PROJET D'ARCHITECTURE PARALLELE :

CODE GENETIQUE, DISTANCE DE HAMMING ET MATRICES STOCHASTIQUES

Encadré par le Professeur : 'Yaspr' Ibnamar

Chadha Sakka

chadha.sakka@ens.uvsq.fr

MASTER 1 CALCUL HAUTE PERFORMANCE ET SIMULATION
UNIVERSITE PARIS-SACLAY

20 JANVIER 2023

I. Introduction:

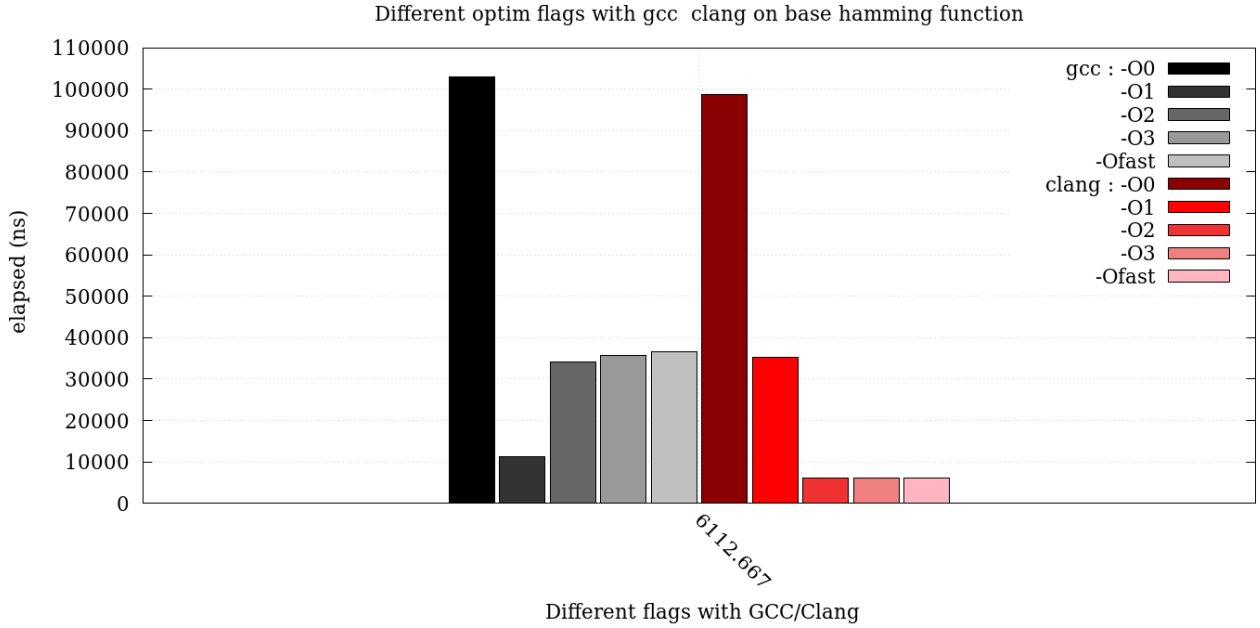
Ce document présente une analyse des performances d'une fonction qui compare deux chaînes de caractères représentant des séquences d'ADN à l'aide de la distance de Hamming. Dans un premier temps, ce rapport présentera l'implémentation de base, ensuite l'architecture cible, les différentes optimisations appliquées au niveau de la compilation et au niveau du code. Et on présentera au fur et à mesure des méthodes appliquées le résultat des mesures effectuées pour les compilateurs gcc et clang avec différents flags d'optimisation.

** Les séquences d'ADN sont de longueur de 2 mètres, cela signifie que les séquences d'ADN sont composées de 2 million de nucléotides. Comme chaque nucléotide est codé sur 2 bits (A, C, G, T), cela signifie que nous aurons besoin de 2 millions de * 2 bits = 4 millions de bits pour stocker une séquence d'ADN de longueur 2 mètres ce qui nécessitera une taille énorme de mémoire pour les stocker et le temps de calcul de la hamming distance peut s'avérer interminable. Il est donc tout à fait naturel de chercher à optimiser notre code pour utiliser efficacement la mémoire disponible et améliorer sa vitesse d'exécution.

