–6.

[113] A. Wedel, T. Pock, C. Zach, H. Bischof, and D. Cremers, "An improved algorithm for TV-L1 optical flow," in *Statistical and geometrical approaches to visual motion analysis*. Springer, 2009, pp. 23–45.

[114] C. Ballester, L. Garrido, V. Lazcano, and V. Caselles, "A TV-L1 optical flow method with occlusion detection," in *Joint DAGM (German Association for Pattern Recognition) and OAGM Symposium*. Springer, 2012, pp. 31–40.

[115] A. Dosovitskiy, P. Fischer, E. Ilg, P. Hausser, C. Hazirbas, V. Golkov, P. Van Der Smagt, D. Cremers, and T. Brox, "Flownet : Learning optical flow with convolutional networks," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 2758–2766.

[116] E. Ilg, N. Mayer, T. Saikia, M. Keuper, A. Dosovitskiy, and T. Brox, "Flownet 2.0 : Evolution of optical flow estimation with deep networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 2462–2470.

[117] T.-W. Hui, X. Tang, and C. 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, pp. 8981–8989.

[118] J. Sachez, E. Meinhardt-Llopis, and G. Facciolo, "TV-L1 optical flow estimation," *Image Processing On Line (IPOL)*, vol. 2013, no. 3, pp. 137–150, 2013.

[119] L. I. Rudin, S. Osher, and E. Fatemi, "Nonlinear total variation based noise removal algorithms," *Physica D : nonlinear phenomena*, vol. 60, no. 1-4, pp. 259–268, 1992.

[120] A. Chambolle, "An algorithm for total variation minimization and applications," *Journal of Mathematical imaging and vision*, vol. 20, no. 1-2, pp. 89–97, 2004.

[121] Derf, "Derf's test media collection https://media.xiph.org/video/derf/."

[122] M. Bouyer, "A low-cost SoC power measurement platform https://www-soc.lip6.fr/trac/soc-conso."

[123] N. Rambaux, D. Galayko, G. Guignan, J. Vaubaillon, L. Lacassagne, P. Keckhut, A. Levasseur-Regourd, A. Hauchecorne, M.Birlan, G. Augarde, S. Barnier, S. B. 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, and B. Zanda, "Meteorix : a cubesat mission dedicated to the detection of meteors," in *42nd Assembly of Committee on Space Research (COSPAR)*, 2018.

[124] A. Petreto, A. Hennequin, T. Koehler, T. Romera, Y. Fargeix, B. Gaillard, M. Bouyer, Q. L. Meunier, and L. Lacassagne, "Energy and execution time comparison of optical flow algorithms on SIMD and GPU architectures," in *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE, 2018, pp. 25–30.

[125] A. Hennequin, L. Lacassagne, L. Cabaret, and Q. 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, pp. 76–81.

[126] 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. Levasseurd-Regourd, J. Rault, and 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, pp. 1–9.

[127] Z. X. et al., "OpenBLAS http://www.openblas.net/."

[128] OpenCV, "https://docs.opencv.org/master/dc/d4d/classcv_1_1optflow_1_1DualTVL1OpticalFlow.html."

[129] S. Piskorski, L. Lacassagne, S. Bouaziz, and D. Etiemble, "Customizing CPU instructions for embedded vision systems," in *Computer Architecture, Machine Perception and Sensors (CAMPS)*. IEEE, 2006, pp. 59–64.

[130] L. Lacassagne, D. Etiemble, and S. Kablia, "16-bit floating point instructions for embedded multimedia applications," in *CAMP : Computer Architecture and Machine Perception*. IEEE, 2005.

[131] D. Etiemble, L. Lacassagne, and S. Bouaziz, "Customizing 16-bit floating point instruction on a NIOS II processor for FPGA image and media processing," in *Estimedia - Embedded Systems for Real-Time Multimedia*, 2005.

[132] D. Etiemble and L. Lacassagne, "Introducing image processing and SIMD computations with FPGA soft-cores and customized instructions," in *Workshop on Reconfigurable Computing Education (WRCE)*, 2006, pp. 1–8.

[133] L. Lacassagne and D. Etiemble, "16-bit floating point operations for low-end and high-end embedded processors," in *ODES : Optimizations for DSP and Embedded Systems*. IEEE/ACM, 2005.

[134] B. Barrois, O. Sentieys, and D. Menard, "The hidden cost of functional approximation against careful data sizing—a case study," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 181–186.

[135] R. Rocher, D. Menard, O. Sentieys, and P. Scalart, "Accuracy evaluation of fixed-point LMS algorithm," in *2004 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 5. IEEE, 2004, pp. V–237.

[136] D. Menard, D. Chillet, and O. Sentieys, "Floating-to-fixed-point conversion for digital signal processors," *EURASIP Journal on Advances in Signal Processing*, vol. 2006, no. 1, p. 096421, 2006.

[137] G. Gupta, "Algorithm for image processing using improved median filter and comparison of mean, median and improved median filter," *International Journal of Soft Computing and Engineering (IJSCE)*, vol. 1, no. 5, pp. 304–311, 2011.

[138] K. He, J. Sun, and X. Tang, "Guided image filtering," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 6, pp. 1397–1409, 2012.

[139] K. He and J. Sun, "Fast guided filter," *arXiv preprint arXiv :1505.00996*, 2015.

[140] T. Q. Pham and L. J. Van Vliet, "Separable bilateral filtering for fast video preprocessing," in *2005 IEEE International Conference on Multimedia and Expo*. IEEE, 2005, pp. 4–8.

[141] N. Fukushima, S. Fujita, and Y. Ishibashi, "Switching dual kernels for separable edge-preserving filtering," in *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015, pp. 1588–1592.

[142] N. N. Schraudolph, "A fast, compact approximation of the exponential function," *Neural Computation*, vol. 11, no. 4, pp. 853–862, 1999.

[143] G. C. Cawley, "On a fast, compact approximation of the exponential function," *Neural computation*, vol. 12, no. 9, pp. 2009–2012, 2000.

[144] njuffa, "Fastest implementation of exponential function using SSE https://stackoverflow.com/questions/47025373/fastest-implementation-of-exponential-function-using-sse."

[145] A. C. I. Malossi, Y. Ineichen, C. Bekas, and A. Curioni, "Fast exponential computation on SIMD architectures," *Proc. of HIPEAC-WAPCO, Amsterdam NL*, 2015.

[146] C. T. Inc, "Astro carrier board for nvidia jetson tx2/tx2i/tx1 http://connecttech.com/product/astro-carrier-for-nvidia-jetson-tx2-tx1/."

[147] AverMedia, "Cm313bw https://www.avermedia.com/professional/product/cm313bw/overview."

[148] A. Petreto, T. Romera, F. Lemaitre, I. Masliah, B. Gaillard, M. Bouyer, Q. L. Meunier, and L. Lacassagne, "A new real-time embedded video denoising algorithm," in *IEEE International Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2019, pp. 1–6.

[149] A. Petreto, T. Romera, F. Lemaitre, M. Bouyer, B. Gaillard, P. Menard, Q. Meunier, and L. Lacassagne, "Real-time embedded video denoiser prototype," in *9th International Symposium-Optronics in Defense and Security (Optro)*, 2020.

[150] A. Petreto, A. Hennequin, T. Koehler, T. Romera, Y. Fargeix, B. Gaillard, M. Bouyer, Q. Meunier, and 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," GdR SOC2, Jun. 2018, poster. [Online]. Available : https://hal.archives-ouvertes.fr/hal-01835240

[151] ——, "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 *2018 Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS)*, 2018.

