Optimisation d'un code de distance de Hamming

Matéo Pasquier

21 Janvier 2023

Contents

1 Introduction

Le but de ce rapport est de mesurer les performances d'un noyau de calcul et tenter de l'optimiser en utilisant différentes méthodes. Nous nous intéresserons à un code écrit en C calculant la distance de Hamming entre deux chaînes de caractères représentants du code génétique.
Nous étudierons pour cela les performances de la version de base, puis celles des optimisations afin de les comparer avant de tout regrouper. Nous essaierons enfin de trouver une version optimale comprenant les optimisations avec les meilleures performances observées.

2 Noyau de calcul initial

Le code génétique est représenté sous forme de "Gray code", i.e une suite de chiffres binaires. Le code de la distance de Hamming est donc simple, nous pouvons faire une boucle sur les deux listes et les comparer en utilisant directement des opérations sur bits, opérations demandants typiquement peu de cycles CPU.

```
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;
}
```

C'est donc un algorithme de complexité linéaire $O(n)$ car il ne demande qu'une seule boucle sur les données. Il n'y a aussi qu'ne seule opération arithémtique (add) en plus des opérateurs logiques.

3 Architecture

Tous les tests ont étés exécutés sur un Intel Core i7 de 8-ème génération avec les caractéristiques suivantes :

Frequency (Ghz)	Cores	Threads	L1 Cache (Kib)	L2 Cache (Mib)	L3 Cache (Mib)
1.8	4	8	32	1	8

4 Optimisations

Nous allons expérimenter plusieurs optimisations pour ce noyau et voir lesquelles améliorent ses performances.

4.1 Compilateur

Pour chaque compilateur utilisé nous allons essayer plusieurs flags d'optimisations (O1, O2, Ofast...). Ces derniers vont apporter des modifications au code assembleur généré qui peuvent êtres plus ou moins pertinentes, nous essaierons de déduire lesquels nous avantagent.
Certains de ces flags permettent entre autre de vectoriser le code (SIMD) ce qui dans certains cas peut grandement aider à améliorer le temps d'exécution.

4.2 Déroulage

Le principe du déroulage est de réduire le nombre de boucles en répétant une même instruction plusieurs fois à la suite. En théorie cela pourrait améliorer les performances car il permet au CPU de faire moins de JUMP, ou branchement dans le code assembleur qui gère la boucle, le processeur préférant en effet un code le plus linéaire possible.
Nous implémenterons deux versions du déroulage, une x4 et une autre x8.

4.2.1 unrollx4

```
for (u64 i = 0; i < n; 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]);
}
```

4.2.2 unrollx8

```
for (u64 i = 0; i < n; 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]);
}
```

4.3 Alignement mémoire

Dû au langage de programmation utilisé (C) qui est de bas-niveau, nous devons allouer nous-même la mémoire qui contiendra les deux vecteurs que nous comparerons.

```
//Allocating memory for sequence bases
s->bases = malloc(sizeof(u8) * sb.st_size);
```

L'amélioration ici serait d'utiliser des fonctions qui allouent cet espace de manière continue dans la mémoire, de sorte que l'accès aux données par le CPU soit plus simple et rapide.

```
//Allocating memory for sequence bases
s->bases = aligned_alloc(sizeof(u8), sizeof(u8) * sb.st_size);
```

4.4 Popcount

Le "popcount" consiste à compter le nombre de bits initialisés à 1 dans un code binaire, fonction qui est au centre du calcul de la distance de Hamming dans notre cas. Le noyau naïf utilise un popcount de base intégré dans GCC, "built-in popcount" qui est une "intrinsic".

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

Nous allons étudier une autre implémentation de popcount utilisant SSE4, un jeu d'instruction x86 censé améliorer les performances SIMD.

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

4.5 Autres améliorations

Une autre amélioration possible serait de paralléliser le code sur plusieurs threads ou cœur en utilisant des librairies comme openMP ou MPI. Ce code est facilement parallélisable car le résultat d'une itération ne dépend pas d'une autre, elles peuvent donc toutes êtres effectuées en parallèle tant que la somme est correctement accumulée dans la bonne section mémoire.

