Rapport LABO4 HPC

Bastien Pillonel

Date : 27.04.2024

Config :

- Distrib PopOS 64bits (Ubuntu 22.04)
- AMD Ryzen 7 5800X 8-Core Processor
- AMD K19 (Zen3) architecture
- gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0
- Library SIMD : immintrin.h

Partie 2 Analyse et amélioration

Introduction

Pour cette partie il est demandé d'analyser et d'optimiser le code de segmentation d'image fournit.

Mise en place d'un setup de test

Pour la mesure de temps je n'avais pas vu que le code serait testé à l'aide de l'utilitaire time. A la place j'ai simplement ajouter dans le code du main une mesure de temps autour de la fonction kmean grâce à la lib time.h. Ensuite le script meas.sh permet de lancer l'application segmentation sur chaque image avec plusieurs taille de cluster. Les résultats des temps de chaque execution sont ensuite logguer dans les fichiers commençant par time_measure.txt.

Une première mesure servant de référence pour le reste à été fait avec l'application non optimisé et est logguée sous time_measure_init.txt.

Analyse du code

Essaie du flag ftree-vectorize Tout d'abord je voulais tester l'efficacité du flag -ftree-vectorize permettant l'auto vectorisation de certaines boucles/bout de code.

Flags de compilation:

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -O3 -g -Wall -fno-inline -ftree-vectorize -ffast-math")
```

Le flag fast-math permet la reduction sur les opérations à virgule flottante.

Après compilation et execution je n'ai pas remarqué d'amélioration de la vitesse.

Logs dans : time_measure_ftree_vect.txt

Vectorisation du code Ensuite j'ai essayé de l'amélioré en utilisant les instruction SIMD. L'application des instruction SIMD ne se prête pas super bien à ce code si l'on souhaite traité plusieurs pixels en même temps car les pixels d'une images sont constitués dans nombre variable de composantes.

Néanmoins il est possible de vectorisé la fonction de distance pour chaque pixel.Cette fonction distance prends en entrée deux vecteurs ayant chacun les composants du pixel dans les octets de poids faible.

Pseudo-code distance_avx2:

```
distance_avx2(vect_128b a, vect_128b b, nbr_composant):
    etendre les 4 octets lsb de a sur des entiers 32bit
    etendre les 4 octets lsb de b sur des entiers 32bit

    soustraire verticalement b à a

    convertir le resultat précendant en vecteur de float

    multiplié chaque élément du vecteur précédant par lui même (puissance 2)

    pour i < nbr_de_composant:
        incrémenté la variable somme avec la valeur l'élément[i] du vecteur précédent
    fin

    returner somme
fin distance_avx2
```

Ensuite on peux remplacer l'appelle à la fonction distance par cette nouvelle fonction dans le code. Le `#define VECTORIZE 1` indique si que l'on souhaite utiliser la version vectorisé de la fonction. Il faut aussi adapté le chargement des données pour correspndre aux arguments de la fonction.

Cette fonction retourne le résultat au carré directement. Car 2 cas sur trois dans le code rappelais la fonction pour pouvoir multiplié les deux resultats ensemble => on appelle la fonction qu'une fois et l'opération racine carré est appelé moins souvent (opération coûteuse).

Après compilation et execution j'ai pu constater une légère amélioration de la vitesse.

Flags de compilation:

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -O3 -Wall -fno-inline -march=native -mavx2")
```

Logs dans : time_measure_kmeanpp_vect.txt

Optimisation de l'allocation dynamique En regardant le code d'un peu plus près j'ai aussi remarqué que dans certaines boucles on allouait pour chaque itération de la mémoire. On peut très bien allouer une première fois hors boucle

la mémoire puis copié les nouvelles valeurs à utilisé à ce même emplacement. L'opération d'allocation dynamique peut faire appelle à des appels systèmes qui vos faire perdre du temps d'execution (bascule en mode noyau => perte de vitesse).

Après avoir sortis les allocations des boucles, fais attention que la mémoire est restitué correctement et qu'aucun pointeur fantôme ne traîne, j'obtiens une très bonne amélioration du temps d'execution.

Flags de compilation:

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -O3 -Wall -fno-inline -march=native -mavx2")
```

logs dans : time_measure_vect_malloc_opti.txt

Partie 3 Développement et réflexion sur l'utilisation SIMD

Mise en place d'un setup de test

Ici j'ai repris le script de la partie 2 mais j'utilise la commande time à la place de mesurer dans le code. Le script à utiliser est edge_measure.sh.

Pour la baseline j'utilise les flags de compilation suivants:

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -O0 -g -Wall -fno-inline")
```

Les logs des resultats se trouvent dans le fichier time_measure_edge.txt

Analyse du code

J'ai repris la version de sobel du labo3 avec l'optimisation du calcul de certaine valeurs redondante dans les boucles pour la fonction filter_pixel_1D qui permet d'appliquer le kernel sur un pixel.

Optimisation SIMD La majeur partie des opérations du code se font dans mes deux fonctions qui servent à appliquer le kernel sur un pixel (filter_pixel_1D et filter_pixel_chained).

J'ai donc décider d'utiliser la multiplication vectorielle afin de multiplier le kernel et la sous matrice de pixel ensemble. La problématique suivante se pose: mon kernel peut avoir une taille différente il faut donc que je décide si je souhaite stocké tout le kernel dans un vecteur (ce qui réduit la taille d'un kernel à 16 élément => maxv2) ou si je stock une ligne de vecteur puis répète l'opération. J'ai choisit la 2e options mon choix est motivé par le fait que je trouve que des kernels de côté 4 maximal reduisent beaucoup la réutilisabilité du code.

Cependant j'ai du contraindre mon code à n'utiliser que des kernels de taille fixe car lorsque je souhaite additionner les valeurs des produits de chaque éléments d'une ligne à partir de l'optimisation -O2 l'opération ne s'effectue pas comme il devrait pour une raison que je n'arrive pas à déterminer.

Comme dans le cas du laboratoire nous n'utilisons que des kernels de taille 3, j'ai décidé de brider mes fonctions selon cette condition.

Les flags de compilation utilisé sont les suivants:

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -O3 -Wall -fno-inline -march=native -mavx2")
```

Les logs des résultats se trouvent dans le fichier time_measure_edge_simd.txt

Nous pouvons constatez une amélioration par rapport à notre baseline dans le temps d'execution.