


École Normale Supérieure de Lyon

Licence 3 Informatique Fondamentale

Rapport de Stage

Optimisation de l'exécution d'une application de flot optique sur une architecture parallèle hétérogène

Enrique GALVEZ

Encadrants
Adrien CASSAGNE
Alix MUNIER
Lionel LACASSAGNE
Maxime MILLET

Laboratoire d'accueil : LIP6

Equipe ALSOC

30 mai 2022 – 8 juillet 2022

Table des matières

Introduction

Les algorithmes de flot optique permettent d'estimer le mouvement dans des paires d'images consécutives d'une séquence vidéo. Ils ont de nombreuses applications, initialement en robotique et aujourd'hui pour les véhicules autonomes, pour l'analyse des mouvements et la détection d'obstacles.

L'exécution efficace de ces algorithmes sur un système embarqué est un problème difficile ; ils sont en effet très calculatoires et la limitation de l'énergie est une contrainte forte.

On considère dans ce stage une plateforme d'exécution hétérogène : une carte Nvidia Jetson TX2 constituée de plusieurs cœurs dont les performances en temps et en puissance sont différentes.

Le but de ce stage était, pour un algorithme de flot optique fixé de type pyramidal, de tester une ou plusieurs stratégies de déploiement, d'en mesurer les performances en temps d'exécution et si possible de les comparer.

1 Contexte

1.1 Motivation : la mission METEORIX

Meteorix[1] est une mission universitaire dédiée à la détection et à la caractérisation des météores. Le but de cette mission est d'embarquer dans un CubeSat (satellite cubique miniature) des outils afin de détecter et caractériser des météores depuis l'espace.

Parmi ces outils, l'équipe ALSOC du LIP6 travaille sur un algorithme de traitement d'image dont le but est de détecter des météores. Basé sur une estimation du flot optique selon la méthode de Horn et Schunk[2], cet algorithme est très calculatoire. Il est donc nécessaire de tester des stratégies de déploiement et d'optimisation de cet algorithme sur différents systèmes embarqués, afin de doter le CubeSat de Meteorix d'un algorithme peu coûteux en temps ainsi qu'en énergie.

Mon objectif durant ce stage a été de tester le déploiement d'une maquette de cet algorithme sur une carte NVIDIA Jetson TX2, ainsi que de proposer une version optimisée de cette maquette, qui pourrait être adaptée sur l'application finale.

1.2 La carte NVIDIA Jetson TX2

La carte NVIDIA Jetson TX2, conçue pour l'embarqué, est dotée d'une architecture hétérogène. Bien que la carte Jetson TX2 date de 2017, ce type d'architecture a de l'avenir, par exemple pour les processeurs Intel, sachant que la $12^{\text{ème}}$ génération est munie d'une architecture hétérogène.

La carte sur laquelle j'ai travaillé était donc munie de deux types de processeurs, pour lesquels les caractéristiques ainsi que les performances étaient différentes. La Table 1 présente les caractéristiques techniques fournies par le constructeur pour les deux types de cœurs.

	Cœurs	Fréquence max	Cache L1	Cache L2
Denver 2	2	2.5 GHz	192 kiB	2 MiB
ARM Cortex A57	4	2 GHz	80 kiB	2 MiB

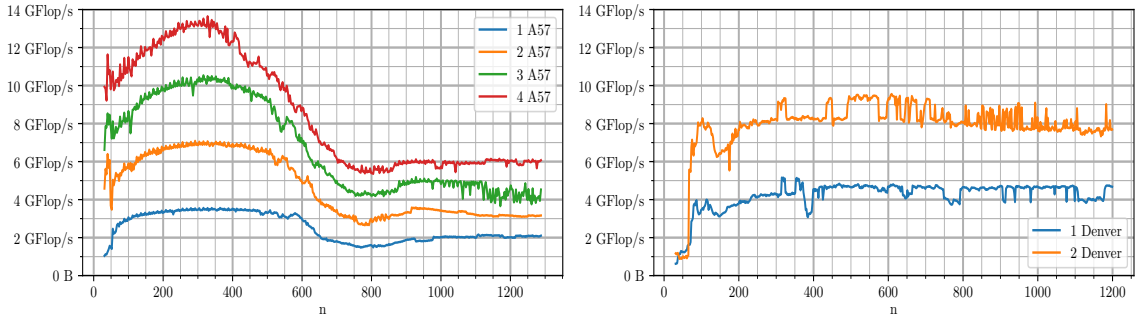
TABLE 1 – Caractéristiques techniques annoncées par le constructeur

Bien que les caractéristiques techniques de ces processeurs soient semblables, nous allons voir que leurs performances ne sont pas identiques, ce qui constitue bien une architecture hétérogène.

Afin de comprendre la différence entre ces deux types de processeurs, il est nécessaire de réaliser un premier benchmark : la mesure du nombre d'opérations par seconde réalisées lors d'une multiplication de matrices en parallèle. En effet, le calcul matriciel effectue beaucoup d'opérations flottantes pour peu d'accès mémoire : $2n^3$ opérations pour $2n^2$ accès mémoire si l'on considère une matrice de taille n. Il permet donc d'estimer la performance "maximale" que les processeurs peuvent atteindre.

