Université de Versailles Saint Quentin en Yvelines
Faculté d'Informatique
Spécialité :
Master Calcul Haute Performance et Simulation

Rapport Projet Architecture Parrallèle

BEN NACER Meriem
N° étudiant : 22207572

Table des matières

Table des figures

Résumé de l'article

Edge detection is widely used in computer vision algorithms to determine the edges of an image. The text discusses the idea of extending standard Sobel kernels to improve edge detection. The results of the extended filters are compared with those of the standard Sobel filter and the statistical results of the improvements are presented the Sobel filter is used to detect edges in an image and is used for a variety of applications. There are also other more effective methods. The paper presents a simplified version of the extended Sobel filter which seems to perform better in practice and has increased execution efficiency. The results of the extended filters are compared to those of standard Sobel filters and other edge detection filters using different sets of images. The Canny Edge detection algorithm is also used to compare the extended filter with the standard approach. The use of Sobel's standard formula for detecting edges in images using the steps of the edge detection algorithm. It is also mentioned how to reduce the noise in the image by using a Gaussian filter and then using the filters to convolve the grayscale image with their kernels on the x and y axes. Reference is also made to the use of the Canny edge detection algorithm using the Sobel filter and custom extended filters for result comparisons on BSDS500 test, training and validation image sets.
The common method for detecting edges in an image is to use the Sobel filter which calculates the difference between pixel intensities. In this article, it is proposed to use modified Sobel filters to detect the contours of images from a different angle. The standard 3x3 Sobel cores are extended to 5x5, 7x7, 9x9, 11x11, 13x13, and 15x15 for both axes. By extending the kernels, the distance between pixels that influence the convolution result is increased, allowing stronger intensity changes to be found in the image. The results obtained with this approach are generally better than those obtained with the filters presented in other works, despite a lack of respect for the geometric gradient formulas.
In this section, we present visual comparisons between the results of convolving an image with the standard Sobel filter and the extended filters defined previously. So we used the BSDS500 image sets as a data set of standard edge detection experiments to compare the extended filters (5x5, 7x7, ..., 15x15) with the standard Sobel filter. The results show that some extended filters provide better edge detection results than the standard 3x3 Sobel filter. We will now use a standard dataset and a benchmark to compare extended filters. The results will be compared to reference images of human-segmented boundaries. The benchmark will choose a number of threshold values between 0 and 255 and thus determine the best possible result for each filter for each image in the set. In this section we present a study in which we compared the performance of different Sobel filters, including a custom 5x5 filter, to the performance of predefined mathematical filters. We found that their custom filter had the best overall F1 score, although some of the other filters may have had better recall or precision. We note that the simplicity of their custom filter likely contributed to its effective performance, but further research is needed to fully understand why it performed better than other filters.

The performance of different Sobel filters extended with the standard 3x3 Sobel filter in the Canny edge detection algorithm, using BSDS500 datasets. The results show that the extended filters have higher accuracy than the standard 3x3 Sobel filter in the Canny algorithm. The 7x7 custom extended filter achieved the highest F1 score on the BSDS500 test set, but the F1 score started to decrease when the filter was extended. The results for the training and validation sets are similar. The extended Sobel filter has better results in detecting edges in images than the standard Sobel filter. They achieved better recall and accuracy by extending the kernels of the Sobel filter, as can be seen in Tables 1-3. Comparing their custom extended Sobel filter with other edge detection filters, we found that their custom 5x5 extended Sobel filter got a better overall F1 score than the other filters. Custom extended Sobel filters are also a good choice for runtime-critical applications because they always have the same number of operations for any extension.

1.Introduction

Le rapport suivant décrit les différentes étapes suivies pour l'optimisation du code Sobel, il s'agit de la l'algorithme de détection des contours d'une image en utilisant des filtres de convolution. Il utilise deux filtres de Sobel, l'un pour détecter les contours horizontaux et l'autre pour détecter les contours verticaux.. Il est utilisé dans de nombreux domaines tels que la vision par ordinateur, la reconnaissance de formes, la robotique, la reconnaissance d'images, la reconnaissance de caractères, la vidéosurveillance et la vidéo analyse.

