RAPPORT DU STAGE DE FIN D'ÉTUDES

PARALLÉLISATION ET OPTIMISATION D'APPLICATIONS DE TYPE STREAMING POUR LES SYSTÈMES EMBARQUÉS MULTI-CŒURS ET HÉTÉROGÈNES

Mathuran KANDEEPAN

Encadrant : Adrien CASSAGNE

Mars-Août 2023

Table des matières

Chapitre 1

Introduction

1.1 Organisme d'accueil

Le LIP6, intégré à Sorbonne Université et associé au Centre National de la Recherche Scientifique (CNRS), est un laboratoire de recherche en informatique jouant un rôle prépondérant dans la recherche en informatique au niveau national et international, créé en 1971 par la fusion de plusieurs laboratoires d'informatique. Les activités de recherche du LIP6 sont articulées autour de quatre axes majeurs :

— Intelligence Artificielle et Sciences des Données (AID) : ce premier axe se consacre à l'étude et au développement de technologies liées à l'intelligence artificielle, telles que l'apprentissage automatique, la vision par ordinateur et l'analyse de données. Il vise à exploiter au mieux les données massives pour des applications variées,

— Architecture, Systèmes et Réseaux (ASN) : l'axe ASN se concentre sur la conception et l'optimisation de systèmes informatiques, de l'architecture matérielle aux réseaux de communication. Son objectif est d'améliorer l'efficacité, la performance et la fiabilité des infrastructures informatiques,

— Sécurité, Sûreté et Fiabilité (SSR) : la sécurité informatique, la sûreté des systèmes et leur fiabilité sont au cœur de cet axe. Il cherche à identifier et à atténuer les vulnérabilités informatiques, garantissant ainsi la protection des données et des systèmes,

— Théorie et Outils Mathématiques pour l'Informatique (TMC) : l'axe TMC s'attache à développer des fondements mathématiques solides pour l'informatique. Il contribue à la création de modèles et d'outils formels pour résoudre des problèmes complexes.

Ce laboratoire joue un rôle clé dans la formation de jeunes chercheurs en informatique en proposant des programmes de doctorat, de master, et de formation continue. Les étudiants et les chercheurs bénéficient d'un matériel de pointe, et de bibliothèques spécialisées. Le stage se déroule au sein de l'équipe Architecture et Logiciels pour Systèmes Embarqués sur Puce (ALSOC), suivant l'axe ASN. L'équipe ALSOC se spécialise dans la recherche sur la conception de systèmes multiprocesseurs intégrés sur puce, mais aussi de l'adéquation entre algorithme et architecture matérielle. Ces systèmes sont spécifiquement conçus pour répondre aux exigences de performance de diverses applications embarquées, notamment le traitement de flux vidéo et multimédia dans

les appareils embarqués, ainsi que le traitement de paquets dans les équipements de télécommunication.

1.2 Contexte

De nos jours, les applications de type streaming occupent une place centrale dans notre mode de vie numérique, qu'il s'agisse de regarder des vidéos, d'écouter de la musique en streaming, de jouer à des jeux en ligne ou même de participer à des réunions virtuelles. Cependant, l'importance de ces applications est encore plus prononcée lorsque nous les envisageons dans le contexte des systèmes embarqués. Les systèmes embarqués, tels que les téléviseurs intelligents, les appareils mobiles, les systèmes de divertissement pour voiture et même les dispositifs médicaux, dépendent de la performance optimale des applications pour offrir des expériences utilisateur de haute qualité. Dans cette perspective, il devient impératif que ces applications soient non seulement fonctionnelles, mais également hautement performantes, car leur efficacité influence directement notre quotidien, notre productivité et même notre sécurité dans le monde de plus en plus connecté d'aujourd'hui.

Toutefois, malgré la disponibilité de systèmes embarqués de pointe, il est souvent remarquable de constater que nous n'exploitons pas pleinement leur potentiel. Bien que les systèmes embarqués modernes aient la capacité de gérer des tâches complexes avec une bonne efficacité, nous sommes parfois amenés à sous-utiliser leurs ressources. Il faudrait utiliser au maximum les ressources des architectures modernes, telles que les multiples cœurs CPU (Central Processing Unit) ou les unités de traitement GPU (Graphics Processing Unit).

Aujourd'hui, 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. Durant ce stage, la parallélisation repose sur de **la programmation à base de tâches**, et plus précisément, c'est le DSL AFF3CT [5, 6] 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 les traitements vidéos et communications).

1.3 Objectif

Le défi actuel réside dans la conception de systèmes qui sont en mesure de transférer et de traiter de vastes quantités de données en un laps de temps réduit, tout en maintenant une faible empreinte énergétique.

Durant ce stage, deux applications de type streaming ont été étudiées et optimisées grâce à la décomposition en graphes de tâches proposée par AFF3CT. Dans un premier temps, la contribution principale est la parallélisation *multi-thread* d'une nouvelle application de détection automatique de météores. La chaîne est d'abord décrite 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 fils d'exécution aux cœurs physiques sont évaluées. Ceci dans une optique de maximisation de l'utilisation des ressources.

Dans un second temps, l'objectif principal est d'exploiter un système hétérogène CPU/GPU, grâce aux bibliothèques AFF3CT et OPENCL, toujours dans le cadre des applications de type streaming. Pour prendre en main et se familiariser avec la programmation GPU, différentes études sont effectuées, dont notamment une portant sur une chaîne de communication numérique utilisée aujourd'hui pour une partie des réseaux 4G/5G. Il est important de souligner qu'aujourd'hui, AFF3CT ne supporte pas les implémentations hétérogènes CPU/GPU. Les travaux présentés introduisent justement les premiers pas vers ces implémentations.

Ce rapport est organisé comme suit. Le Chapitre 2 présente une nouvelle application de détection automatique de météores les pour systèmes embarqués depuis l'atmosphère, ainsi que les optimisations implémentées pour répondre aux contraintes de cette application de type streaming sur CPU. Le Chapitre 3 décrit une chaîne de communication numérique et introduit la programmation GPU pour tendre vers un système hétérogène CPU/GPU. Enfin, le Chapitre 4 conclut sur les travaux présentés.

Chapitre 2

Parallélisation CPU d'une application de détection automatique de météores

Lors des pluies d'étoiles filantes, un essaim de météores est visible dans l'atmosphère terrestre depuis la surface de la Terre. Ces objets venant de l'espace intéressent un bon nombre de scientifiques, spécialement des astrophysiciens cherchant des réponses concernant les propriétés des objets dans l'univers [4]. Cependant, les conditions météorologiques peuvent rendre l'étude de ces objets extraterrestres plus difficile. Pour optimiser ces études et remédier à ce problème, il faudrait observer ces météores depuis la haute atmosphère ou l'espace.

2.1 Introduction

2.1.1 Contexte

Un **météore** est un phénomène lumineux se produisant lorsqu'un corps solide pénètre dans l'atmosphère, qui s'identifie par une traînée incandescente. Ce corps est généralement de la matière issue d'un astéroïde ou d'une comète, appelé **météoroïde**, mais pas toujours (les débris spatiaux par exemple). Ce phénomène lumineux est dû à la ionisation de certaines particules atmosphériques lors du passage du corps. L'intensité lumineuse et la couleur d'un météore dépendent de la composition chimique du météoroïde et des particules atmosphériques ionisées sur son passage. On distingue deux types de météore :

— **Les étoiles filantes** : météoroïdes de toutes petites dimensions, de l'ordre du grain de sable, se volatilisant en une ou deux secondes. Parmi les sources d'étoiles filantes, on relève les poussières cosmiques laissées par les comètes. En traversant l'atmosphère, elles produisent un essaim de météores, plus communément appelé **pluie de météores**.

— **Les bolides** : de l'ordre du centimètre à plusieurs dizaines de mètres. Les objets plus gros se fragmentent dans les couches denses de l'atmosphère. Si le météoroïde n'est pas entièrement consumé par son entrée dans l'atmosphère, il se peut qu'un fragment survive et tombe sur le sol terrestre. Ces corps extraterrestres tombés sont appelés des **météorites**.

Avec l'augmentation des missions spatiales, la détection de météores est devenu nécessaire afin d'évaluer le flux d'objets rentrant dans l'atmosphère et pour planifier en toute sécurité les activi-

tés spatiales. Pour établir un système détectant des météores, il est nécessaire d'avoir une caméra, un ordinateur et d'une chaîne de traitement. 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 sur un avion volant à haute altitude [24], soit embarquée sur un ballon-sonde [19]. Cependant, les apparitions des 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 durer des années pour les astronomes. Une automatisation de cette détection est donc nécessaire, d'où le développement d'une chaîne de traitement d'images intitulée Fast Meteor Detection Toolbox (FMDT) [1].

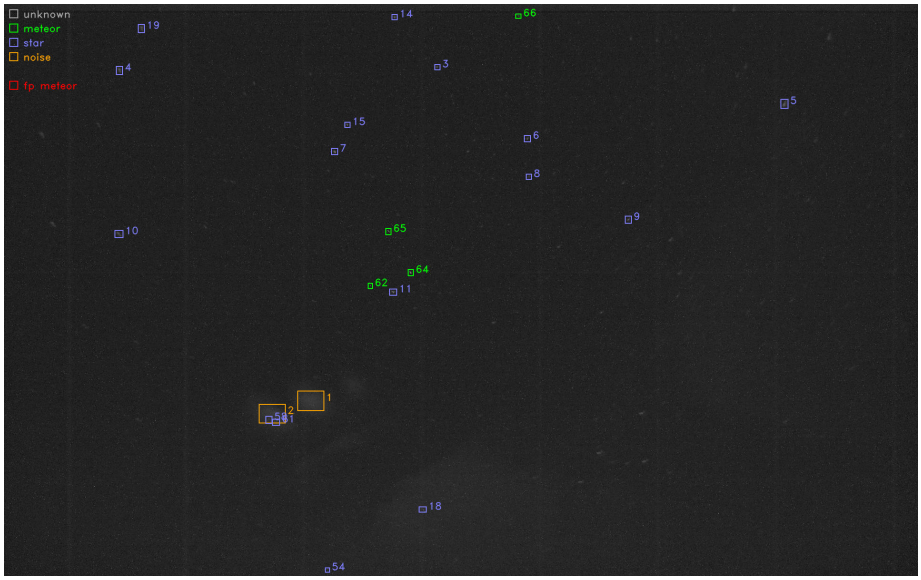


FIGURE 2.1 – Exemple de la détection météore avec FMDT.

2.1.2 Travaux connexes

Jusqu'à maintenant, la majorité des chaînes de traitement d'images développées sont exécutées depuis le sol avec des caméras immobiles [9, 18, 1]. La démarche souvent utilisée consiste à comparer des images prises à des intervalles de temps rapprochés. En détectant les pixels non stationnaires, il est possible d'identifier les météores. Cette méthode, dite par analyse de différences temporelles, est utilisée dans le projet FRIPON [8] par la chaîne de traitement intitulée *FreeTure*. 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). Il faut donc prendre en compte un nouveau facteur qui rend la détection plus complexe : le mouvement de la caméra. Ce dernier fait que tous les pixels bougent d'une image à l'autre.

METEORIX [21] est le premier projet universitaire développé par Sorbonne Université dédié à la détection de météores depuis l'espace à bord d'un CubeSat 3U (nanosatellite cubique de côté 30 cm, voir Figure 2.2). Le projet est toujours en phase de définition du système (phase B) de la

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

charge utile. Le nanosatellite embarque une caméra dans le domaine du visible et un ordinateur exécutant une chaîne de traitement d'images pour la détection en temps réel. En utilisant la caméra dirigée vers la Terre, la chaîne de traitement s'appuie des méthodes basées sur le flot optique (algorithme de *Horn & Schunk* [10]) et des statistiques angulaires pour classifier les objets détectés en mouvement. La chaîne de traitement implémentée est temps réel [16] sur les vidéos HD de l'université de Chiba et consomme moins de 7 Watts.

