Architectures Parallèles

Optimisation d'un filtre de Sobel

Corentin BEAULIEU - 21 90 69 66

Sommaire

1 Introduction

Le but de ce document est de présenter une tentative d'optimisation d'une application appliquant filtre de Sobel appliqué à une vidéo. Plusieurs étapes clés seront présentées ici, chacune d'elle est associée avec une version du code que vous trouverez sur le dépôt dans lequel vous avez dû trouver ce document.

1.1 Filtre de Sobel

Dans notre cas, on utilise le filtre de Sobel pour faire de la détection de contours dans une vidéo. Cela consiste tout d'abord à passer la dite vidéo en nuances de gris afin d'améliorer la précision de la méthode. Ensuite, pour chaque pixel, on calcule le gradient de son intensité. Enfin, si la magnitude de ce gradient est supérieur à une valeur fixée, alors, on peut considérer qu'il se trouve le long d'un changement brusque d'intensité et par conséquent sûrement sur un contour. On colore donc ce pixel en blanc pour le faire ressortir. On applique ce procédé à tous les pixels constituant toutes les images de notre vidéo d'entrée et on écrit le résultat dans un fichier de sortie. Cette application ne supporte que les vidéos au format 1280×720.

1.2 Choix des métriques

On a calculé plusieurs métriques pour mesurer nos performances. Tout d'abord, on a celles qui sont liées au calcul d'une image (minimum, moyenne, maximum en temps, bande passante et écart-type) puis celles plus globales avec le nombre d'images par secondes et le temps total d'exécution.

Afin d'assurer plus de stabilité, nous avons fait la moyenne de dix exécution pour chaque version du code. Ces mesures sont obtenus grâce à un script *measure.sh* qui redirige toutes les sorties des différentes versions. Ce script gère la compilation et l'exécution des versions ainsi que le calcul moyen des différentes métriques.

La qualité de la sortie est jugée par un humain et peut facilement être jugé à nouveau en lançant la version que l'on souhaite tester. Une méthode aurait été de comparer les hashs de la sortie pour chaque version pour être certain d'avoir exactement la même sortie. Néanmoins, certaines modifications ont apporté un changement dans le fichier de sortie qui n'est donc pas forcément identique entre les versions.

1.3 Description de la machine

Les caractéristiques principales de la machines où ont été effectuées les mesures sont les suivantes

Modèle	Cores	Fréquence	Vectorisation	L1	L2	L3	DRAM	Stockage
i5 4590	4	3.5GHz	SSE, SSE2, AVX, AVX2	32kio	256kio	6144kio	16Mio	HDD 7200 tr/min

Certaines de nos optimisations viseront précisément cette architecture.

Dans un premier temps, nous verrons des améliorations apportées au kernel au travers de la bande passante du calcul d'une image ensuite, on verra des améliorations qui touchent de façon plus globale le temps d'exécution du programme avec la mesure de nombre d'images par secondes et le temps total d'exécution du calcul. Toutes les mesures ont été faites sur le même fichier test (celui fourni et converti à l'aide du script *cvt_vid.sh*). Pour éviter que les mesures soient influencées par le fichier de sortie, celui-ci est supprimer avant chaque mesure et devra donc être recréé par le programme.

2 Présentation des versions

2.1 Options de compilation

Afin d'améliorer nos performances, on a tout d'abord décidé de changer les options de compilation. Le *O1* est devenu un *O3* demandant au compilateur d'essayer plus d'optimisations différentes. On a aussi rajouter *ffast-math* pour permettre des optimisations sur les calculs mathématiques au détriment de la précision (ce qui dans notre cas parait comme un échange valide). Enfin, on a rajouté *funroll-loops* et *ftree-vectorize* afin de tester à nouveau les heuristiques de déroulage de boucle et de vectorisation et peut-être pouvoir les passer là où elles ont pu échouer dans le *O3*.

Une piste non explorée est l'utilisation d'autres compilateurs que *GCC*. Cela aurait pu nous amener un gain de performance aussi.

Cette modification est la première effectuée car elle permettra d'amplifier les différentes optimisations possibles.

Avec ces nouvelles options de compilation, on souhaite aider le compilateur à dérouler les boucles et vectoriser les calculs. Pour cela, on doit aligner la mémoire. Cela est fait grâce aux appels à `_mm_malloc` et `_mm_free` déjà présents dans la *baseline* mais sûrement sous-exploité par le compilateur.

