# HPC : Laboratoire 4

SIMD

Rayane Annen

5 mai 2024

## Table des matières



## Benchmarks

Machine utilisée pour les tests :

- Architecture : Intel x86_64
- CPU : i9-9900K 8 Cores / 16 Threads @ 3.60GHz / Turbo @ 5.00GHz (CoffeeLake)
  - Cache L1 : 32 kB
  - Cache L2 : 256 kB
  - Cache L3 : 16 MB
- OS: Debian 12
- Compilateur :
  - gcc 12.2.0
  - target: x86_64-linux-gnu
  - Flags de compilation: `-O3 -g -Wall -fno-inline -march=native`
  - Librairies: `stb`, `math.h`

# Analyse

## Segmentation

### Tests effectués

Comme pour les précédents laboratoires, pour chaque modifications effectuées une batterie de tests est lancée, voici le résumé des tests effectués :

- Nombre de cluster : 5

Images :

| Nom du fichier | Dimensions [pixels] | Nombre de composantes par pixel |
|---|---|---|
| `half-life.png` | $2000 \times 2090 = 4038000$ | 3 (8-bit RGB) |
| `medalion.png` | $1267 \times 919 = 1164373$ | 3 (8-bit RGB) |
| `nyc.png` | $1150 \times 710 = 816500$ | 3 (8-bit RGB) |
| `sample_640_2.png` | $640 \times 426 = 272640$ | 3 (8-bit RGB) |

Pour obtenir des résultats consistents, on teste 50 runs et on prend la moyenne.

### État initial

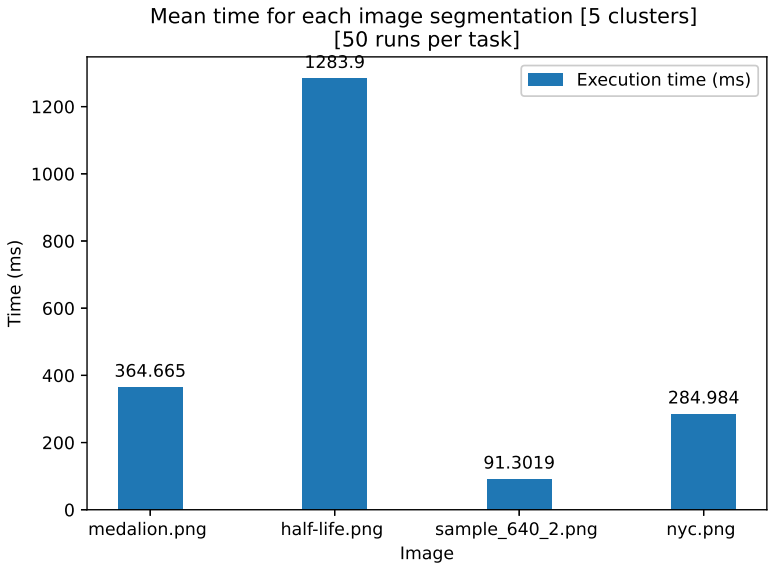


Figure 1: Performances (temps) du programme initial

### Somme des poids vectorisée dans `kmeans_pp`

Dans cette partie, lors du calcul de `total_weight` (somme des distances) au lieu de parcourir l'image pixel par pixel, on la parcourt 8 pixels par 8 pixels en utilisant des opérations SIMD sur 256 bits.

```
rem = img_dim - ((img_dim) % 8); // 8 pixels per iteration
for (j = 0; j < rem; j += 8) {
  dist_v = _mm256_loadu_ps(distances + j);
  dist_v = _mm256_hadd_ps(dist_v, dist_v);
  total_weight += dist_v[0];
}

for (j = rem; j < img_dim; ++j) {
  total_weight += distances[j];
}
```

