Projet d'Architecture Parallèle

Filtre de Sobel

EJJED Zakaria

https://github.com/Lilifus/sobel_optimisation

Sommaire

1 Introduction

1.1 Présentation du projet

Dans ce rapport, nous allons étudier les différentes méthodes d'optimisation pour l'algorithme de détection de bords de Sobel. Ce filtre est couramment utilisé pour mettre en évidence les contours d'une image, et il est souvent utilisé en vision par ordinateur et en analyse d'image. Nous allons examiner les différentes techniques d'optimisation, telles que la parallélisation, la vectorisation et la restructuration de données, et mesurer leur impact sur les performances de l'algorithme. Nous utiliserons aussi différents compilateurs ainsi que différentes options de compilations. Enfin, nous allons présenter les résultats obtenus et discuter des implications pour l'optimisation de l'algorithme de détection de bords de Sobel.

1.2 Caractéristique du processeur.

Modèle	Architecture	cache L1	cache L2	cache L3	frequence	Vectorisation
AMD FX-8350	Piledriver	128Kio	8Mio	8Mio	4GHz	SSE, SSE2, AVX

1.3 Présentation de l'algorithme de détéction de bords de Sobel

L'algorithme de détection de bords de Sobel est un algorithme de traitement d'image utilisé pour mettre en évidence les contours d'une image. Il se base sur l'utilisation de filtres de convolution pour calculer les gradients de l'image dans les directions horizontale et verticale. Les gradients sont ensuite combinés pour obtenir un résultat final qui met en évidence les bords de l'image.

Le filtre de Sobel est constitué de deux matrices 3×3 qui sont utilisées pour calculer les gradients dans les directions horizontale et verticale. Ces matrices sont généralement définies comme suit:

$$\text{f1}= \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, \text{ et f2}= \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

La convolution de l'image avec ces matrices permet de calculer les gradients dans les directions horizontale et verticale. Le gradient final est calculé en utilisant la somme des carrés des gradients dans les deux directions. Enfin, une threshold est utilisée pour déterminer si un pixel est un bord ou non. Les pixels dont la valeur de gradient dépasse la threshold sont considérés comme des bords et sont marqués en blanc dans l'image de sortie, les autres pixels sont marqués en noir.

En résumé, l'algorithme de détection de bords de Sobel utilise des filtres de convolution pour calculer les gradients de l'image dans les directions horizontale et verticale, puis combiner ces gradients pour obtenir un résultat final qui met en évidence les bords de l'image en utilisant une threshold pour déterminer si un pixel est un bord ou non.

1.4 Objectifs de l'optimisation

Les objectifs de l'optimisation d'un programme de filtre de Sobel sont généralement les suivants:

1.Améliorer les performances en termes de temps d'exécution: Le but est de diminuer le temps nécessaire pour traiter une image en utilisant l'algorithme de Sobel.

2.Utiliser efficacement les capacités matérielles: L'optimisation doit permettre d'utiliser efficacement les ressources matérielles telles que les coeurs CPU et les instructions vectorielles pour améliorer les performances.

3.Scalabilité: L'optimisation doit permettre d'élargir le nombre de threads pour une meilleure utilisation des ressources matérielles en cas de besoin.

4.Fiabilité: L'optimisation doit garantir que le résultat final de l'algorithme est le même que celui obtenu avec l'algorithme original. En somme, les objectifs de l'optimisation d'un programme de filtre de Sobel sont d'améliorer les performances en utilisant efficacement les ressources matérielles, de garantir la fiabilité et la flexibilité du résultat final, et de faciliter la portabilité du code.

1.5 Méthodologie utilisé

Tout d'abord, nous avons tester les différentes options de compilations sur différent compilateurs, ensuite il a été nécessaire d'étudier le fonctionnement du programme et les améliorations possibles, pour cela on a utilisé MAQAO qui un outil permettant l'analyse statique et dynamique de programme executé et qui a permis d'identifier les instructions complexes ainsi que les goulots d'étranglements tout en faisant des retours sur le niveau d'optimisation du programmes.
Par la suite, on a essayé d'incorporé différentes optimisations les unes après les autres mais sans les combinés, enfin on a observé la vitesse d'execution et mesuré la bandwidth sur des versions combinant différentes optimisations pour en tirer certaines conclusions.