2.2 Prise en compte du greyscale

On a remarqué que après le passage au gris, les trois composantes de chaque pixel sont identiques. De là, on s'est dit qu'il n'était pas nécessaire de faire le calcul du filtre de Sobel sur ces trois valeurs identiques mais seulement sur l'une d'elle que l'on réécrit en triple au final. De plus, ainsi on évite un éventuel décalage de couleur et on conserve des images uniquement en nuances de gris.

Pour faire cela, on a modifié la fonction `greyscale_weighted`. Elle alloue (de façon aligné) un tableau d'un tiers de la taille d'une image. Elle fait ensuite le calcul des pixels en gris un à un et les stocke dans le tableau nouvellement créé. Ce tableau est retourné par la fonction et l'image d'entrée peut être désallouée. On obtient ici une première amélioration quant à l'espace mémoire utilisé.

Ensuite, on a modifié la fonction `sobel_baseline` en `sobel_grey` pour prendre en compte ce changement. Le tableau passé à la fonction est maintenant notre tableau contenant qu'une valeur par pixel. On calcule alors trois fois moins de valeurs que précédemment. Afin d'obtenir une vidéo valable en sortie, on doit écrire la valeur calculée trois fois pour obtenir un pixel gris correspondant à notre détection de contours.

2.3 Optimisation du coeur de calcul

Dans cette partie, nous allons voir plusieurs petits changement que nous avons opéré afin d'améliorer le temps d'exécution et/ou l'encombrement mémoire de notre programme.

Dans cette optique, nous avons utilisé des spécificités du langage C. Les variables ont eu leur taille changée pour correspondre au plus proche à ce qu'elles vont contenir. Nous avons rendu constantes celle qui le pouvait. On a ensuite utilisé le mot clé `restrict` pour spécifier qu'un pointeur est le seul accès vers un objet et ainsi aider le compilateur dans ces heuristiques d'optimisations notamment de vectorisation.

On a passé l'image de nuances de gris à noir et blanc ce qui change énormément le résultat. Néanmoins, cela nous permet de nous affranchir d'un branchement en fin de boucle.

Ensuite, on a remplacé la racine carrée au coeur du calcul du filtre par une approximation. Plus exactement, on calcule le carré du *threshold* et on le compare à la somme des carré des gradients.

Comme le *threshold* est positif on a la relation suivante

$$\sqrt{g_x^2 + g_y^2} > threshold \Rightarrow \sqrt{g_x^2 + g_y^2} > \sqrt{threshold^2}$$

Or, la fonction racine carrée est strictement croissante sur $\mathbb{N}$. D'où

$$\sqrt{g_x^2 + g_y^2} > \sqrt{threshold^2} \Rightarrow g_x^2 + g_y^2 > threshold^2$$

Ce résulat permet de nous affranchir d'un calcul coûteux de racine carrée. De plus, comme nous sommes passé en noir et blanc, nous n'avons plus besoin du résultat de cette racine carrée.

Une autre approximation est de calculer une norme de manhattan plutôt que la norme deux comme actuellement. On aurait alors, $mag = |g_x| + |g_y|$. Néanmoins, après un test, on a vu que ce changement ne permettait pas de gain de performance significatif tout en détériorant fortement la qualité du résultat (on voit apparaître plus de bruit dans notre filtre). Il est toutefois possible que ce non gain de performance soit dû à notre implémentation. Il pourrait être intéressant de creuser cette piste plus en profondeur dans le futur.

On a aussi retirer les zéros des matrices utilisées pour la convolution ce qui nous permet d'avoir un gain de place. Nous avons modifié la convolution pour ne pas calculer les multiplications par zéros précédemment présentes et ainsi gagner un tiers des calculs de cette dernière.

Afin de préserver une sorte de qualité dans notre résultat malgré l'approximation, on a décidé d'implémenter le filtre avec des matrices de 7×7 présenté dans le papier de Victor BOGDAN, Cosmin BONCHIS et Ciprian ORHEI qui est trouvable sur le dépôt sous le nom *paper.pdf* et résumé dans *summary.pdf*. On a pas observé de baisse significative de nos performances tout en ayant une détection de contour nettement plus précise.