SOURCE [14] (Stuttgart Operated University Research Cubesat for Evaluation and Education) est un autre projet Cubesat 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 *SpaceMEDAL* obtient de bons résultats de détection mais n'atteint pas la cadence temps réel (1.25 images/seconde) [20]. Les vidéos fournies par l'université de Chiba sont utilisées comme référence, mais ne sont pas suffisantes pour tester leur chaîne de traitement. C'est pour cette raison qu'un outil d'incrustation de météores artificiels (ArtMESS) a été développé. Ce dernier simule des météores perçus depuis un système d'observation en orbite basse de la Terre.

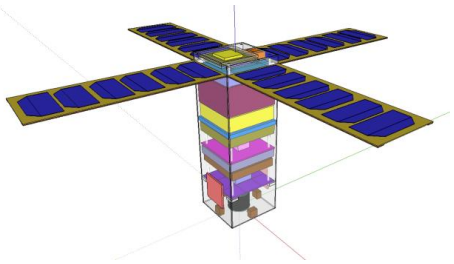


FIGURE 2.2 – Une vue CAO (Conception Assistée par Ordinateur) du satellite METEORIX.

Dans les projets METEORIX et SOURCE, les caméras sont dirigées vers la Terre. Cela implique l'utilisation d'algorithmes capables de différencier plusieurs mouvements différents sur une même scène : 1) la rotation de la terre, 2) le mouvement de la caméra, 3) les nuages, 4) les éclairs et 5) les objets entrant dans l'atmosphère (dont les météores). Bien souvent cela est traité par un algorithme de flot optique capable d'estimer un vecteur vitesse en tout point de l'image. Il est bon de noter que cette opération est très coûteuse en calcul.

Contrairement à METEORIX et SOURCE, la caméra est dirigée vers l'espace. Les scènes à traiter sont donc plus « simples » (un fond noir avec des étoiles). Dans ce cas, il est possible de déterminer le mouvement global entre deux images consécutives en se basant sur les étoiles et ainsi éviter de calculer une estimation du flot optique coûteuse.

2.1.3 Problématique

Ce projet universitaire Fast Meteor Detection Toolbox, 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 l'univers [4]. Pour la communauté scientifique, déployer un système embarqué dans l'atmosphère détectant les météores à très bas coût est une avancée importante. 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 de 1920×1200 pixels à 25 images/seconde pour satisfaire la contrainte de temps réel. Les ressources en énergie nécessaires au fonctionnement du système sont aussi limitées. À titre d'exemple, dans un projet universitaire de nano-satellite [21, 16], nous avons relevé que la puissance instantanée disponible pour la chaîne de traitement est inférieure à 10 Watts.

2.2 Chaîne de traitement naïve

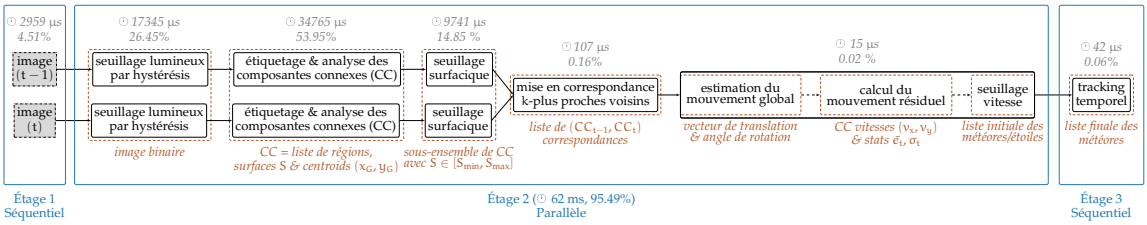


FIGURE 2.3 – Chaîne de traitement détaillée de FMDT avec répartition du temps d'exécution des tâches en séquentiel sur un seul cœur sur Raspberry Pi 4.

La chaîne de traitement développée utilise différents algorithmes efficaces d'un point de vue calculatoire. La Figure 2.3 présente la chaîne de traitement utilisée pour la détection de météores. Elle prend en entrée un flux vidéo en niveaux de gris et retourne en sortie la liste des météores détectés.

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 permet de sélectionner (ou non) un pixel en fonction de son voisinage : s'il est connexe à des pixels supérieurs au seuil haut.. Cela permet de réduire le bruit sur l'image, surtout au niveau de l'atmosphère terrestre. Ainsi, un masque binaire est récupéré en sortie du seuillage contenant des pixels blancs (255) ou noirs (0).

La deuxième étape consiste à passer d'une représentation binaire de l'image à une liste d'objets appelés des Composantes Connexes (CC) via un algorithme d'étiquetage en composantes connexes (ECC). L'algorithme *Light Speed Labeling* **[?]** est utilisé car il combine étiquetage et analyse en composantes connexes (ACC). Dans notre cas, des caractéristiques des régions sont la surface S en pixels, le centre d'inertie de coordonnées (x_G, y_G) et le rectangle englobant de chaque CC $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$.

La troisième étape est un seuillage surfacique ne gardant que les CC dont la surface est comprise entre S_{min} et S_{max}. Les groupes de pixels inférieurs à S_{min} sont du bruit. Les groupes de pixels supérieurs à S_{max} sont les nuages ou les composantes connexes représentant la Terre. 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 consiste à mettre en correspondances ces CC avec un algorithme des *k-plus proches voisins* (k-PPV). L'idée est d'associer les CC proches entre 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 calcule le mouvement global (un vecteur translation $\vec{T}$ et un angle de rotation θ) de l'image due au balancement et au déplacement de la caméra [2]. Le calcul de ces trois composantes est possible grâce aux associations établies précédemment.

La sixième étape compense le mouvement global de l'image en calibrant l'image I_t sur l'image I_{t-1}. Après recalage, les CC associées, dont la distances est quasi nulle, représentent des étoiles. À contrario, les CC qui sont à des distances significativement plus grandes sont des objets 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 contenant les informations sur cet objet est créée. Cela permet d'éliminer des faux positifs. Cette étape s'occupe de la création, de la mise à jour, de l'extrapolation et de la terminaison des pistes. Les objets mobiles qui ne suivent pas une direction rectiligne sont supprimés. Ainsi, la liste des météores est obtenue.

2.3 Optimisations

2.3.1 Décomposition en graphe de tâches

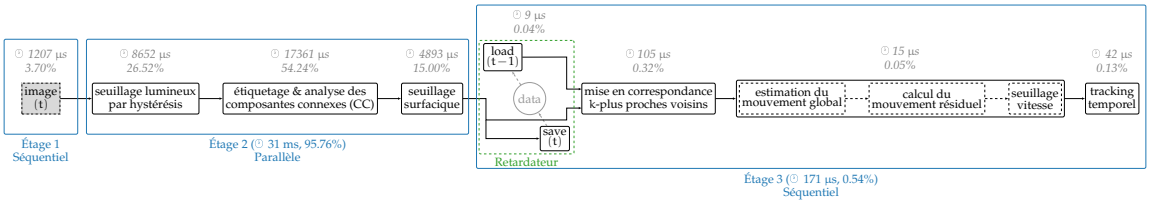


FIGURE 2.4 – Chaîne de traitement avec rotation des données avec répartition du temps d'exécution des tâches en séquentiel sur un seul cœur sur Raspberry Pi 4.

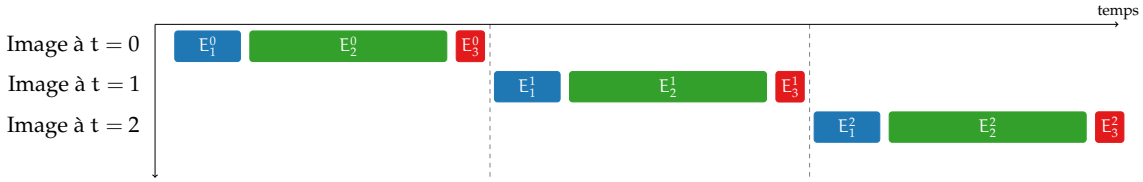
Sur la Figure 2.3, le seuillage lumineux, l'ECC, l'ACC et le seuillage morphologique sont appliqués sur les images I_{t-1} et I_t. Or, à $t+1$, les mêmes traitements sont ré-appliqués sur l'image I_t. On remarque donc que l'on calcule inutilement deux fois les CC sur l'image I_t. Étant donné que ces traitements sont coûteux, il est préférable de ne pas les recalculer et de mémoriser les CC pour l'itération suivante. La Figure 2.4 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 ce rapport, le graphe de tâches correspondant à la Figure 2.3 est référencé comme la version 1 et le graphe de tâches correspondant à la Figure 2.4 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 le Raspberry Pi 4. Pour

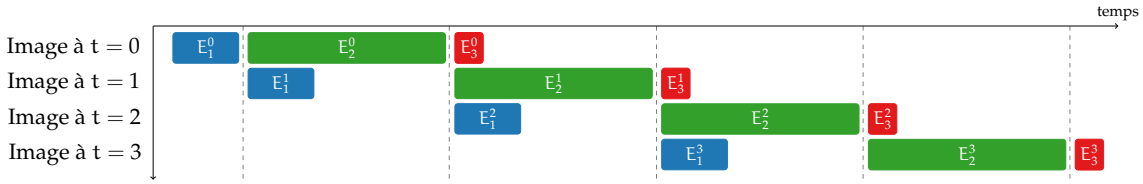
la version 1, la latence est de 65 ms alors que pour la version 2, la latence est de 32 ms. La version 2 est donc 2× plus rapide que la version 1.

2.3.2 Parallélisation des graphes de tâches

Pipeline



(a) Exécution séquentielle



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

FIGURE 2.5 – 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).

En séquentiel, les tâches de la chaîne de traitement sont exécutées les unes après les autres. Ce type d'exécution est l'illustré par la Figure 2.5a. 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 fils d'exécution (voir Figure 2.5b). Le débit théorique est alors le débit de l'étage le plus lent. La Figure 3.1 montre qu'une exécution pipelinée permet d'augmenter le débit d'images par rapport à une exécution séquentielle.

Réplication de tâches

Dans la chaîne de détection proposée, certaines tâches peuvent être exécutées en parallèle sur différents fils d'exécution et sur différentes images. C'est notamment le cas des tâches de l'étage 2 dans les Figures 2.3 et 2.4. C'est ce que l'on appelle le parallélisme de données (aussi connu sous le nom de parallélisme *fork-join*). C'est ce type de parallélisme qui est le plus souvent exploité lors de l'utilisation de la célèbre bibliothèque OpenMP.

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 2, soit l'étage

dont la latence est la plus élevée, il est possible de tirer parti au maximum des architectures multi-cœurs.