2.Optimisations potentielles

2.1.Alignement de la mémoire

L'alignement de la mémoire est l'organisation des données en mémoire pour maximiser les performances en utilisant les caractéristiques de l'architecture de la machine. En C, les données peuvent être transférées dans des emplacements de mémoire qui ne correspondent pas nécessairement à des limites de mémoire précises. Cela peut entraîner des opérations de mémoire plus lentes, car l'ordinateur doit effectuer des opérations supplémentaires pour accéder aux données.
Il existe plusieurs façons d'améliorer l'alignement de la mémoire dans un code C pour l'optimisation :

- Utiliser des pragmas ou des directives de compilation pour aligner les structures de données sur des limites de mémoire précises.
- Utiliser des types de données prédéfinis qui sont alignés sur des limites de mémoire précises.

- Utiliser des fonctions de mémoire spécifiques à l'architecture pour allouer de la mémoire alignée, comme _mm_malloc() pour les processeurs x86.

  pour notre cas : remplacer *_mm_malloc* par *_aligned_malloc* pour comparer la performance en utilisant les deux fonctions. *u8 *frame = (u8*)_aligned_malloc(size, 16) ;*
- Utiliser des bibliothèques de mémoire alignée pour gérer l'allocation et la libération de la mémoire.

2.2.Suppression des instructions coûteuses : sqrt, division

Pour s'en passer de la fonction mathématique *sqrt()* dans ce code, on peut utiliser l'approximation de Newton-Raphson pour calculer la racine carrée d'un nombre. La méthode suivante peut être utilisée pour remplacer sqrt :

```c
float approx_sqrt(float x)
{
    float approx = x;
    int i;
    for(i = 0; i < 10; i++)
    {
        approx = 0.5 * (approx + x / approx);
    }
    return approx;
}
```

FIGURE 1 –

Ce sera le meme cas pour la fonction *sqrt* dans les autres parties de code. Il existe plusieurs façons d'améliorer la performance de la division dans un code C. Certaines méthodes incluent :

- Utiliser des opérations de décalage de bits : En utilisant des opérations de décalage de bits, vous pouvez diviser par 2 en décalant les bits vers la droite, ou diviser par 4 en décalant les bits de 2 positions vers la droite, etc.
- Utiliser une multiplication inverse : Plutôt que de diviser, vous pouvez également utiliser une multiplication inverse, qui est plus rapide que la division.
- Utiliser des techniques d'approximation comme la méthode de Newton-Raphson pour approximer la division.
  Exemple : $f64\ inv_n = 1.0/(f64)n;$
  $m* = inv\_n;$

2.3.Vectorisation (SSE, AVX, ou AVX512 pour les architectures x86_64)

pour optimiser ce code en utilisant la vectorisation :

- Utilisez des instructions AVX2 ou AVX-512 pour traiter plusieurs pixels à la fois. Les instructions AVX2 et AVX-512 permettent de traiter 8 et 16 pixels respectivement en même temps, ce qui peut entraîner une accélération considérable par rapport aux instructions AVX qui ne traitent que 8 pixels à la fois.

- Utilisez des instructions de chargement et de stockage alignées pour accélérer les accès mémoire.
- Les instructions _mm256_load_ps et _mm256_store_ps permettent de charger et de stocker des données alignées sur des limites de 32 octets, ce qui peut entraîner une accélération considérable par rapport aux instructions _mm256_loadu_ps et _mm256_storeu_ps qui ne garantissent pas l'alignement.
  Voici un exemple d'utilisation de la vectorisation dans notre code en utilisant les instructions SSE pour effectuer cette opération :

```
for (u64 i = 0; i < H * W * 3; i += 12) {
    vector_gray = _mm_set1_ps(frame[i]);
    _mm_storeu_ps(frame + i, vector_gray);
    _mm_storeu_ps(frame + i + 4, vector_gray);
    _mm_storeu_ps(frame + i + 8, vector_gray);
}
```

FIGURE 2 –

Ce code utilise l'instruction _mm_set1_ps pour charger la valeur gray dans un vecteur ___m128. Il affecte ensuite cette valeur à une plage de 12 éléments dans le tableau frame en utilisant _mm_storeu_ps.

