Parallélisation d'une nouvelle application embarquée pour la détection automatique de météores

Mathuran Kandeepan, Clara Ciocan, Adrien Cassagne, Lionel Lacassagne

Compas'2023 : Parallélisme / Architecture/ Système
LISTIC/USMB - Annecy France, 4-7 juillet 2023

Parallélisation d'une nouvelle application embarquée pour la détection automatique de météores

Mathuran Kandeepan, Clara Ciocan, Adrien Cassagne et Lionel Lacassagne

Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
prenom.nom@lip6.fr

Résumé
Cet article présente les techniques mises en œuvre pour paralléliser une nouvelle chaîne de traitement de l'image. L'application permet de détecter automatiquement les météores depuis un flux vidéo non stabilisé. La cible finale de cette chaîne de traitement est un système sur puce à faible consommation énergétique pouvant être embarqué sur un ballon-sonde, ou bien dans un avion. Les méthodes mises en place pour répondre aux contraintes de traitement temps réel ($\geq$ 25 images par seconde) tout en restant sur des niveaux de consommation énergétique faible ($\leq$ 10 Watts) sont présentées. En outre, la chaîne de traitement est découpée en graphe de tâches puis elle est parallélisée. Les résultats obtenus démontrent l'efficacité de la parallélisation. Par exemple, sur la Raspberry Pi 4, la chaîne de traitement atteint 42 images par seconde et consomme moins de 6 Watts pour des séquences vidéos Full HD.

Mots-clés : Vision par ordinateur, systèmes embarqués, parallélisme, multi-thread, pipeline.

1. Introduction

Avec l'augmentation des missions spatiales, la détection de météores est devenue nécessaire afin d'évaluer le flux d'objets rentrant dans l'atmosphère et pour planifier en toute sécurité les activités spatiales. Aujourd'hui, il est possible d'acquérir des heures de vidéos contenant des météores en positionnant une caméra soit sur Terre et dirigée vers le ciel, soit à bord d'un avion volant à haute altitude [16], soit embarquée sur un ballon-sonde [13]. Cependant, les apparitions de météores représentent une infime partie de la durée totale des enregistrements vidéos. L'exploitation de ces vidéos ainsi que l'analyse image par image peut être fastidieuse pour les astronomes. Une automatisation de cette détection est donc nécessaire, d'où le développement de la chaîne de traitement d'images intitulée Fast Meteor Detection Toolbox (FMDT) [1].
Ce projet universitaire, développé par Sorbonne Université en collaboration avec l'Institut de Mécanique Céleste et de Calcul des Éphémérides (IMCCE), a plusieurs enjeux majeurs. Pour les astrophysiciens, étudier les météores peut apporter des réponses concernant les propriétés des objets dans le système solaire [3]. De plus, déployer un système embarqué à bas coût détectant les météores dans l'atmosphère serait une avancée significative.
En effet, envoyer un ballon-sonde équipé d'une caméra et d'un système sur puce n'est pas très coûteux (quelques milliers d'euros). **La chaîne de traitement doit être capable d'analyser un flux vidéo *Full HD* (1920×1200 pixels) à 25 images/seconde** pour satisfaire la contrainte