Depuis cette partie, une version en blocs a été expérimenté. Le but était de garantir la localité des données dans le cache en découpant l'image en bloc de taille constante. Cette version a mené lors de nos tests à une baisse drastique de la performance dans le kernel de calcul, la bande passante étant divisée par un peu plus de huit. Cette version a donc été abandonné et n'a pas servi de base aux versions suivantes.

2.4 Entrées/Sorties

Le but dans cette partie est de limiter la latence due aux entrées sorties. A la suite des modifications que nous allons expliquer nous ne nous attendons pas à une amélioration du temps de calcul d'une frame (voir même l'inverse) mais une diminution globale du temps d'exécution du programme dans son entièreté.

Tout d'abord, on a retiré les affichages à chaque image calculée qui sont assez coûteuses pour une information redondante que l'on obtient sous forme de moyenne à la fin de l'exécution du programme.

Ensuite, on a souhaité bufferiser un peu plus les accès au stockage de masse. En effet, jusqu'ici on lisait une image à la fois, on la calculait et puis on l'écrivait. On veut donc faire plus d'une image par accès au stockage. On a choisi de mapper un morceau du fichier en mémoire en utilisant *mmap*. On prend presque arbitrairement une taille de 360 images pour la taille maximum que nous chargeons en mémoire. Dans les fait, cette taille correspond exactement au fichier test. De plus, c'est un multiple de quatre ce qui permettra une parallélisation plus simple dans la partie suivante. Enfin, 360 images d'une résolution de 1280×720 représente environ 1Go donc avec nos deux projections, on atteint une empreinte mémoire d'environ 2Go pour l'application entière. Cette taille pourrait être changée pour mieux correspondre à d'autres cas et machines.

On écrit exactement la taille de ce que l'on lit. On utilise pour l'écriture le flag `MAP_SHARED`. L'écriture se fait donc via une écriture mémoire dans le kernel de calcul qui est ensuite synchronisé avec le disque de stockage. L'écriture a donc lieu maintenant au coeur du calcul d'une image ce qui induit une baisse de la performance lors du calcul d'une frame mais on s'attend à voir une augmentation drastique de la performance de notre application dans son entièreté.

2.5 Parallélisation

Maintenant que nos performances ont été améliorées sur une version séquentielle du programme, nous allons paralléliser.

Nous avons décider de paralléliser au niveau des frames soit la boucle la plus externe du programme. Chaque thread sera chargé de calculer le résultat du filtre sur un paquet d'images contiguës. Notre machine disposant de quatre coeurs, la décision a été prise de ne créer que quatre threads pour éviter l'overhead liée à leur création.

Chaque thread reçoit une structure contenant le nombre d'images qu'il a à calculer et les adresses des débuts de ses parties dans les données d'entrées, le tableau des *samples* pour le calcul de la performance et dans les données de sortie. Il est alors chargé de calculer chacune des images dont il a la responsabilité et de les écrire par la même occasion.

Les morceaux du fichier d'entrée et de sortie que nous avons chargé en mémoire précédemment sont donc divisé en quatre parties contiguës. Avec la taille choisie dans la partie précédente, chaque threads calcule et écrit 90 frames. Dans cette implémentation assez basique, de nombreux points sont améliorables comme il sera discuté dans la partie 4.

3 Présentation des résultats

3.1 Performance du noyau de calcul

Nous discuterons d'abord des performances des quatre premières versions à l'aide des mesures de bande passante.