La Figure 1 présente les courbes obtenues en mesurant le temps d'exécution t (en moyenne sur 50 exécutions) d'un programme qui effectue la multiplication de deux matrices de taille n puis en calculant :

$$\text{Flop/s} = \frac{\text{ops}}{t} = \frac{2n^3}{t}$$



(a) Cœurs A57 (b) Cœurs Denver 2

FIGURE 1 – Performance en terme de calcul selon le type de cœur

Nous voyons que les cœurs Denver 2 sont individuellement légèrement plus performants que les cœurs A57. On retrouve des performances cohérentes avec celles annoncées par le constructeur (cf. Table 1), étant donné que le nombre maximum d'opérations par seconde que peut effectuer un processeur est proportionnel à sa fréquence.

Nous observons cependant sur la Figure 1 que la performance maximale que peuvent atteindre les processeurs A57 chute lorsque n devient grand. Ce phénomène s'explique par le fait que les données traitées sont trop volumineuses pour tenir dans les caches, le programme est alors limité par la vitesse d'accès en mémoire au lieu d'être limité par la puissance de calcul des processeurs. De plus, la Table 1 explique l'absence de ce phénomène pour les cœurs Denver 2. En effet,la taille du cache L1 étant bien plus grande que pour les A57, pour les valeurs de n considérées dans la Figure 1, les données peuvent être stockées dans le cache L1 des coeurs Denver 2.

1.3 Clé de l'optimisation : la localité des données

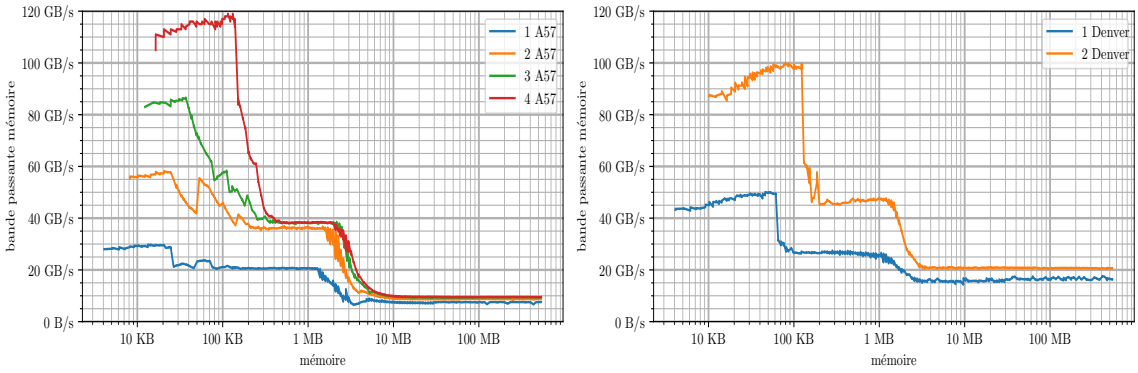
D'après le benchmark que l'on a effectué ainsi que les spécifications techniques du constructeur, la différence entre les cœurs ne se situe pas tant au niveau de la fréquence des processeurs, mais plutôt au niveau de la taille des caches ainsi que de la vitesse pour accéder à des données dans ceux-ci.

Pour mesurer une vitesse d'accès à des données, la notion de *bande passante mémoire* peut être utile : il s'agit de calculer le nombre d'opérations en mémoire effectuées par seconde. Le calcul s'effectue de la même manière que précédemment :

$$\text{bandwidth} = \frac{\text{memops} \times \text{taille d'une donnée}}{t}$$

Où bandwidth est la bande passante mémoire et *memops* ainsi que t sont respectivement le nombre d'opérations en mémoire et le temps mis pour les effectuer par un algorithme.

Afin d'estimer la bande passante maximale que l'on pourrait espérer, on trace en Figure 2 les courbes de la bande passante mémoire pour un algorithme qui recopie des vecteurs de taille croissante, en fonction de la place qu'occupent ces vecteurs en mémoire.



(a) Cœurs A57 (b) Cœurs Denver 2

FIGURE 2 – Bande passante mémoire selon le type de cœur

Une première chose que l'on peut remarquer sur les deux graphiques de la Figure 2 est la présence de 3 palliers, correspondant aux différents niveaux de localité des données.

En effet, le premier pallier est celui où la bande passante mémoire est la plus élevée, il correspond au cache L1, le plus proche du processeur, propre à chaque cœur, ce qui explique la multiplication des performances en parallèle.

Le deuxième pallier correspond quant à lui au deuxième niveau de cache, L2, propre à chaque cœur pour les Denver et partagé par paquets de 2 cœurs pour les A57. On remarque bien que lorsque plus de deux cœurs A57 sont utilisés et que les données traitées sont de l'ordre de la taille du cache L2, il est inutile d'utiliser plus de 2 cœurs A57, car les performances ne sont pas améliorées.

Enfin, le dernier pallier correspond à la bande passante dans la RAM, et puisque la mémoire RAM est partagée par tous les cœurs, on voit qu'utiliser du parallélisme lorsque le programme est limité par des accès RAM n'améliore pas ses performances.