1. https ://github.com/alsoc/fmdt

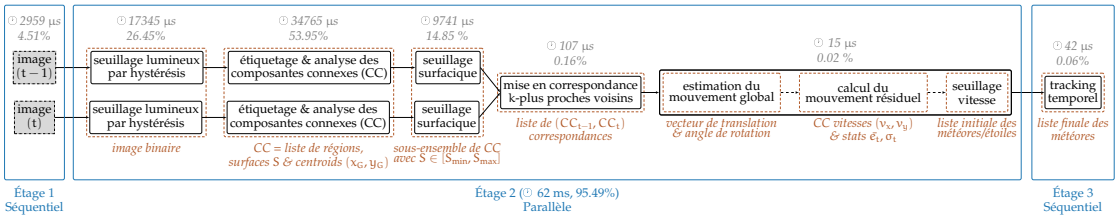
de temps réel. Les ressources en énergie nécessaires au fonctionnement du système sont aussi limitées. À titre d'exemple, dans un projet universitaire de nanosatellite [15, 11], **la puissance instantanée disponible pour la chaîne de traitement est inférieure à 10 Watts**. Pour répondre à ce défi, la chaîne de détection doit être rapide et efficace.
Les architectures CPU modernes sont majoritairement multi-cœur. **Il est indispensable d'exploiter le parallélisme multi-thread** pour en tirer parti. Il existe plusieurs modèles de programmation pour adresser ce type de parallélisme. Le plus commun repose sur l'utilisation d'une bibliothèque de threads (ex. : les threads POSIX). Il est aussi possible d'utiliser des directives à la compilation avec OPENMP par example. Cette solution a pour avantage d'être très concise. Des langages comme OPENCL permettent de programmer une grille de threads et de s'abstraire de l'architecture (CPU, GPU, FPGA, etc.). Enfin, il existe aussi des solutions spécifiques à un domaine (*Domain Specific Langages* ou DSL en anglais). Les DSL proposent des constructeurs parallèles spécifiques à une classe d'applications. Dans cet article, la parallélisation repose sur de **la programmation à base de tâches**, et plus précisément, c'est le DSL AFF3CT [4, 5] qui est utilisé. Ce dernier se présente sous la forme d'une bibliothèque (DSL enfoui) et est conçu pour supporter les applications de type *streaming* (incluant le traitement vidéo). Enfin, AFF3CT est une bibliothèque C++ et il est naturel d'encapsuler FMDT qui est écrit en langage C.
La contribution principale de cet article est la parallélisation *multi-thread* de la chaîne de traitement FMDT. Dans un premier temps le système est décrit en un graphe de tâches. Ensuite, plusieurs décompositions de ce graphe de tâches sont étudiées. Enfin, deux techniques de parallélisation sont implémentées et combinées :

1. le travail à la chaîne (noté *pipeline* par la suite),
2. la réplication des tâches (aussi appelé parallélisme *fork-join* dans la littérature anglaise).

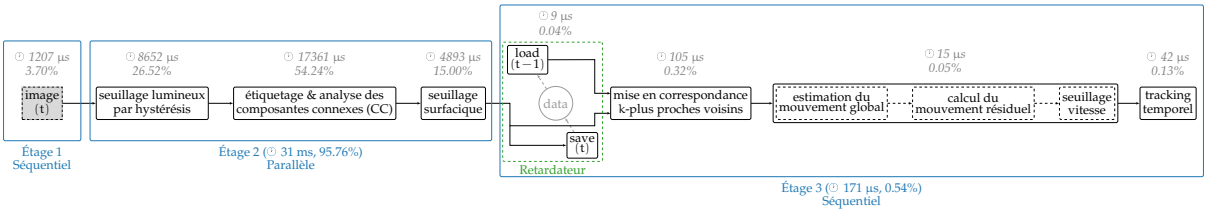
Différentes configurations d'affectations des threads aux cœurs physiques sont évaluées.
Cet article est organisé comme suit. La Section 2 présente un état de l'art des applications de détection de météores existantes. La Section 3 introduit la version initiale de la chaîne de détection, ainsi que les premières optimisations apportées pour satisfaire les contraintes d'embarquabilité. La Section 4 décrit les expérimentations effectuées et les résultats obtenus. Enfin, la Section 5 conclut sur les travaux présentés.

2. Travaux connexes

Jusqu'à maintenant, la majorité des chaînes de traitement d'images développées sont exécutées depuis le sol avec des caméras fixes [7, 12, 1, 6]. Un des problèmes de la détection depuis la surface de la Terre est qu'elle est sensible aux perturbations météorologiques. C'est pour cela qu'il est intéressant de déployer des systèmes de détection depuis la haute atmosphère (au dessus des nuages). Ainsi, la caméra est en mouvement, à l'inverse de la détection sur Terre. Cela simplifie l'observation, mais complexifie la chaîne de traitement.
METEORIX [15] est le premier projet universitaire développé par Sorbonne Université dédié à la détection de météores depuis l'espace à bord d'un nanosatellite. Le projet est toujours en phase de définition du système. Le nanosatellite doit embarquer une caméra dans le domaine du visible et un ordinateur exécutant une chaîne de traitement d'images pour la détection en temps réel. Avec une caméra dirigée vers la Terre, la chaîne de traitement s'appuie des méthodes basées sur le flot optique. Cependant, la chaîne de traitement implémentée n'est temps réel que jusqu'à la résolution HD (1280×720 pixels) [11] alors que la cible de cet article est la résolution Full HD (1920×1080 pixels).