[152] A. Petreto, T. Romera, F. Lemaitre, I. Masliah, B. Gaillard, M. Bouyer, Q. L. Meunier, and L. Lacassagne, "Débruitage temps réel embarqué pour vidéos fortement bruitées," in *2019 Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS)*, 2019.

[153] Wikipedia, "Peak signal to noise ratio https://fr.wikipedia.org/wiki/Peak_Signal_to_Noise_Ratio."

Glossaire

Albédo Pouvoir réfléchissant d'une surface. 6

API *Application Programm Interface*. Interface de programmation d'application, ensemble de normes qui permettent 90

Block Matching Méthode de mise en correspondance de plusieurs voisinages de pixels au sein d'une image ou parmi plusieurs. 8, 54, 55, 58

BNL Bas Niveau de Lumière. 5, 6, 37

DRI Détection Reconnaissance Identification. Critère principalement utilisé dans un contexte militaire pour évaluer les performances d'un système d'observation. 6

Intensité arithmétique Indique, pour une implémentation donnée, le rapport entre le nombre de d'opérations effectuées et le nombre d'accès mémoire. 85, 86, 89

LWIR *Long Wave InfraRed*. Désigne l'infrarouge grandes ondes : longueur d'onde $\approx [8-15]\mu m$ 5, 6

MIMD *Multiple Instruction on Multiple Data*. Modèle de parallélisme multi-tâches. 13, 14, 17, 31

MSE *Mean Square Error*. Erreur quadratique moyene : ici, de reconstruction. 67–74, 91–94, 109–112, 117, 120

MWIR *Mid-Wave InfraRed*. Désigne l'infrarouge ondes moyennes : longueur d'onde $\approx [3-8]\mu m$ 5, 6

NIR *Near InfraRed*. Désigne le proche infrarouge : longueur d'onde $\approx [0,75-1,4]\mu m$ 5

OpenMP *Open Multi-Processing*. Interface de programmation pour la gestion du parallélisme (principalement MIMD). 14

PSNR *Peak Signal to Noise Ratio*. Mesure de distorsion utilisée en image numérique [153]. Permet de quantifier les performances d'une transformation sur une image par rapport à l'image d'origine. Il s'agit d'une mesure de similitude exprimée en décibels, plus sa valeur est haute, plus l'image évaluée est proche de celle avec laquelle elle est comparée. 7, 11, 117–121, 134, 136, 137

SIMD *Single Instruction on Multiple Data*. Modèle de parallélisme permettant d'appliquer une même instruction à plusieurs données. 3, 13–17, 31, 46, 52, 53, 55, 82, 84, 89, 95, 96, 100, 105, 107, 108

SIMDisation Écriture d'instructions SIMD à la main par le développeur sans se reposer sur l'aide du compilateur. A ne pas confondre avec la vectorisation. 15, 52, 86, 106, 155

SPMD *Single Program on Multiple Data*. Modèle de parallélisme multi-tâches où tous les processeurs exécutent en parallèle le même programme sur des données différentes. 14, 89, 100, 106–108, 113, 136

StabLK Nom donné dans ce manuscrit à la méthode de stabilisation utilisée dans RTE-VD. 3, 36, 40, 43, 44, 51–53, 108, 127, 135, 136, 138

SWIR *Short Wave InfraRed*. Désigne l'infrarouge ondes courtes : longueur d'onde $\approx [1,4-3]\mu$m 5

Vectorisation Génération d'instructions SIMD à partir d'un code scalaire par un compilateur. A ne pas confondre avec la SIMDisation. 15

VIS Désigne le domaine visible du spectre lumineux $\approx [380-780]$nm. 5, 6

Warp Recalage d'image par interpolation selon un champs de vecteur donné. Utilisé dans TV-L^1 pour stabiliser la méthode et est essentiel dans une approche multi-échelles. 61, 62, 64, 65, 70–72, 74, 82, 84, 90, 91, 93, 95, 96, 99, 100, 110, 111

Annexe A

Temps de calcul et consommation de TV-L^1 : Résultats complets

Dans cette annexe, nous présentons les différentes performances obtenues pour le calcul de 3 itérations de TV-L^1. Les résultats sont indiqués pour les 3 plateformes évaluées (AGX, NANO, TX2) et pour différentes fréquences. Pour chaque série de résultats nous précisons : le CPU utilisé, le nombre et la nature des cœurs utilisés ainsi que la finesse de gravure.

Les critères évalués sont :

— Le temps de traitement en cycles par point (CPP) en fonction de la taille des images traitées.
— L'énergie consommée en nano-joules par pixel (nJ/Pix) en fonction de la taille des images traitées.
— La puissance consommée en Watts (W) en fonction de la taille des images traitées.
— La puissance consommée en Watts (W) pour le traitement d'une image 1008×1008 en fonction de la fréquence.

Les implémentations évaluées sont :

— *Base* : implémentation standard non optimisée.
— *Simd* : équivalent de *base* avec SIMDisation.
— *Pipe* : *simd* + pipeline d'itérations.
— *Pipe_mono* : *Pipe* + optimisation mono-buffer.

Toutes les implémentations bénéficient du parallélisme de *threads*.