II. Implémentation initiale :

Cet algorithme est de complexité linéaire et nécessite une seule opération arithmétique (add) par itération en plus des opérandes logiques.

```
u64 hamming(u8 *a, u8 *b, u64 n)
{
  u64 h = 0;
  for (u64 i = 0; i < n; i++)
    h += __builtin_popcount(a[i] ^ b[i]);
  return h;
}
```



III. Architecture :

Modèle	Fréquence GHz	Taille L1 KiB	Taille L2 KiB	Taille L3 MiB	SIMD
Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz	2.60GHz	192 KiB (6 instances)	1,5 MiB (6 instances)	12 MiB (1 instance)	SSE AVX AVX2

IV. Optimisations :

Il est important de noter que l'optimisation dépendra des caractéristiques de notre jeu de données et de l'architecture de notre système. D'où il est indispensable de tester différents compilateur, flags d'optimisations et jeux de données pour trouver celle qui convient le mieux à nos besoins.

1. Au niveau du compilateur :

a. Alignement mémoire :

On peut ajouter les flags suivants avec gcc :

-falign-functions=16 –falign-loops=16 –falign-jumps falign-labels

b. Compiler auto-vectorisation

On peut utiliser les drapeaux d'optimisation suivants :

-ftree-vectorize –fopt-info-vec-optimized

Le compilateur recherchera les vectorisations possibles de boucles. Le code source reste le même, mais le code compilé est complètement différent.

c. Déroulage

On peut ajouter ce flag : -funroll-loops pour indiquer au compilateur de faire un déroulage automatique sur les boucles.

d. Différents flags d'optimisation :

Pour les deux compilateurs : gcc et clang que j'ai utilisé, j'ai testé plusieurs flags d'optimisation (O0, O1, …, Ofast) et chacun apporte ses modifications sur le code assembleur, qui peuvent des fois nous avantager et des fois rendre notre code moins performant. J'ai remarqué même si j'ai l'impression que mes mesures sont corrompues que clang est plus performant que gcc sur la version. J'ai également

remarqué qu'en général les flags d'optimisation O2 et O3 apportent pratiquement les mêmes modifications avec gcc. Par contre, clang est optimal accompagné des flags –O3 ,-Ofast.

2. Au niveau du code :

a. Alignement mémoire :

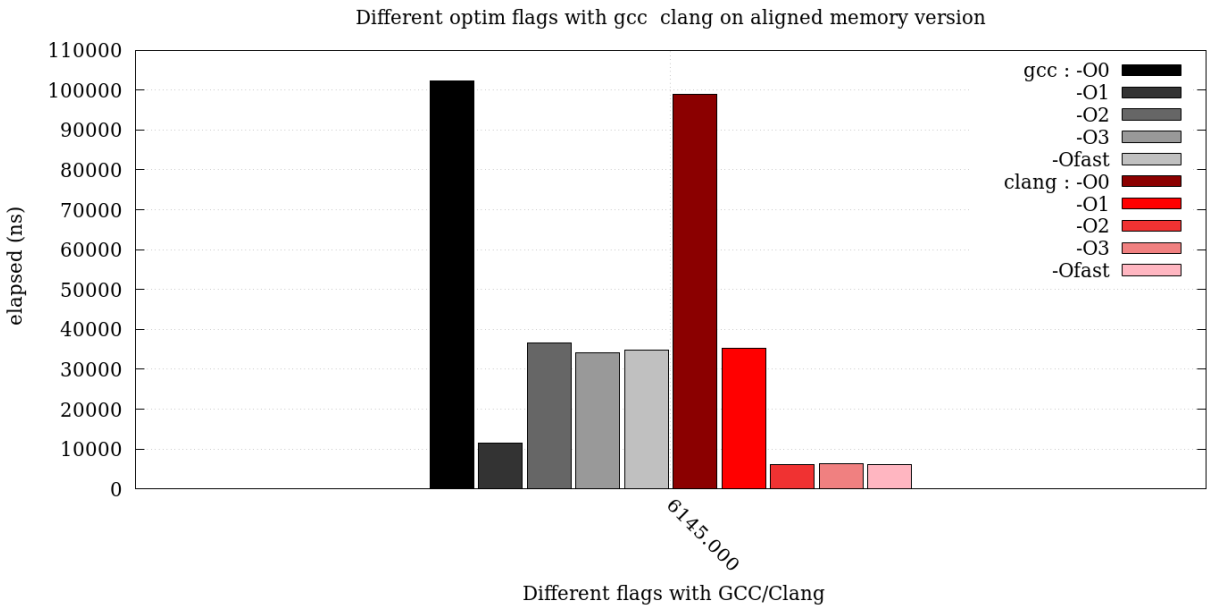
Dans la version initiale du code on alloue de la mémoire pour la séquence en faisant appel à malloc ce qui ne garantit pas l'alignement en mémoire :

```
s->bases = malloc (sizeof(u8) * sb.st_size);
```