(a) Chaîne de traitement initiale (version 1).



(b) Chaîne de traitement optimisée (version 2).

FIGURE 1 – Différentes décompositions de la chaîne de détection de météores (avec les étages E_1, E_2 et E_3 du pipeline). La latence de chaque tâche est indiquée en gris (1 cœur du Raspberry Pi 4 en Full HD).

SOURCE [10] est un autre projet de nanosatellite dédié à la démonstration d'observations de météores depuis l'espace et à la quantification du flux de météores. En utilisant également le flot optique, la chaîne de traitement obtient de bons résultats de détection mais n'atteint pas la cadence temps réel (1.25 images/seconde) sur des vidéos HD [14].
Dans les projets METEORIX et SOURCE, la caméra est dirigée vers la Terre et l'algorithme de flot optique permet de différencier plusieurs mouvements : la rotation de la Terre, le mouvement en orbite de la caméra, les éclairs, les villes et les objets entrant dans l'atmosphère (dont les météores et les débris spatiaux). Si ce type d'algorithme est efficace, il est aussi coûteux en temps de calcul. Dans cet article, contrairement à METEORIX et SOURCE, la caméra regarde au *limbe* : son axe est tangentiel à la Terre, la scène observée comprend l'espace ainsi que l'atmosphère où sont visibles les météores ainsi que certaines étoiles.

3. Contributions

3.1. Description de la chaîne de traitement

La chaîne de traitement développée utilise différents algorithmes efficaces d'un point de vue calculatoire. La Figure 1a présente la chaîne de traitement utilisée pour la détection de météores. Elle prend en entrée un flux d'images en niveaux de gris et retourne en sortie la liste des météores.
La première étape consiste à appliquer un seuillage lumineux sur toute l'image pour distinguer les régions d'intérêt (météores, étoiles, planètes, satellites, ...). Un seuillage par hystérésis est utilisé. Ce dernier consiste en un filtrage des pixels inférieurs à un seuil bas. Parmi les régions d'intérêt restantes, seules celles contenant au moins un pixel supérieur à un seuil haut sont retenues.
La deuxième étape consiste à passer d'une représentation binaire de l'image à une liste de régions appelées des composantes connexes (CC) via un algorithme d'étiquetage en composantes connexes (ECC). L'algorithme *Light Speed Labeling* [8] est utilisé car il combine étiquetage et analyse en composantes connexes (ACC). Dans notre cas, les caractéristiques des régions sont la surface S en pixels, le centre d'inertie de coordonnées (x_G, y_G) et le rectangle englobant $([x_{min}, x_{max}] \times [y_{min}, y_{max}])$ de chaque CC.

La troisième étape est un seuillage surfacique ne gardant que les CC dont la surface est comprise entre S_{min} et S_{max}. Ces trois premières étapes sont appliquées sur deux images successives I_{t-1} et I_t pour récupérer deux ensembles de CC.
La quatrième étape est la mise en correspondances (algorithme des k-*plus proches voisins* ou k-PPV) des CC de deux images consécutives. Pour chaque CC de l'image I_t ayant trouvé une association dans I_{t-1}, un vecteur distance est calculé.
La cinquième étape estime le mouvement global (un vecteur translation $\vec{T}$ et un angle de rotation θ) de l'image du au balancement et au mouvement de la caméra [2].
La sixième étape recale l'image I_t sur l'image I_{t-1}. Après recalage, les couples de CC dont la distances est quasi nulle représentent des étoiles. À contrario, les couple de CC dont les distances sont significativement plus grandes sont des régions en mouvement et potentiellement des météores. Ce vecteur distance représente alors une estimation de la vitesse de l'objet.
La septième étape applique un seuillage en fonction de la vitesse de chaque CC pour supprimer les objets immobiles (étoiles, planètes, etc.).
La huitième et dernière étape est le *tracking* temporel. Si une CC est détectée en mouvement sur au moins 3 images, alors une piste est créée.