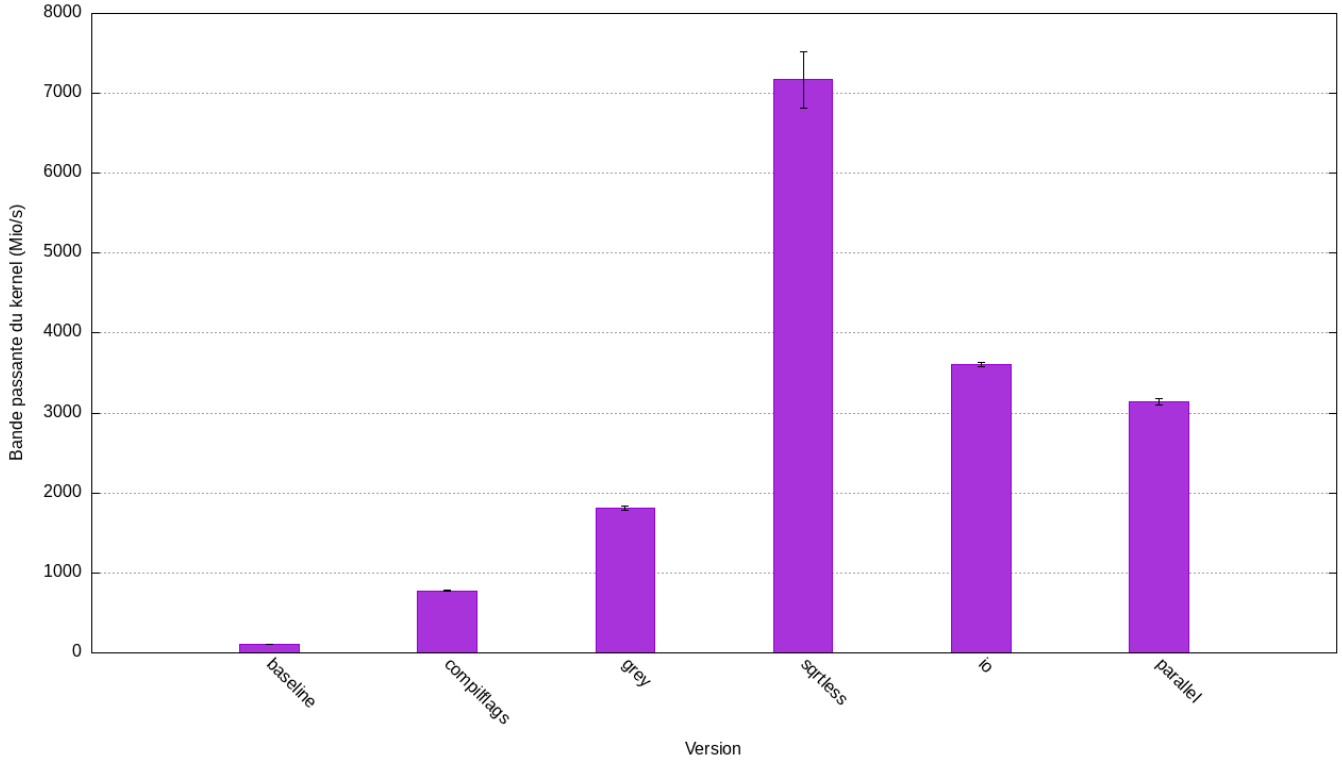

Fig. 1 – Bande passante du noyau de calcul pour les différentes versions

On voit dans le graphique 1, que chacune des quatre premières versions a eu une influence positive sur la bande passante dans le noyau de calcul par rapport à la précédente.

Dans les fait, on observe une augmentation de $\times 7.7$ entre la version de base et celle où plus d'options de compilations ont été utilisées.

On voit une augmentation de $\times 2.3$ entre cette dernière et la version prenant en compte la spécificité du passage en nuance de gris. Cela est plutôt cohérent avec le fait qu'on ait environ divisé par trois le nombre d'opérations au sein du kernel.

Finalement, on voit que l'approximation de la racine carrée et les autres modifications qui ont été apportées dans la version *sqrtless* par rapport à la précédente, nous ont permis de multiplier la bande passante au sein du kernel par environ quatre. Tous ça donne que cette dernier version a une bande passante dans le kernel plus de 70 fois supérieure à celle de la version de base.

Dans la 2, le constat n'est pas tout à fait le même. La version *sqrtless* est environ deux fois plus rapide dans son entièreté que la version *baseline*. Néanmoins, elle n'a pas d'augmentation nette de sa bande passante totale par rapport aux versions avec options de compilations et prise en compte des nuances de gris.

On arrive à la conclusion que les performances globales de l'application sont limitées par les accès au stockage. En effet, l'accélération de $\times 10$ entre la version avec options de compilation et la version avec l'approximation de la racine carrée n'a aucune réelle incidence sur la bande passante globale de l'application. On en déduit facilement que si les calculs ne sont pas limitant alors ce sont peut-être les accès disques, ce qui est cohérent avec l'utilisation d'un disque dur sur la machine sur laquelle ont été réalisé les mesures.