5 Performances

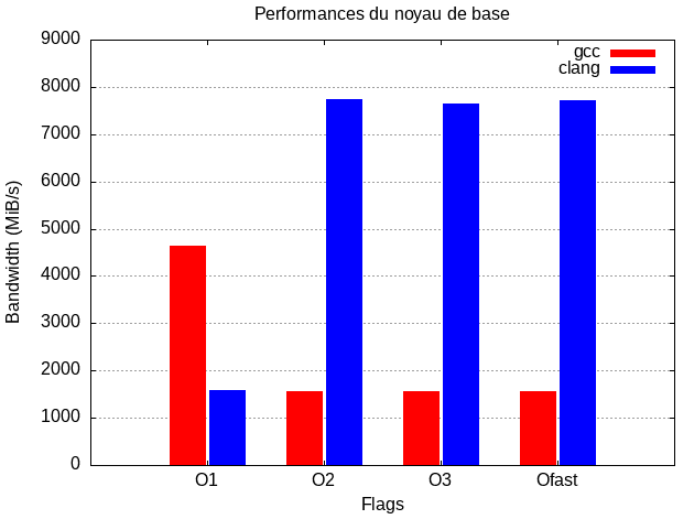
5.1 Méthodologie

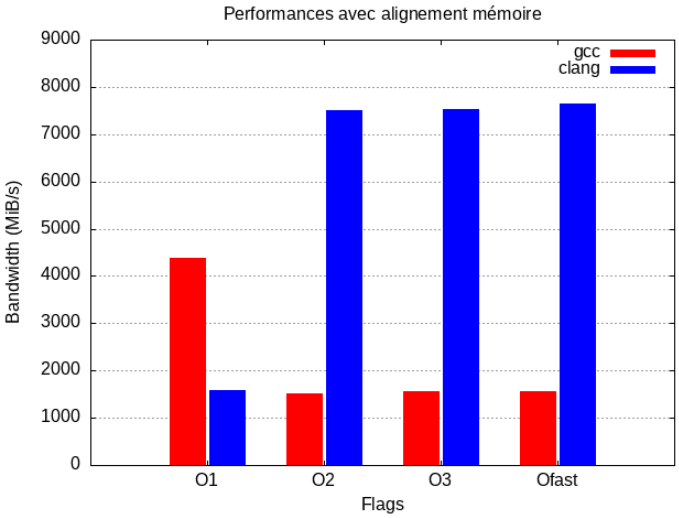
Nous avons donc à disposition plusieurs noyaux de calcul qui nous espérons auront de meilleures performances que celui de base. Afin de mesurer leurs performances, nous utiliserons la méthode suivante :

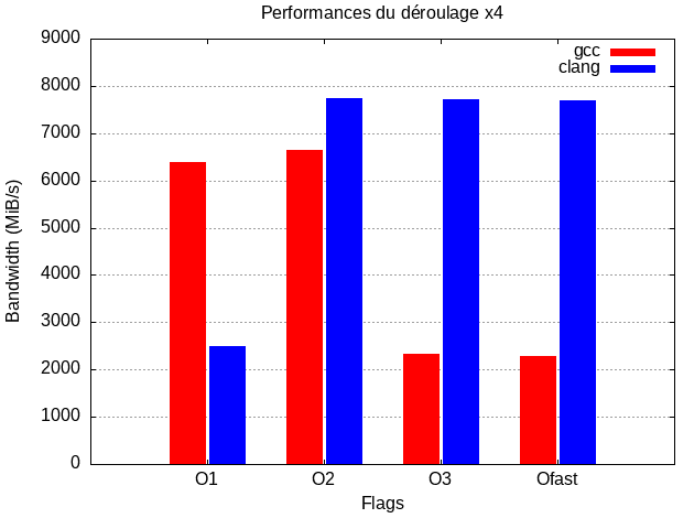
- Pinner le processus sur un cœur spécifique pour éviter que le système d'exploitation ne le change durant l'exécution.
- Fixer la fréquence du CPU pour être sûr d'utiliser ses meilleures capacités.
- Toutes les versions utiliserons le même jeu d'entrée afin de s'assurer que le résultat final soit toujours correct et donc que l'implémentation est bonne.
- Nous utiliserons d'abord une taille d'entrée de 100MiB pour simuler la capacité du code de manière générale sur un code génétique arbitrairement long, puis nous verrons des tailles plus faibles qui pourraient faciliter l'accès aux caches.
- Nous utiliserons deux compilateurs C, gcc et clang.
- Pour chaque compilateur, nous utiliserons plusieurs flags d'optimisation : O1, O2, O3 ainsi que Ofast.
- Chaque version sera exécutée plusieurs fois afin d'obtenir une performance moyenne plus stable.