On souhaite alors améliorer notre gestion de mémoire tout en allouant le même espace mais de manière continue de façon que l'accès à cette zone soit rapide et optimal.

```
s->bases = aligned_alloc (sizeof(u8), sizeof(u8) * sb.st_size);
```

On peut également utiliser : la directive #pragma pack(16)



b. Popcount optimal ?

Cette fonction utilise la fonction __builtin_popcount () qui est une fonction prédéfinie de GCC et Clang qui renvoie le nombre de bits à 1 dans un mot binaire. Cette fonction peut varier en performance selon l'architecture de la machine, elle est généralement plus rapide que de compter les bits à 1 en utilisant des boucles.

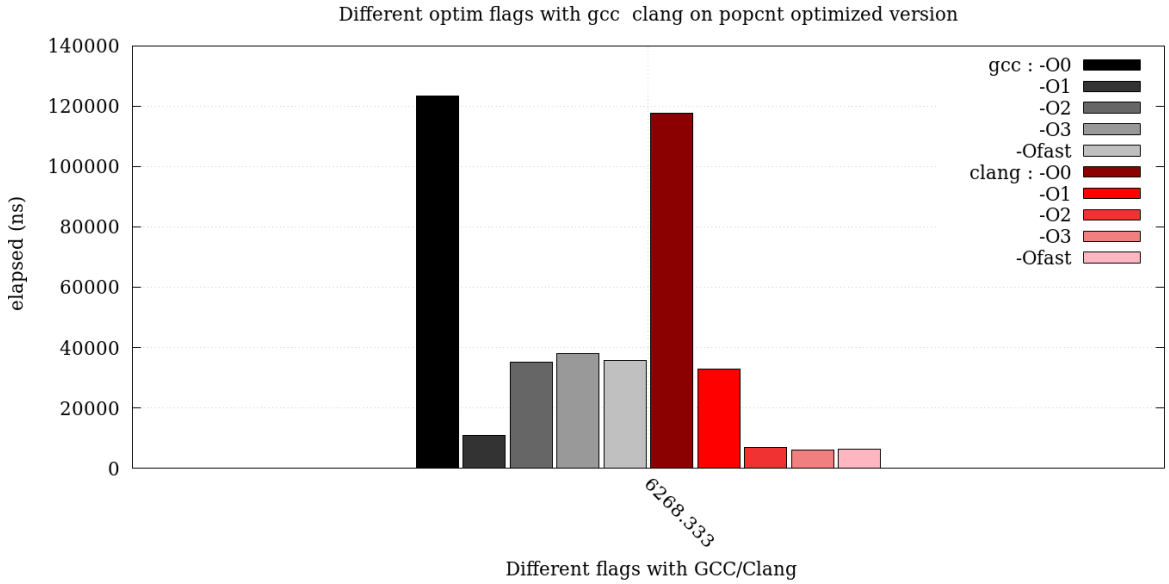
h+= **__builtin_popcount**(a[i] ^ b[i])

Afin de savoir s'il est optimal, on peut utiliser des bibliothèques externes qui ont déjà implémentées des fonctions de calcul de distance de Hamming optimisées pour différents types de processeur. Par exemple, la bibliothèque SSE4.2 dans GCC et clang en faisant appel aux fonctions :

h+= **_mm_popcnt_u32** (a[i] ^ b[i])

h+= **_mm_popcnt_u64** (a[i] ^ b[i])