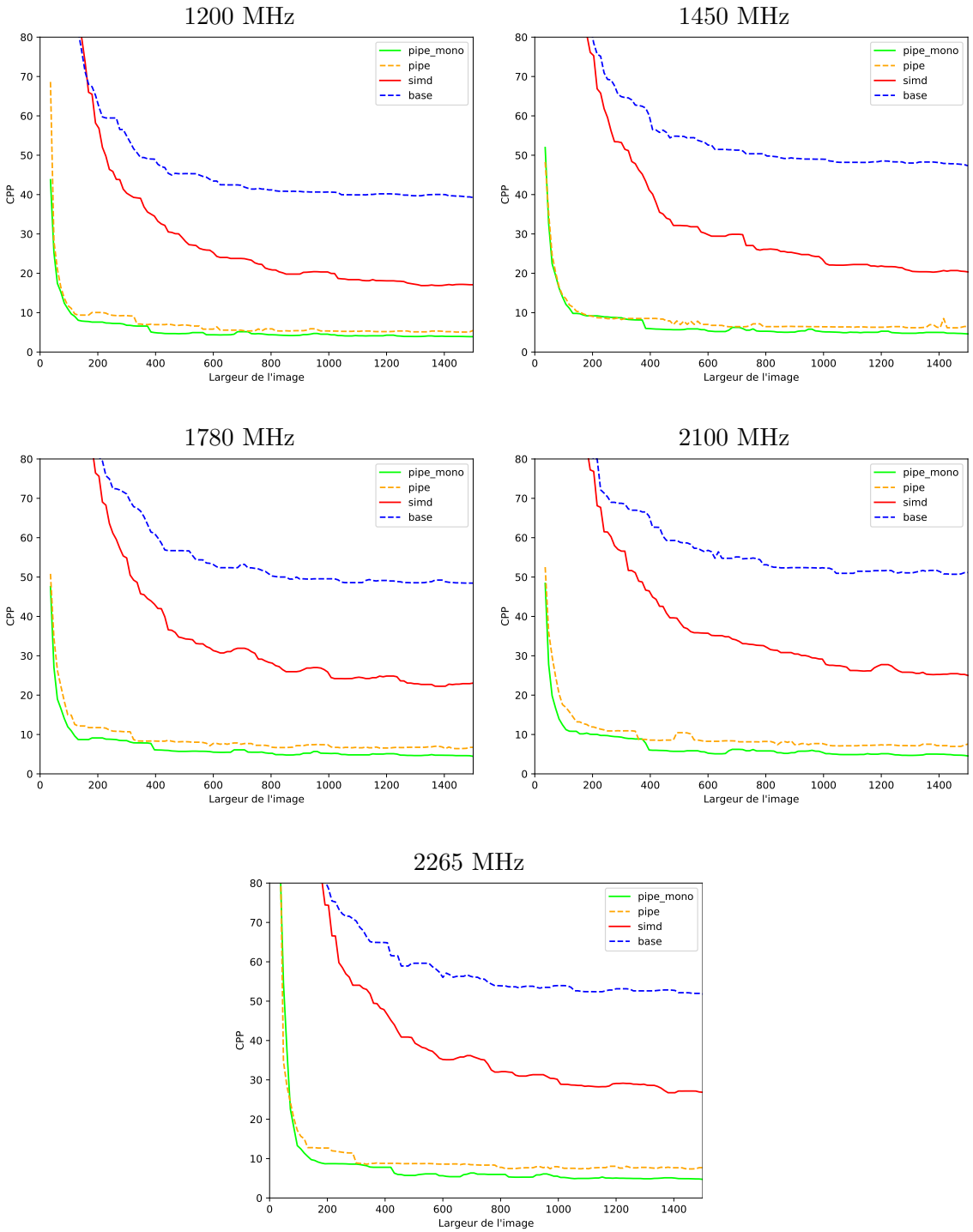


FIGURE A.1 – CPP en fonction de la taille des images pour l'AGX : 8×ARM Carmel 12 nm.

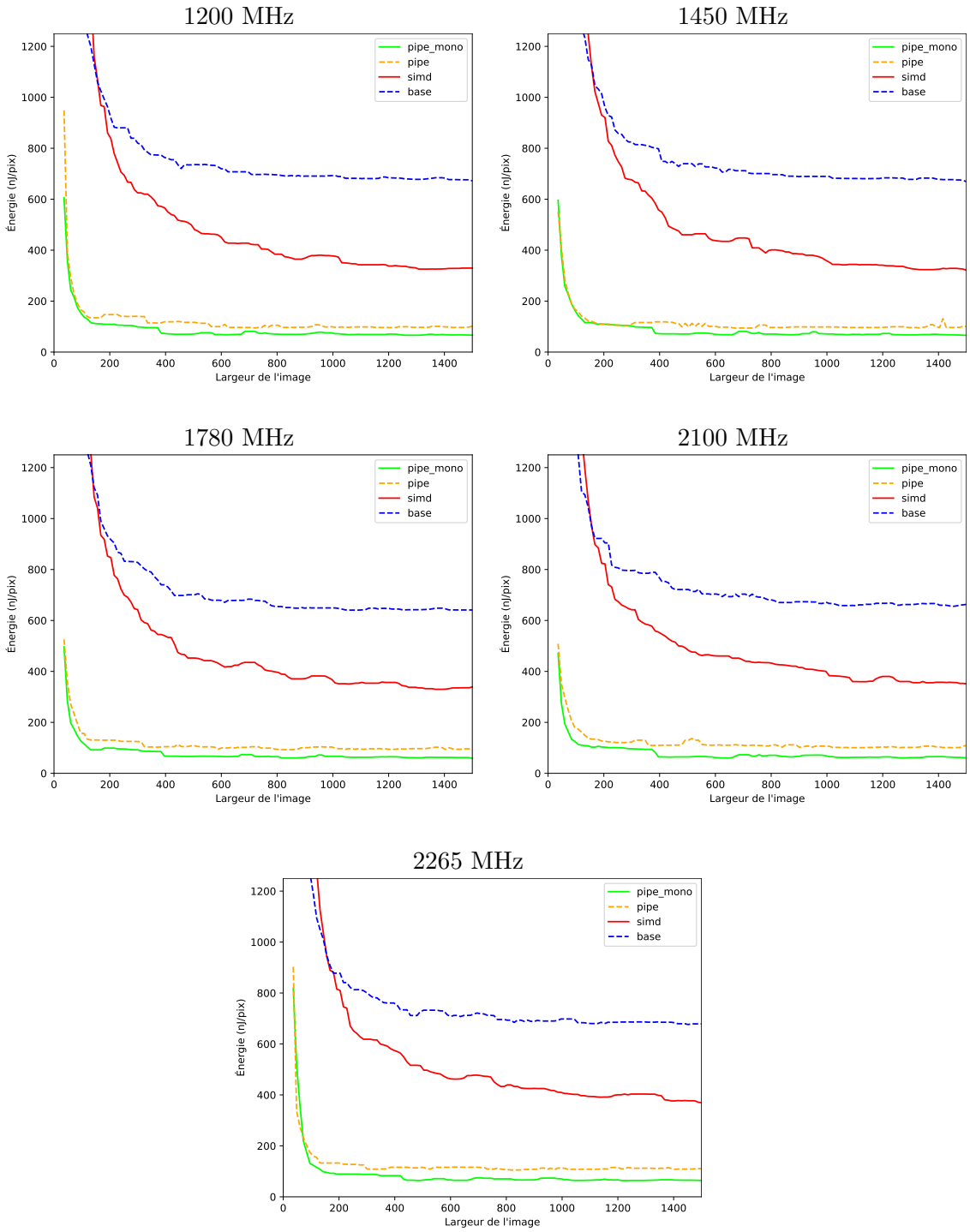


FIGURE A.2 – Énergie en fonction de la taille des images pour l'AGX : 8×ARM Carmel 12 nm.