Implémentation avec la bibliothèque AFF3CT

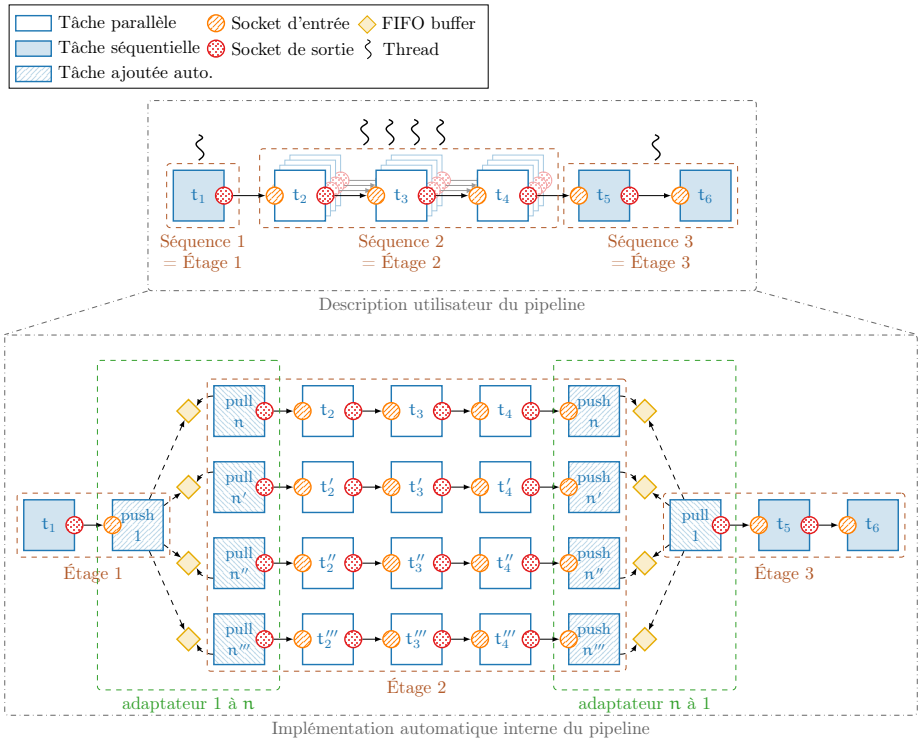


FIGURE 2.6 – Construction du pipeline par AFF3CT

Le *pipeline* avec différents étages et de la réplication de tâches ont été implémentés à l'aide de la bibliothèque AFF3CT [5]. AFF3CT est une bibliothèque C++ et basée sur le principe de programmation orientée objet. AFF3CT définit différents composants : 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. Les trois étages définis précédemment correspondent à trois séquences.

Pour se servir de cette bibliothèque (voir Figure 2.6), l'utilisateur doit d'abord décrire la chaîne de traitement en graphe de tâches. Il doit ensuite regrouper les tâches dans les étages du *pipeline* et sélectionner le nombre de fils d'exécution (threads) par étage. Il est possible d'associer des threads à des cœurs physiques et de choisir le type de synchronisation entre les étages du pipeline (active/passive).

Dans le cadre de l'application FMDT, il a été choisi de découper la chaîne de traitement en 3

étages, illustré sur la Figure 2.4 Le premier étage est l'acquisition du flux vidéo en entrée qui est une étape séquentielle basée sur la notion du temps, donc impossible à paralléliser. Le deuxième étage est composé du seuillage par hystérésis, de l'étiquetage/l'analyse des composantes connexes et du seuillage surfacique. Les expérimentations effectuées montrent que cet étage est le plus coûteux en raison de plusieurs parcours des images entières engendrant énormément d'accès mémoire. Une fois qu'une image est en entrée, tout l'étage peut s'exécuter sans interruption, donc cette partie est exécutable en parallèle. Le troisième étage est aussi séquentiel et contient le reste des étapes. Pour le *tracking*, il y a une notion temporelle et un historique de détections donc il n'est pas possible de paralléliser cette partie. Au vu des courtes durées d'exécution des étages 1 et 3, le gain majeur de ces optimisations doit se trouver dans la parallélisation de l'étage 2, autrement dit l'étage le plus long.

2.4 Expérimentations et résultats

Référence	Nom	Date	Gravure	CPUs	Fréquence
XU4	Odroid-XU4	Feb. 2016	28 nm	4 × *LITTLE* ARMv7 Cortex-A7 4 × *Big* ARMv7 Cortex-A15	1.4 GHz 1.5 GHz
RPI	Raspberry Pi 4 model B	Jun. 2019	28 nm	4 × ARMv8 Cortex-A72	1.5 GHz
NANO	Nvidia Jetson Nano	Mar. 2019	20 nm	4 × ARMv8 Cortex-A57	1.479 GHz
M1	Apple Silicon M1 Ultra	Mar. 2022	5 nm	4 × *E-core* ARMv8 Icestorm 16 × *P-core* ARMv8 Firestorm	≈ 2.0 GHz ≈ 3.0 GHz

TABLE 2.1 – Différents SOC évalués.

La Table 2.1 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 [24]. Enfin, seule la version 2 de la chaîne de traitement est considérée (voir Figure 2.4). Pour comparer les résultats obtenus sur n images traitées, 4 métriques ont été choisies :

1. le débit en nombre d'images par seconde que la chaîne peut traiter,
2. la latence moyenne de la chaîne $\mathcal{L}$ (durée de traitement $t_{fin} - t_{dbut}$) en millisecondes,
3. la puissance moyenne $\mathcal{P}$ (tension $\mathcal{U}\times$ intensité $\mathcal{I}$) en Watts,
4. la consommation d'énergie par image $\mathcal{E}$ (durée totale d'exécution $t \times \mathcal{P}/n$) en milliJoules.

La Figure 2.7 montre les résultats obtenus selon les 4 métriques énoncées précédemment. $\mathcal{D}$, $\mathcal{L}$, $\mathcal{P}$ 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. 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.

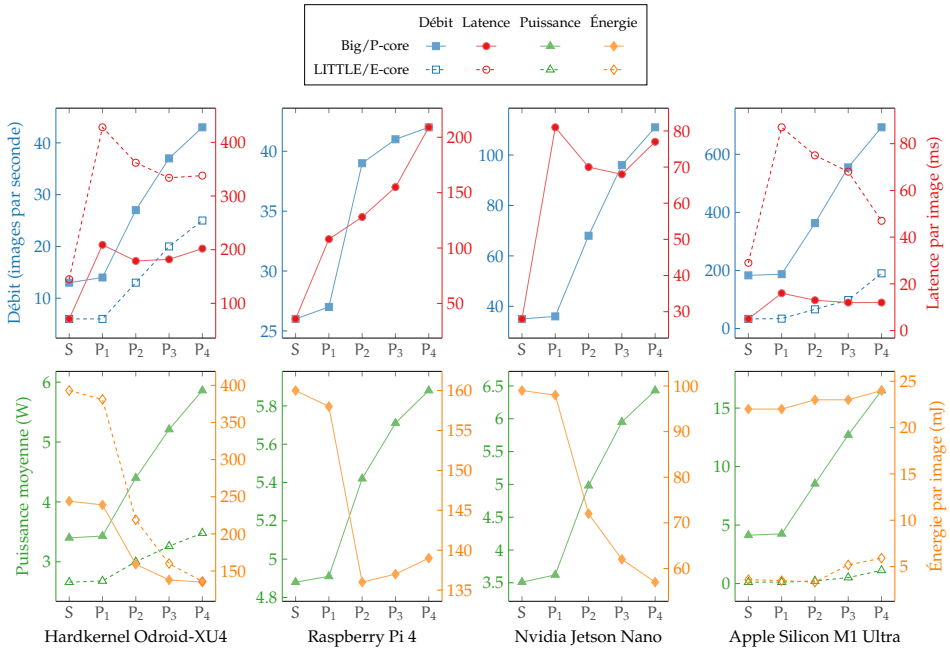


FIGURE 2.7 – Performances Full HD de la chaîne de détection (version 2, c.f. Figure 2.4) en terme de débit, latence, puissance moyenne et énergie consommée pour quatre cibles embarquées.

Débit et latence

La Figure 2.7 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 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 2.7, 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.

2.5 Conclusion et perspectives

Ce chapitre a introduit Fast Meteor Detection Toolbox, une nouvelle chaîne de traitement d'images développée pour la détection de météores. Cet outil est adapté pour la détection de météores sur des systèmes équipés de caméras en mouvement, par exemple sur des ballons-sondes. Contrairement aux autres chaînes existantes employant l'algorithme du flot optique, FMDT utilise un recalage d'images pour détecter les objets en mouvement entre deux images consécutives.

Grâce à la parallélisation des CPU multi-cœurs (par travail à la chaîne et réplication de tâches), la chaîne de détection respecte les contraintes de temps réel et de basses consommations énergétiques pour convenir aux besoins des systèmes embarqués. La version P_4 atteint 111 images par seconde sur la Nvidia Jetson Nano pour une puissance moyenne de 6.51 Watts. Un speedup de $\times$6.5 est obtenu par rapport à la version naïve.

Les travaux présentés dans ce chapitre ont fait l'objet d'une soumission d'article accepté dans une conférence francophone d'informatique en parallélisme, architecture et système [11].

En utilisant les récents travaux sur l'étiquetage et l'analyse des composantes connexes, il est possible de repousser les limites actuelles de FMDT [13] pour améliorer les performances de l'étage le plus long. De plus, les systèmes de nos jours contiennent des architectures hétérogènes CPU/GPU. La seconde partie du stage consiste à prendre en main et à exploiter cette hétérogénéité fournie par les GPU.

Chapitre 3

Parallélisation CPU/GPU d'une chaîne de communication numérique

Une radio logicielle, ou Software-Defined Radio (SDR) en anglais, représente un système de communication numérique adaptable, exploitant des méthodes de traitement du signal au sein de structures numériques programmables. Avec l'évolution des normes de communication complexes et la puissance croissante des processeurs conventionnels, l'idée d'échanger l'efficacité énergétique des architectures spécialisées contre la flexibilité et la facilité de mise en œuvre sur des processeurs généraux devient de plus en plus attrayante. Même lorsque la mise en œuvre d'un traitement numérique est finalement réalisée sur une puce dédiée, une version logicielle préliminaire de ce traitement est nécessaire pour garantir le bon fonctionnement de la fonctionnalité. Cette étape est généralement effectuée par le biais de simulations de type Monte-Carlo, bien que celles-ci soient souvent gourmandes en temps de calcul, nécessitant parfois plusieurs jours, voir même semaines, pour évaluer les performances d'un modèle fonctionnel de système [7].

Durant le stage, l'objectif est de transférer une partie du traitement de cette simulation sur GPU pour tendre vers un système hétérogène.

3.1 Introduction à la programmation GPU

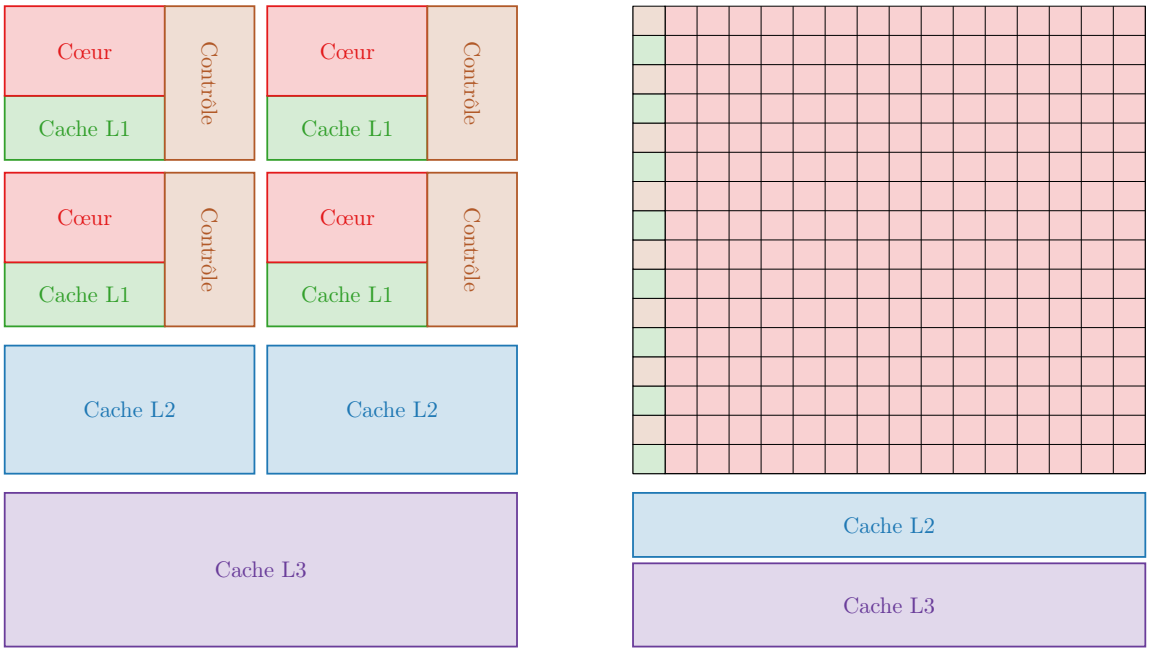
Les GPU sont des composants matériels conçus pour accélérer le traitement de données spécifiques, en particulier les opérations liées aux graphismes et au calcul parallèle. Bien que leur origine soit liée au rendu graphique, les GPU ont évolué pour devenir des accélérateurs de calcul généralistes capables de gérer une grande variété de tâches complexes et intensives en calcul.