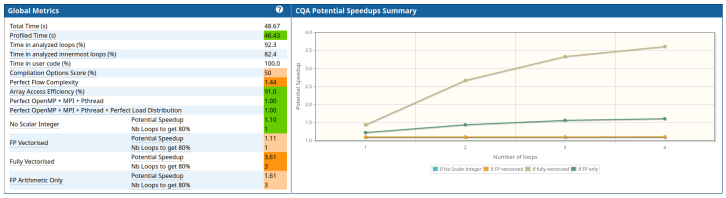
2 Analyse du programme de base

2.1 Analyse globale

Le programme fourni était avait tout d'abord une execution très lente, avec 0.11368115 secondes d'execution par frame, avec 360 frames à convertir et 46MiB/s de bandwidth (valeur que nous avons réutiliser pour calculer le speedup approximatif).

2.2 Mesure de performances

En utilisant MAQAO (Modular Assembly Quality Analyzer and Optimizer), on peut observer que dès la compilation des flags d'optimisations sont manquant, que le programme pourrait profiter d'un peu de vectorisation, et en plongeant un peu plus en profondeurs on observe aussi que le programme est ralenti par des instructions comme `sqrt`.



3 Optimisation grâce aux options de compilation

3.1 Présentation des options de compilations

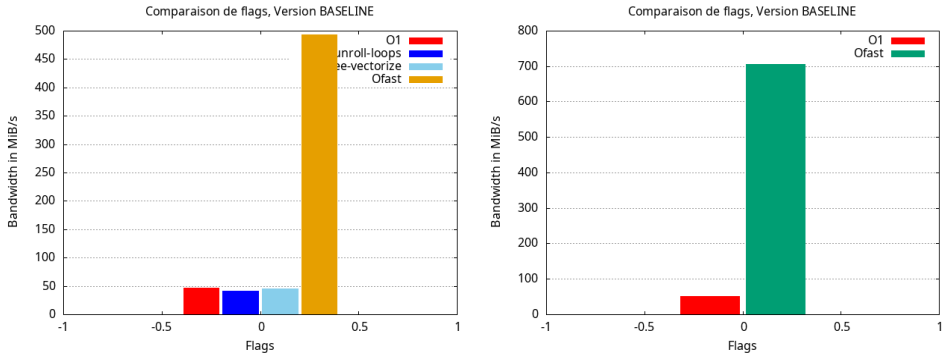
Tout d'abord, afin d'optimiser un code à la compilation il existe l'option `-O` qui se décline en 5 niveaux d'optimisations: `-O0`, `-O1`, `-O2`, `-O3` et `-Ofast`. Chaque niveau d'optimisation regroupe un lot d'options d'optimisation plus explicite, comme `-funroll-loops`.

3.2 Utilisation des options

Afin d'optimiser notre programme on a principalement utilisé et comparé les niveaux `-O1` et `-O3` avec entre autres l'utilisation occasionnel de `-funroll-loops` et `-ftree-vectorize` qui permettent respectivement le déroulage de boucles et la vectorisation du programme par le compilateur. On a aussi utilisé de manière presque systématique l'option `-march=native` afin d'avoir des optimisation adapté à l'architecture de la machine.

3.3 Analyse des performances

Avec l'utilisation de gcc, on remarque que la différence entre l'utilisation de `-O1` ou de `-O1 -funroll-loops/-ftree-vectorize` ne provoque pas de différence notable, par contre l'utilisation d'`-Ofast` provoque un speedup de 10.7. Lors de l'utilisation de clang, on observe aussi un speedup considérable (15.35) lors du passage à `-Ofast` qui implique avec lui une grande quantité d'optimisation à la compilation.



Pour ces mesures les versions suivantes de compilateurs ont été utilisé:

Compilateur	Version
gcc	12.2.1
clang	15.0.7

4 Optimisation avec déroulage

4.1 Présentation de l'optimisation par déroulage

L'optimisation par déroulage (loop unrolling) est une technique utilisée pour améliorer les performances d'un programme en réduisant le nombre d'itérations d'une boucle. Elle consiste à répéter plusieurs fois le corps de la boucle, avec des variables d'itération différentes, plutôt qu'une seule fois, cela permet de réduire le nombre de sauts de boucle et d'améliorer les performances du programme.

L'optimisation par déroulage est particulièrement utile pour les boucles qui effectuent des calculs simples et répétitifs, comme les boucles de convolution dans l'algorithme de filtre de Sobel. En déroulant ces boucles, on peut réduire le nombre d'instructions nécessaires pour effectuer les calculs, ce qui entraîne une amélioration des performances.