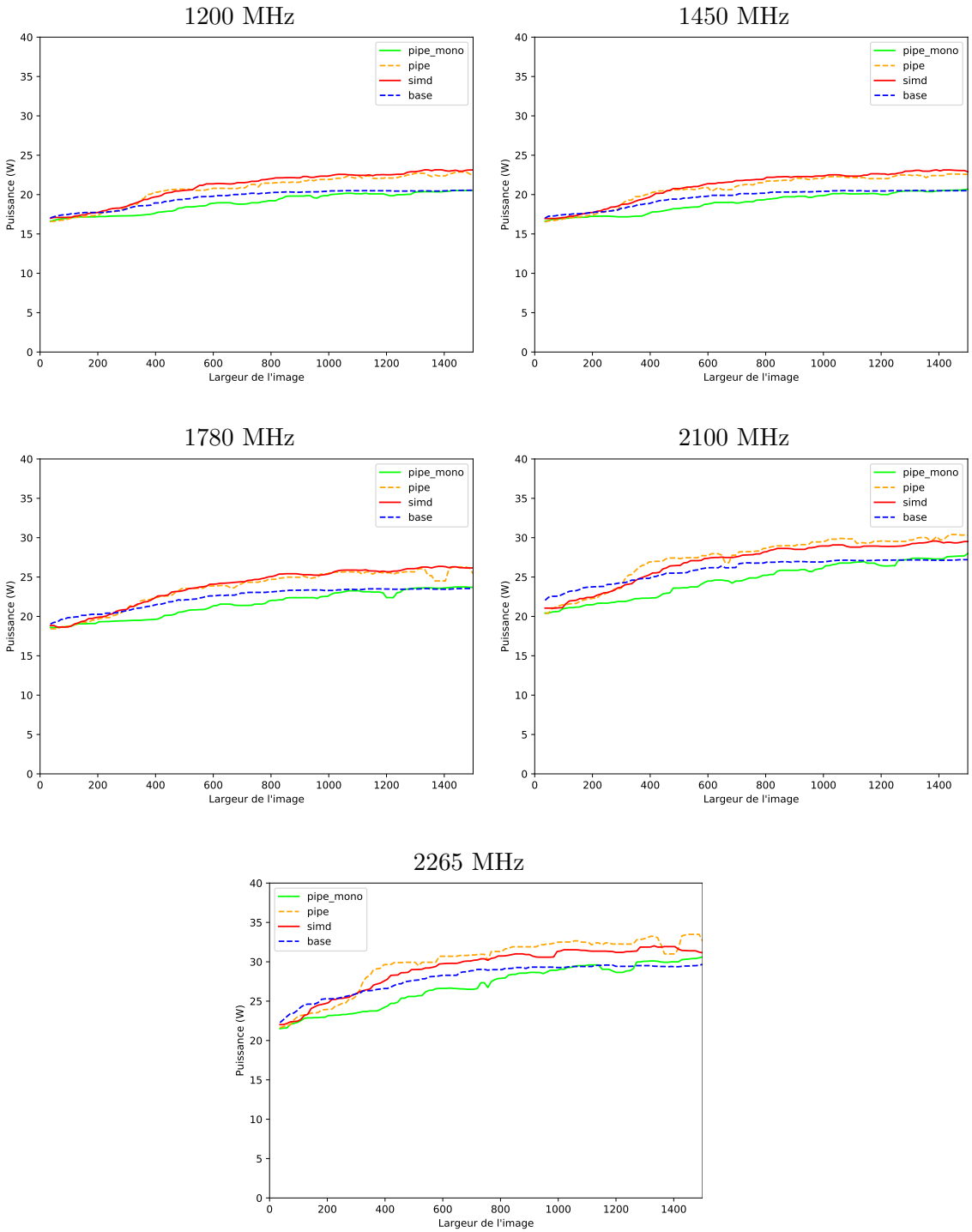


FIGURE A.3 – Puissance en fonction de la taille des images pour l'AGX : 8×ARM Carmel 12 nm.

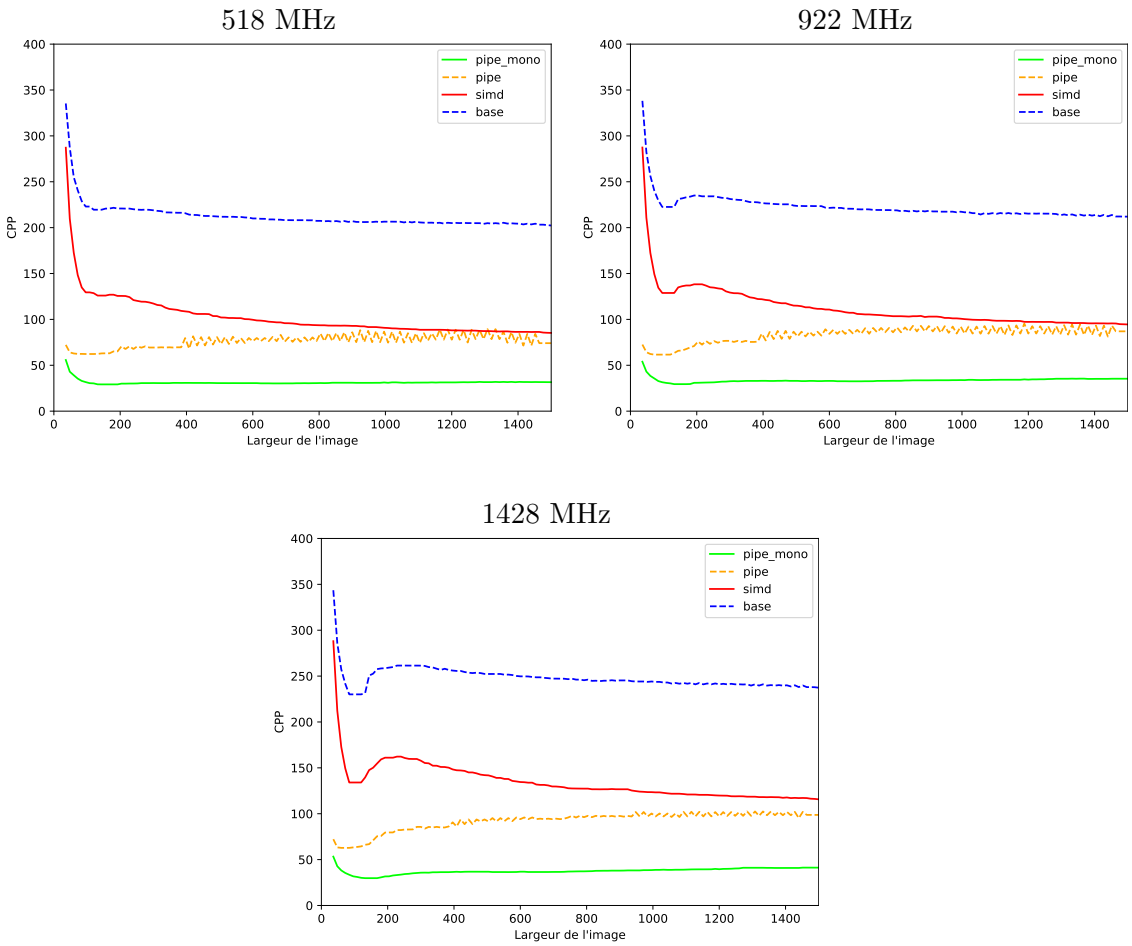


FIGURE A.4 – CPP en fonction de la taille des images pour la NANO : 4×ARM A57 12 nm.