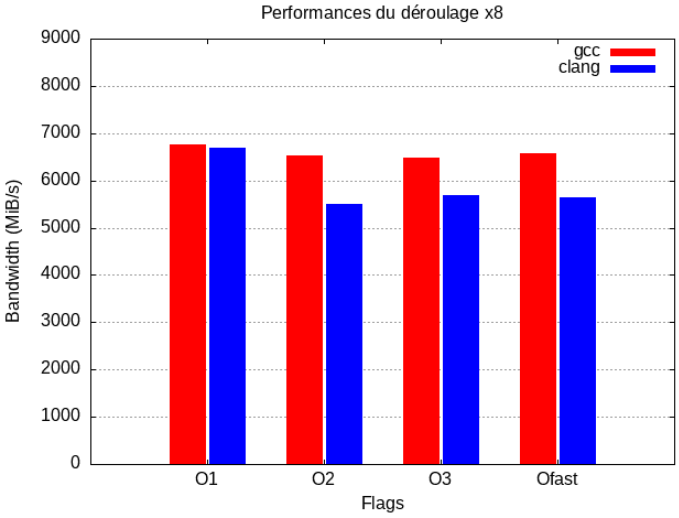
5.2 Résultats

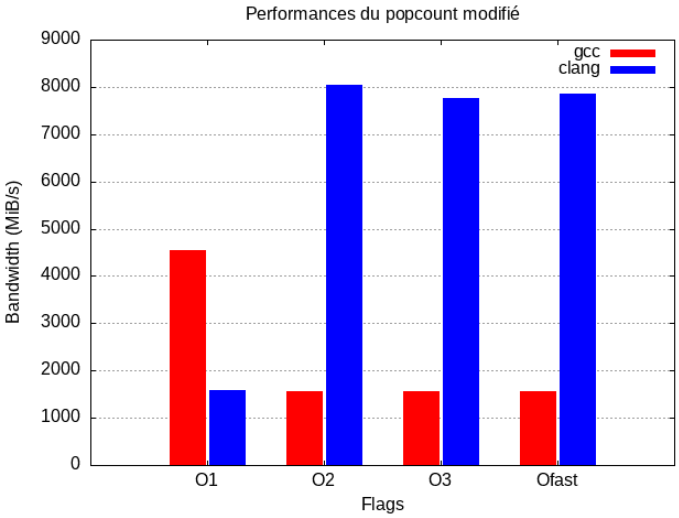
Nous avons donc cinq optimisations différentes, plus un noyau de calcul où toutes ces dernières ont été rassemblées. Le temps d'exécution du code a été calculé en nanosecondes, mais nous avons rajouté une métrique de comparaison plus pertinente. Nous comparerons donc la bande passante de chaque version, en MiB/s, représentant la quantité de donnée pouvant être calculée par le programme toutes les secondes.