3.1.1 Architecture des GPU

Comparées aux CPU, ces architectures mettent en œuvre de manière beaucoup plus intensive les classes de parallélisme SIMD (Single Instruction Multiple Data) et MIMD (Multiple Instruction Multiple Data). Alors que les CPU modernes comptent généralement entre 4 et 8 cœurs, voire jusqu'à 64 pour les processeurs haut de gamme, les GPU disposent de plusieurs centaines, voire de plusieurs milliers de cœurs de calcul pour les modèles haute performance.

Les CPU sont des processeurs généralistes qui cherchent à réduire la latence des instructions d'un programme pour améliorer les performances. La majeure partie de l'espace sur puce des CPU est consacrée à la mémoire cache. En revanche, les GPU sont des processeurs auxiliaires spécialisés qui cherchent à optimiser le débit des données traitées en parallèle pour augmenter les performances. La surface de puce consacrée aux calculs est donc considérablement plus importante que celle réservée à la mémoire cache pour les GPU.

Une autre différence majeure entre les CPU et les GPU réside dans les quantités de mémoire disponibles et leur bande passante par cœur de calcul. Comme mentionné précédemment, les GPU sont conçus pour maximiser les débits, tant pour les instructions que pour la mémoire. Les cœurs de calcul et les mémoires sont plus compacts et plus simples du côté des GPU.



(a) Architecture d'un CPU multi-coeur.

(b) Architecture d'un GPU.

FIGURE 3.1 – Représentation des architectures d'un CPU multi-coeur et d'un GPU modernes

Malgré les avancées constantes dans la conception des GPU, il est devenu de plus en plus évident que ces composants puissants sont soumis à la contrainte fondamentale de la loi d'Amdahl. Cette loi, formulée par Gene Amdahl dans les années 1960, stipule que la vitesse globale d'un système parallèle est limitée par la fraction séquentielle de son exécution, quelle que soit la quantité de parallélisme que l'on puisse introduire. En d'autres termes, même si les GPU sont équipés de milliers de cœurs de calcul, leur efficacité est inévitablement bridée par les parties du calcul qui ne peuvent pas être parallélisées. Cette limitation pose un défi constant aux développeurs de logiciels et aux concepteurs de matériel qui cherchent à exploiter pleinement la puissance des GPU, les obligeant à optimiser et à équilibrer habilement les tâches parallèles avec celles qui sont séquentielles pour atteindre les meilleures performances possibles.

3.1.2 Modèle de programmation

Plusieurs langages de programmation GPU ont été développés pour simplifier le développement sur GPU. CUDA (Compute Unified Device Architecture) est à la fois le modèle de programmation et l'API de NVIDIA pour le domaine du GPGPU (General-purpose Processing on Graphics Processing Units). D'autres plateformes et modèles de programmation sont disponibles comme notamment OPENCL ou encore OPENMP. Pour la chaîne de communication numérique étudiée, l'utilisation de la bibliotèque OPENCL a été choisi pour des raisons de portabilité. Cela facilite le *benchmarking* sur des architecture différentes.

OpenCL (Open Computing Language), développé par Khronos Group, est une bibliothèque *open source* pour la programmation parallèle sur des architectures matérielles hétérogènes. Elle permet aux développeurs de tirer parti de la puissance de calcul des processeurs multi-cœurs (CPU), des unités de traitement graphique (GPU), des DSP (processeurs de traitement numérique du signal) et d'autres accélérateurs matériels. Son modèle est très proche de celui de CUDA.

Ces modèles de programmation se focalisent sur l'utilisation de fils d'exécution légers (*threads*), qui sont automatiquement organisés et répartis sur les cœurs physiques du GPU. Ces *threads* fonctionnent en parallèle sur des données distinctes, représentant ainsi une variation du modèle SIMD. Nvidia a nommé ce modèle SIMT (Single Instruction Multiple Threads). La programmation parallèle ne repose plus sur l'utilisation d'instructions parallèles spécifiques pour le traitement simultané de plusieurs données. Par conséquent, le parallélisme n'est plus directement apparent dans le code, car c'est le matériel lui-même qui gère le traitement parallèle des instructions provenant des différents *threads* du programme. Dans ce contexte, les programmeurs écrivent des fonctions, appelées *kernels*, destinées au GPU. À la différence de la programmation SIMD pour les CPU, où le programmeur manipule des registres SIMD contenant plusieurs éléments scalaires, dans ce modèle, la vision parallèle de la programmation est implicite [22].

3.1.3 Prise en main de la bibliothèque OpenCL

Pour prendre en main la programmation GPU à l'aide de la bibliothèque OPENCL, prenons l'exemple simple de la fonction SAXPY. Cette dernière est une opération mathématique courante utilisée en calcul numérique. Le terme SAXPY est une abréviation de "Single-precision A·X Plus Y", ce qui signifie que cette fonction effectue une opération linéaire sur deux vecteurs et additionne le résultat à un troisième vecteur, le tout en utilisant une précision simple (généralement des nombres à virgule flottante en simple précision).

```
1 #define VECTOR_SIZE 2048*1000
2
3 void saxpy(float alpha, float *A, float *B, float *C){
4     for (int i = 0; i < VECTOR_SIZE; i++){
5         C[i] = alpha * A[i] + B[i];
6     }
7 }
```

Listing 3.1 – Fonction SAXPY en C

Dans la suite de cette partie, un exemple de code C, détaillé étape par étape, est fourni pour exécuter un kernel GPU grâce à la bibliothèque OPENCL :

1. Écrire le code *kernel* dans un fichier *saxpy.cl* :

```
__kernel void saxpy_kernel( float alpha,
                    __global float *A,
                    __global float *B,
                    __global float *C)
{
    int index = get_global_id(0);
    C[index] = alpha* A[index] + B[index];
}
```

2. Dans le *main.c* de l'application CPU, identifier et spécifier la plateforme matérielle sur laquelle les calculs parallèles s'exécuteront :

```
cl_platform_id *platforms = NULL;
cl_uint num_platforms;
clGetPlatformIDs(0, NULL, &num_platforms);
platforms = (cl_platform_id *) malloc(sizeof(cl_platform_id)*
    num_platforms);
clGetPlatformIDs(num_platforms, platforms, NULL);
```

3. Choisir le périphérique sur lequel travailler. Ici, le premier GPU est choisi :

```
cl_device_id *device_list = NULL;
cl_uint num_devices;
clGetDeviceIDs( platforms[0], CL_DEVICE_TYPE_GPU, 0,NULL, &
    num_devices);
device_list = (cl_device_id *) malloc(sizeof(cl_device_id)*
    num_devices);
clGetDeviceIDs( platforms[0],CL_DEVICE_TYPE_GPU, num_devices,
    device_list, NULL);
```

4. Créer le contexte, qui est essentiellement une interface entre votre application et la plateforme OPENCL :

```
cl_context context = clCreateContext( NULL, num_devices,
    device_list, NULL, NULL, NULL);;
```

5. Créer une file de commande sur le périphérique concerné. Toutes les commandes à un périphérique sont soumises par cette file :

```
cl_command_queue command_queue = clCreateCommandQueue(context,
    device_list[0], NULL, NULL);
```

6. Allouer les tampons (*buffers*) de données dans le contexte pour stocker les entrées et les sorties du kernel :

```
cl_mem A_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY,
    VECTOR_SIZE * sizeof(float), NULL, NULL);
cl_mem B_clmem = clCreateBuffer(context, CL_MEM_READ_ONLY,
    VECTOR_SIZE * sizeof(float), NULL, NULL);
cl_mem C_clmem = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
    VECTOR_SIZE * sizeof(float), NULL, NULL);
```

7. Créer un *kernel* en compilant le code source *saxpy.cl* :

```
char *saxpy_source;
load_kernel_source_cl(saxpy_source, "saxpy.cl");
cl_program program = clCreateProgramWithSource(context, 1,(const
    char **)&saxpy_source, NULL, NULL);
clBuildProgram(program, 1, &(device_list[device_chosen]), NULL,
    NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "saxpy_kernel", NULL);
```

8. Charger les tampons d'entrées dans la file de commande du périphérique pour transférer les données de l'application CPU vers le GPU :

```
clEnqueueWriteBuffer(command_queue, A_clmem, CL_TRUE, 0,
    VECTOR_SIZE * sizeof(float), A, 0, NULL, NULL);
clEnqueueWriteBuffer(command_queue, B_clmem, CL_TRUE, 0,
    VECTOR_SIZE * sizeof(float), B, 0, NULL, NULL);
```

9. Ajouter les arguments au kernel :

```
clSetKernelArg(kernel, 0, sizeof(float), (void *)&alpha);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&A_clmem);
clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&B_clmem);
clSetKernelArg(kernel, 3, sizeof(cl_mem), (void *)&C_clmem);
```

10. Exécuter le kernel :

```
size_t global_size = VECTOR_SIZE;
clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global_size
    , NULL, 0, NULL, NULL);
```

11. Charger le tampon de sortie dans la file de commande du périphérique pour transférer les données du GPU vers l'application CPU :

```
clEnqueueReadBuffer(command_queue, C_clmem, CL_TRUE, 0, VECTOR_SIZE
    * sizeof(float), C, 0, NULL, NULL);
```

12. Libérer les ressources utilisés :

```
clFlush(command_queue);
clFinish(command_queue);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseMemObject(A_clmem);
clReleaseMemObject(B_clmem);
clReleaseMemObject(C_clmem);
clReleaseCommandQueue(command_queue);
clReleaseContext(context);
free(platforms);
free(device_list);
```

Si les données dans les tampons changent, les étapes 8 à 11 sont répétées en boucle, alors que les autres étapes ne sont faites qu'une seule fois. La bibliothèque OPENCL dispose aussi d'un outil de profilage pour mesurer le temps d'exécution du kernel.

3.2 Chaîne de communication numérique

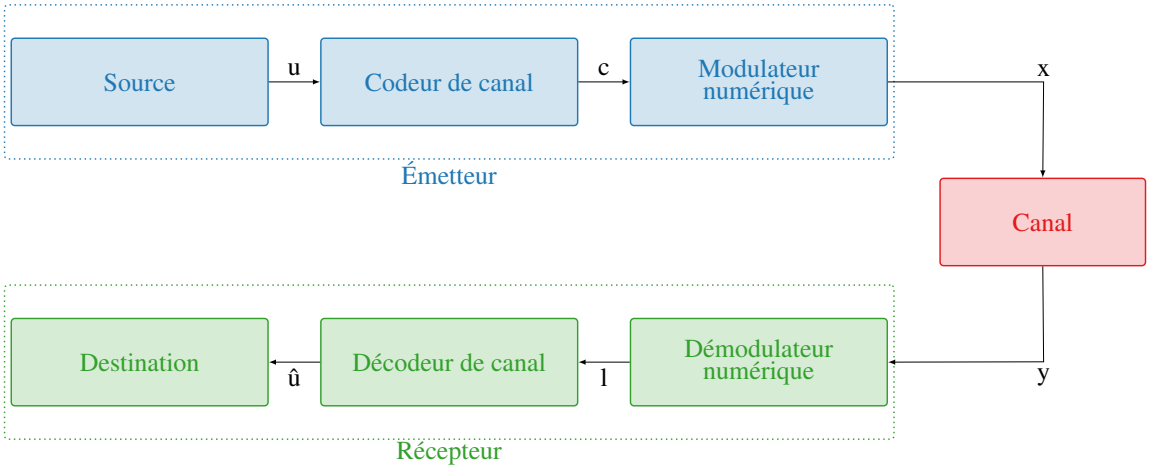


FIGURE 3.2 – Système de communication numérique.