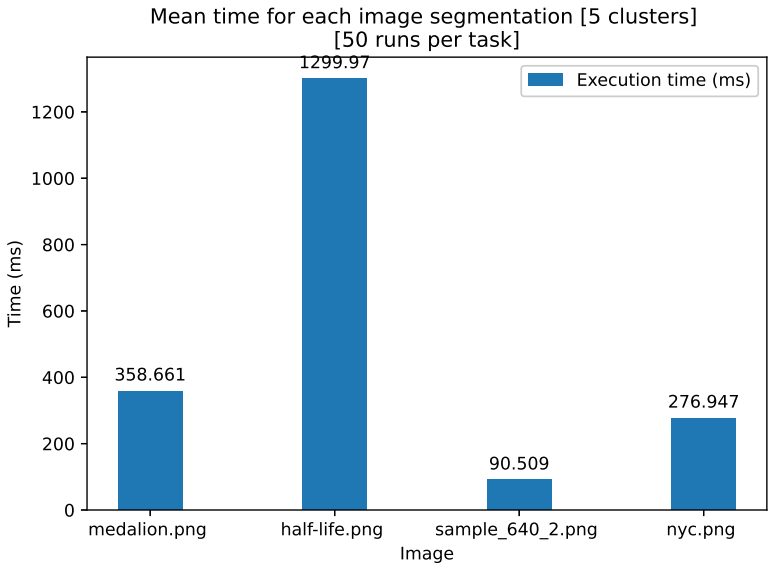


Figure 2: Performances (temps) avec la somme des poids vectorisée

On constate que pour la plupart des images la distance est négligeable, ce n'est toutefois pas le cas pour l'image `nyc.png` qui obtient un gain assez important étonnamment.

### Amélioration du calcul des distances euclidiennes dans `kmeans_pp`

Certaines distances étaient calculées puis mise au carré, j'ai ajouté un code permettant de calculer une distance au carré directement, évitant un appel inutile `sqrt` :

```c
float distance_sqr(uint8_t *p1, uint8_t *p2) {
    float r_diff = p1[0] - p2[0];
    float g_diff = p1[1] - p2[1];
    float b_diff = p1[2] - p2[2];
    return r_diff * r_diff + g_diff * g_diff + b_diff * b_diff;
}
```

Voici les résultats obtenus :

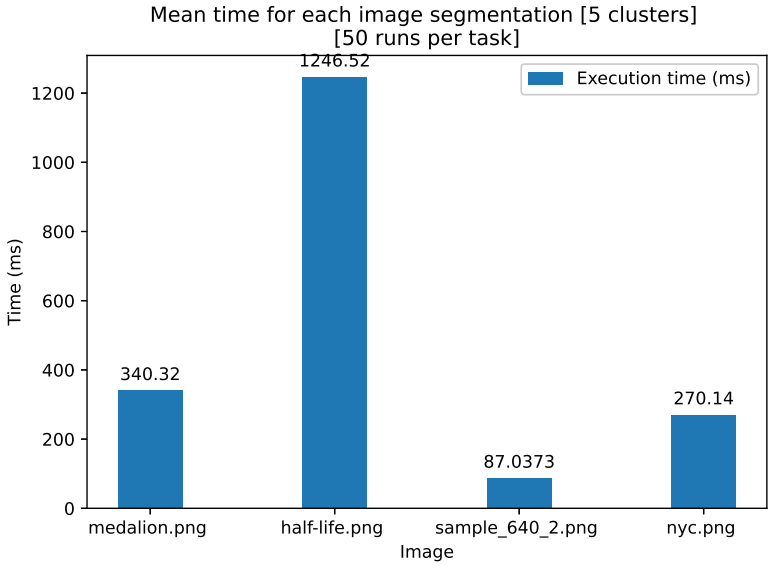


Figure 3: Performances (temps) du programme avec les distances optimisées

Cette modification a eu une influence sur les performances, on obtient un léger gain, pouvant de 50 ms pour les grandes images à 3 ms pour les plus petites. Ce gain est attendu, étant donné que nous n'effectuons plus de calcul avec des racines carrées, on économise un appel de fonction et le temps de traitement de la fonction elle-même.