Pour mettre en place l'optimisation par déroulage dans un programme, il est nécessaire de répéter manuellement plusieurs fois le corps de la boucle, en modifiant les variables d'itération à chaque répétition. Il est important de noter que l'optimisation par déroulage peut entraîner une augmentation de la taille du code en raison de la duplication du corps de la boucle, il est donc important de choisir judicieusement les boucles à dérouler et de limiter le nombre de répétitions pour éviter un surdimensionnement du code.

Il est également important de noter que l'optimisation par déroulage peut avoir un impact sur la mémoire cache, en raison de la duplication du corps de la boucle, il peut y avoir des conflits de cache et une diminution de la performance. Il est donc important de tester différents nombres de répétitions pour trouver un compromis entre les performances et l'utilisation de la mémoire cache.

En résumé, l'optimisation par déroulage est une technique utilisée pour améliorer les performances d'un programme en réduisant le nombre d'itérations d'une boucle en répétant plusieurs fois le corps de la boucle avec des variables d'itération différentes. Cependant, il est important de choisir judicieusement les boucles à dérouler et de limiter le nombre de répétitions pour éviter un surdimensionnement du code et des conflits de cache.

4.2 Modification du code avec déroulage

Afin de dérouler la fonction de convolution, la valeur d'`UNROLL` a été initialisé à 8 et le corps de la boucle intérieur a été répété 8 fois avec une seconde boucle pour s'occuper des restes.

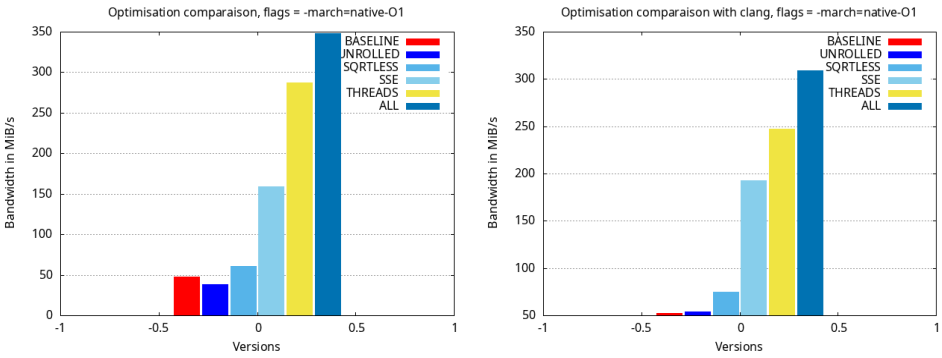
La fonction ensuite renommé `convolve_unroll` est implémenté de la façon suivante:

4.3 Analyse des performances suite au déroulage

Contrairement à ce que l'on aurait pu imaginer les performances du filtres de sobel se réduisent avec l'utilisation du déroulage, on passe ainsi de 46MiB/s de bandwidth à 38.65MiB/s ce qui revient à un "speedup" de 0.84. La raison de ce ralentissement pourrait être dû au compilateur qui même avec l'option O1 est capable de dérouler la boucle de manière plus optimale, et en la déroulant à la main on complexifie la fonction et on empêche le compilateur d'optimiser correctement la boucle.

Algorithm 1 convolve_unroll

```
1:  function CONVOLVE_UNROLL(m, f, fh, fw)
2:      r ← 0
3:      for i ← 0 to fh − 1 do
4:          for j ← 0 to fw − 1 step UNROLL do
5:              r ← r + m[INDEX(i, j, W ∗ 3)] ∗ f[INDEX(i, j, fw)]
6:              r ← r + m[INDEX(i, j + 1, W ∗ 3)] ∗ f[INDEX(i, j + 1, fw)]
7:              r ← r + m[INDEX(i, j + 2, W ∗ 3)] ∗ f[INDEX(i, j + 2, fw)]
8:              r ← r + m[INDEX(i, j + 3, W ∗ 3)] ∗ f[INDEX(i, j + 3, fw)]
9:              r ← r + m[INDEX(i, j + 4, W ∗ 3)] ∗ f[INDEX(i, j + 4, fw)]
10:             r ← r + m[INDEX(i, j + 5, W ∗ 3)] ∗ f[INDEX(i, j + 5, fw)]
11:             r ← r + m[INDEX(i, j + 6, W ∗ 3)] ∗ f[INDEX(i, j + 6, fw)]
12:             r ← r + m[INDEX(i, j + 7, W ∗ 3)] ∗ f[INDEX(i, j + 7, fw)]
13:         end for
14:         for j ← fw − (fw&(UNROLL − 1)) to fw − 1 do
15:             r ← r + m[INDEX(i, j ∗ 3, W ∗ 3)] ∗ f[INDEX(i, j, fw)]
16:         end for
17:     end for
18:     return r
19: end function
```