Aujourd'hui, un système de communication numérique peut être représenté par trois éléments essentiels, comme illustré dans la Figure 3.2 : un émetteur, un canal et un récepteur. Au départ, une source génère un message numérique **u** à transmettre, sous forme d'une séquence de bits. Le codeur de canal le transforme en un mot de code **c** afin de le rendre plus résistant aux erreurs éventuelles pouvant apparaître au niveau du canal. Pour permettre la transmission des informations via le canal, il est impératif de préparer le flux de données. Par exemple, dans le contexte des communications sans fil, ce flux doit être converti en un signal haute fréquence qui peut être émis par une antenne de taille raisonnable. C'est là que le modulateur numérique entre en jeu, produisant un vecteur de symboles **x**. Le canal altère ce signal avec du bruit et des distorsions, ce qui donne le signal **y**. Du côté du récepteur, les composants effectuent les opérations inverses pour récupérer le message décodé **û**. Si aucune erreur ne s'est produite pendant la transmission, ou si des erreurs se sont produites mais ont été corrigées, alors **û = u** [7].

Dans le codage de canal, également connu sous le nom de *Forward Error Correction* (FEC), K bits d'information sont encodés dans l'émetteur. Cela résulte en un mot de code c de N bits. P = N - K est le nombre de bits de redondance ajoutés en tant qu'informations supplémentaires et R = K/N est le taux de codage. Plus le taux de codage R est élevé, moins le nombre de bits P est élevé. Les performances sont mesurées en estimant le taux d'erreur résiduel au niveau du récepteur. Il est possible d'observer deux taux différents : 1) le taux d'erreur binaire BER (Bit Error Rate) et 2) le taux d'erreur de trame FER (Frame Error Rate). Le BER est calculé en considérant les K bits d'information de manière indépendante, par exemple un BER de 10^{-3} signifie qu'il y a en moyenne une erreur binaire par millier de bits d'information transmis. Le FER est calculé

en considérant l'ensemble de la trame ; s'il y a au moins une erreur dans la trame en cours, elle sera comptée comme une trame erronée. Un FER de 10^{-2} signifie qu'il y a en moyenne une trame erronée par cent trames transmises. Ces taux dépendent de nombreux facteurs : le bruit du canal, le type de modulation, le type de code, le taux de codage R, etc. Plus les taux d'erreur binaire et de trame sont faibles, plus la capacité de correction du système est élevée [7].

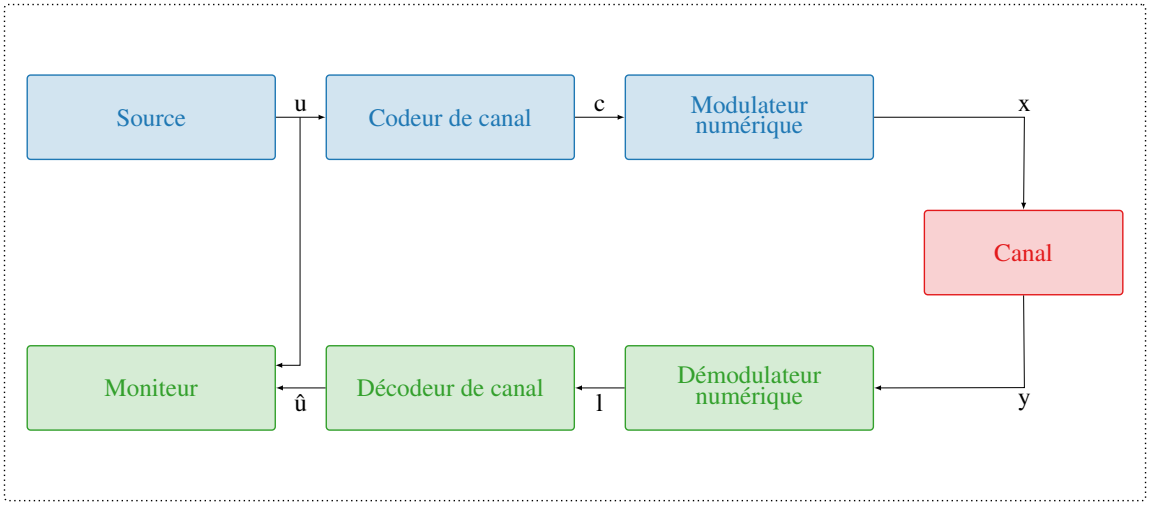


FIGURE 3.3 – Chaîne de simulation du système de communication numérique

Les différentes méthodes pour encoder le message ne sont pas présentées dans ce rapport. Cependant, il faut savoir qu'il existe de nombreux modèles de codage avec différentes caractéristiques. Ces codes sont évalués et comparés par simulation fonctionnelle avant d'être déployés dans de véritables systèmes. L'idée est d'évaluer les performances du système en générant de nombreuses trames aléatoires et en appliquant des échantillons de bruit aléatoire sur ces trames (méthode Monte-Carlo). Les trames bruitées sont décodées et la séquence de sortie des bits $\hat{\mathbf{u}}$ est comparée aux bits d'information initiaux $\mathbf{u}$. Le nombre d'erreurs est ensuite utilisé pour mettre à jour la valeur de BER/FER jusqu'à ce qu'elles atteignent une valeur stable. Cette simulation est représentée dans la Figure 3.3. Cette fois-ci, le moniteur connaît les K bits d'information de sortie à partir de la source, à la différence de la Figure 3.2.

Pour simuler les perturbations appliquées au message $\mathbf{x}$ passant par le canal de transmission, différents modèles peuvent être utilisés. Le modèle choisi pour bruiter le message est souvent le canal par ajout de bruit blanc gaussien AWGN (*Additive White Gaussian Noise*). La génération de ce bruit peut être divisée en deux parties : 1) la génération de variables aléatoires uniformément distribuées et 2) la transformation en une variable aléatoire gaussienne. Un bruit uniformément distribué peut être généré par un générateur de nombres pseudo-aléatoires PRNG (*PseudoRandom Number Generator*) tel que le *Mersenne Twister 19937* (MT19937) [15]. Ensuite, la méthode de *Box-Muller* [3] transforme les nombres aléatoires uniformément distribués en nombres aléatoires normalement distribués. Pour générer deux variables aléatoires normalement distribuées indépendantes Z_0 et Z_1 à partir de deux variables aléatoires uniformément distribuées U_0 et U_1 dans l'intervalle $]0, 1]$, les équations sont les suivantes :

$$Z_0 = \sqrt{-2\ln(U_0)} \cdot \cos(2\pi U_1), \tag{3.1}$$
$$Z_1 = \sqrt{-2\ln(U_0)} \cdot \sin(2\pi U_1). \tag{3.2}$$

3.3 Optimisations

3.3.1 Décomposition en graphe de tâches et parallélisation

Décomposition en graphe de tâches et profiling

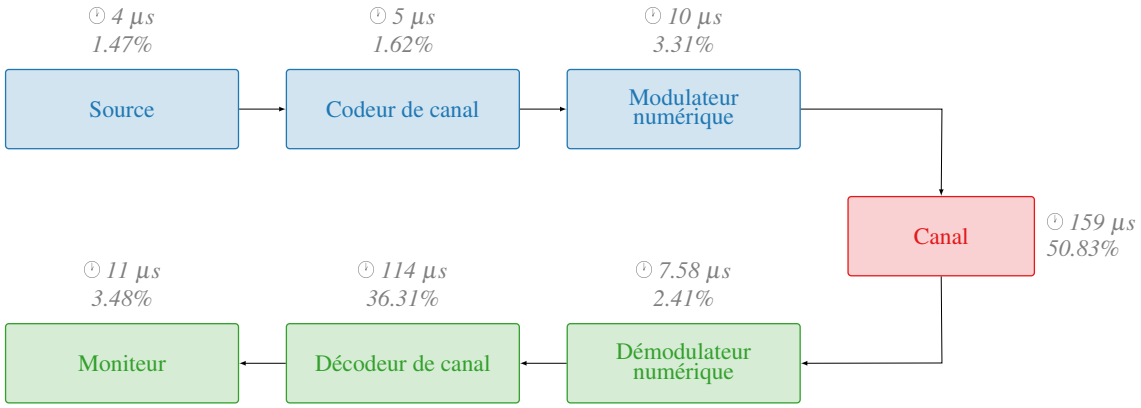


FIGURE 3.4 – Graphe de tâches de la chaîne de communication numérique avec répartition du temps d'exécution des tâches en séquentiel sur un seul cœur Big de l'Odroid-XU4.

Sur la Figure 3.4, comme dans le Chapitre 2, la simulation de la chaîne de communication est décomposée en graphe de tâches. Avec la version séquentielle sur un seul cœur CPU, il est possible de visualiser le temps passé dans chaque tâche dans l'optique d'optimiser les parties les plus longues.

La tâche qui prend le plus de temps est celle du canal avec 50% du temps d'utilisation total. Ce constat s'explique par l'implication d'un grand nombre de calculs dans le canal pour 1) la génération de variables aléatoires uniformément distribuées (Mersenne Twister 19937) et 2) la transformation en une variable aléatoire gaussienne (Box-Muller). La Figure 3.5 montre la répartition du temps d'exécution au sein du canal. La séparation du canal en deux tâches différentes permet d'effectuer des optimisations différentes sur chaque partie du canal.

Parallélisation du graphe de tâches avec pipeline et réplication

En séquentiel, les tâches de la chaîne numérique sont toujours exécutées les unes après les autres. Cette fois-ci, en regroupant des plusieurs tâches en étages et utilisant le principe du *pipeline* (voir Section 2.3.2), un meilleur débit est obtenu. En délimitant les étages comme sur la Figure 3.6, 3 étages avec des temps d'exécutions similaires sont obtenus. Cette version *pipeline* est toujours

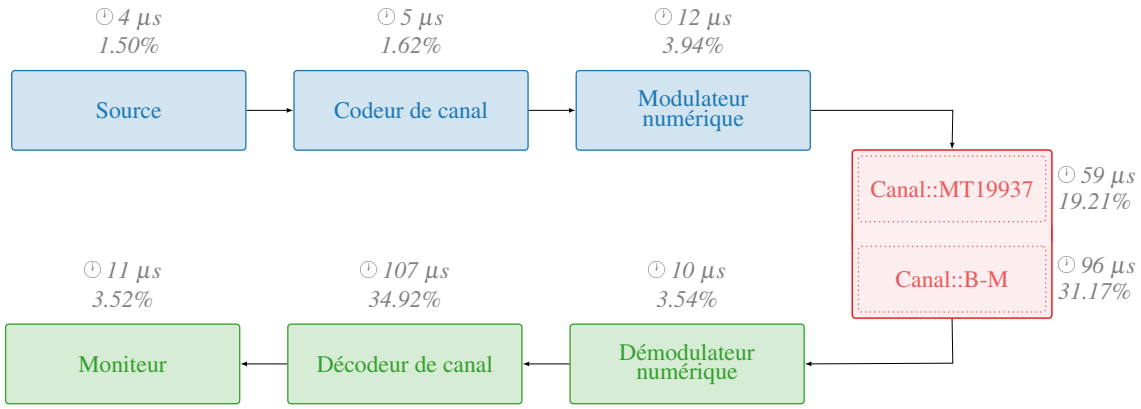


FIGURE 3.5 – Graphe de tâches de la chaine avec Mersenne Twister (MT19937) et Box-Muller (B-M), avec répartition du temps d'exécution des tâches en séquentiel sur un seul cœur Big de l'Odroid-XU4.