3.2. Optimisations

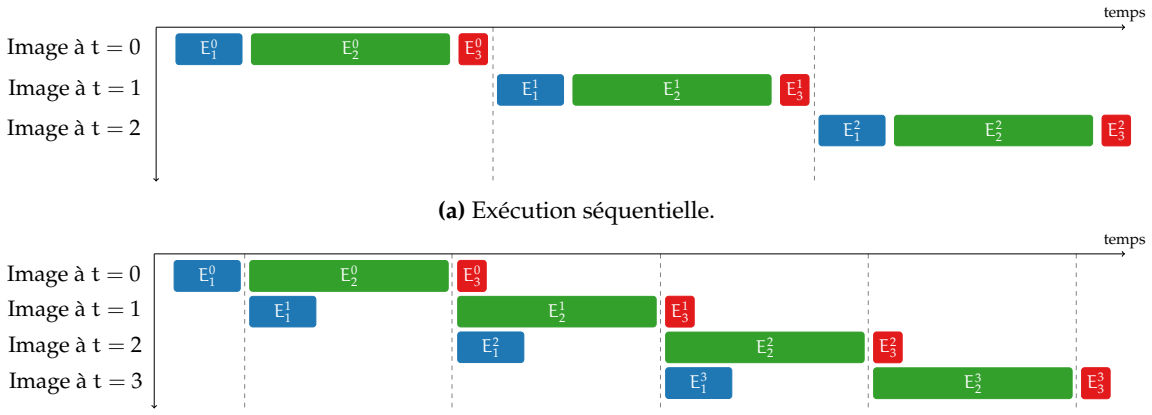
3.2.1. Décomposition en graphe de tâches

Sur la Figure 1a, le seuillage lumineux, l'ECC, l'ACC et le seuillage morphologique sont appliqués sur les images I_{t-1} et I_t. Or, à $t+1$, les mêmes traitements sont ré-appliqués sur l'image I_t. On remarque donc que l'on calcule inutilement deux fois les CC sur l'image I_t. Étant donné que ces traitements sont coûteux, il est préférable de ne pas les recalculer et de mémoriser les CC pour l'itération suivante. La Figure 1b illustre la nouvelle chaîne de traitement. Un couple de tâches, appelé *Retardateur*, a été ajouté entre le seuillage morphologique et k-PPV. Après le seuillage, la tâche *load* est exécutée. Cette dernière lit les CC correspondants à l'image I_{t-1}. Ensuite, la tâche *save* est exécutée. Elle sauvegarde les CC correspondants à l'image I_t pour le traitement de la future image à $t+1$. Enfin, la tâche k-PPV peux s'exécuter avec les bonnes entrées : les CC à $t-1$ et les CC à t.
Dans la suite de cet article, le graphe de tâches correspondant à la Figure 1a est référencé comme la version 1 et le graphe de tâches correspondant à la Figure 1b est référencé comme la version 2. Sur les deux graphes de tâches, le pourcentage de temps et le temps de chaque tâche est indiqué. Ces temps ont été mesurés depuis une exécution séquentielle sur la Raspberry Pi 4. Pour la version 1, la latence est de 65 ms alors que pour la version 2, la latence est de 32 ms. La version 2 est donc $2\times$ plus rapide que la version 1.

3.2.2. Parallélisation du graphe de tâches

Pipeline

En séquentiel, les tâches de la chaîne de traitement sont exécutées les unes après les autres. Ce type d'exécution est illustré par la Figure 2a. Dans un contexte multi-cœur, il est possible de regrouper des tâches consécutives pour former des *étages* et appliquer le principe du travail à la chaîne (*pipeline*). Lors d'une exécution pipelinée, l'étage E_e^{t+1} peut s'exécuter en même temps que l'étage E_{e+1}^t (avec e le numéro de l'étage et t le numéro temporel de l'image à traiter). Ce type de parallélisme a pour avantage de conserver les dépendances de données. En régime permanent, cela se traduit par une exécution en parallèle de tous les étages sur différents threads (voir Figure 2b). Le débit théorique est alors celui de l'étage le plus lent. La Figure 2 montre que l'exécution pipelinée atteint un débit plus élevé que l'exécution séquentielle. Dans ce travail, la



(a) Exécution séquentielle.

