Laboratoire 4 : SIMD

HPC 2023

Francesco Monti

9.05.2023

Partie 1

Compilation avec SIMD

Pour pouvoir compiler les instructions SIMD avec GCC on doit mettre le flag de compilation correspondant. Dans notre cas, si on veut que GCC utilise les extensions SSE3 on ajoute :

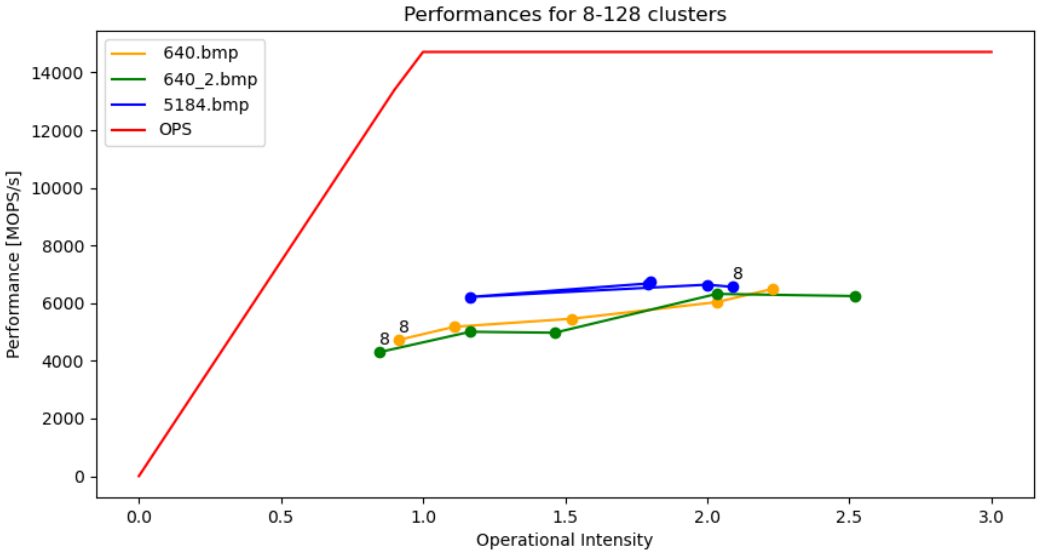
```
CFLAGS= -O3 -msse3 -std=c11 -Wall -Wextra -pedantic -g -I../include
```

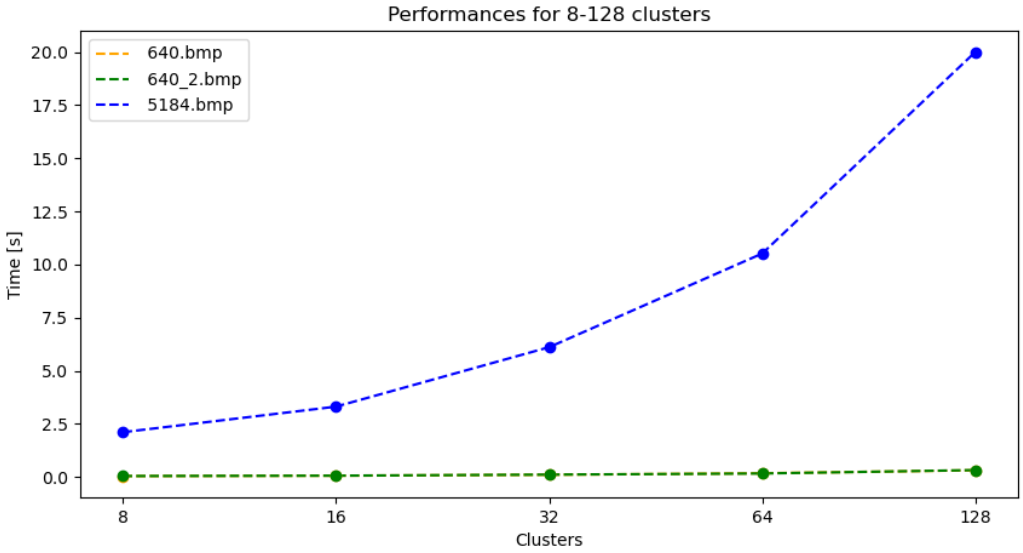
au *Makefile*. On peut aussi utiliser les flags `-mavx` et `-mavx2` pour utiliser les extensions AVX et AVX2 respectivement.

Pour utiliser les extensions SIMD dans le code avec les fonctions intrinsèques on doit inclure le fichier `immintrin.h` pour les extensions SSE et AVX.

Analyse du code

En regardant une première fois le code on s'aperçoit des nombreuses boucles qui s'exécutent sur des tableaux. Ces boucles sont des cas typiques où on peut utiliser les instructions SIMD pour accélérer le code. En effet, les instructions SIMD permettent de faire des opérations sur plusieurs données en même temps. Dans notre cas, on peut faire des opérations sur 4 `float` en même temps. On peut donc faire 4 fois moins d'itérations dans les boucles et gagner ainsi des performances. Mais pour cela il faut faire attention à ce que nous tableaux soient alignés correctement sur 128 bits. Si on regarde les performances du code avant optimisation on obtient ces graphiques :





On peut voir que les performances ne sont pas nulles, mais on peut les améliorer.

Calcul de la distance

Alignement des tableaux

Pour utiliser efficacement les fonctions SIMD on doit s'assurer que les tableaux sont alignés correctement sur 128 bits. Pour ce faire on remplace la fonction `malloc` par la fonction `aligned_alloc` qui permet d'allouer de la mémoire alignée. On doit lui passer l'alignement en paramètres en bytes, dans notre cas *16*. On pourrait également utiliser la fonction `__mm_malloc` mais il faudrait remplacer les appels à `free` par `__mm_free`.

Code motion

Une première optimisation qui est possible est de chercher et remplacer toutes les occurences d'un calcul immuable par une variable. Dans notre cas on peut voir que dans les fonction `kmeans` et `kmeans_pp` on calcule souvent la taille totale de l'image, soit `width * height`. On peut donc remplacer ces calculs par une variable `total_pixels` qui ne sera calculée qu'une fois. On peut également voir que lors du calcul de distance, on calcule souvent deux fois la distance avant de la mettre au carré. On peut donc calculer la distance une fois et la mettre au carré ensuite. On peut le voir ici :

// image_segmentation/src/k-means.c#L78-L79

```
while (r > distances[index] && index < total_pixels)
{
```

Vectorisation

On peut vectoriser une partie du code, spécialement les boucles qui parcourent tous les pixels de l'image. On peut voir dans la fonction kmeans quand on calcule le poids de tous les pixels par exemple. On peut le voir par exemple dans la boucle qui calcule les distances entre le premier centre et tous les pixels :

```
// image_segmentation/src/k-means.c#L37-L52

__m128 v_dist;
for (int i = 0; i < total_pixels_aligned; i += 4)
{
    float dist1 = distance(image[i], centers[0]);
    float dist2 = distance(image[i + 1], centers[0]);
    float dist3 = distance(image[i + 2], centers[0]);
    float dist4 = distance(image[i + 3], centers[0]);
    v_dist = _mm_set_ps(dist4, dist3, dist2, dist1);
    _mm_mul_ps(v_dist, v_dist);
    _mm_store_ps(&distances[i], v_dist);
}
for (int i = total_pixels_aligned; i < total_pixels; i++)
{
    float dist = distance(image[i], centers[0]);
    distances[i] = dist * dist;
}
```

Ici on a remplace l'exécution normale de la boucle par une vectorisation avec des instructions SSE. On remplace notamment les 4 multiplications des distances