5 Optimisation avec SSE

5.1 Présentation de l'instruction set SSE

Le jeu d'instructions SSE (Streaming SIMD Extensions) est un jeu d'instructions pour processeurs x86 qui peut être utilisé pour calculer plusieurs données simultanément. Introduit par Intel en 1999 pour le processeur Pentium III, SSE permet de traiter les données à l'aide de registres 128 bits pouvant stocker 4 nombres à virgule flottante (32bits) ou 8 entiers (16 bits). Les instructions SSE sont utilisées pour accélérer les opérations arithmétiques (addition, soustraction, multiplication) et les calculs mathématiques tels que les opérations de comparaison avec des nombres à virgule flottante ou des entiers utilisant des registres de 128 bits. Vous pouvez également utiliser les instructions SSE pour effectuer des opérations de chargement et de stockage économes en mémoire et des opérations de conversion de type de données. Les instructions SSE sont

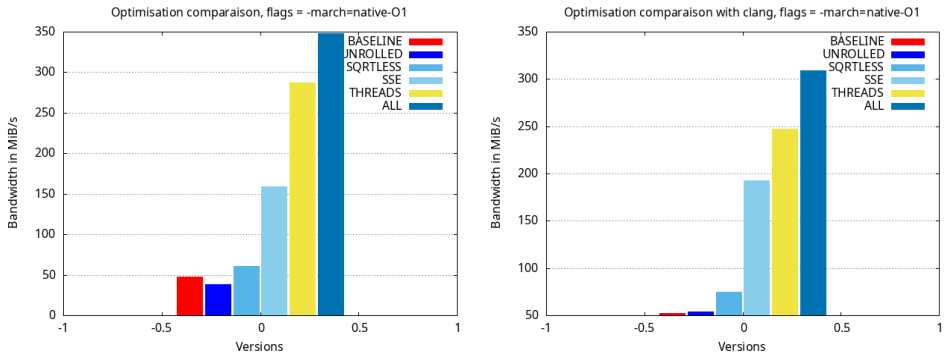
utilisées pour accélérer le calcul dans des domaines tels que la vision par ordinateur, la reconnaissance de formes, la simulation physique et le calcul scientifique. Les instructions SSE vous permettent de traiter plusieurs éléments de données simultanément, ce qui peut améliorer les performances de calcul par rapport à l'utilisation d'instructions scalaires (traitement d'une donnée à la fois).

5.2 Utilisation du SSE

Les fonctions `sobel_baseline` et `convolve_baseline` ont été réimplémenté en intrasic SSE afin d'essayer de vectoriser le programme, n'étant pas très à l'aise avec l'intrasic, il a été décidé de reproduire l'execution de la même façon mais avec de la vectorisation, il est donc possible que des fonctions beaucoup plus efficaces que celles utilisées soit utilisable. De plus on a pris la décision de conserver la fonctions sobel qui appelle la fonction convolve pour obtenir un i32. Nous avons d'abord implémenter l'intrasic de cette façon pour s'assurer du fonctionnement de l'intrasic ainsi que de la justesse des résultats, mais cela empêche malheureusement une vectorisation totale qui n'a pas été implémenté par manque de temps, de connaissance et de pratique.

5.3 Analyse des performances avec SSE

L'implémentation du SSE - même si incomplète - permet un speedup de 3.5, avec une bandwidth de 159MiB/s et un temps d'execution de 0.03317469175s/frame, d'après MAQAO un speedup de 1.85 serait possible si on arrivait à vectoriser complètement le programme.



6 Optimisation grâce au multithreading

6.1 Présentation de pthreads

pthreads est une bibliothèque qui permet de créer et gérer des threads (ou des fils d'exécution) dans un programme. Les threads sont des sous-processus légers qui partagent les mêmes ressources mémoire qu'un processus principal. Ils peuvent être utilisés pour améliorer les performances d'un programme en exécutant des tâches de manière parallèle.