(b) Exécution à la chaîne (parallélisme *pipeline*).

FIGURE 2 – Exécutions séquentielle et pipelinée (avec E_1^t, E_2^t et E_3^t, les étages 1, 2 et 3 à l'instant t).

chaîne de traitement a été découpée en 3 étages (voir Figure 1).

Réplication de tâches

Dans la chaîne de détection étudiée, certaines tâches peuvent être exécutées en parallèle sur différents threads et sur différentes images. C'est notamment le cas des tâches de l'étage E_2 dans la Figure 1. C'est ce que l'on appelle le parallélisme de données.
Ainsi, il est possible d'exécuter des tâches sur plusieurs cœurs, de sorte que l'exécution se ramifie à certain points du graphe pour se rejoindre et reprendre l'exécution séquentielle ensuite. C'est ce qu'on appelle la réplication de tâches. En ramifiant l'exécution de l'étage E_2, soit l'étage dont la latence est la plus élevée, il est possible d'augmenter le débit.

Implémentation avec la bibliothèque AFF3CT

AFF3CT permet de décrire des applications selon le formalisme de *flux de données*. Pour cela, plusieurs composants sont définis : la *séquence*, la *tâche* et la *socket*. Une tâche est un traitement effectué sur des données. Chaque étape de la chaîne de traitement est une tâche. Les tâches sont caractérisées par des sockets. Une socket définit un chemin par lequel des tâches peuvent consommer et/ou produire des données. Une séquence est un ensemble de tâches exécutées avec un ordonnancement déterminé à sa construction.
De plus, AFF3CT vient avec un support d'exécution multi-thread comprenant une implémentation du pipeline pouvant être combinée à de la réplication automatique de séquence. Cela à la condition que toutes les tâches qui composent la séquence soient sans état interne. Une séquence correspond à un étage du pipeline.
Dans ces travaux, un pipeline à 3 étages a été instancié. Et, dans le second étage du pipeline, la réplication de tâches a été activée. Les synchronisations entre l'étage 1-2 et entre l'étage 2-3 ont été configurées avec de l'attente passive, permettant ainsi de maximiser l'utilisation des ressources au détriment d'un léger surcoût en latence. Lorsqu'un thread s'endort, le système d'exploitation peut ré-affecter le cœur correspondant à un autre thread actif. Enfin, la synchronisation entre les étages du pipeline est assurée par un algorithme de type producteur-consommateur. Il y a des *buffers* pour l'échange de données entre les threads. La taille de ces *buffers* a été fixée à 1, permettant ainsi de minimiser la latence sans qu'il n'y ait d'impact significatif sur le débit. Cela est vrai pour un petit nombre de cœurs homogènes ($n \leq 4$).

Réf.	Nom complet	Date	Gravure	CPUs	Fréq.	RAM (taille & débit)	
XU4	Hardkernel Odroid-XU4	2016	28 nm	4 × *LITTLE* ARMv7 Cortex-A7 4 × *Big* ARMv7 Cortex-A15	1.4 GHz 1.5 GHz	2 GB	3.5 GB/s
RPi4	Raspberry Pi 4 model B	2019	28 nm	4 × *Big* ARMv8 Cortex-A72	1.5 GHz	8 GB	3.9 GB/s
Nano	Nvidia Jetson Nano	2019	20 nm	4 × *Big* ARMv8 Cortex-A57	≈ 1.5 GHz	4 GB	9.0 GB/s
M1	Apple Silicon M1 Ultra	2022	5 nm	4 × *E-core* ARMv8 Icestorm 16 × *P-core* ARMv8 Firestorm	≈ 2.0 GHz ≈ 3.0 GHz	64 GB	344.0 GB/s

TABLE 1 – Spécifications des différents SoC évalués.