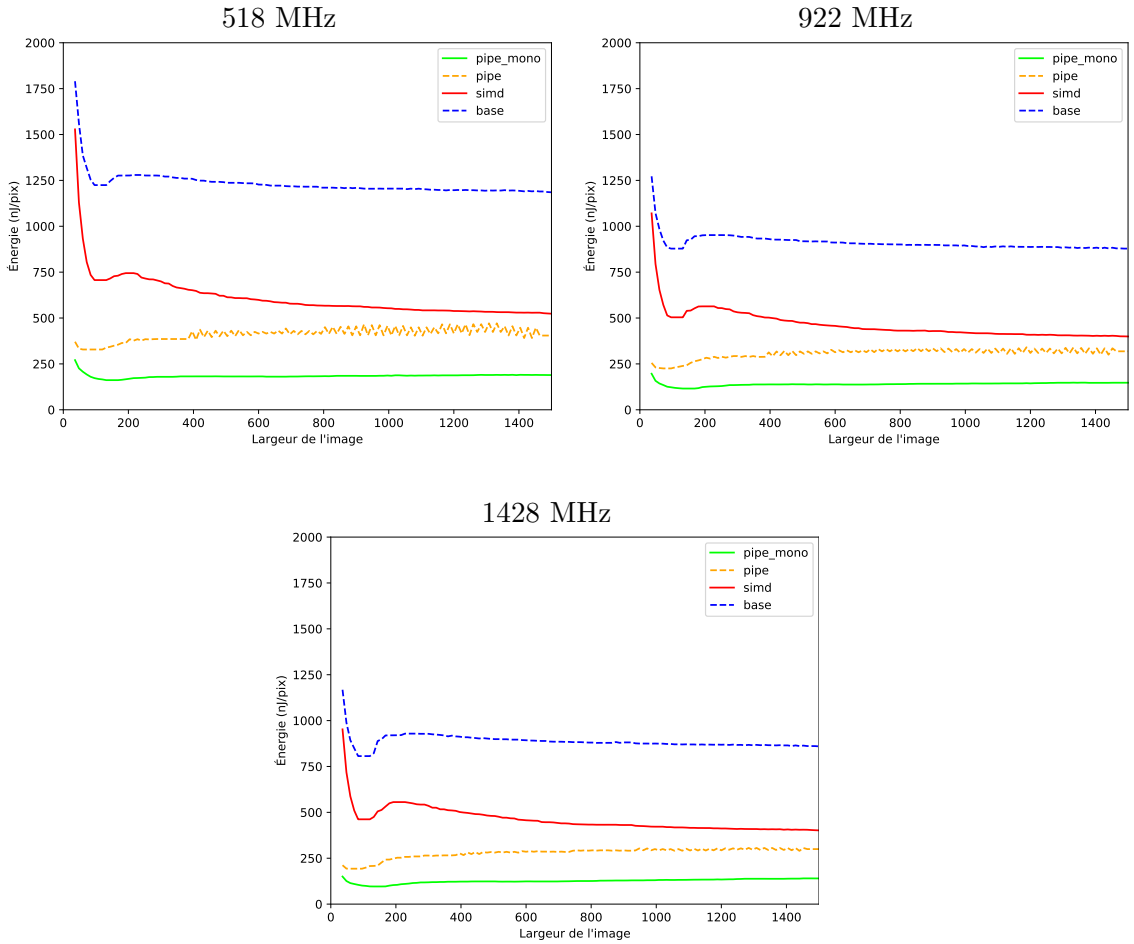


FIGURE A.5 – Énergie en fonction de la taille des images pour la NANO : 4×ARM A57 12 nm.

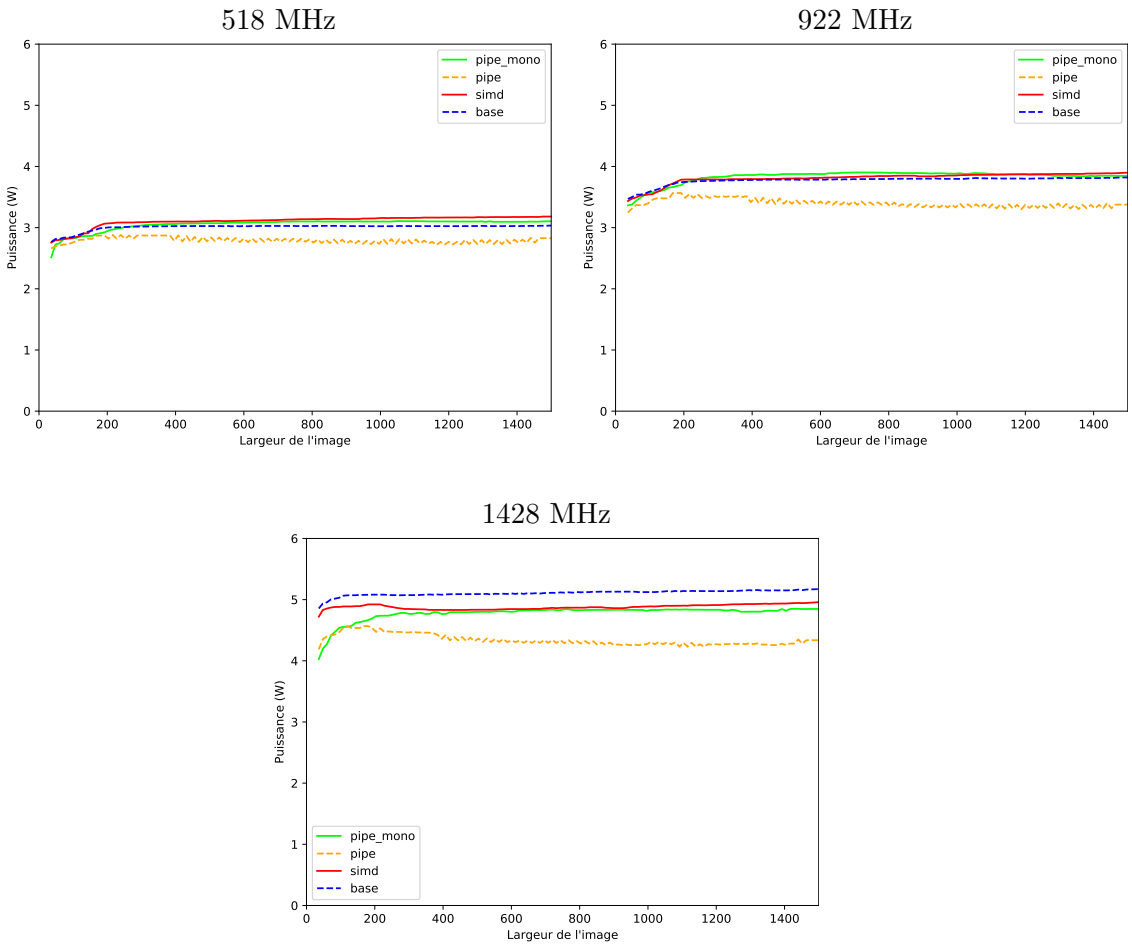


FIGURE A.6 – Puissance en fonction de la taille des images pour la NANO : 4×ARM A57 12 nm.

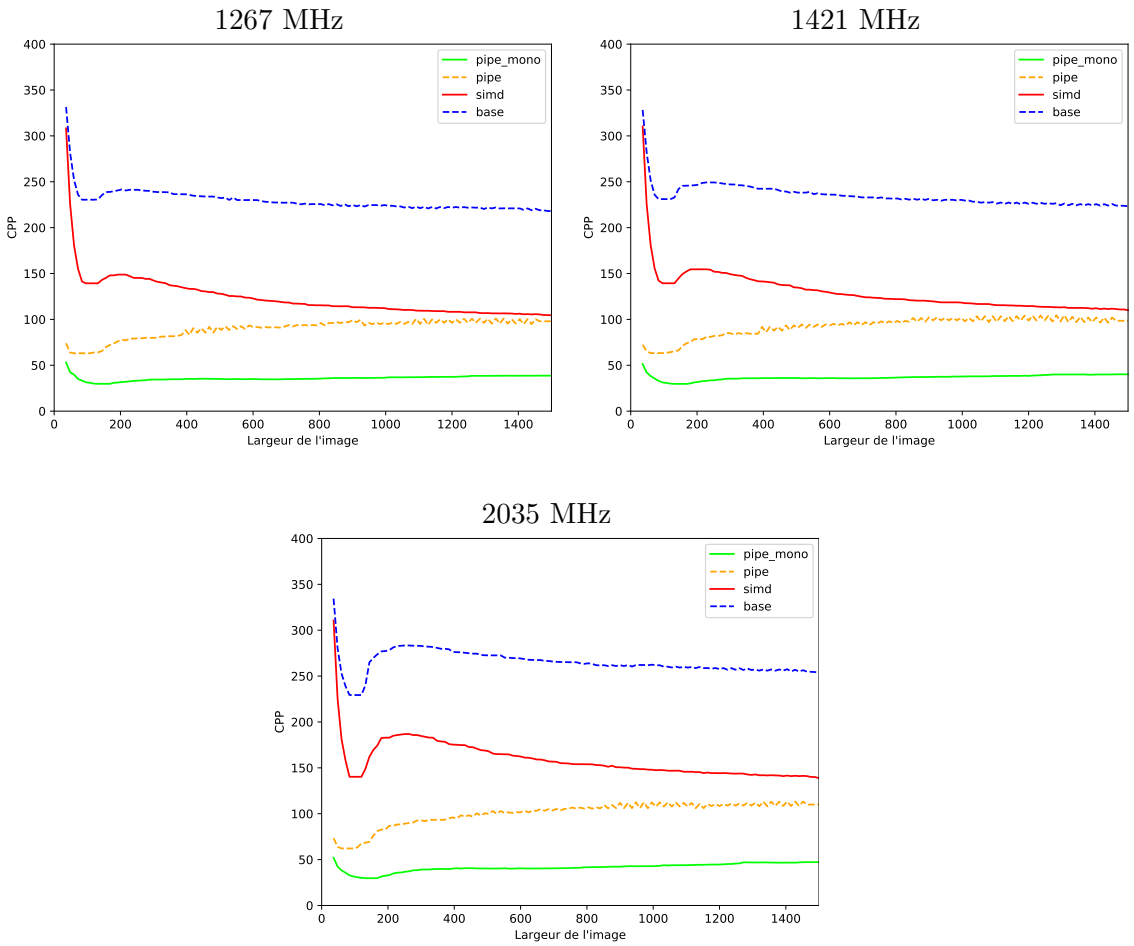


FIGURE A.7 – CPP en fonction de la taille des images pour la TX2 : 4×ARM A57 16 nm.