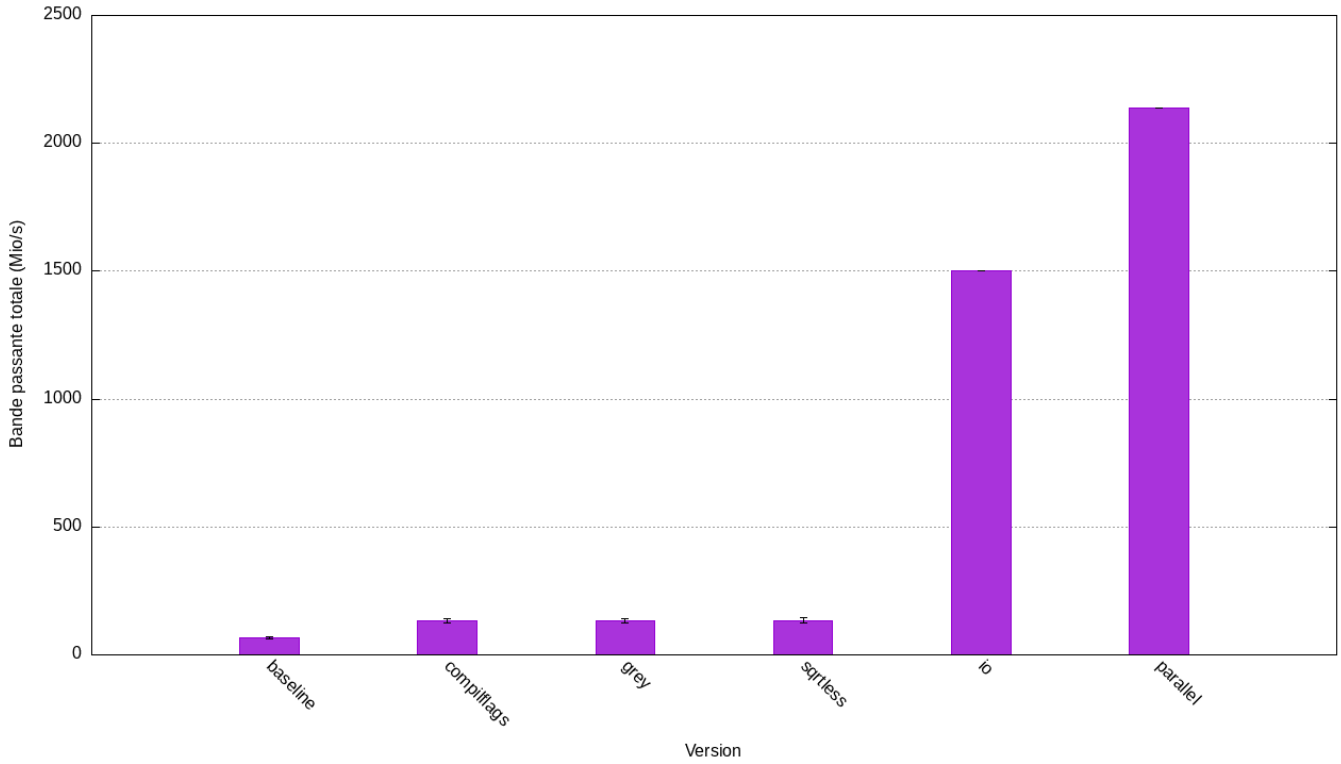

Fig. 2 – Bande passante total de l'application pour les différentes versions

3.2 Performance de l'application

Nous allons maintenant nous intéresser aux dernières versions grâce au mesures du temps d'exécution total et du nombre d'images calculées par secondes.