### Vectorisation du calcul des distances euclidiennes

Une fois le calcul de la mise au carrée effectuée, j'ai décidé de vectoriser le calcul des distances euclidiennes, celle-ci permet de traiter 8 calculs de distances à la fois, le distance est toujours calculée entre un pixel donné (le premier centre ou le nouveau centre du cluster) et un vecteurs de 8 pixels.

On peut résumer les opérations SIMD avec le schéma suivant :

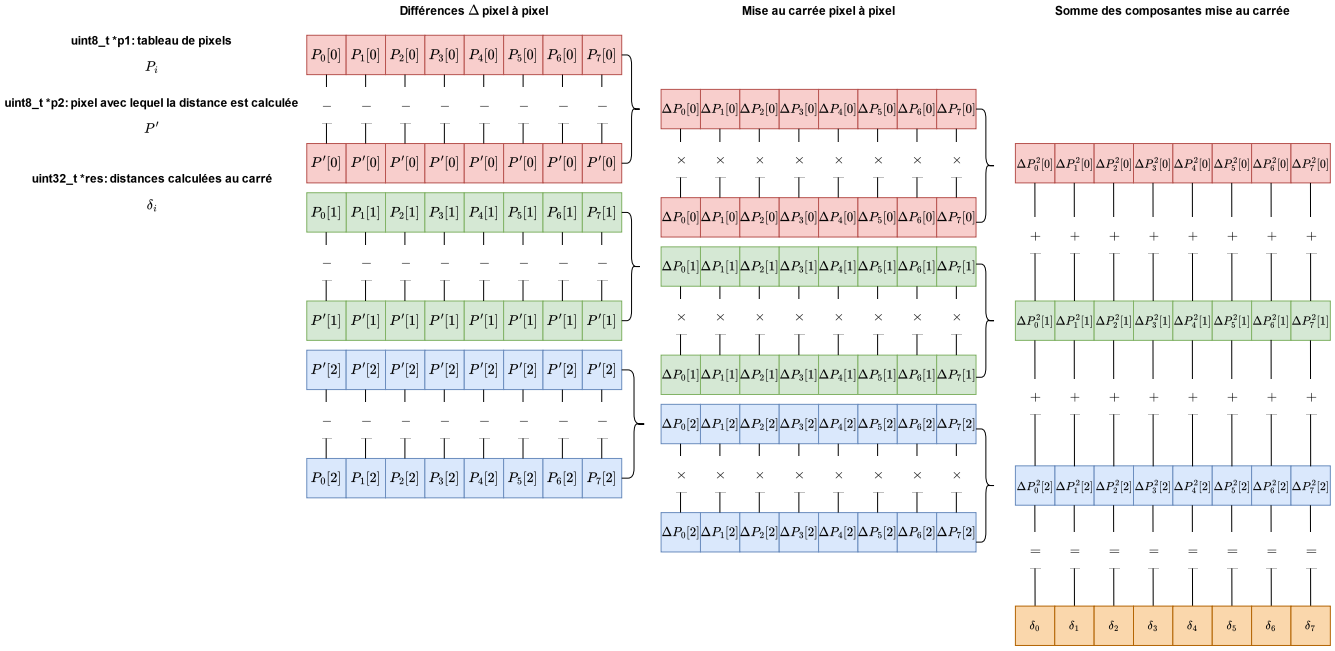


Figure 4: Distance euclidienne SIMD

Les résultats obtenus sont les suivants :