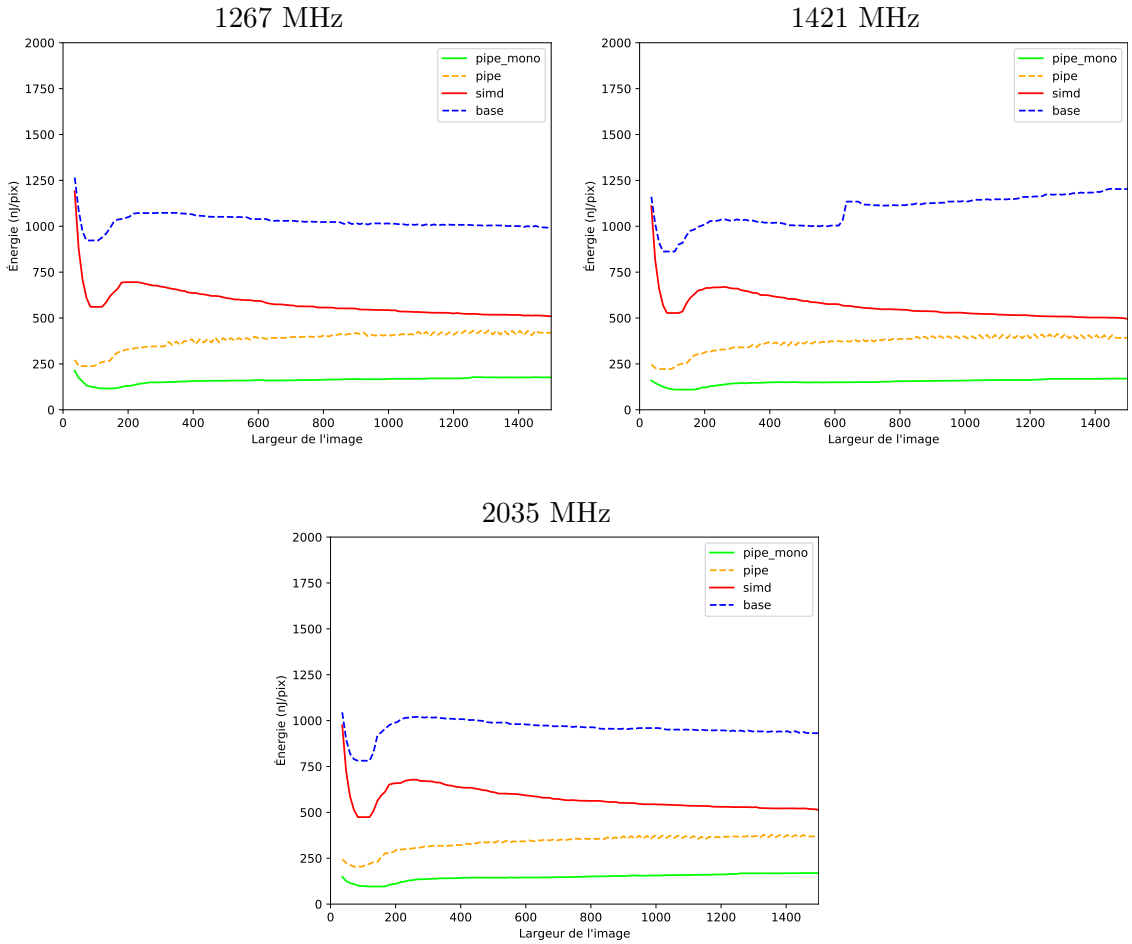


FIGURE A.8 – Énergie en fonction de la taille des images pour la TX2 : 4×ARM A57 16 nm.

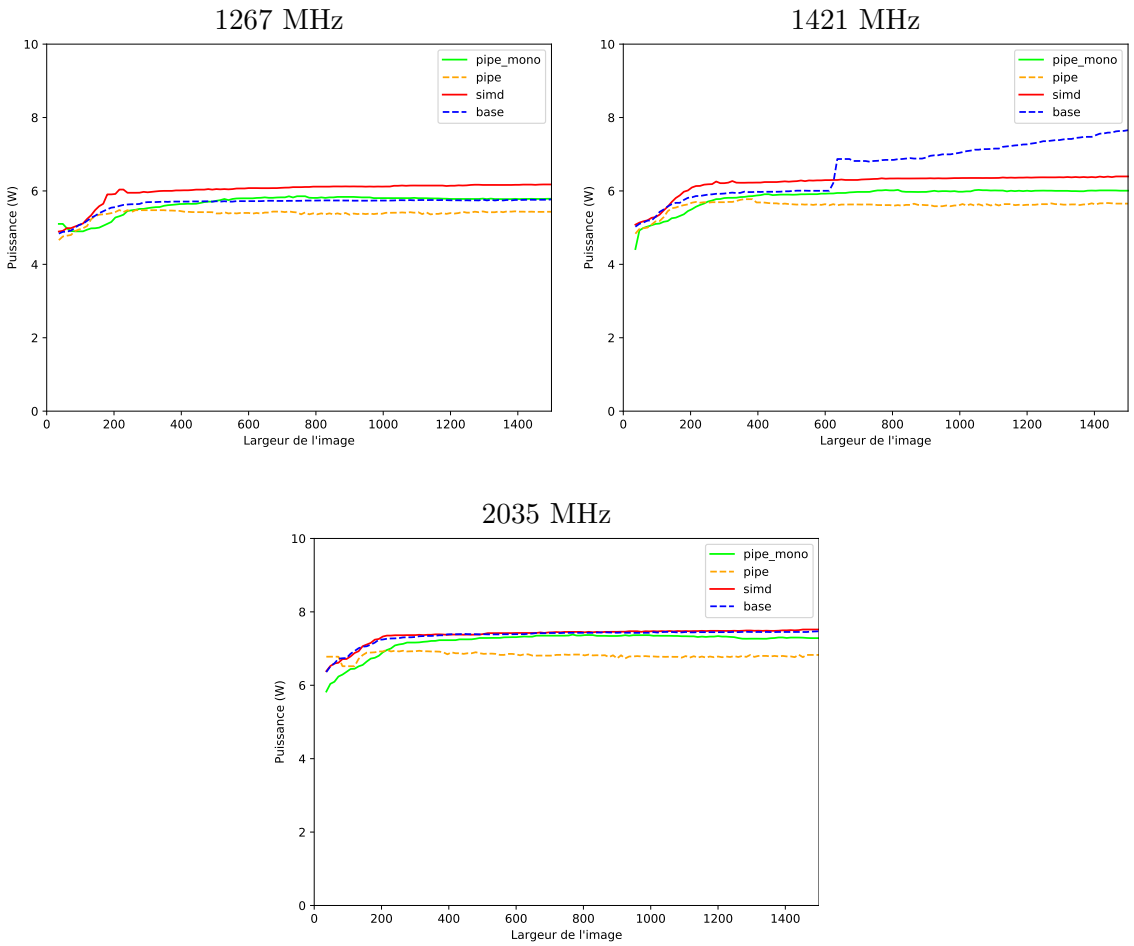


FIGURE A.9 – Puissance en fonction de la taille des images pour la TX2 : 4×ARM A57 16 nm.