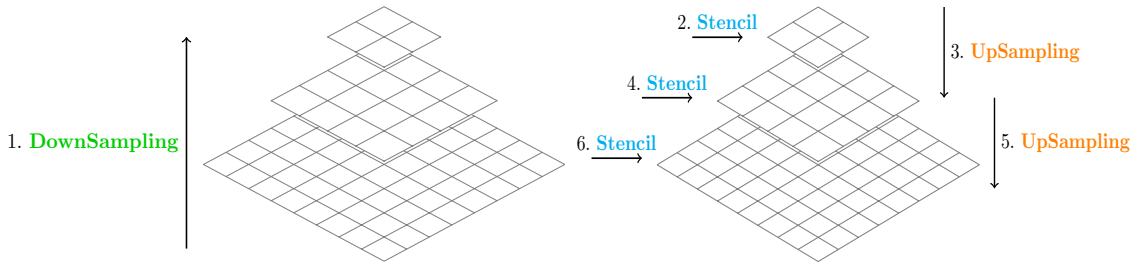
En résumé : Lorsque l'on cherche à paralléliser un programme effectuant beaucoup d'accès mémoire, il est nécessaire de lui faire réutiliser au maximum des données stockées dans les caches afin de bénéficier de hausses de performances significatives. On appelle ça *améliorer la localité des données*.

1.4 L'application de flot optique

L'algorithme conçu pour être embarqué sur METEORIX est un algorithme de flot optique de type pyramidal. C'est à dire que son fonctionnement repose sur une *pyramide d'image* : une représentation multi-résolution de l'image traitée (cf. Figure 3). On peut donc représenter l'exécution de ce programme à l'aide d'un schéma représentant une pyramide, où chaque étage est une image à une résolution différente.

On distinguera dans ce programme l'application de trois opérateurs à l'image traitée :
- **DownSampling** Un opérateur qui diminue l'image à la manière d'un *average-pooling* 2×2
- **Stencil** Un opérateur convolutif qui produit en sortie une image de même taille que celle en entrée, il est généralement itéré plusieurs fois sur la même image
- **UpSampling** Un opérateur qui agrandit l'image en reconstituant des pixels par interpolation à partir de ses voisins

Au cours de son exécution, le programme parcourt la pyramide d'image dans deux sens différents : d'abord des plus grandes images vers les plus petites avec l'opération **DownSampling** puis des plus petites vers les plus grandes en alternant les opérations **UpSampling** et **Stencil**. La Figure 3 illustre cet ordre d'application des opérateurs.



(a) Phase ascendante (b) Phase descendante

FIGURE 3 – Fonctionnement de l'algorithme

Dans le but d'optimiser uniquement ces opérations, un programme maquette n'effectuant que le calcul ainsi que le parcours de la pyramide m'a été fourni. Toutes les mesures et optimisations décrites dans la suite ont été réalisées à partir de ce programme maquette.

2 Optimisation de l'application

2.1 Comment optimiser l'algorithme ?

Avant de chercher à optimiser ou paralléliser les calculs de l'algorithme, il est nécessaire de déterminer son *profil*, c'est à dire quelles opérations sont les plus coûteuses, et quel élément technique limite les performances.

Pour commencer, il convient de comparer les durées d'exécution des différents opérateurs afin de déterminer quelles opérations optimiser en priorité :

	temps de calcul (en % du temps total)
DownSampling	14 %
Stencil	81 %
UpSampling	4 %

TABLE 2 – Proportion du temps de calcul que représente l'exécution des opérateurs

La Table 2 nous montre que la majorité du temps de calcul correspond à l'exécution de **Stencil**, optimiser la phase descendante est donc une priorité. Il est alors nécessaire de comprendre comment fonctionne l'opérateur **Stencil**.

Stencil est un opérateur de type convolutif de noyau $\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$. Il produit en sortie une image de même taille que celle d'entrée, où chaque pixel de la sortie est calculé en faisant la moyenne (pondérée selon le noyau) des 8 pixels voisins et de lui-même sur l'image d'entrée.

Son comportement peut être illustré simplement à l'aide de la fonction `stencil_line` qui applique **Stencil** sur une ligne de l'image X (cf. Algorithme 1).

```
void stencil_line(uint32** X, int i){
    for (int j = 0; j < w; j++){
        int r = 1*X[i-1][j-1] + 2*X[i-1][j] + 1*X[i-1][j+1]\
                2*X[i  ][j-1] + 4*X[i  ][j] + 2*X[i  ][j+1]\
                1*X[i+1][j-1] + 2*X[i+1][j] + 1*X[i+1][j+1]
        X[i][j] = r/16;
}}
```

ALGORITHME 1 – La fonction appliquant le stencil à une ligne de l'image

De manière plus simple, on peut représenter sur le schéma en Figure 4 les images d'entrée (à gauche) et de sortie (à droite) lors de l'application de l'opérateur stencil sur un cas simple : une image de taille (4×4) avec des bords (représentés en pointillés, valent 0 et ne servent que pour les calculs).

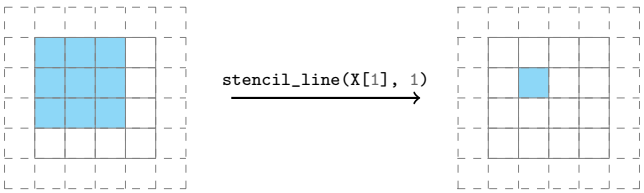


FIGURE 4 – L'application du stencil

L'Algorithme 1 ainsi que le schéma de la Figure 4 permettent de mieux comprendre pourquoi **Stencil** est l'opérateur le plus coûteux : il réalise beaucoup d'accès mémoire (9 lectures et 1 écriture par pixel de l'image de sortie) ainsi que des opérations de calcul (6 additions et 5 multiplications par pixel de l'image de sortie).