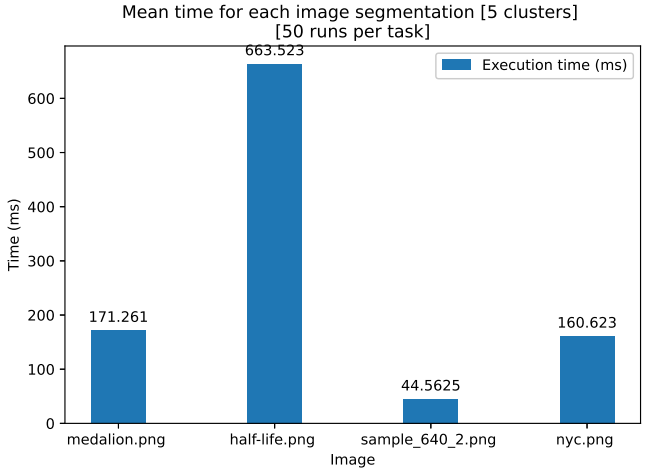


Figure 5: Performances (temps) du programme avec le calcul des distances vectorisé

Comme on peut le voir les performances sont fortement accrues, le temps de calcul a été environ divisé par deux. En plus du fait que les calculs n'utilisent plus de racines carrées mais directement la somme des différences au carrée, le traitement de 8 pixels à la fois a permis de fortement accélérer l'opération.

## Détection de contours (`edge_detection`)

### Tests effectués

Comme pour les précédents laboratoires, pour chaque modifications effectuées une batterie de tests est lancée, voici le résumé des tests effectués :

Images d'entrée :

| Nom du fichier | Dimensions [pixels] | Nombre de composantes par pixel |
|---|---|---|
| `half-life.png` | $2000 \times 2090 = 4038000$ | 3 (8-bit RGB) |
| `medalion.png` | $1267 \times 919 = 1164373$ | 3 (8-bit RGB) |
| `nyc.png` | $1150 \times 710 = 816500$ | 3 (8-bit RGB) |
| `sample_640_2.png` | $640 \times 426 = 272640$ | 3 (8-bit RGB) |

Nombre de type :

- Tableau 1D : `1`
- Liste chaînée : `2`

Sur la base de ces deux variables (l'image et le type de structures de données) une matrice de tests a été effectuée. Chaque test est effectué 50 fois et le résultat gardé est la moyenne de toutes les runs.

### État initial

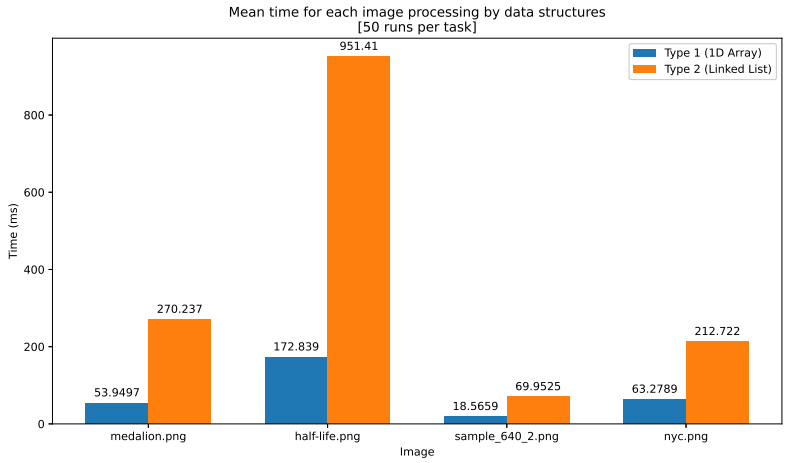


Figure 6: Performances (temps) du programme initial

### Vectorisation du filtre grayscale pour les tableaux 1D

J'ai vectorisé le code qui applique le filtre pour transformer une image en niveau de gris.