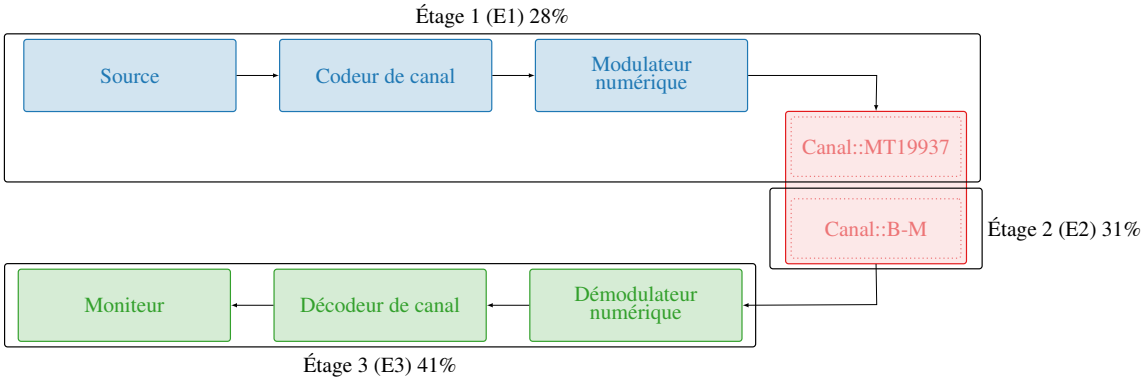


FIGURE 3.6 – Graphe de tâches de la chaîne de communication numérique, avec répartition du temps d'exécution des étages en pipeline sur 3 cœurs Big de l'Odroid-XU4.

limitée par l'étage le plus long qui est E3. Cependant, le parallélisme sur les 3 cœurs CPU doit apporter en théorie une amélioration d'un facteur d'environ 2.5× du débit car les 3 cœurs physiques sont toujours actifs et n'attendent presque jamais. Les résultats sont présentés dans la suite de ce rapport, dans la Section 3.4.

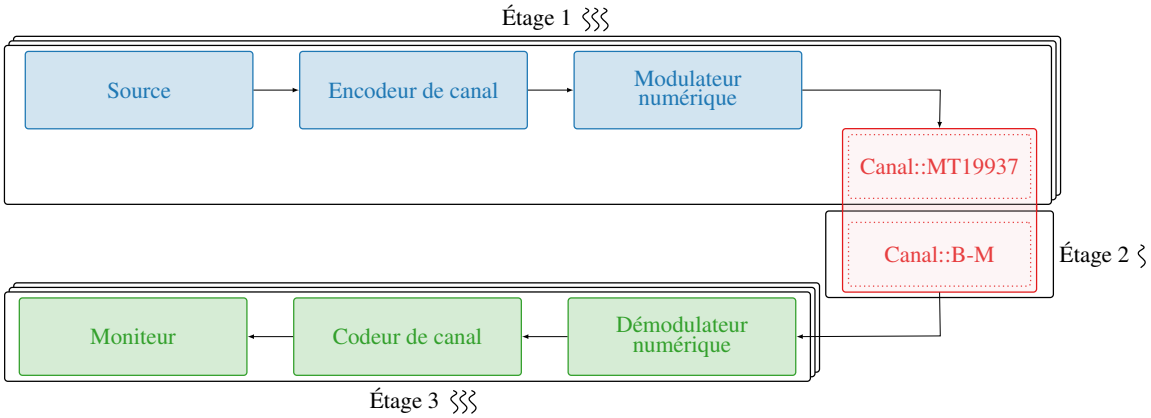


FIGURE 3.7 – Graphe de tâches de la chaîne numérique de communication avec réplication des tâches (Ici 3 threads pour les étages 1 et 3, 1 thread pour l'étage 2).

En utilisant la réplication de tâches (voir Section 2.3.2), il est possible d'améliorer les performances et de tirer parti au maximum des architectures multi-cœurs. À l'aide de la bibliothèque AFF3CT, la réplication de tâches a été implémentéé. Les tâches répliquées sont illustrées dans la Figure 3.7. La méthode de Box-Muller du canal n'est pas répliquée dans l'optique de porter cette tâche sur GPU. Ainsi, un *thread* d'un cœur CPU est libéré et peut se consacrer aux autres étages.

3.3.2 Système hétérogène CPU/GPU

Comme le montre les Équations 3.1 et 3.2, les calculs de la méthode Box-Muller utilisent des fonctions mathématiques assez courantes (cosinus, sinus, racine carrée, logarithme népérien) et sont appliqués sur des données indépendantes. Cette caractéristique des calculs, parfaitement adaptée aux capacités massivement parallèles des GPU, incite à tirer parti de cette architecture spécialisée. En transférant cette charge de travail vers le GPU, une accélération de notre traitement est attendu tout en optimisant l'utilisation des ressources matérielles disponibles. Pour cela, la bibliothèque OPENCL (voir Section 3.1.3) est utilisée. Voici le code d'un *kernel* naïf :

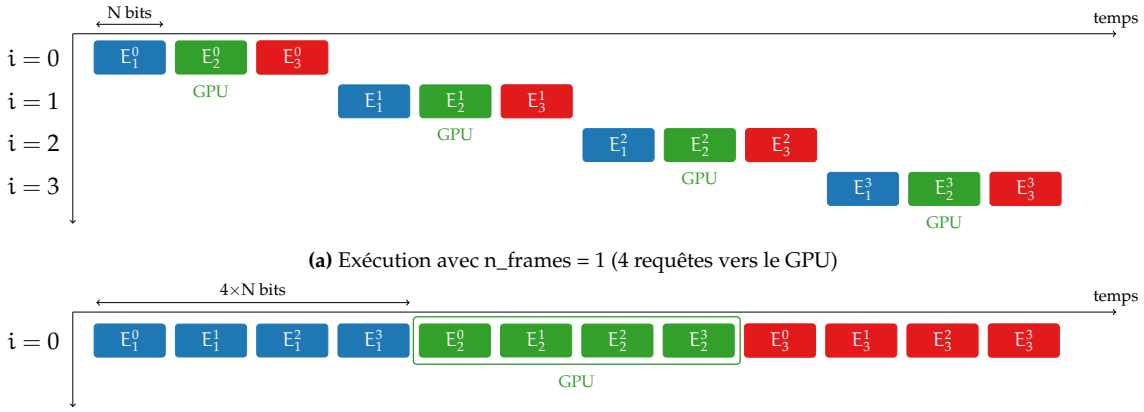
```
1  __kernel void add_noise_kernel(__global const float2 *UNI,
2                                 __global const float2 *X_N,
3                                 __global       float2 *Y_N,
4                                 const float sigma,
5                                 const float mu,
6                                 const float twopi)
7  {
8          int i = get_global_id(0);
9
```

```
10          float2 u = UNI[i];
11          float u1 = u.x;
12          float u2 = u.y;
13
14          float2 x = X_N[i];
15          float x1 = x.x;
16          float x2 = x.y;
17
18          float radius = sqrt(log(u1) * -2.0f) * sigma;
19          float theta  = u2 * twopi;
20
21          float sintheta = sin(theta);
22          float costheta = cos(theta);
23
24          float2 n;
25          n.x = radius * sintheta + mu + x1;
26          n.y = radius * costheta + mu + x2;
27
28          Y_N[i] = n;
29  }
```

Le méthode de Box-Muller génère 2 variables aléatoires. Pour exploiter efficacement le parallélisme fourni par le GPU, il faut que chaque *thread* travaille sur la génération de 2 variables. OPENCL offre justement des types de données qui répondent à cette problématique. Le *float2* est un type de données vectoriel qui permet de stocker et de manipuler efficacement des paires de *float*. Par exemple, un *float2* peut contenir les coordonnées x et y d'un point dans un espace bidimensionnel. Dans le cas de la méthode de Box-Muller, les *float2* permettent de stocker des paires de *float* en entrée et en sortie. Il est possible d'encore optimiser ce code *kernel*. Cependant, la version actuelle est considérée comme assez performante donc les recherches n'ont pas été axées vers l'optimisation du *kernel*. Il existe un compilateur et un simulateur de *kernel* chez ARM (*Mali Offline Compiler*) qui permettent de comparer les performances de différents *kernels* en les simulant.

Traitement de n trames en une itération

Aujourd'hui, transférer des données du CPU vers le GPU et vice versa est assez coûteux (à minima le temps d'une copie mémoire). Il faut donc limiter le plus possible les transferts. Pour cela, comme illustré sur la Figure 3.8, un regroupement de données de différentes trames est possible. Soit n_frames le nombre de paquets traités en même temps. Au lieu d'envoyer n_frames fois un paquet de taille N, il est plus intéressant d'envoyer une seule fois un paquet de taille n_frames$\times$N. Plus n_frames est grand, moins il y de transferts (mais en contrepartie les transferts sont gros). Ce regroupement de données de différentes trames permet de réduire le nombre de requêtes soumises au GPU. De plus, de manière générale, traiter plusieurs trames en même temps peut aussi permettre d'utiliser les registres SIMD (en attribuant chaque élément du registre à une trame différente).



(a) Exécution avec n_frames = 1 (4 requêtes vers le GPU)

(b) Exécution avec n_frames = 4 (1 requête vers le GPU)

FIGURE 3.8 – Exécutions avec des tailles de paquets différentes (avec E_1^p, E_2^p et E_3^p, les étages 1, 2 et 3 appartenant au paquet p).

Optimisations spécifiques à OpenCL

L'API OPENCL prend en charge deux mécanismes permettant à l'application hôte d'interagir avec les tampons OPENCL. Elles peuvent :

— Lire et écrire des tampons en utilisant *clEnqueueReadBuffer* et *clEnqueueWriteBuffer*,

— Mapper et démapper des zones mémoires en utilisant *clEnqueueMapBuffer* et *clEnqueueUnmapMemObject*.

Les fonctions de lecture et d'écriture impliquent une copie des données vers et depuis les tampons OpenCL. Cela signifie généralement un déplacement des données depuis l'espace mémoire du CPU vers la mémoire où réside généralement un tampon OpenCL. Les fonctions de mappage/démapage mappent la mémoire d'un tampon dans l'espace d'adresse de l'application hôte et permettent à l'application hôte de lire et d'écrire directement dans/à partir du contenu du tampon. Cette méthode ne nécessite pas deux zones de stockage contenant les mêmes données (une dans l'espace mémoire du CPU et l'autre dans l'espace mémoire du GPU). De plus, elle ne nécessite pas de déplacement supplémentaire des données entre les deux zones de stockage (l'espace mémoire est partagé par le CPU et le GPU).