On peut cependant remarquer que ce code n'est pas optimal car il pourrait davantage réutiliser certaines données (les accès mémoire correspondant aux deux colonnes de droite du stencil sont redondants entre deux itérations consécutives de la boucle). Il est donc nécessaire d'optimiser ce code en séquentiel avant de tenter de paralléliser son exécution. On pourra, dès lors, se référer à l'article [3], et plus particulièrement à l'algorithme 3 de l'article qui décrit exactement les optimisations à effectuer sur `stencil_line` en séquentiel : stockage des valeurs à réutiliser, réduction en moyenne sur les colonnes et rotation des valeurs réduites.

2.2 Parallélisation "naïve" de la phase descendante

Une bonne manière d'accélérer un programme très calculatoire peut être de paralléliser son exécution, c'est à dire de répartir les calculs sur plusieurs cœurs. Pour ce faire, on peut utiliser la bibliothèque OpenMP[4], qui permet de paralléliser des calculs à l'aide d'une simple directive dans le code.

Cependant, paralléliser un programme n'est pas toujours une tâche facile. Il faut en effet choisir les calculs que l'on veut paralléliser, comment le faire et surtout prendre en compte les parties du programme qui sont *intrinsèquement séquentielles*, c'est à dire qui ne peuvent pas être parallélisées.

L'Algorithme 2 présente le code que l'on cherche à paralléliser, constituant la phase descendante de l'algorithme, c'est à dire l'alternance de **Stencil** et **UpSampling**.

```
1  for(int level = nb_level - 1; level >= 0; level--) {
2
3      for(int iter = 0; iter < nb_iter; iter++){
4          for(int i = 0; i < h; i++)
5              stencil_line(X[level], i);
6
7      } if(level) {
8          for (int i = 0; i < h; i++)
9              upsample_line(X[level], i, X[level-1]);
10     }
11 }
```

Algorithme 2 – Le code pour la phase descendante de l'algorithme

Lorsque l'on regarde ce code, on a envie de paralléliser les deux boucles `for` correspondant aux itérations de `stencil_line` et de `upsample_line` sur les lignes de l'image traitée. On cherche donc à paralléliser les boucles `for` des lignes 4 et 8. Il "suffit" donc de rajouter les directives OpenMP adaptées à la parallélisation de ces boucles. (voir Algorithme 3)

```
for(int level = nb_level - 1; level >= 0; level--) {

    for(int iter = 0; iter < nb_iter; iter++){
        #pragma omp parallel for
        for(int i = 0; i < h; i++)
            stencil_line(X[level], i);

    } if(level) {
        #pragma omp parallel for
        for (int i = 0; i < h; i++)
            upsample_line(X[level], i, X[level-1]);
    }
}
```

ALGORITHME 3 – Première version parallélisée de l'algorithme : version *classic*

On dispose ainsi d'une première version parallélisée de la phase descendante de l'algorithme, que l'on appellera la version *classic*.

Cette version a l'avantage d'être la plus facile à mettre en place, mais présente quelques inconvénients :

— La discontinuité dans le parallélisme introduite par la présence de deux boucles `for` introduit deux *barrières* : une à la fin de chaque boucle `for`. En effet, dès qu'un cœur arrive à ce niveau, il doit "attendre" les autres pour effectuer la suite du calcul.

— Du point de vue de la localité des données, la version *classic* n'est pas optimale car chaque cœur parcourt une partie de l'image avant une *barrière* mais rien ne garantit qu'il reparcourra la même où qu'il pourra réutiliser des données précédentes lors du calcul suivant.

2.3 Améliorer la localité de données en changeant l'ordre de calcul

Afin de rendre l'algorithme plus optimal du point de vue de la localité des données, une méthode peut être de changer l'ordre des calculs. Pour comprendre les changements dans le code, on peut comparer l'exécution en séquentiel de ces deux versions :

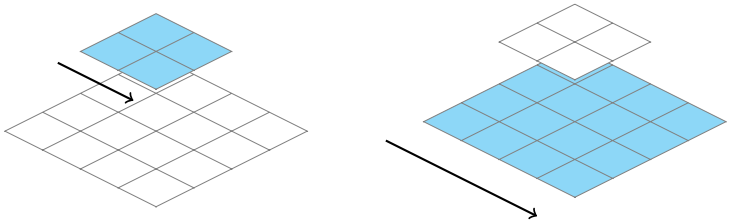


FIGURE 5 – Fonctionnement en séquentiel de la version *classic*

Comme l'illustre la Figure 5, dans cette version *classic*, chaque application d'un opérateur (**Up-Sampling** ou **Stencil**) sur l'image est réalisée entièrement avant de passer à l'opération suivante.

En parallèle, la seule différence est que les cœurs de calcul se répartiront le calcul lors de chaque opération. Comme évoqué précédemment, le maintien d'une bonne localité des données entre deux opérations ne peut pas être assuré.