Les calculs SIMDs peuvent être résumés avec le schéma suivant :

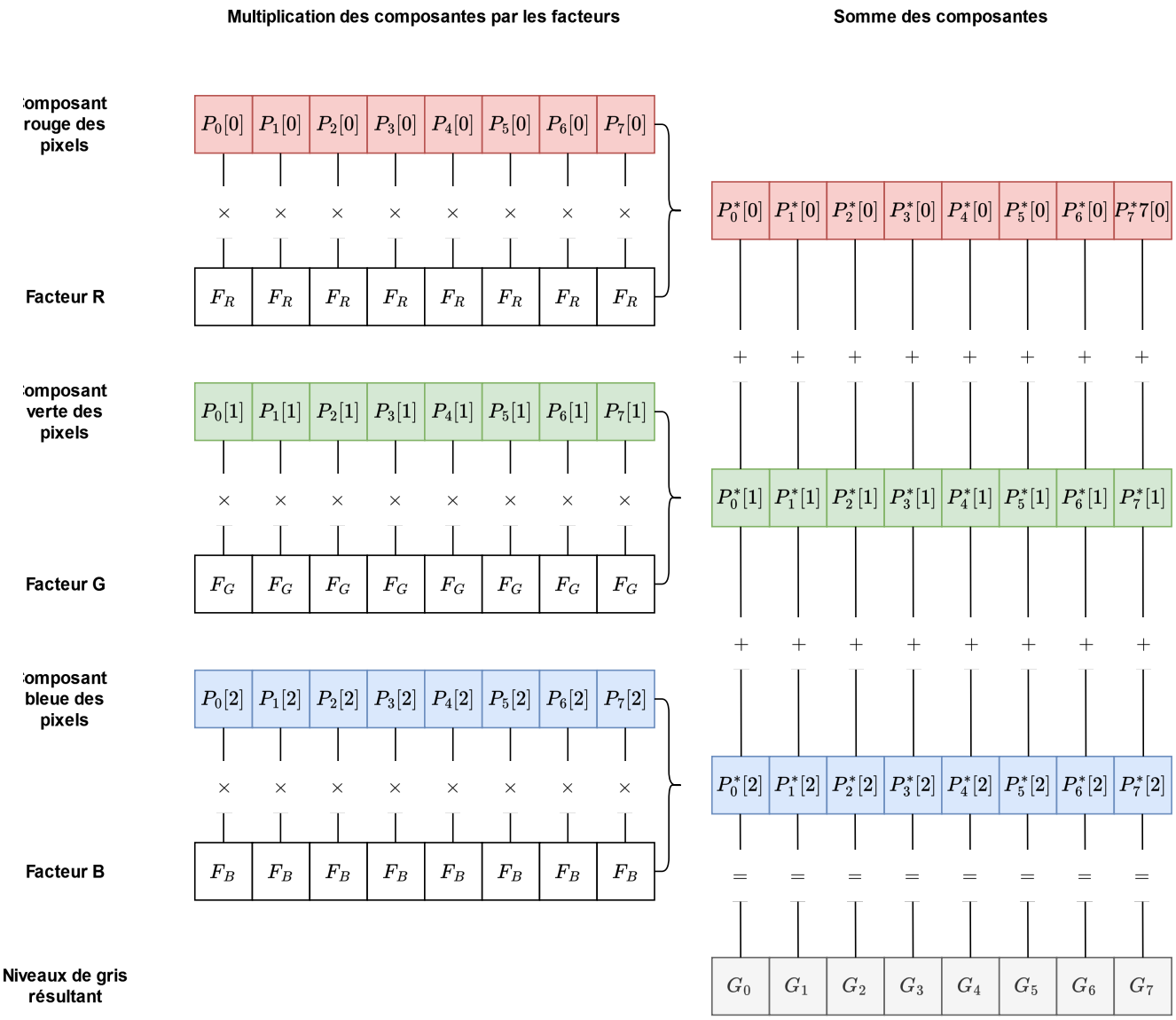


Figure 7: Diagramme des calculs SIMD pour appliquer le filtre grayscale

Les résultats sont les suivants :