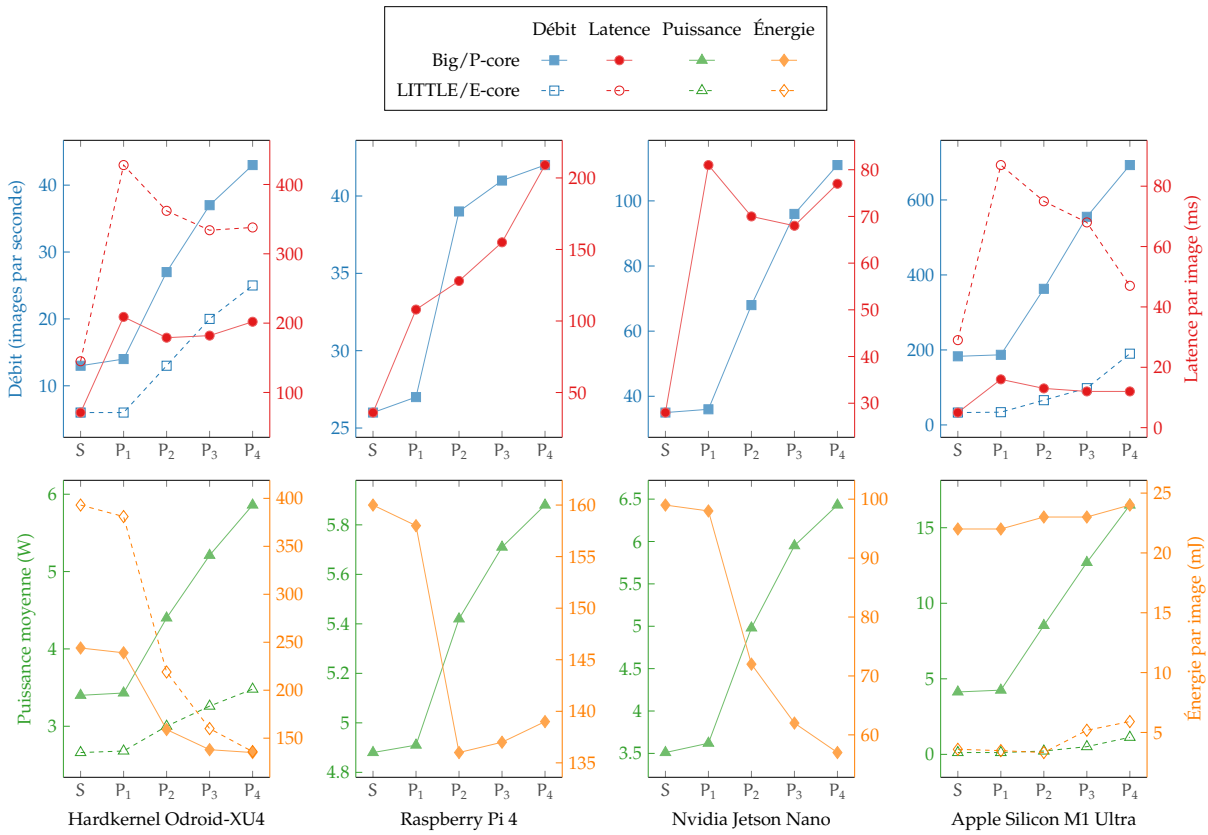


FIGURE 3 – Performances Full HD de la chaîne de détection (version 2, c.f. Figure 1b) en terme de débit, latence, puissance moyenne et énergie consommée pour quatre cibles embarquées.

4. Expérimentations et résultats

La Table 1 présente les différentes architectures embarquées évaluées. La XU4 et le M1 sont hétérogènes et disposent : (1) de cœurs efficaces du point de vue énergétique (*LITTLE* ou *E-core*) et (2) de cœurs puissants/rapides (*Big* ou *P-core*). Le code est compilé avec GCC v9.4.0 et avec les options d'optimisations suivantes : `-O3 -funroll-loops -march=native`. Toutes les expérimentations ont été faites sur une vidéo Full HD où 100% des météores sont détectés [16]. Enfin, seule la version 2 de la chaîne de traitement est considérée (voir Figure 1b).
Les résultats obtenus sont étudiés au travers de 4 métriques : le débit moyen $\mathcal{D}$ en nombre d'images par seconde, la latence moyenne $\mathcal{L}$ pour traiter une image, la puissance moyenne $\mathcal{P}$ en Watts, la consommation énergétique moyenne $\mathcal{E}$ pour traiter une image en milliJoules (mJ).
La Figure 3 montre les résultats obtenus selon les 4 métriques énoncées précédemment. La complexité calculatoire ne varie pas (ou très peu) en fonction des images. En effet, les scènes