2.4.Parallélisation à l'aide d'OpenMP sur la boucle la plus externe.

Il est possible d'appliquer la parallélisation à l'aide d'OpenMP sur la boucle la plus externe pour accélérer le calcul de la somme. Voici un exemple de comment ajouter la parallélisation à cette boucle en utilisant OpenMP :

```
#include <omp.h>

i32 r = 0;

#pragma omp parallel for reduction(+:r)
for (u64 i = 0; i < fh; i++) {
    for (u64 j = 0; j < fw; j++) {
        r += m[INDEX(i, j, fw)] * f[INDEX(i, j, fw)];
    }
}
```

FIGURE 3 –

La directive pragma omp parallel for indique à OpenMP de paralléliser la boucle for en utilisant autant de threads que de cœurs disponibles. La directive reduction(+ :r) indique à OpenMP de combiner les résultats calculés par chaque thread pour obtenir le résultat final. En utilisant la parallélisation, le temps d'exécution de cette boucle sera réduit en fonction du nombre de cœurs disponibles.

3.Compilation et exécution

3.1.Compilation

principalement avec make, on va utiliser différents compilateurs (au moins deux) : gcc, clang.

3.2.Execution

Tout d'abord, on va convertir la vidéo d'entrée au format RVB brut en utilisant le script suivant (en s'assurant que ffmpeg est installé sur la distribution Linux) :
./cvt_vid.sh r2v out/output.raw out/output.mp4

Ensuite, vous pouvez exécuter le programme comme suit :

./sobel in/in.raw out/out.raw

Une fois la conversion terminée, on peux lire la vidéo pour vérifier si la sortie est valide en utilisant mplayer (à installer) :

out mplayer/out.mp4

Après avoir optimisé dans notre code pour la dététction des contours avec le filtre Sobel, nous allons générer un code pour avoir des graphes qui nous permet de comparer entre les résultats de l'éxécution sur gcc et sur clang. Voici 2 graphs représentatif :

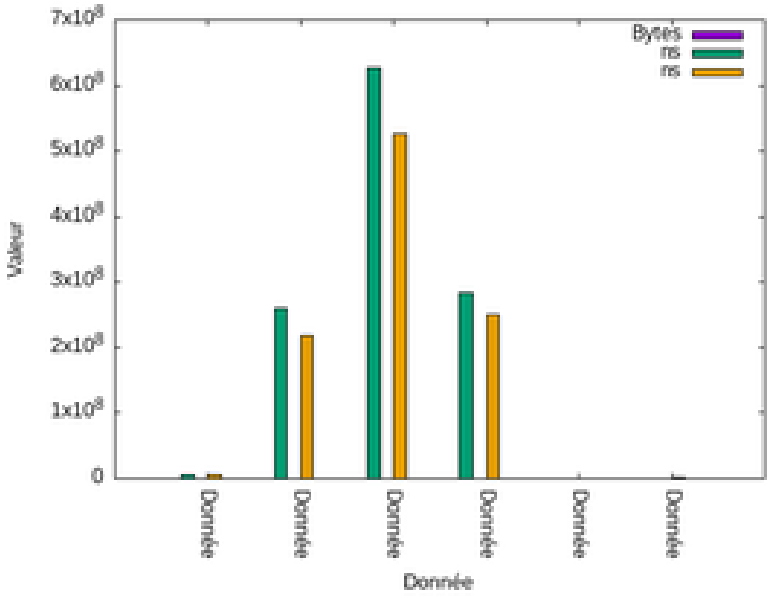

FIGURE 4 –