Il existe des situations où l'utilisation de *clEnqueueReadBuffer*/*clEnqueueWriteBuffer* est préférable. Pour les tampons plus petits, le surcoût lié aux copies supplémentaires est minime, et les commandes supplémentaires enfilées dans la file de commandes présentent un certain surcoût. Dans les exemples des Listings 3.2 et 3.3, il y a 4 commandes enfilées pour le déplacement des données dans le cas du mappage/démapage, tandis qu'il n'y en a que deux pour le déplacement des données dans le cas de la lecture/écriture.

```
1 int *buf_CPU = (int*) malloc(bufsize);
2 // Operations sur buf_CPU
3 clmem buf_GPU = clCreateBuffer(context, CL_MEM_READ_WRITE, bufsize,
    NULL, NULL);
```

```
4  clEnqueueWriteBuffer(command_queue, buf_GPU, CL_TRUE, 0, bufsize,
       buf_CPU, 0, NULL, NULL);
5  clSetKernelArg(kernel, 0, sizeof(buf_GPU), buf_GPU);
6  clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global_size,
       NULL, 0, NULL, NULL);
7  clEnqueueReadBuffer(command_queue, buf_GPU, CL_TRUE, 0, bufsize,
       buf_CPU, 0, NULL, NULL);
8  // Operations sur buf_CPU
```

Listing 3.2 – Exemple avec clEnqueueReadBuffer/ clEnqueueWriteBuffer.

```
1  clmem buf_GPU = clCreateBuffer(context, CL_MEM_READ_WRITE |
       CL_MEM_ALLOC_HOST_PTR, bufsize, NULL, NULL);
2  int *buf_CPU = (int *)clEnqueueMapBuffer(command_queue, buf_GPU, 0,
       CL_MAP_WRITE, 0, bufsize, 0, NULL, NULL, NULL);
3  // Operations sur buf_CPU
4  clEnqueueUnmapMemObject(command_queue, buf_GPU, buf_CPU, 0, NULL, NULL)
       ;
5  clSetKernelArg(kernel, 0, sizeof(buf_GPU), buf_GPU);
6  clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global_size,
       NULL, 0, NULL, NULL);
7  // L'@ buf_CPU a pu changer depuis la derniere fois, bien qu'elle
       pointe toujours vers la meme case en meroire RAM
8  buf_CPU = (int *)clEnqueueMapBuffer(command_queue, buf_GPU, 0,
       CL_MAP_READ, 0, bufsize, 0, NULL, NULL, NULL);
9  // Operations sur buf_CPU
10 clEnqueueUnmapMemObject(command_queue, buf_GPU, buf_CPU, 0, NULL, NULL)
       ;
```

Listing 3.3 – Exemple avec clEnqueueMapBuffer/ clEnqueueUnmapMemObject.

La gestion de la synchronisation est essentielle pour garantir que les différentes tâches parallèles s'exécutent correctement et pour garder une cohérence des données. Il existe plusieurs mécanismes de synchronisation en OpenCL. Il y a les événements (*event*) qui permettent d'attendre qu'une commande spécifique soit terminée avant de poursuivre. Ces *events* sont utiles pour synchroniser les commandes et les tâches. Par exemple, il est possible d'attendre la fin de l'exécution d'un kernel avant de continuer (voir Listing 3.4).

```
1  cl_event event;
2  clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL, &global_size,
       NULL, 0, NULL, &event);
3  clWaitForEvents(1, &event);
```

Listing 3.4 – Exemple de synchronisation avec clWaitForEvents.

Avec clEnqueueReadBuffer/clEnqueueWriteBuffer, il est possible de bloquer l'opération de lecture/écriture. Les paramètres CL_TRUE et CL_FALSE spécifient si la commande de lecture/écriture doit être exécutée de manière synchrone (bloquante) ou asynchrone (non bloquante).

Enfin, le *callback* est utile pour gérer des opérations asynchrones. Le principe du *callback* se réfère à une technique de programmation où une fonction A est passée en tant que paramètre à

une fonction B, et cette fonction A est exécutée uniquement quand la fonction B est terminée. Ce mécanisme est géré par des évènements. Ainsi, quand le code du *kernel* est terminé, il est possible de lancer directement la lecture du buffer de façon asynchrone.

3.4 Expérimentations et résultats

Référence	Nom	Date	CPUs	Fréq CPUs	GPUs	Fréq GPUs
XU4	Odroid-XU4	Fév 2016	4 × *LITTLE* ARMv7 Cortex-A7 4 × *Big* ARMv7 Cortex-A15	1.4 GHz 1.5 GHz	Mali-T628 MP6	600 MHz
OPI5	Orange Pi 5 Plus	Mai.2023	4 × *LITTLE* ARMv7 Cortex-A55 4 × *Big* ARM Cortex-A76	1.8 GHz 2.4 GHz	Mali-G610 MP4	1000 MHz

TABLE 3.1 – Différents SOC évalués.

La Table 3.1 présente les différentes architectures embarquées évaluées. L'Odroid-XU4 (XU4) et l'Orange Pi 5 plus (OPI5) sont hétérogènes et disposent : 1) de cœurs efficaces du point de vue énergétique (*LITTLE*), 2) de cœurs puissants/rapides (*Big*) et 3) de GPU. Le code est compilé avec GCC v9.4.0 et avec les options d'optimisations suivantes : `-O3 -funroll-loops -march=native`.

3.4.1 Single precision A.X + Y (saxpy)

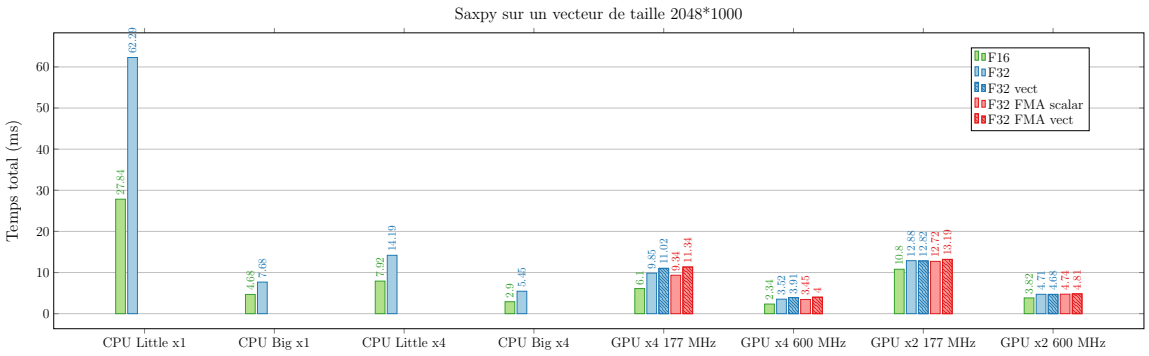


FIGURE 3.9 – Histogramme du temps d'exécution de la fonction *saxpy* en fonction de différentes configurations sur l'Odroid-XU4

Pour prendre en main la programmation GPU et la bibliothèque OPENCL, une étude a été effectuée sur l'efficacité de la fonction saxpy selon différentes configurations et implémentations. Les résultats sont présentés dans la Figure 3.9. Par défaut, un *float* ou *float32* (nombre en virgule flottante simple précision) est codé sur 32 bits. Les *half* ou *float16* (nombre en virgule flottante *half-precision* codé sur 16 bits) sont une alternative pour économiser les ressources mémoires mais au coût de perdre en précision de calculs. Cependant, certaines applications ne nécessitent pas de calculs très précis. Ainsi, utiliser les *float16* améliore les performances. La Figure 3.9 montre que la version utilisant les *float16* est effectivement plus rapide que les versions utilisant les *float32*,

que cela soit sur CPU ou GPU. Le gain est inférieur à 2 mais reste non négligeable. On observe aussi qu'à la fréquence maximale, le GPU est plus rapide que le CPU utilisant les 4 cœurs *Big*. Cela montre qu'il est important de migrer certains calculs sur GPU pour améliorer les performances d'une application limitée par la memoire globale.

3.4.2 Chaîne de communication numérique

Les expérimentations de la simulation de la chaîne de communication numérique ont été menées sur 6 configurations différentes :

— CPU Seq : version séquentielle sur un seul cœur CPU Big (voir Figure 3.5),

— CPU pip : version pipeline sur 3 cœurs CPU Big (voir Figure 3.6),

— GPU v1 : version avec Box-Muller sur GPU avec *clEnqueuReadBuffer/ clEnqueueWriteBuffer* (voir Listing 3.2) ; un seul paquet de taille N×n_frames est envoyé (voir Figure 3.8) ; synchronisation avec CL_TRUE,

— GPU v2 : version avec Box-Muller sur GPU avec *clEnqueueMapBuffer/ clEnqueueUnmapMemObject* (voir Listing 3.3) ; un seul paquet de taille N×n_frames est envoyé ; synchronisation avec *clWaitForEvents*,

— GPU v3 : version avec Box-Muller sur GPU avec *clEnqueueMapBuffer/ clEnqueueUnmapMemObject* ; n_frames paquets de taille N sont envoyés ; synchronisation avec *clWaitForEvents* (voir Listing 3.4),

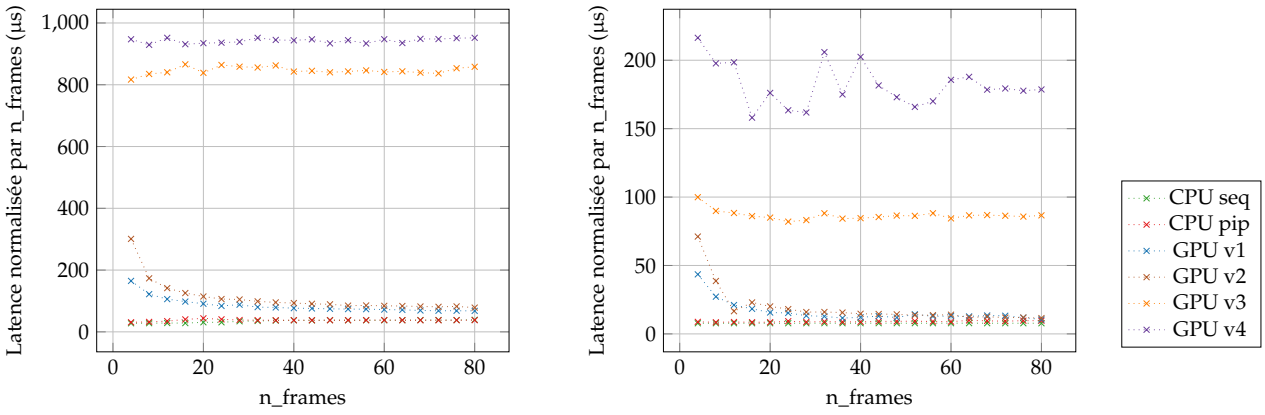
— GPU v4 : version avec Box-Muller sur GPU avec *clEnqueueMapBuffer/ clEnqueueUnmapMemObject* ; n_frames paquets de taille N sont envoyés ; synchronisation avec *callback*.

Ces différentes versions ont été vérifiées en comparant leur taux BER et FER (voir Section 3.2). Pour comparer les performances obtenues, 2 métriques ont été choisies :

1. le temps d'exécution du Box-Muller du canal.
2. le débit en mégaBytes par seconde (MB/s) que la chaîne complète peut traiter.

La Figure 3.10 représente la latence normalisée par n_frames en fonction de n_frames. Plus les points de la figure sont bas, moins de temps est passé dans la tâche. Pour les versions CPU, la taille du paquet n'a pas d'impact sur le temps normalisé par n_frames. Le temps d'exécution est linéaire. Cependant, pour les versions GPU v1 et v2, plus la taille des données augmente, moins on passe de temps dans la tâche. Pour une grande taille de données, des performances quasi similaires sur GPU et CPU sont obtenues sur le OPI5. On peut voir aussi que les versions utilisant une seule requête GPU (versions GPU v1 et v2) sont plus efficaces que celles utilisant n_frames requêtes (versions GPU v3 et v4). Quelque soit la version GPU, il y a une copie inutile et que se rapprocher de la latence de la version CPU est donc un bon résultat.