observées sont presque identiques : quelques petites régions lumineuses (les étoiles ou les météores) sur un fond majoritairement foncé. Pour que les mesures soient significatives, chaque expérience est exécutée pendant 30 secondes.
$\mathcal{D}$, $\mathcal{L}$, $\mathcal{P}$ et $\mathcal{E}$ sont étudiées en fonction de l'exécution séquentielle (notée S) et de plusieurs exécutions pipelinées (notées P_i) de la chaîne de traitement. Lors d'une exécution pipelinée, 1 thread est affecté à l'étage E_1 et un autre thread est affecté à l'étage E_3. Le nombre de threads de l'étage E_2 peut varier grâce à la réplication (voir Section 3.2.2). Ainsi, i indique le nombre de threads affectés à l'étage E_2. Par exemple P_3 signifie qu'il y a 1 thread pour l'étage E_1, 3 threads pour l'étage E_2 et 1 thread pour l'étage E_3 (soit 5 threads au total). En fonction du nombre de cœurs physiques de la cible, il peut y avoir plus de threads que de cœurs physiques.

Débit et latence

La Figure 3 montre que le pipeline P_1 n'apporte pas un gain significatif en terme de débit par rapport à la version séquentielle S. De plus, la latence de P_1 est triplée par rapport à une exécution séquentielle. Pour la détection de météores, la latence n'est pas critique tant que l'on peut mettre quelques images (< 10) en attente dans une mémoire tampon (la taille de la RAM des SoC évalués est largement suffisante). Pour des instances de P_i avec $i \geq 2$, les débits sont nettement meilleurs que la version S. La réplication dans l'étage E_2 du pipeline a un impact direct sur le débit final. En fonction des cibles, la latence peut soit diminuer légèrement lorsque i grandit (XU4, Nano et M1) soit, au contraire, augmenter (RPi4). Sur la RPi4, la bande passante de la mémoire globale est trop faible pour alimenter efficacement plus de 2 cœurs dans l'étage E_2. Pour toutes les cibles étudiées, il existe au moins une configuration qui tient la cadence temps réel de 25 images par seconde.

Puissance et consommation énergétique

Les mesures de puissance $\mathcal{P}$ sont faites à la prise d'alimentation pour la XU4, la RPi4 et la Nano. Pour le M1, c'est l'outil `powermetrics` d'Apple qui est utilisé. Ce dernier mesure uniquement la consommation du CPU. L'énergie $\mathcal{E}$ est fonction de la puissance moyenne $\mathcal{P}$ et du débit $\mathcal{D}$. Sur la Figure 3, on constate que généralement quand le nombre de cœurs utilisé augmente alors la puissance moyenne $\mathcal{P}$ augmente. C'est ce qui est attendu. Pour la consommation énergétique, on observe plutôt la tendance inverse : plus il y a de ressources, moins on consomme d'énergie. Cependant, cela n'est pas toujours vrai, par exemple pour la RPi4 et pour le M1, on perd en efficacité à partir de 3 cœurs dans l'étage E_2. Comme la contrainte en débit est respectée pour 2 cœurs, il n'est pas intéressant d'affecter plus de cœurs à cet étage. À l'exception des *P-cores* du M1, toutes les cibles respectent la contrainte de puissance fixée à 10 Watts.

5. Conclusion et perspectives

Dans cet article, des méthodes de parallélisation (pipeline et réplication) ont été présentées puis implémentées sur une nouvelle chaîne de détection de météores. L'application a été évaluée sur plusieurs cibles embarquées. Grâce à l'implémentation multi-threads, toutes les cibles respectent les contraintes en terme de débit et de puissance sur des séquences vidéos Full HD. Sur certaines cibles (Nvidia Jetson Nano et Apple M1 Ultra), il est envisageable de traiter des résolutions plus importantes comme la QHD (2560×1440 pixels). Dans des travaux futurs, il serait possible de combiner du code CPU SIMD [9] pour l'étiquetage en composantes connexes avec de la programmation sur GPU pour certains traitements réguliers comme les seuillages.