L'idée dans la seconde version, que l'on appellera la version *ord*, est donc de diviser l'image en blocs, et de calculer en profondeur dans chacun des blocs. Ainsi les données calculées à la fin d'une opération seront directement réutilisées pour la suivante.

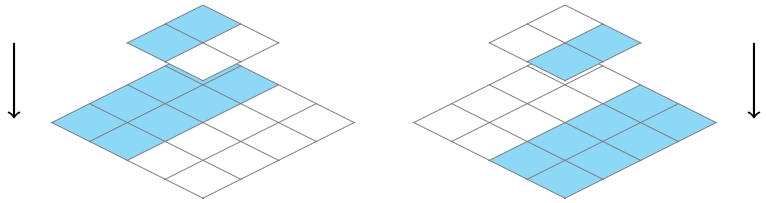


FIGURE 6 – Fonctionnement en séquentiel de la version *ord*

La Figure 6 montre le fonctionnement en séquentiel de cette nouvelle version : *ord*. Dans cette version, les différents cœurs se répartiront au début de l'algorithme les zones de l'image sur lesquelles chacun devra se restreindre pour tout le reste du calcul.

```
#pragma omp parallel for
for(int k = 0; k < nb_blocs){
    for(int level = nb_level - 1; level >= 0; level--) {

        for(int iter = 0; iter < nb_iter; iter++){
            for(int i = blocStart(k); i < blocEnd(k); i++)
                stencil_line(X[level], i);

        } if(level) {
            for (int i = blocStart(k); i < blocEnd(k); i++)
                upsample_line(X[level], i, X[level-1]);
        }
}}
```

ALGORITHME 4 – Deuxième version parallélisée de l'algorithme : version *ord*

Comme annoncé précédemment, le parallélisme se fait sur la première boucle `for` de l'Algorithme 4, celle qui correspond aux zones de l'image à se répartir.

En parallélisant ainsi selon la boucle des blocs, chaque cœur de calcul aurait pour tâche le calcul d'une petite portion de l'image, et la possibilité de réutiliser les données pour calculer le résultat des opérateurs suivants.

2.4 Implémentation de cette méthode

Il reste cependant un "détail" à régler dans l'Algorithme 4 si l'on souhaite que son implémentation produise le même résultat que la version *classic* (i.e. un résultat correct).

En effet, à cause des dépendances de données introduites par l'opérateur **Stencil**, il sera nécessaire d'introduire des calculs supplémentaires, redondants.

Afin de comprendre cela, schématisons le calcul d'un pixel résultant de l'application successive de 3 opérateurs stencil sur une image.

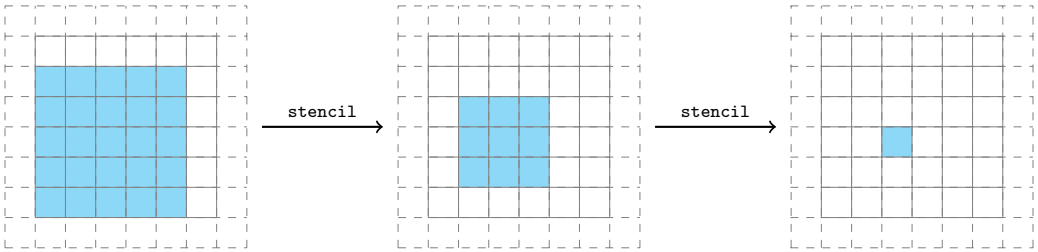


FIGURE 7 – Calcul d'un pixel après 2 itérations de **Stencil**

La Figure 7 illustre bien les dépendances de données introduites par l'opérateur stencil, étant donné que chaque pixel est calculé à partir de 9 pixels sur l'image précédente.

Afin de garantir la correction de *ord*, il est donc nécessaire d'introduire des zones de calculs supplémentaires, appelées **shadow-zones**. Ces shadow-zones peuvent être représentées sur le schéma de la Figure 8 :

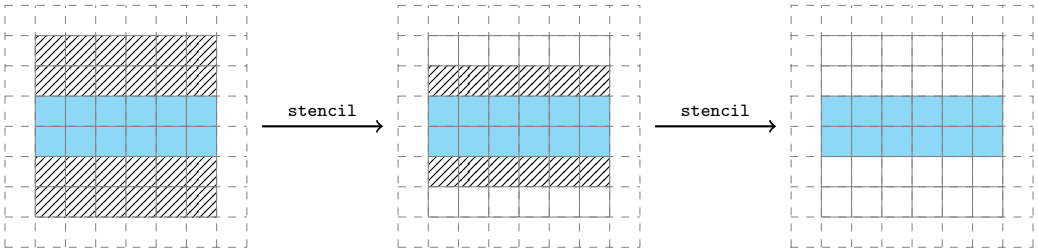


FIGURE 8 – Shadow-zones pour le calcul de 2 itérations de **Stencil**

Si le calcul de ces shadow-zones est nécessaire à la correction de la version *ord*, on espère que son coût sera compensé par le gain de performance obtenu grâce à l'amélioration de la localité des données par rapport à la version *classic*.

On peut cependant, à ce stade, prévoir que ce coût ne sera pas toujours compensé et dépendra fortement de la division du calcul.

En effet, sur de très grandes images, on anticipe que le coût des calculs supplémentaires sera négligeable, ce qui ne semble pas être le cas sur de petites images.