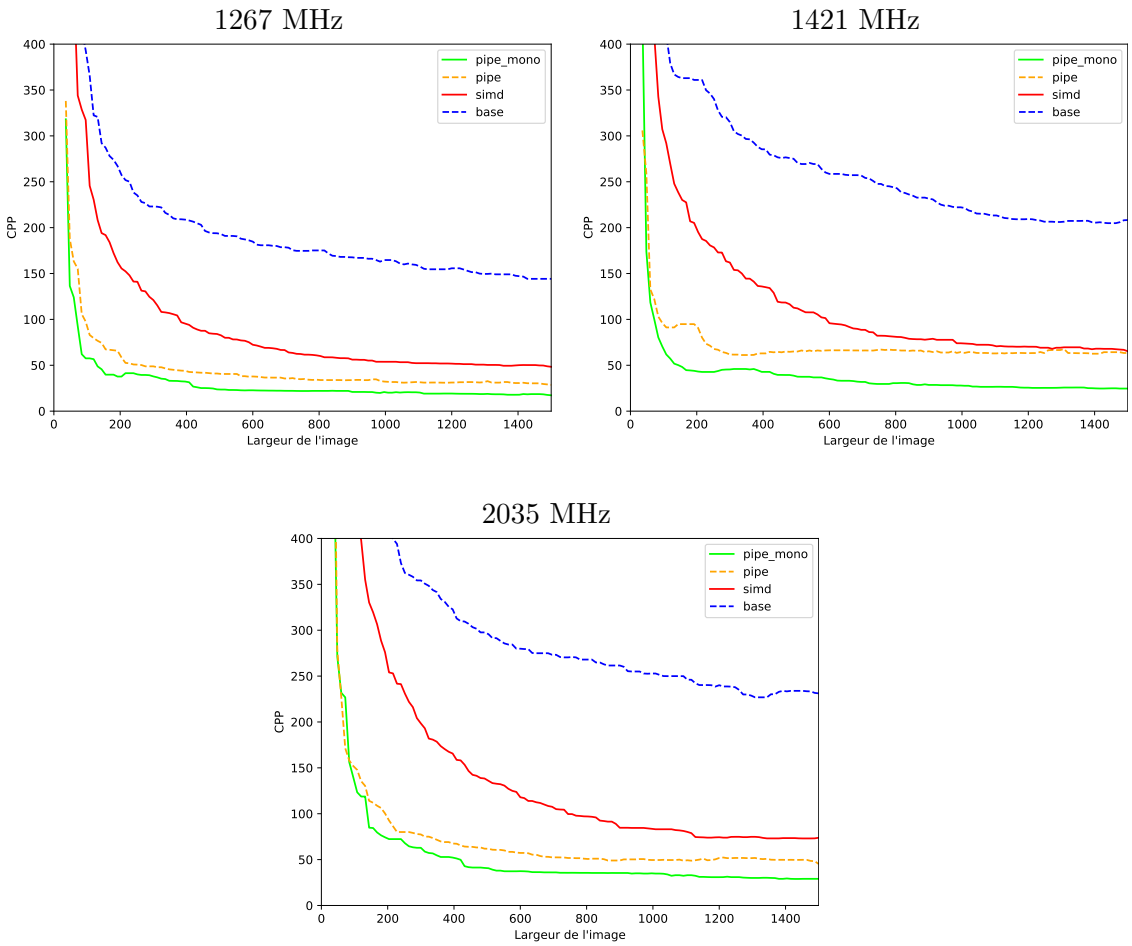


FIGURE A.10 – CPP en fonction de la taille des images pour la TX2 : 2×ARM Denver 16 nm.

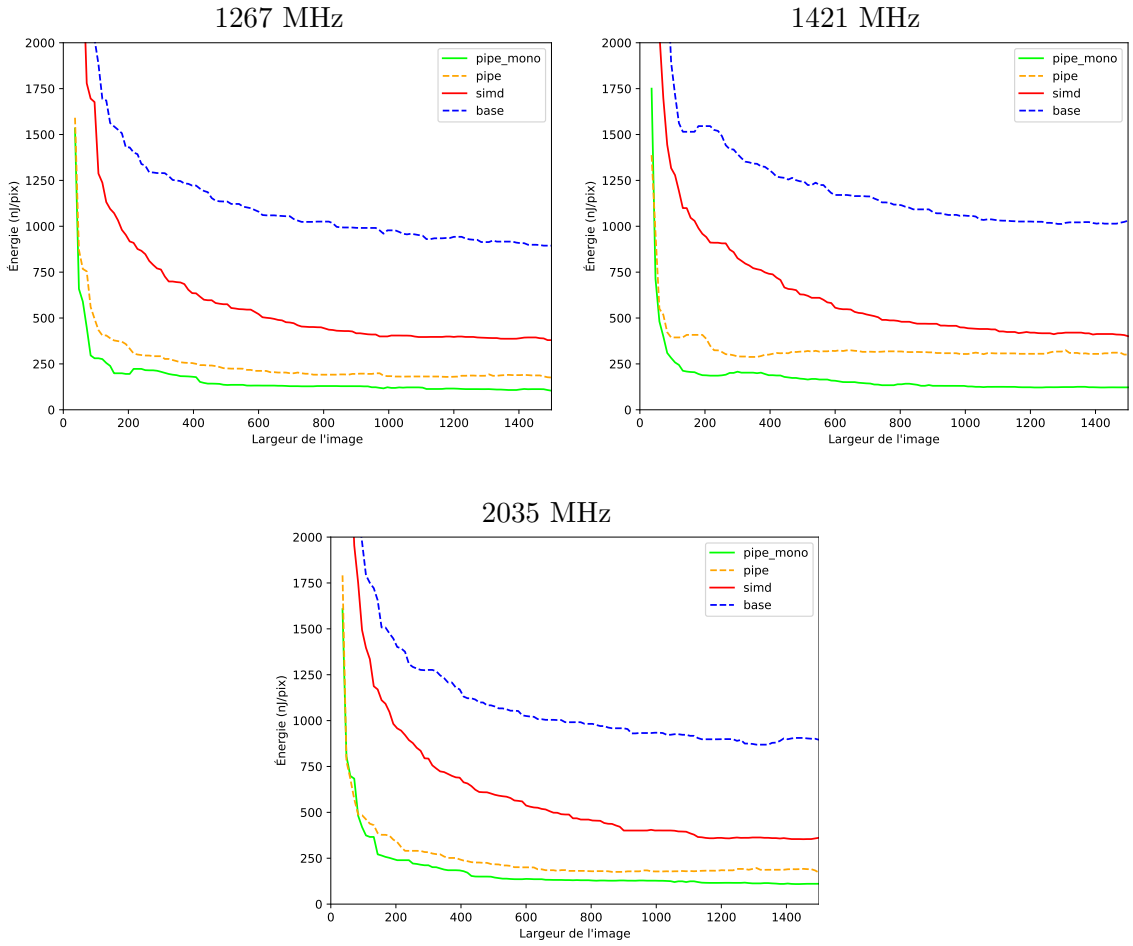


FIGURE A.11 – Énergie en fonction de la taille des images pour la TX2 : 2×ARM Denver 16 nm.