Bibliographie

1. Audureau (Y.), Marmo (C.), Bouley (S.), Kwon (M.-K.), Colas (F.), Vaubaillon (J.), Birlan (M.), Zanda (B.), Vernazza (P.), Caminade (S.) et al. – Freeture : A free software to capture meteors for fripon. – In *International Meteor Conference*, pp. 39–41, 2014.
2. Bloch (I.). – Recalage et fusion d'images médicales, 2020. https ://perso.telecom-paristech.fr/bloch/P6Image/recalageImMed.pdf.
3. C. Hongru (N. R.) et Vaubaillon (J.). – Accuracy of meteor positioning from space- and ground-based observations. 2020.
4. Cassagne (A.), Hartmann (O.), Léonardon (M.), He (K.), Leroux (C.), Tajan (R.), Aumage (O.), Barthou (D.), Tonnellier (T.), Pignoly (V.), Le Gal (B.) et Jégo (C.). – AFF3CT : A fast forward error correction toolbox ! *Elsevier SoftwareX*, vol. 10, 2019, p. 100345.
5. Cassagne (A.), Tajan (R.), Aumage (O.), Barthou (D.), Leroux (C.) et Jégo (C.). – A DSEL for high throughput and low latency software-defined radio on multicore CPUs. *Wiley Concurrency and Computation : Practice and Experience (CCPE)*, juillet 2023, p. e7820.
6. Colas (F.), Zanda (B.), Bouley (S.), Jeanne (S.), Malgoyre (A.), Birlan (M.), Blanpain (C.), Gattacceca (J.), Jorda (L.), Lecubin (J.) et al. – Fripon : a worldwide network to track incoming meteoroids. *Astronomy & Astrophysics*, vol. 644, 2020, p. A53.
7. Gural (P. S.). – An operational autonomous meteor detector : Development issues and early results. *WGN, Journal of the International Meteor Organization*, vol. 25, 1997, pp. 136–140.
8. Lacassagne (L.) et Zavidovique (B.). – Light Speed Labeling for RISC architectures. – In *International Conference on Image Analysis and Processing*, pp. 3245–3248. IEEE, 2009.
9. Lemaitre (F.), Hennequin (A.) et Lacassagne (L.). – How to speed Connected Component Labeling up with SIMD RLE algorithms. – In *Workshop on Programming Models for SIMD/Vector Processing*, pp. 1–8. ACM, 2020.
10. Liegibel (M.), Petri (J.), Hoffmann (P.), Geier (N.) et Klinkner (S.). – Meteor observation with the source cubesat–developing a simulation to test on-board meteor detection algorithms. – In *Symposium on Space Educational Activities*. Universitat Politècnica de Catalunya, 2022.
11. Millet (M.), Rambaux (N.), Cassagne (A.), Bouyer (M.), Petreto (A.) et Lacassagne (L.). – High performance computer vision application for Meteor detection from a cubesat. – In *Committee on Space Research*, 2022.
12. Molau (S.). – The meteor detection software metrec. – In *International Meteor Conference*, pp. 9–16, 1999.
13. Ocaña (F.), de Miguel (A. S.), Project (D.) et al. – Balloon-borne video observations of geminids 2016, 2019. arXiv :1911.10064.
14. Petri (J.). – *Satellite Formation and Instrument Design for Autonomous Meteor Detection*. – phdthesis, Universität Stuttgart, 2022.
15. Rambaux (N.), Vaubaillon (J.), Lacassagne (L.), Galayko (D.), Guignan (G.), Birlan (M.), Capderou (M.), Colas (F.), Deleflie (F.), Deshours (F.), Hauchecorne (A.), Keckhut (P.), Levasseurd-Regourd (A.), Rault (J.) et Zanda (B.). – Meteorix : a cubesat mission dedicated to the detection of meteors and space debris. – In *ESA Space Safety Programme Office, NEO and Debris Detection Conference*, pp. 1–9, 2019.
16. Vaubaillon (J.), Loir (C.), Ciocan (C.), Kandeepan (M.), Millet (M.), Cassagne (A.), Lacassagne (L.), da Fonseca (P.), Zander (F.), Buttsworth (D.), Loehle (S.), Tóth (J.), Gray (S.), Moingeon (A.) et Rambaux (N.). – A 2022 τ-herculid meteor cluster from an airborne experiment : Automated detection, characterization, and consequences for meteoroids. *Astronomy and Astrophysics*, 2023.