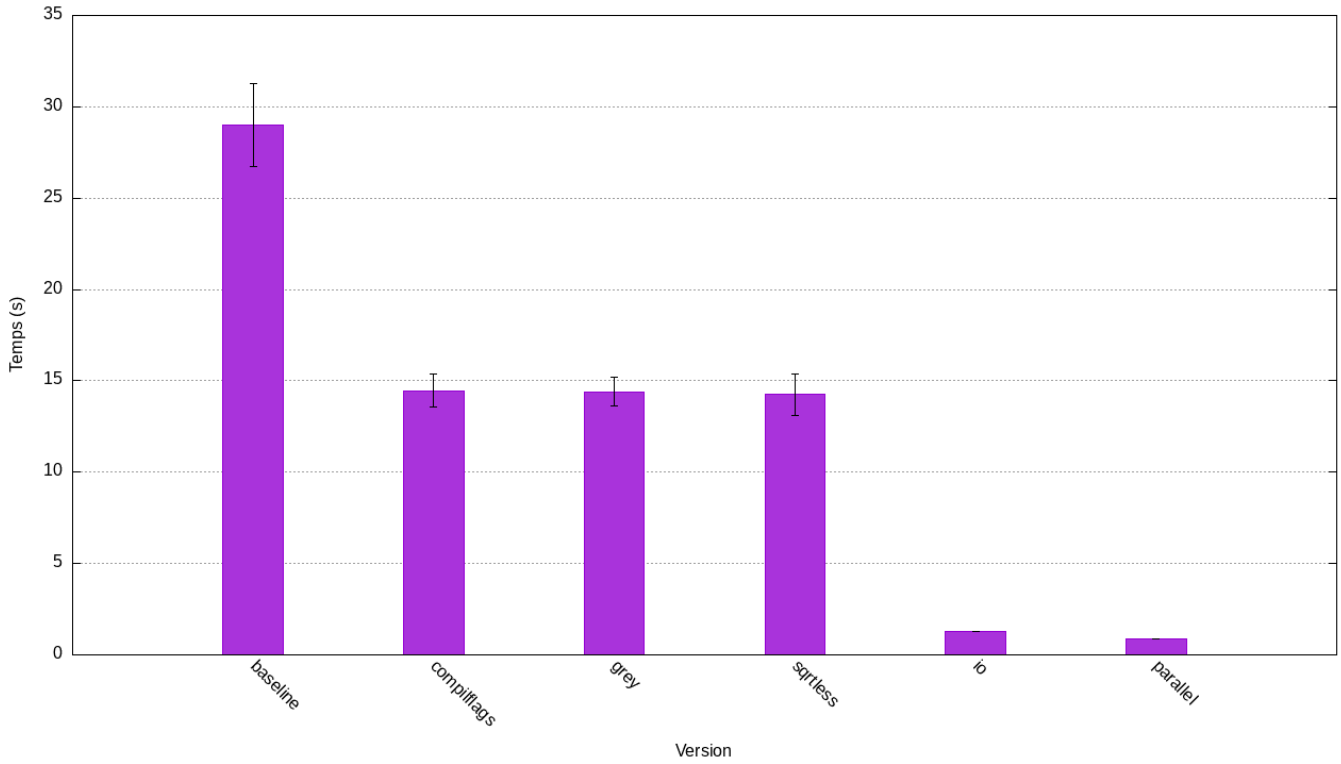

Fig. 3 – Temps total d'exécution de l'application pour les différentes versions

Pour commencer, revenons sur le graphque 1. On observe que la version *io* est environ deux fois moins performante que la version *sqrtless* dans le noyau de calcul. Cela correspond à nos attentes, en effet, l'écriture a été déplacé à

l'intérieur du calcul du filtre du fait de l'utilisation de `mmap`. On observe aussi que la dernière version *parallel* est moins performante encore dans le kernel.

La 2 quant à elle montre quelque chose de bien différent que l'on retrouve dans les autres graphiques.