Et on pourra comparer les performances des deux méthodes pour déterminer si notre popcount initial est optimal.



c. Déroulage de boucles x 4 et x 8

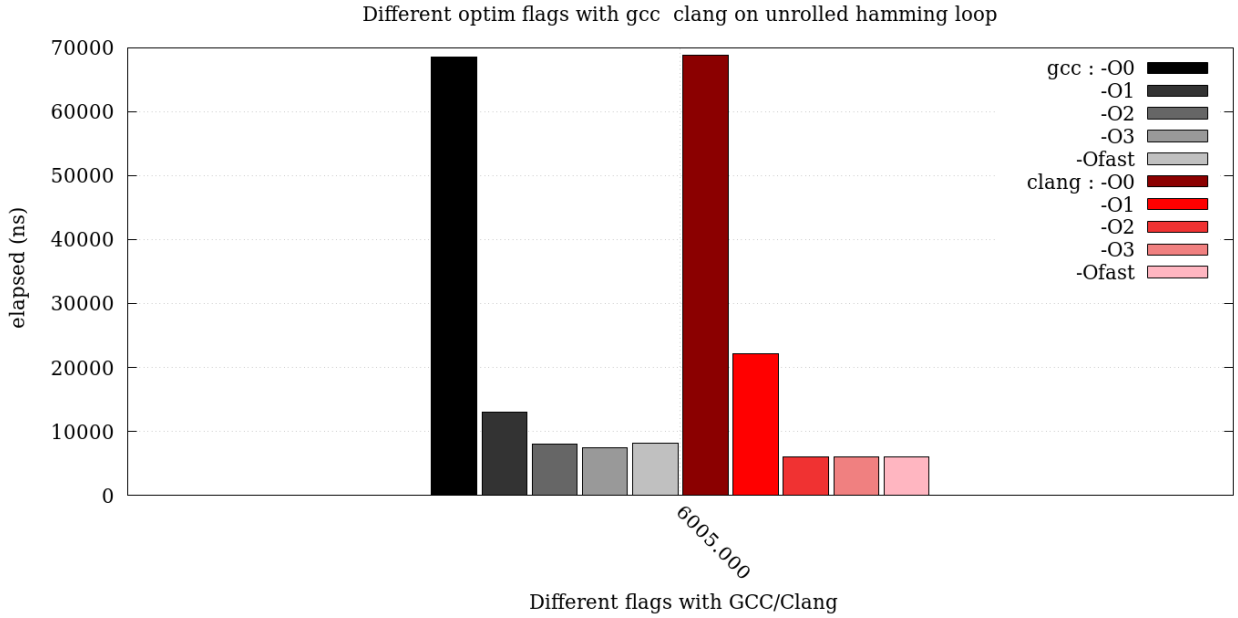
En règle générale, le déroulage d'une boucle peut aider à augmenter le parallélisme au niveau des instructions et à réduire le nombre d'échecs de cache en réduisant le nombre d'accès à la boucle. Mais cela peut également augmenter la taille du code généré. Par conséquent, nous devons choisir une valeur qui offre un bon équilibre entre les performances et la taille du code et mesurer les performances de notre code avec différents facteurs de déroulement pour trouver la valeur optimale pour votre cas d'utilisation spécifique.

Cache lines = (Size of loop iteration) / (Size of cache line)

J'ai eu recourt à un déroulage de boucles à la main. Mais j'aurais pu ajouter #pragma GCC unroll 4.