L'utilisation de pthreads permet de diviser les tâches d'un programme en plusieurs parties qui peuvent être exécutées simultanément par différents threads. Cela permet d'utiliser plus efficacement les ressources système, comme les processeurs, les coeurs, les mémoires cache et les accès disques, pour améliorer les performances.

Les threads peuvent être utilisés pour effectuer des tâches telles que des calculs, des entrées/sorties, des communications réseau, etc. Il est également possible de synchroniser les threads pour gérer les dépendances entre les tâches et éviter les conflits d'accès aux ressources partagées.

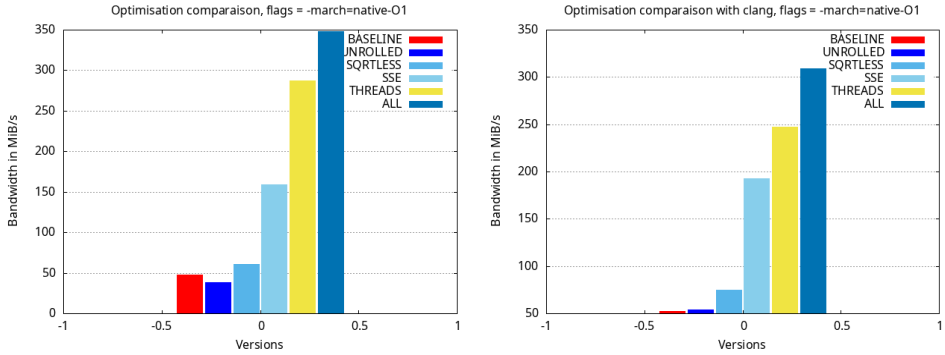
Dans le cas présent, l'utilisation de pthreads permet de paralléliser l'application du filtre de Sobel sur des images, en divisant les tâches d'application du filtre en plusieurs parties qui peuvent être exécutées simultanément par différents threads. Cela permet d'améliorer les performances en utilisant plus efficacement les ressources système pour traiter les images plus rapidement.

6.2 Utilisation des threads

Afin d'implémenter la parallelisation du programme à travers le multithreading et l'utilisation de la bibliothèque pthreads on a implémenter une fonction `sobel_pthreads` dans laquelle on déclare les threads qui vont se partager équitablement les pixels, on stocke aussi les données dont les threads auront besoin dans une struct et on les déploit ensuite dans la fonction `sobel_threads` qui est implémenté de la même manière que `sobel_baseline` et qui fait aussi appelle à `convolve_baseline`.

6.3 Analyse des performances du multithreading

L'utilisation de threads (dans notre cas 8 threads) devrait théoriquement nous permettre un speedup de 8 mais on obtient un speedup de 6, probablement dû à une distribution non-optimale des tâches. On atteint ainsi un temps d'execution de 0.01833659790s/frames, on traite donc environ 54,5 FPS.



7 Conclusion

7.1 Version finale

Enfin, on pourrait penser qu'afin de tirer le meilleur speedup il faudrait mixer toutes ces fonctions ensembles, mais malheureusement ce n'est pas si simple, tout d'abord car en complexifiant nos fonctions on réduit l'impact que peut avoir le compilateur sur notre programme ce qui nous privent de beaucoup d'optimisation, ensuite il n'est pas toujours facile de combiner differente méthode comme le multithreading, le déroulage et l'intrasic SSE.
Dans notre cas on a décidé d'utiliser le multithreading en retirant l'instruction `sqrt` qui est 12 fois plus couteuse qu'une multiplication, on compare donc mag avec le carré du threshold pour assigner 255 ou 0 au lieu de passer par cette opération trop couteuse. En alliant cela à l'option `-Ofast` on obtient un speedup de 38.1, ce qui nous permet un temps d'execution de 0.003007301611s/frame soit environ 332.5FPS.

7.2 Pistes d'optimisation futur

Malgré que les pistes d'optimisations vu lors de ce projet n'ont pas toujours été complètement exploré, on peut tout de même envisagé d'autres moyen d'optimisation, on pourrait par exemple envisager des optimisations pour des images de plus grandes tailles ou de tailles variable, on pourrait optimiser le programme pour des processeurs spécifique comme des processeurs ARM our des processeurs graphique (GPU). La piste de l'optimisation parallèle pourrait être bien plus approfondi avec OpenMP, MPI ou CUDA par exemple. En somme malgré le temps passé à optimiser ce programme de filtre de Sobel il reste encore beaucoup de moyen de l'optimiser dont certains pourrait être bien plus efficace que ce que nous avons fait.