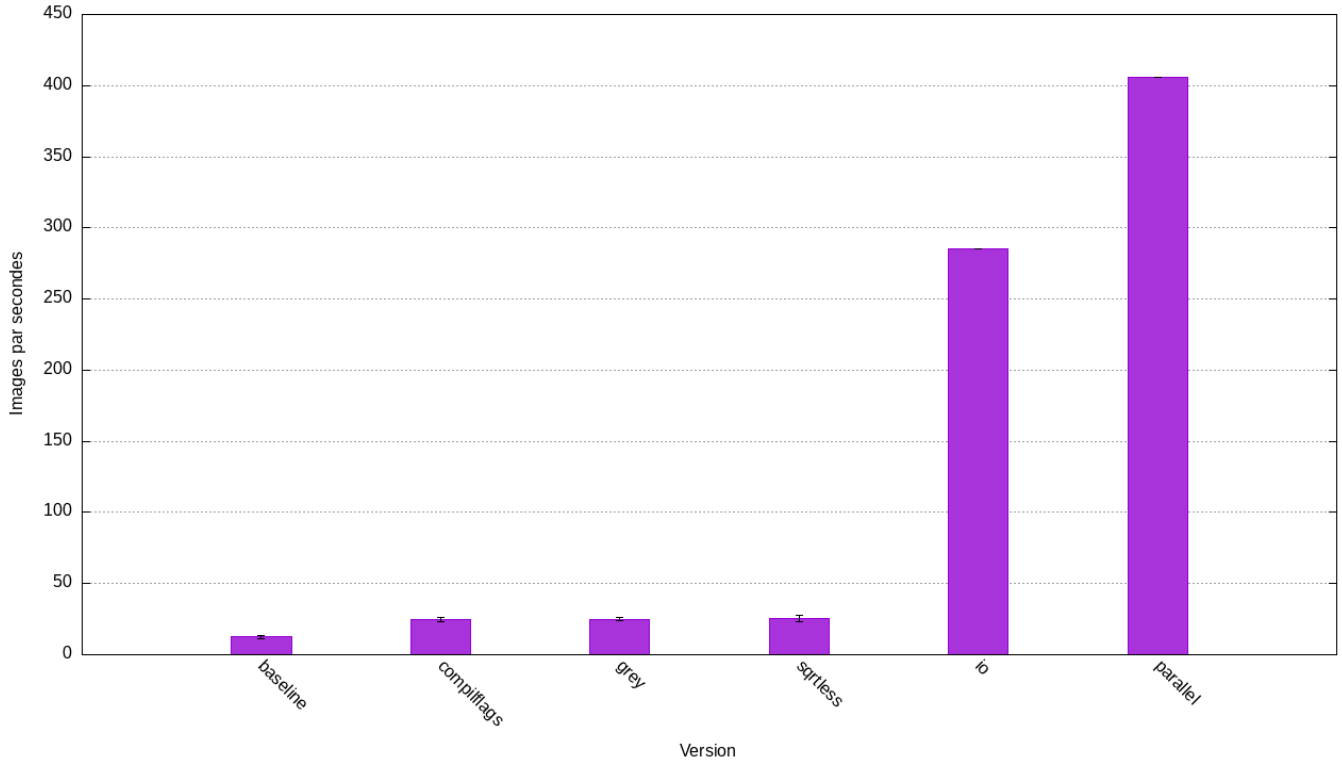

Fig. 4 – Nombre moyen d'images calculées par seconde par les différentes versions de l'application

Comme vu précédemment, on a la confirmation dans le graphique 3 que la version *baseline* est environ deux fois plus lente dans son ensemble que les versions *compilflags*, *grey* et *sqrtless* qui sont quant à elle à peu près aussi rapide les unes que les autres.

On avait émis l'hypothèse que les accès au disque prenait le pas sur le calcul du noyau. On peut confirmer cette hypothèse. En effet, on voit que la bufferisation des entrées/sorties nous a permis d'obtenir une accélération de $\times 10$ sur temps total par rapport aux dernières versions sans.

Enfin, la version parallélisée partiellement est plus rapide que la version bufferisée simplement. Néanmoins, malgré le fait que nous parallélisions avec quatre threads sur une machine possédant quatre coeurs, on observe que le speedup est seulement de l'ordre 1.5. On peut expliquer cela par le fait que la parallélisation ait été faite à la main et que la section parallèle ne couvre pas toutes les fonctions coûteuses. Par exemple, le calcul des nuances de gris se fait encore de façon séquentielle.

On voit toutefois que l'on réussi à obtenir jusque-là une augmentation du nombre d'images traitées à la secondes par l'application, entre la toute première version et la dernière version parallèle, de l'ordre de $\times 33$.

Pour conclure, on voulons rappeler que les résultats que nous avons obtenu et les analyses que nous avons pu en tirer sont fortement liés à la machine sur laquelle nous avons compilé et exécuté le code. En effet, certaines options de compilation (principalement `march=native`) peuvent amener des optimisations plus efficaces sur les architectures que les supports notamment avec l'utilisation de registres de vectorisation plus longs.

4 Pistes d'amélioration

Dans cette section, nous nous efforcerons de trouver des pistes d'amélioration. On va s'intéresser à des potentiels optimisations qui n'ont pas pu être testées ou alors pas suffisamment faute de temps.

Comme dit dans 2.1, on aurait pu tester d'autres compilateur (*clang*, *icc* ou *icx*) pour voir lequel est celui qui nous apporte les meilleures performances selon la version.