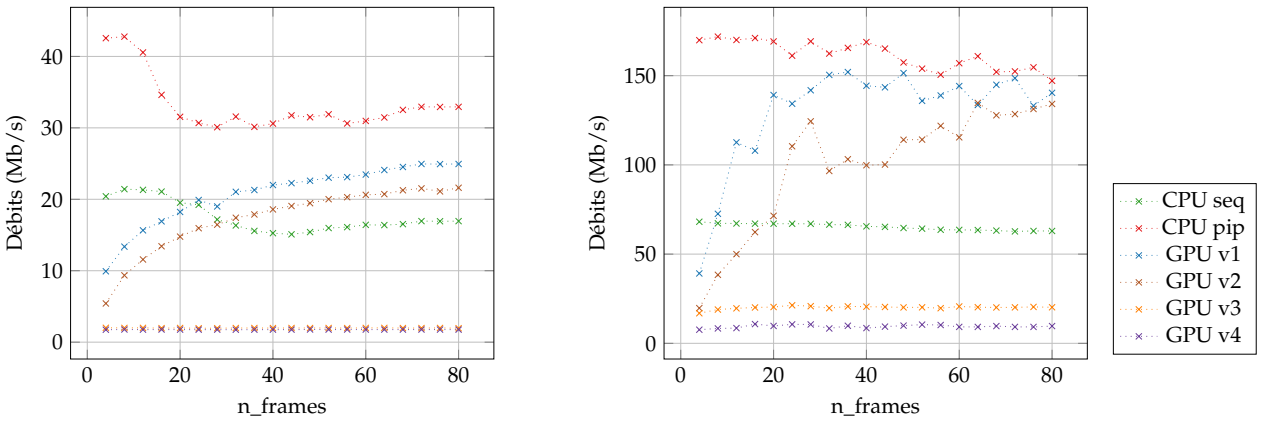
La métrique la plus importante pour la simulation d'une chaîne numérique reste le débit. La Figure 3.11 présente les débits obtenus pour les différentes versions étudiées. Plus les points sont hauts, plus le débit est élevé. On remarque que les versions GPU v3 et v4 ne sont pas efficaces par rapport à la version basique CPU seq, peu importe la taille du paquet. Encore une fois, avoir n_frames requêtes GPU est un facteur limitant pour les performances. Sur le XU4, le débit est différent selon la taille des paquets. Comme visualisé dans la Figure 3.10, plus les paquets sont gros, plus les versions GPU v1 et v2 sont efficaces. Sur le OPI5, les versions GPU v1 et V2 sont 2× plus rapide que la version initiale CPU seq pour des paquets de grandes tailles. La version la plus



(a) Odroid-XU4.

(b) Orange Pi 5 Plus.

FIGURE 3.10 – Temps d'exécution de Box-Muller sur le canal.



(a) Odroid-XU4.

(b) Orange Pi 5 Plus.

FIGURE 3.11 – Performances de la chaîne complète

rapide reste la version CPU pip peu importe la taille du paquet, même si on observe un meilleur débit sur des petits paquets de données. Avec encore quelques optimisations en plus, les versions GPU v1 et v2 tendent vers une meilleure performances sur des gros paquets que sur le CPU. On rappelle qu'il y une copie inutile avant et après le *kernel* GPU. Même si les versions GPU ne sont pas les plus rapides, l'utilisation du GPU permet de libérer un cœur CPU.

3.5 Conclusion

Ce chapitre a introduit la programmation GPU grâce à la bibliothèque OPENCL. Pour tendre vers un système hétérogène CPU/GPU, les travaux ont été portés sur la simulation d'une chaîne de traitement numérique utilisée dans les réseaux de communication numérique actuels.

Plusieurs configurations ont été étudiées, dont la parallélisation des CPU multi-cœurs, le portage d'une tâche sur GPU et le traitement de paquets de tailles différentes. Actuellement, la parallélisation des CPU multi-cœurs apporte le plus gros gain. Cependant, les résultats présentés montrent qu'en exploitant de façon intelligente les GPU, les performances des version GPU tendent vers celles des versions CPU multi-cœurs. Plusieurs versions, dont 2 utilisant le GPU, obtiennent un gain en débit $2\times$ supérieur à la version naïve sur la Orange Pi 5 Plus.

Il est encore possible d'améliorer encore les performances liées au GPU, en optimisant le code *kernel* exécuté sur ce dernier ou en réduisant les transferts de données entre CPU et GPU. Ces derniers sont des facteurs non négligeables limitant les performances globales d'une système hétérogène CPU/GPU. Ces problèmes n'ont pas été réglés car cela impliquait des changements conséquents dans le DSEL AFF3CT.

Des mesures de consommation sont aussi nécessaires pour assurer la pérennité et la viabilité de ces systèmes hétérogènes.

Chapitre 4

Conclusion

Pendant ce stage, nous avons cherché à concevoir des systèmes en mesure de transférer et de traiter des vastes quantités de données en un laps de temps réduit sur des systèmes embarqués multi-cœurs et hétérogènes. Cette étude a porté sur deux applications de type streaming d'actualité.

D'une part, nous sommes maintenant capable de traiter en temps réel un flux d'images dans le contexte d'une application de détection automatique de météores. La décomposition en graphe de tâches et la parallélisation des tâches sur différents cœurs CPU ont permis d'accomplir cet objectif. Les études présentées montrent aussi que ces optimisations ont permis à l'embarquabilité de l'application sur des systèmes embarqués comme la Raspberry Pi 4. Ces travaux ont fait l'objet d'une soumission d'article accepté dans une conférence francophone d'informatique en parallélisme, architecture et système. [11]

D'autre part, nous avons introduit la programmation GPU avec OpenCL dans le cadre d'une simulation d'une chaîne de communication numérique. La parallélisation des tâches sur CPU a encore montré son efficacité et son importance pour maximiser l'utilisation des ressources matérielles. Un système hétérogène CPU/GPU est introduit avec des performance prometteuses. Il arrive tout juste à rivaliser avec un système CPU multi-cœurs mais l'objectif sur le long terme est de dépasser de loin les performances de ce dernier.

Ces travaux présentées sont une introduction vers un système hétérogène CPU/GPU. En effet, plus les données sont vastes, plus les GPU sont efficaces. Or, les résultats présentés sur la programmation GPU ne concernent que des trames de bits (1D). Il serait intéressant de porter des tâches de l'application de détection de météores sur GPU. En travaillant sur des images (2D), on s'attend à une meilleure exploitation des ressources GPU.

Aujourd'hui, la bibliothèque AFF3CT ne supporte pas pleinement les GPU. En effet, la gestion de la mémoire ne prend pas en compte les tâches de type GPU, ce qui oblige de faire des copies inutiles. L'automatisation des allocations mémoires dans le cadre de tâches GPU est une piste prometteuse.

Bibliographie

[1] Yoan Audureau, Chiara Marmo, Sylvain Bouley, Min-Kyung Kwon, François Colas, Jérémie Vaubaillon, Mirel Birlan, Brigitte Zanda, Pierre Vernazza, Stephane Caminade, et al. Freeture : A free software to capture meteors for fripon. In *Proceedings of the International Meteor Conference, Giron, France, 18-21 September 2014*, pages 39–41, 2014.

[2] Isabelle Bloch. Recalage et fusion d'images médicales, 2020.

[3] George EP Box and Mervin E Muller. A note on the generation of random normal deviates. *The annals of mathematical statistics*, 29(2) :610–611, 1958.

[4] N. Rambaux C. Hongru and J. Vaubaillon. Accuracy of meteor positioning from space- and ground-based observations. 2020.

[5] A. Cassagne, O. Hartmann, M. Léonardon, K. He, C. Leroux, R. Tajan, O. Aumage, D. Barthou, T. Tonnellier, V. Pignoly, B. Le Gal, and C. Jégo. AFF3CT : A fast forward error correction toolbox ! *Elsevier SoftwareX*, 10 :100345, October 2019.

[6] A. Cassagne, R. Tajan, O. Aumage, D. Barthou, C. Leroux, and C. Jégo. A DSEL for high throughput and low latency software-defined radio on multicore CPUs. *Wiley Concurrency and Computation : Practice and Experience (CCPE)*, 2023. e7820.

[7] Adrien Cassagne. *Optimization and parallelization methods for software-defined radio*. PhD thesis, Université de Bordeaux, 2020.

[8] Florent Colas, Brigitte Zanda, Sylvain Bouley, Simon Jeanne, Adrien Malgoyre, Mirel Birlan, Cyrille Blanpain, Jérôme Gattacceca, Laurent Jorda, Julien Lecubin, et al. Fripon : a worldwide network to track incoming meteoroids. *Astronomy & Astrophysics*, 644 :A53, 2020.

[9] Peter S Gural. An operational autonomous meteor detector : Development issues and early results. *WGN, Journal of the International Meteor Organization, p. 136-140.*, 25 :136–140, 1997.

[10] B. K. P. Horn and Brian G. Schunk. Determining optical flow. *ACM Computing Surveys (CSUR)*, 17(1-3) :185–203, 1981.

[11] Mathuran Kandeepan, Clara Ciocan, Adrien Cassagne, and Lionel Lacassagne. Parallélisation d'une nouvelle application embarquée pour la détection automatique de météores. In *COMPAS 2023-Conférence francophone d'informatique en Parallélisme, Architecture et Système*, 2023.

[12] Lionel Lacassagne and Bertrand Zavidovique. Light speed labeling : Efficient connected component labeling on RISC architectures. *Springer Journal of Real-Time Image Processing (JRTIP)*, 6(2) :117–135, June 2011.

[13] F. Lemaitre, A. Hennequin, and L. Lacassagne. How to speed Connected Component Labeling up with SIMD RLE algorithms. In *Workshop on Programming Models for SIMD/Vector Processing*, pages 1–8. ACM, 2020.

[14] Marcel Liegibel, Jona Petri, Philipp Hoffmann, Niklas Geier, and Sabine Klinkner. Meteor observation with the source cubesat–developing a simulation to test on-board meteor detection algorithms. In *4th Symposium on Space Educational Activities*. Universitat Politècnica de Catalunya, 2022.

[15] Makoto Matsumoto and Takuji Nishimura. Mersenne twister : a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1) :3–30, 1998.

[16] M. Millet, N. Rambaux, A. Cassagne, M. Bouyer, A. Petreto, and L. Lacassagne. High performance computer vision application for Meteor detection from a cubesat. In *44th Assembly of Committee on Space Research (COSPAR)*, 2022.

[17] Maxime Millet, Nicolas Rambaux, Adrien Cassagne, Manuel Bouyer, Andrea Petreto, and Lionel Lacassagne. Meteorix : High performance computer vision app. for Meteor detection from a CubeSat. In *COSPAR*, 2022.

[18] Sirko Molau. The meteor detection software metrec. In *Proceedings of the International Meteor Conference, 17th IMC, Stara Lesna, Slovakia, 1998*, pages 9–16, 1999.

[19] Francisco Ocaña, Alejandro Sánchez de Miguel, Daedalus Project, et al. Balloon-borne video observations of geminids 2016. *preprint arXiv :1911.10064*, 2019.

[20] Jona Petri. Satellite formation and instrument design for autonomous meteor detection. 2022.

[21] N. Rambaux, J. Vaubaillon, L. Lacassagne, D. Galayko, G. Guignan, M. Birlan, P. Boisse, M. Capderou, F. Colas, F. Deleflie, F. Deshours, A. Hauchecorne, P. Keckhut, A.C. Levasseurd-Regourd, J.L. Rault, B. Zanda, and all students of the Meteorix team. Meteorix : A cubesat mission dedicated to the detection of meteors and space debris. In *ESA NEO and Debris Detection Conference- Exploiting Synergies -ESA/ESOC, Darmstadt, Germany*, 2019.

[22] Thomas Romera. *Adéquation algorithme architecture pour flot optique sur GPU embarqué*. PhD thesis, Sorbonne Univeristé, 2023.

[23] J. Vaubaillon, C. Loir, C. Ciocan, M. Kandeepan, M. Millet, A. Cassagne, L. Lacassagne, P. Da Fonseca, F. Zander, D. Buttsworth, S. Loehle, J. Toth, S. Gray, A. Moingeon, and N. Rambaux. A 2022 τ-herculid meteor cluster from airborne experiment : automated detection, characterization and consequences for meteoroids. *Astronomy & Astrophysics*, September 2022. (Submitted).

[24] J. Vaubaillon, C. Loir, C. Ciocan, M. Kandeepan, M. Millet, A. Cassagne, L. Lacassagne, P. Da Fonseca, F. Zander, D. Buttsworth, S. Loehle, J. Tóth, A. Moingeon, and N. Rambaux. A 2022 τ-herculid meteor cluster from an airborne experiment : automated detection, characterization, and consequences for meteoroids. *Astronomy and Astrophysics (A & A)*, 2023.