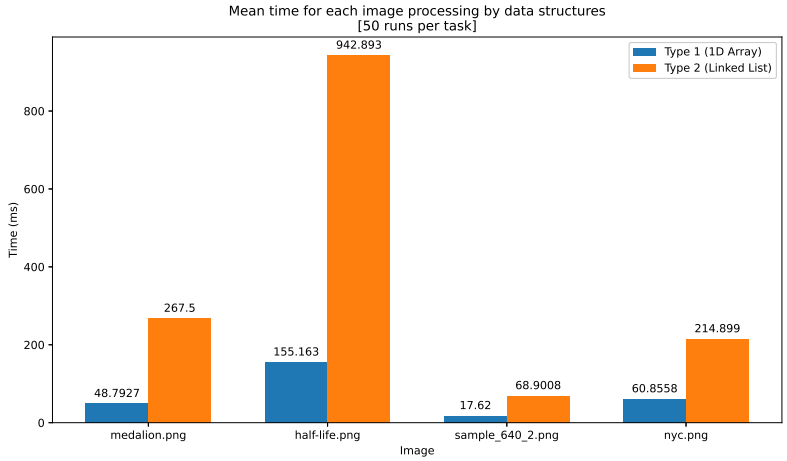


Figure 8: Performance (temps) avec le filtre grayscale SIMD

Comme on peut le voir le gain se voit mais reste très minime, toutefois il reste encore un peu de marge de manoeuvre pour améliorer les performances je pense en vectorisant les filtres suivants.

## Conclusions finales

Je vais maintenant conclure par présenter de manière synthétique les performances obtenues après chaque modification pour les deux programmes.

Concernant le programme permettant de segmenter les images :

| Amélioration | Temps `half-life.png` | Temps `medalion.png` | Temps `nyc.png` | Temps `sample_640_2.png` |
|---|---|---|---|---|
| État initial | 1289 ms | 364 ms | 285 ms | 91 ms |
| Somme vectorisée | 1299 ms (-0.76%) | 358 ms (+1.67%) | 276 ms (+3.26%) | 90 ms (+1.11%) |
| Distance au carrée | 1246 ms (+3.45%) | 340 ms (+7.05%) | 270 ms (+5.55%) | 87 ms (+4.59%) |
| Distance SIMD | 663 ms (+94.4%) | 171 ms (+112.86%) | 161 ms (+77.01%) | 44 ms (+106.81%) |

Le pourcentage d'amélioration indique à quel point les performances sont meilleures par rapport à l'état initial, il est calculé de la manière suivante : $\frac{|T_0 - T_1|}{T_1} \times 100$

Le plus gros apport est le calcul des distances SIMD qui offre des performances remarquables. Je ne suis pas certain qu'il reste de la marge de manoeuvres, cela doit être possible de vectoriser des opérations notamment dans la fonction `kmeans`.

Concernant le programme permettant de détecter les contours (on ne s'intéresse qu'aux tableaux 1D)

| Amélioration | Temps `half-life.png` | Temps `medalion.png` | Temps `nyc.png` | Temps `sample_640_2.png` |
|---|---|---|---|---|
| État initial | 173 ms | 54 ms | 63 ms | 18 ms |
| Grayscale SIMD | 155 ms (+11.61%) | 49 ms (+10.20%) | 60 ms (+5%) | 18 ms (0%) |

Les performances sont un peu meilleures, toutefois, il ne reste plus beaucoup de marge de manoeuvres pour améliorer quoi que ce soit dans cette partie du code, il faudrait voir s'il est encore possible de vectoriser des calculs mais cela me semble compliqué (surtout avec le skip des pixels en bordures).

Concernant le code avec les listes chaînées je ne me suis pas trop attardé dessus mais il doit être possible d'améliorer encore quelques parties mais je pense que cela soit encore plus limité étant donnée qu'on ne travaille pas avec une vue complète de l'image.