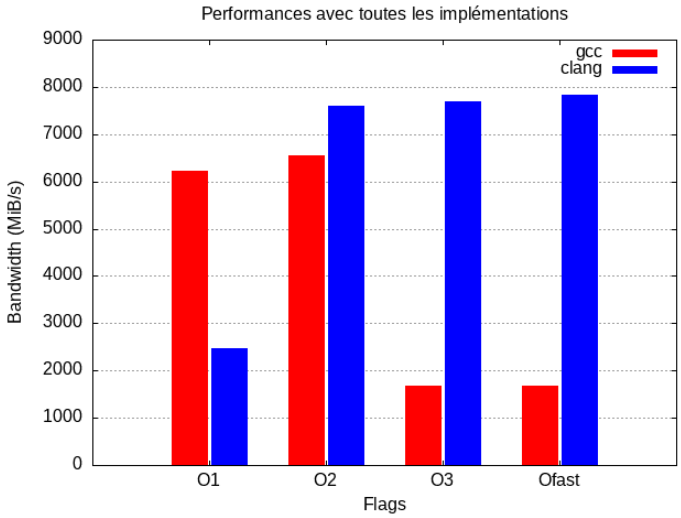

Performances du noyau de base
Bandwidth (MiB/s)
Flags
gcc
clang
O1
O2
O3
Ofast
0
1000
2000
3000
4000
5000
6000
7000
8000
9000


Performances avec alignement mémoire
Bandwidth (MiB/s)
Flags
gcc
clang
O1
O2
O3
Ofast
0
1000
2000
3000
4000
5000
6000
7000
8000
9000


Performances du déroulage x4
Bandwidth (MiB/s)
Flags
gcc
clang
O1
O2
O3
Ofast
0
1000
2000
3000
4000
5000
6000
7000
8000
9000


Performances du déroulage x8
Bandwidth (MiB/s)
Flags
gcc
clang
O1
O2
O3
Ofast
0
1000
2000
3000
4000
5000
6000
7000
8000
9000

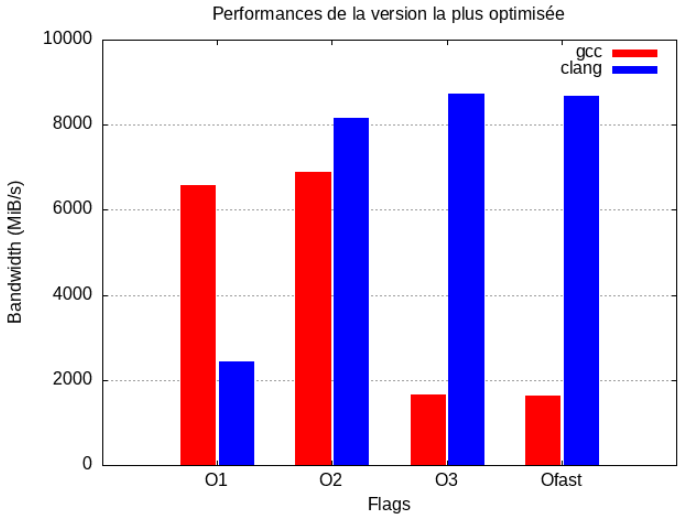




Nous remarquons de prime abord que le compilateur clang est globalement plus performant que gcc sur la version naïve, avec jusqu'à un facteur cinq. Nous notons que la meilleure performance du noyau naïf est de 7750.951 MiB/s avec le flag O2. Clang, de manière générale, obtient de meilleures performances que GCC à part en déroulage x8. Il est intéressant de noter quel le déroulage x8 est moins performant que le x4, ce qui peut être causé par un trop peu de cache d'instruction du processeur, ce qui causerait plus d'accès mémoire et donc "annule" l'intérêt du déroulage.
Le déroulage x4 quand à lui améliore les performances de manière globale peu importe le compilateur ou flag, il conviendrait donc de garder cette optimisation dans le futur.