2.5 Mesurer les performances des algorithmes

Maintenant que l'on dispose de deux algorithmes pour optimiser le calcul de la phase descendante de l'algorithme de flot optique, on va chercher à comparer leurs performances.

Pour ce faire, on va calculer le "nombre d'opérations utiles par seconde", c'est à dire le rapport du nombre d'opérations effectuées par la version d'origine sur le temps d'exécution de l'algorithme dont on veut évaluer la performance.

$$\text{GOp/s utiles} = \frac{\text{nb ops dans la version classic}}{\text{durée d'exécution}}$$

En plus de nous fournir une métrique normalisée, calculer des GOp/s nous permet de comparer les algorithmes entre eux et avec les benchmarks de la partie 1, afin de vérifier l'ordre de grandeur des performances.

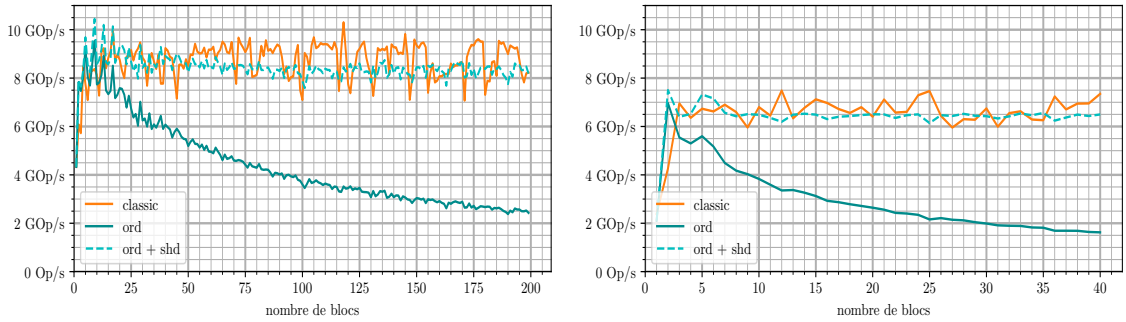
2.6 Comment partager le calcul ?

Revenons à l'implémentation de *ord*. La manière dont le calcul est divisé pour être réparti sur les différents processeurs est aussi un facteur affectant le coût rajouté par la version *ord*.

En effet, la quantité de calculs rajoutés par les shadow-zones est proportionnelle au nombre de blocs de l'image dans lesquels *ord* va calculer en profondeur.

Avant de comparer les performances des versions *ord* et *classic*, il est donc nécessaire de déterminer, par l'expérience, combien de blocs utiliser dans la version *ord*.

Ainsi, on peut observer l'impact des shadow-zones en considérant une image de taille 2048, en partageant le calcul sur 4 cœurs (2 Denver et 2 A57), et en faisant tourner les différentes versions de l'algorithme avec plusieurs configurations.



(a) 6 niveaux, 6 itérations (×6)

(b) 3 niveaux, 5, 10 puis 20 itérations

FIGURE 9 – Influence du nombre de blocs sur la performance

Sur la Figure 9, les courbes pleines correspondent aux GOp/s "utiles", comme définies dans la partie précédente, tandis que les courbes en pointillés correspondent aux GOp/s "réelles", c'est à dire en prenant en compte le calcul des shadow-zones.

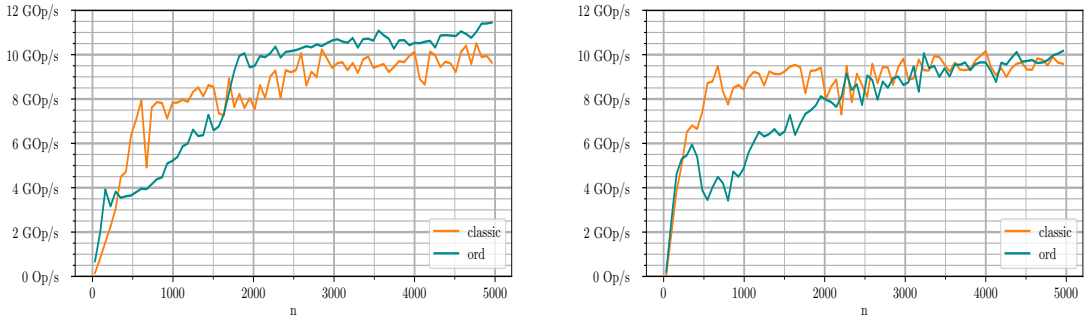
$$\text{GOp/s réelles} = \frac{\text{nb ops dans la version classic } + \text{ nb ops pour shadow-zones}}{\text{durée d'exécution}}$$

La Figure 9 montre l'impact des shadow-zones sur la version *ord* en fonction du nombre de blocs. On constate que cet impact devient très important au delà de 5 ou 10 blocs selon la configuration de l'algorithme. On utilisera donc dans la suite 6 blocs pour tracer les courbes de performance de la version *ord*.

Remarque : Les courbes de la Figure 9, comme toutes les courbes de ce document, ne sont pas parfaitement lisse pour des raisons liées à la mesure. Ce phénomène peut en partie être expliqué par le fonctionnement des caches. Il est décrit plus précisément par Andrea Petreto dans sa thèse[5] au point 3.3.5, dans la sous-section "AGX et difficultés de mesure".