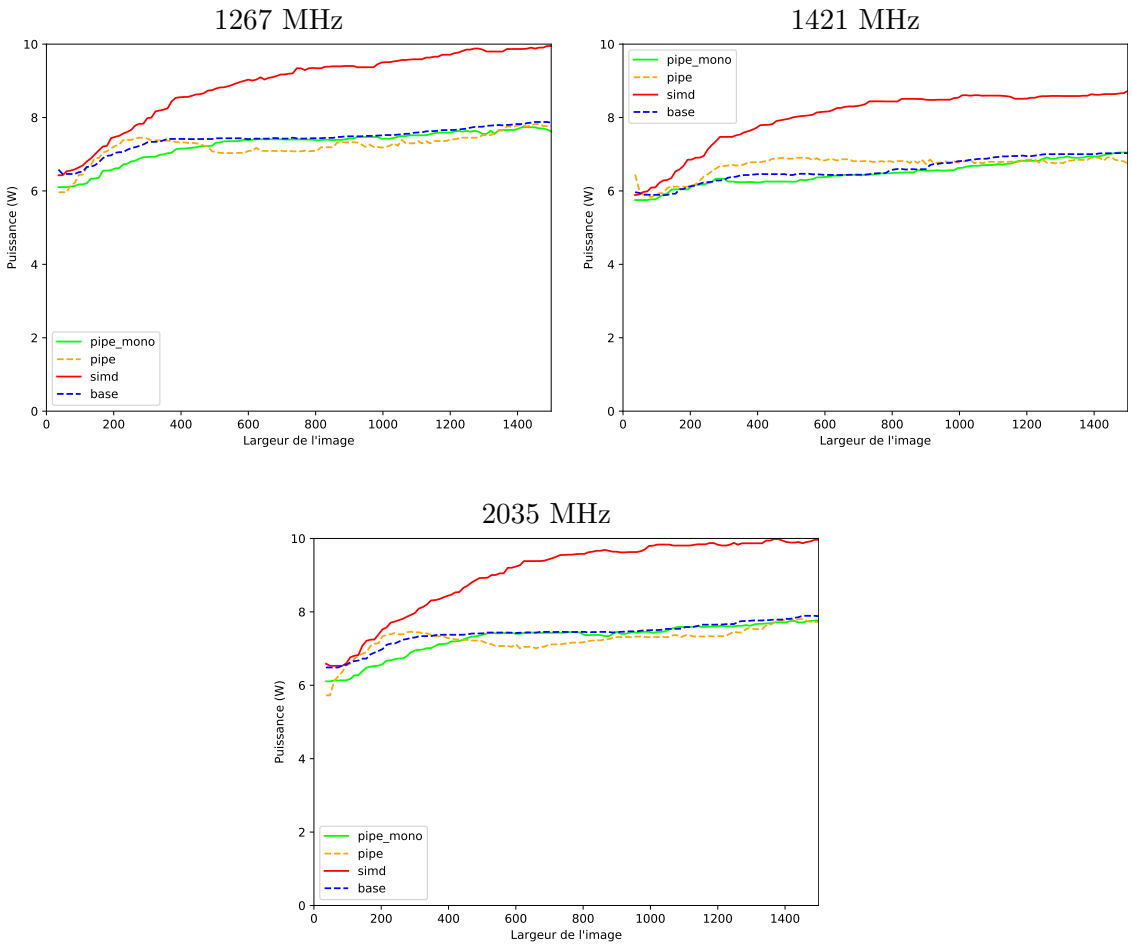


FIGURE A.12 – Puissance en fonction de la taille des images pour la TX2 : 2×ARM Denver 16 nm.

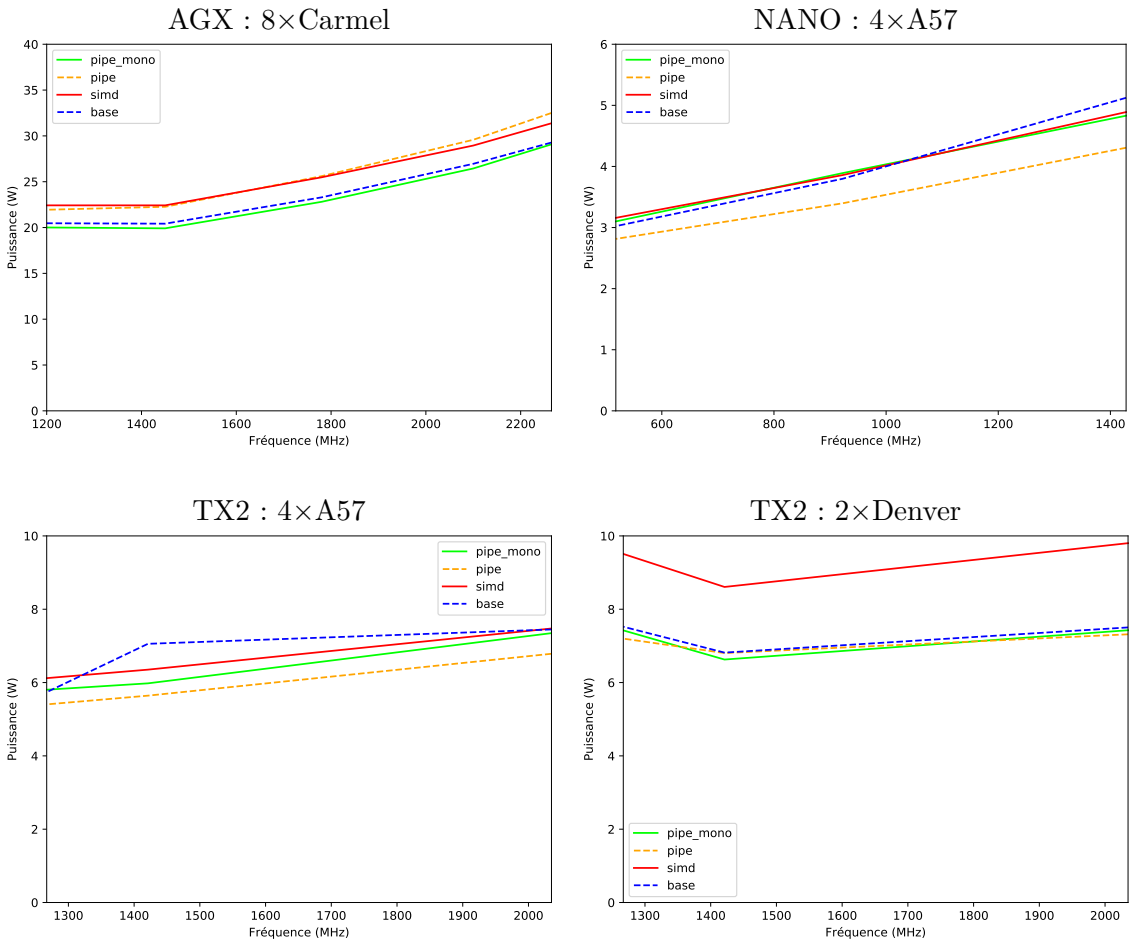


FIGURE A.13 – Puissance en fonction de la fréquence du CPU. Résultats pour des images de dimension 1008×1008.