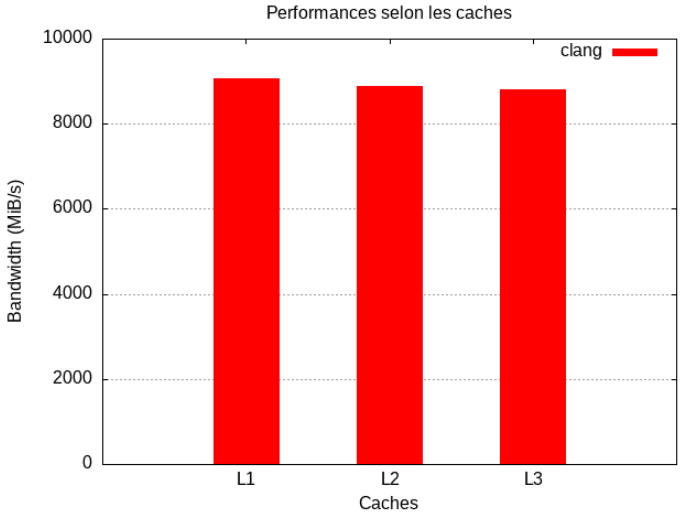
L'alignement mémoire quand à lui ne donne pas les résultats espérés en faisant nettement baisser les capacités du programme. Le nouveau popcount quand à lui rend le programme plus performant, avec un pic à 8035.533 MiB/s, encore une fois avec le compilateur clang et le flag O2. Nous voyons enfin que mettre toutes les optimisations les unes avec les autres ne donne pas de résultat convainquant, ce qui était attendu car certains noyaux font baisser les performances par rapport à la version de base.

Nous allons maintenant créer la version la "plus optimale" en ne gardant que les changements qui ont impacté positivement les performances du programme, i.e le popcount ainsi que le déroulage x4.



Nous voyons donc que cette version est au dessus des autres, avec un pic de bande passante à 8737.854 MiB/s avec clang et un flag O3 cette fois-ci.

Jusqu'ici nous avons étudié les performances pour une grande taille d'entrée censée représenter une utilisation typique du programme, maintenant nous allons choisir des tailles spécifiques en rapport aux capacités des caches de la machine. En utilisant la version la plus optimisée et avec trois entrées différentes (32Kib, 1MiB, 8MiB), nous obtenons les performances suivantes :



Nous voyons que nous obtenons la meilleure performance tous contextes confondus avec l'entrée de 32KiB qui possède una bande passanter moyenne de 9079.815 MiB/s. Ce résultat était attendu; la chaïne de caractère d'entrée pouvant rentrer entièrement dans le cahce L1 le processeur n'a pas besoin de faire autant d'accès mémoire qu'usuellement. Nous observons que même si sim-

ilaires, les performances baissent au fur et à mesure que l'on augmente la taille d'entrée. Le processeur utilise les données du cache L1, si toutes ne peuvent pas y rentrer il doit perdre du temps à aller chercher dans le cache L2 et ainsi de suite. Une taille d'entrée de 8MiB pouvant rentrer entièrement dans le cache L3 obtient toujours de meilleures performances qu'une taille arbitraire testée auparavant (8796.677 MiB/s contre 8737.856 MiB/s) car le CPU n'a pas à faire d'accès mémoire dans la DRAM.

6 Conclusion

Nous avons amélioré un noyau de calcul de distance de Hamming sur du code génétique en implémentant diverses optimisations spécifiques au matériel utilisé, puis avons trouvé que la version la plus performante était celle avec un déroulage x4 plus un popcount SSE4, le tout compilé grâce à clang avec un flag d'optimisation O3.
Nous avons aussi déduit qu'une plus grande taille de jeu d'entrée baissait la bande passante moyenne du programme dû aux multiples accès mémoires nécessaires pour le processeur. Une petite taille d'entrée pouvant être entièrement stockée dans le cache L1 permet donc d'obtenir les meilleures performances car le CPU perd moins de temps à devoir aller chercher toutes les données dans les différents caches et RAM de la machine au court de l'exécution.

Une autre optimisation possible serait de paralléliser ce programme afin de répartir la charge de travail sur les plusieurs threads ou cœurs disponibles. Cet algorithme s'y prêterait bien car chaque processus n'a pas besoin de communiquer avec les autres durant l'exécution, juste les différentes sommes doivent êtres accumulées au même endroit à la fin des boucles.

7 Références

- He, M. (2004) Genetic Code, Hamming Distance and Stochastic Matrices
- Git du projet : `https://github.com/Datky/M1_AP_TP`