2.7 Performance des algorithmes

La Figure 10 représente les performances des algorithmes *classic* et *ord* en utilisant 4 cœurs (2 Denver et 2 A57) de la carte, sur des cas usuels, c'est à dire avec un nombre de niveaux et un nombre d'itérations de **Stencil** du même ordre que pour l'application complète.



(a) 6 niveaux, 6 itérations (×6)

(b) 3 niveaux, 5, 10 puis 20 itérations

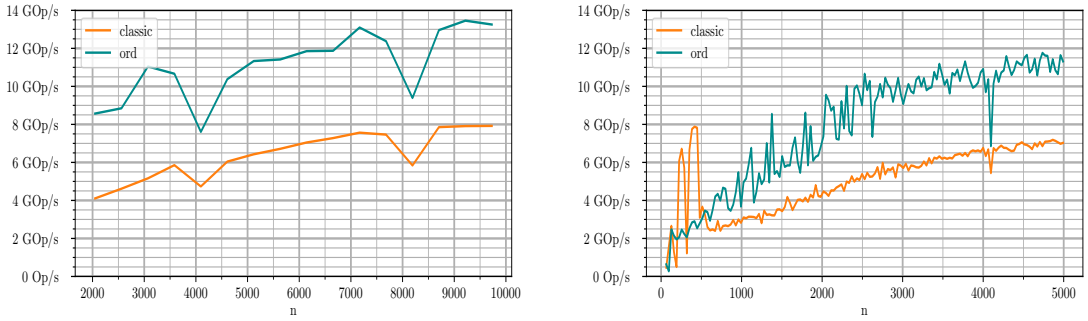
FIGURE 10 – Performance des algorithmes pour des cas usuels

On constate sur la Figure 10 que la version *ord* est équivalente voire meilleure que la version *classic* surtout sur de grandes images, mais moins bonne sur de petites images. Cette observation rejoint la prédiction faite à la partie **2.4**, sur de grandes images les calculs rajoutés par les shadow-zones est négligeable alors qu'il est conséquent sur de petites images.

En outre, on pouvait déjà établir le fait que la version *ord* est optimale lorsque le coût des shadow-zones est largement compensé par une meilleure localité des données par rapport à la version *classic*, c'est à dire quand :

— les images sont grandes
— l'algorithme effectue peu d'itérations de **Stencil**
— l'algorithme calcule beaucoup de niveaux de la pyramide.

On peut donc tracer en Figure 11 les performances des deux algorithmes dans des cas favorables à *ord* afin de constater, on l'espère, une nette amélioration pour la version *ord*. On utilise, pour tracer ces courbes, 2 cœurs Denver et 2 cœurs A57.



(a) 6 niveaux, 6 itérations par niveau

(b) 3 niveaux, 1 itération par niveau

FIGURE 11 – Performance des algorithmes dans des cas favorables à *ord*

On constate bien sur la Figure 11 que la version *ord* est bien plus rapide que la version *classic*. Dans ces deux configurations, l'impact des shadow-zones est négligeable par rapport au gain de performance dû à l'amélioration de la localité des données, comme on pouvait le prévoir. Ces résultats montrent donc qu'il existe des configurations dans lesquelles utiliser la version *ord* permet d'observer des gains de performance significatifs.

Conclusion

Mes travaux durant ce stage ont permis de tester deux stratégies de déploiement, de manière parallélisée, de l'algorithme de traitement d'image qui sera embarqué dans le CubeSat de la mission Meteorix. Ces travaux seront donc utiles à la thèse de Maxime Millet, qui travaille notamment sur la conception de l'algorithme de traitement d'image de Meteorix [6]. De plus, les concepteurs du logiciel pourront choisir, en confrontant mes résultats aux configurations utilisées pour l'algorithme ainsi qu'aux composants de la carte embarquée, s'ils utilisent la version *ord*, la version *classic* ou s'ils privilégient une toute autre approche, par exemple en fournissant le flux vidéo en entrée du programme au lieu de lui fournir les images une par une.

Les résultats obtenus montrent en effet que la version *ord* assure une meilleure localité de données que la version *classic*. Plus généralement, on constate que l'ajout de calculs (ici les calculs des shadow-zones) peut permettre de concevoir un programme plus rapide, lorsque le coût supplémentaire est compensé par une meilleure localité de données. Cette observation contre-intuitive découle directement de l'architecture des ordinateurs, et plus spécifiquement de la différence de vitesse d'accès aux données dans les caches par rapport à la mémoire, ainsi que de la manière dont les processeurs réutilisent des données conservées en cache.

Dans le cadre de ce stage, j'ai appris des méthodes d'optimisation de l'exécution parallèle, en me familiarisant avec l'outil OpenMP dans un premier temps, puis en comprenant que garantir une bonne localité de données est un des enjeux clés de l'optimisation de programmes en parallèle. Cette compréhension des enjeux ainsi que du matériel à ma disposition s'est faite en essayant plusieurs benchmark sur la carte, ainsi qu'en concevant le benchmark MATMUL pour mesurer la fréquence maximale des processeurs en parallèle. J'ai ensuite commencé par établir l'équivalent d'une version *ord* pour la phase ascendante, car le calcul du **DownSampling** n'introduit pas de dépendances de données et donc la comparaison entre les deux manières de paralléliser montrait l'importance de garantir une bonne localité de données. Après avoir optimisé la phase ascendante, dont l'impact sur la durée totale d'exécution de l'algorithme est faible, je me suis concentré sur la phase descendante qui est, de loin, la plus coûteuse. La plus grande difficulté a été l'implémentation de la version *ord* en garantissant son résultat malgré les dépendances de données (introduction des shadow-zones). Enfin, j'ai exploré les configurations de l'algorithme afin de déterminer quand la version *ord* était optimale ou trop coûteuse puis j'ai confronté les résultats de ces expériences avec mes prévisions.