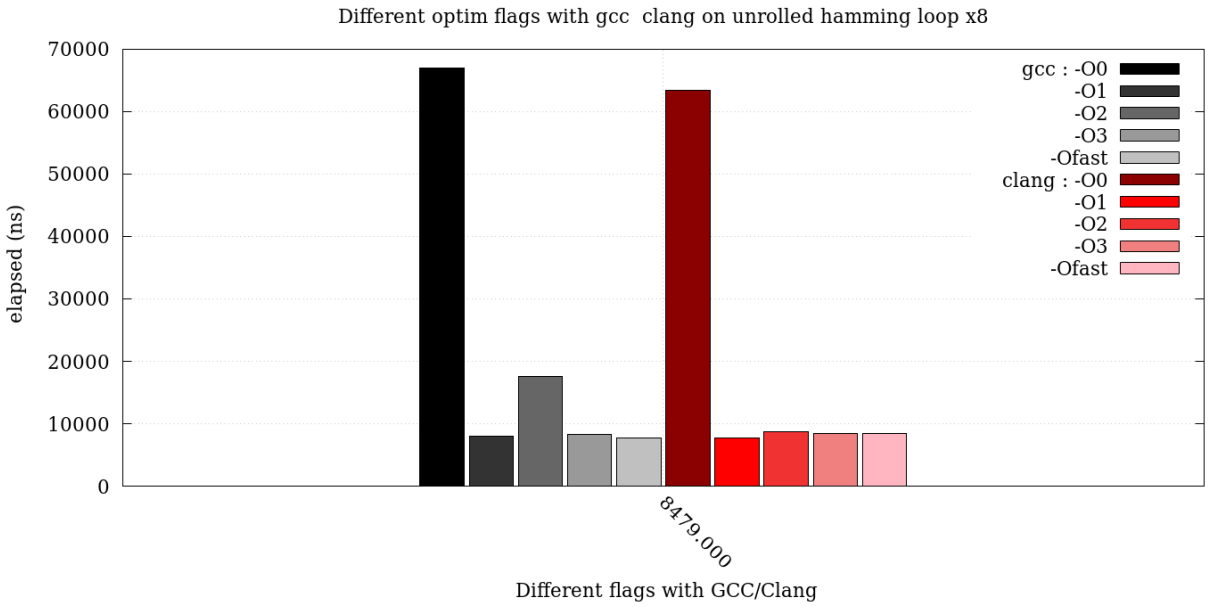
```
u64 hamming_unroll4(u8 *a, u8 *b, u64 n)
{
  u64 h = 0;
  for (u64 i = 0; i < (n - (n & 3)); i+=4)
    {
        h += __builtin_popcount(a[i] ^ b[i]);
        h += __builtin_popcount(a[i+1] ^ b[i+1]);
        h += __builtin_popcount(a[i+2] ^ b[i+2]);
        h += __builtin_popcount(a[i+3] ^ b[i+3]);
    }
  for (u64 j = (n - (n & 3)); j < n; j++)
        h +=  __builtin_popcount(a[j] ^ b[j]);

  return h;
}
```



```
u64 hamming_unroll8(u8 *a, u8 *b, u64 n)
{
  u64 h = 0;
  for (u64 i = 0; i < (n - (n & 7)); i+=8)
    {
        h += __builtin_popcount(a[i] ^ b[i]);
        h += __builtin_popcount(a[i+1] ^ b[i+1]);
        h += __builtin_popcount(a[i+2] ^ b[i+2]);
        h += __builtin_popcount(a[i+3] ^ b[i+3]);
        h += __builtin_popcount(a[i+4] ^ b[i+4]);
        h += __builtin_popcount(a[i+5] ^ b[i+5]);
        h += __builtin_popcount(a[i+6] ^ b[i+6]);
        h += __builtin_popcount(a[i+7] ^ b[i+7]);
    }
  for (u64 j = (n - (n & 7)); j < n; j++)
        h +=  __builtin_popcount(a[j] ^ b[j]);

  return h;
}
```



d. Vectorisation

On peut utiliser des instructions de traitement en parallèle telles que SSE ou AVX, pour calculer la distance de Hamming plus rapidement en parallèle sur plusieurs morceaux de données à la fois.

Si l'on met de côté quelques cas très rares, stocker et travailler sur des nombres sur 128 ou 256 bits n'a généralement pas beaucoup d'intérêt, et ce n'est d'ailleurs pas vraiment à cela que sert AVX : Advanced Vector eXtension.