Dans la partie 2.3, nous avons abordé la possibilité d'utiliser une distance de manhattan pour le calcul de magnitude du gradient. De rapides tests n'ont pas montré de réels gains par rapport à la méthode actuellement en place mais il peut être intéressant de creuser plus en détail cette possibilité. De plus, on a changé la matrice pour une étendue. Il est possible que contrairement aux premières observations, cela ait par la suite eu un coût supplémentaire sur le temps d'exécution de l'application. On peut rajouter la version avec un calcul par blocs qui n'a pas eu le gain de performance escompté bien au contraire. Cette version pourrait être retravailler, la baisse de performance se trouvant peut-être plus dans l'implémentation que dans la méthode elle-même.

Ensuite, dans la partie 2.4, on a changé les entrées/sorties pour travailler sur plus de données à la fois. La taille d'un de nos buffers a été choisi de façon assez arbitraire. Il peut être intéressant de mener des tests pour trouver la taille optimum. Ce genre de changement peut ne pas vraiment ce voir sur notre fichier test. Il est même fortement probable que la taille choisie pour ce fichier, avec notre implémentation, ne soit pas optimal pour d'autres entrées.

Il est tout à fait envisageable de tester d'autre façon de charger et écriture une grosse partie des fichiers d'un coup. Il est par exemple possible d'utiliser comme dans les premières versions des `fread` et `fwrite` pour cela.

Dans la partie 2.5 portant sur la parallélisation de notre programme, nous avons présenté une version très simpliste qui peut être amélioré en de nombreux points.

Tout d'abord, le passage de l'image en nuances de gris se fait toujours de façon séquentielle. Il est possible de paralléliser la boucle interne, la boucle sur les frames ou encore incorporer cette fonction dans le calcul des frames par les threads que l'on utilise actuellement. La méthode qui semble la plus optimale est la dernière, chaque thread gère le passage en nuances de gris du paquet d'images qu'il doit calculer. Néanmoins, cela demande de revoir comment sont gérées les entrées/sorties. Soit, la fonction `greyscale_weighted` renvoie toujours un pointeur vers un tableau alloué et alors, il nous faut démapper le fichier d'entrées lorsque plus aucun thread n'en a besoin, avec peut-être l'utilisation d'une barrière ou d'une variable globale. Soit, on change le comportement de la fonction pour ce rapprocher de celui de départ, c'est-à-dire que la version nuancée sera stockée dans une partie de l'espace mémoire dans lequel on a projeté le fichier d'entrée. Cette deuxième solution demande une plus grande utilisation de la mémoire centrale.

Ensuite, le reste du fichier, lorsque qu'il ne reste plus de morceau de 360 frames à calculer, est calculé de façon séquentielle. Ce changement n'aura pas d'effet sur notre fichier test mais il peut avoir une grosse influence si l'entrée a par exemple un peu moins d'un multiple de 360 images en tout. Il conviendrait donc de paralléliser cette partie ou d'intégrer le calcul des miettes dans la section déjà parallélisée.

Enfin, l'utilisation des threads de la librairie standard n'est pas forcément la meilleure solution. Il peut être intéressant de comparer les performances que l'on a avec cette méthode et l'utilisation d'une autre bibliothèque de calcul parallèle comme *OpenMP*.

Une autre question que l'on peut se poser dans la poursuite d'une meilleure performance est à quel point peut-on sacrifier de la qualité du résultat final ? Avec les modifications que nous avons apporté, nous avons fortement changé le résultat de l'application. Néanmoins, il est possible de dégrader volontairement la sortie pour obtenir un gain en performance. Il est par exemple possible de renvoyer une vidéo où une image sur deux n'est pas calculée mais seulement la copie de la précédente. Ou encore, on peut diviser la résolution. Ces potentielles améliorations d'un point de vue de la performance serait néanmoins d'énorme dégradation de la qualité de la sortie.

Enfin, on aimerait mettre ici en lumière la possibilité de mesurer les performances de l'application autrement. On pourrait ainsi utiliser des outils externes plutôt que de faire des mesures nous-même au sein de l'application. Ensuite, le script de prise de mesures est très largement perfectible, il faut souvent le lancer plusieurs fois avant d'obtenir une erreur assez basse pour que les résultats soient exploitables. On a remarqué notamment un soucis de stabilité pour les deux dernières versions.