Afin de compléter les résultats obtenus pendant ce stage, on aurait pu envisager des mesures énergétiques afin de comparer les algorithmes obtenus. On prendrait ainsi en compte dans l'optimisation la contrainte essentielle qu'est l'énergie pour l'exécution d'applications sur des systèmes embarqués.

Références

[1] 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. Levasseur-Regourd, J.-L. Rault, and B. Zanda, "Meteorix : a cubesat mission dedicated to the detection of meteors and space debris," in *1st ESA NEO and Debris Detection Conference*, (Darmstadt, Germany), p. 9 p., ESA Space Safety Programme Office, Jan. 2019.

[2] I. Bournias, R. Chotin, and L. Lacassagne, "FPGA Acceleration of the Horn and Schunck Hierarchical algorithm," in *International Symposium on Circuits and Systems (ISCAS)*, (Daegu, South Korea), IEEE, May 2021.

[3] L. Lacassagne, D. Etiemble, H. Zahraee, A. Dominguez, and P. Vezolle, "High Level Transforms for SIMD and Low-Level Computer Vision Algorithms," in *Symposium on Principles and Practice of Parallel Programming / WPMVP*, (Orlando, Florida, United States), p. 8, Feb. 2014.

[4] A. Mancheron, "Une introduction à la programmation parallèle avec Open MPI et OpenMP," *GNU/Linux Magazine*, pp. 16–35, Nov. 2018.

[5] A. Petreto, *Débruitage vidéo temps réel pour systèmes embarqués*. Theses, Sorbonne Université, June 2020.

[6] M. Millet, N. Rambaux, A. Cassagne, M. Bouyer, A. Petreto, and L. Lacassagne, "Meteorix : High performance computer vision application for Meteor detection from a CubeSat," in *44th Scientific Assembly of the Committee on Space Research (COSPAR)*, (Athens, Greece), July 2022.

Annexes

Le LIP6 et l'équipe ALSOC

Le LIP6, unité mixte de recherche (UMR 7606) de Sorbonne Université et du Centre National de la Recherche Scientifique (CNRS), est un laboratoire de recherche en informatique situé dans le 5$^{\text{ème}}$ arrondissement de Paris, sur le campus Pierre et Marie Curie de Sorbonne Université. Le LIP6 se consacre à la modélisation et la résolution de problèmes fondamentaux motivés par les applications, ainsi qu'à la mise en œuvre et la validation des solutions au travers de partenariats académiques et industriels.

La recherche est réalisée au sein de 22 équipes (dont 3 communes avec Inria Paris) articulées autour de quatre axes transverses :

— Intelligence artificielle et sciences des données (AID)
— Architecture, systèmes et réseaux (ASN)
— Sécurité, sûreté et fiabilité (SSR)
— Théorie et outils mathématiques pour l'informatique (TMC)

Constituée de 13 permanents et 7 doctorants, l'équipe ALSOC du LIP6 se positionne sur les axes ASN, SSR et TMC en consacrant sa recherche aux systèmes embarqués.

La thématique de recherche de l'équipe ALSOC porte sur les méthodes de conception des systèmes multiprocesseurs intégrés sur puce, et plus particulièrement les manycores, qui répondent aux besoins en performance de nombreuses applications embarquées (traitement de flux vidéo et multimédia dans les équipements nomades, traitement de paquets dans les équipements télécoms). La conception de ces systèmes nécessite la mise en oeuvre de méthodes de conception conjointe du matériel et du logiciel. Les problèmes à résoudre portent sur l'architecture matérielle, les protocoles de communication, les systèmes d'exploitation embarqués, et plus généralement, les techniques de déploiement des applications, la prise en compte des contraintes temps réel, la vérification formelle des systèmes, le test après fabrication et la génération de code optimisé pour différentes architectures cibles.

Durant mon stage, des échanges fréquents avec Adrien Cassagne m'ont grandement aidé à avancer. L'outil Mattermost permettait de plus de partager *en direct* mes résultats avec l'ensemble de mes encadrants, afin d'avoir un regard extérieur et critique sur mon travail. À ces interactions s'ajoutent les réunions avec mes encadrants qui ont permis d'orienter mon travail. Au cours de mon stage, j'ai aussi pu échanger librement avec l'ensemble des membres de l'équipe ALSOC, notamment lors de moments de convivialité. Enfin, j'ai beaucoup échangé avec des membres de l'équipe SYEL (Systèmes Électroniques) avec qui je partageais le même bureau.

Je tiens d'ailleurs à remercier toutes les personnes qui m'ont accompagné dans ce travail et qui ont permis le bon déroulement de mon stage.

Codes sources

`https://gitlab.aliens-lyon.fr/egalve01/ressources-stage-l3/-/tree/master/Code`