Une instruction AVX 256 bits peut ainsi travailler sur 8 données de 32 bits en simultanée, ce qui permet d'améliorer significativement la

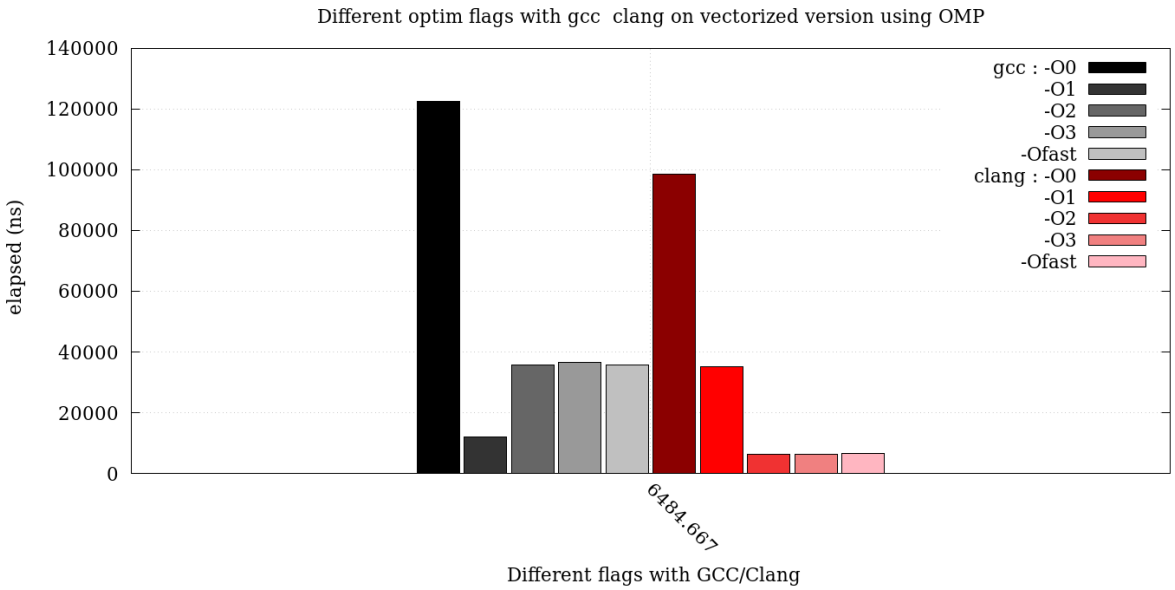
rapidité d'exécution d'un programme, pour peu que l'on ait besoin d'effectuer 8 opérations identiques en parallèle !

Dans ce cas, le compilateur interprète le code C/C++ (il s'agit généralement de boucles qui répètent une instruction) pour générer automatiquement du code en langage machine utilisant des instructions vectorielles.

En théorie, l'idée est excellente. En pratique, le code existant n'est pas forcément écrit pour être vectorisé. En dehors des cas simples, il faudra souvent que le développeur réécrive son code pour retirer des dépendances.

e. Parallélisme avec Open MP

Avec gcc : #pragma omp parallel for : c'est une directive d'Open MP qui indique au compilateur de créer un ensemble de threads qui s'exécuteront en parallèle.



V. Autres optimisations possibles :

1. Inline assembly

C'est la solution la plus simple techniquement, plutôt que de demander l'impossible au compilateur, nous pouvons décider d'écrire nous-même en assembleur certains morceaux de notre programme. Les gains que l'on peut obtenir sont excellents en pratique mais le passage du code C au code assembleur reste assez complexe.

2. Intrinsics

Il s'agit d'une option un peu plus flexible. Plutôt que d'avoir à écrire des morceaux de code en assembleur, les intrinsèques fournissent des raccourcis en langage C vers les instructions AVX. Leur manipulation diffère d'un compilateur à un autre, ce qui peut limiter ou améliorer la portabilité et la flexibilité du code.

VI. Conclusion

Les différents méthodes d'optimisation appliquées sur le code de hamming distance ont effectivement amélioré la performance de celui-ci. Il reste qu'à assembler tous les optimisations en un seul code et regarder si notre code devient encore plus performant ou bien le fait de mettre toute les directives d'optimisation et de parallélisme dans le même code rend celui-ci plus coûteux en termes de ressources utilisées.

VII. Remerciements

Je remercie Gülçin Gedik pour m'avoir permis d'effectuer les mesures sur sa machine.