

ISTY UVSQ
UNIVERSITÉ PARIS-SACLAY

M1 Calcul Haute Performance, Simulation
PROJET DE L'ARCHITECTURE PARALLÈLE
RAPPORT DE PROJET

Code génétique, distance de Hamming et
matrices stochastiques

Réalisé par :

Mme. RIM ACHRRAB

Encadré par :

M. IBNAMMAR M.Salah

Promotion : 2022/2023

Résumé

Dans ce présent rapport, on a vu l'importance des différents options utilisées pour optimiser les codes et augmenter ses performances. J'ai testé plusieurs versions du code et je les ai comparé avec la version initiale du code (sans ajout des options d'optimisation du processus).

Mots clés performances, GCC, Clang, flags :

Abstract

In this report, we have to prove the role of the different options that we can use for optimizing the code and return the processus more performant than the first code used without performance options. I tested a different versions of the code and I compared them with the first version of the code to analyse the performance of every program.

Keywords performance, flags, GCC, CLANG :

Table des matières

Résumé	i
Abstract	i
Introduction générale	1
1 Résumé de l'article	2
2 Réalisation	3
2.1 Explication du programme "genseq.c"	3
2.2 Explication du code "dist.c"	3
2.3 Fichier de type DNA	3
2.4 L'alignement mémoire	4
2.5 Remarque initiale du code	4
2.6 Optimisation	4
3 Analyse des performances	5
3.1 Code initial	5
3.1.1 Compilateur GCC	5
3.1.2 Compilateur CLANG	6
3.1.3 Comparaison entre GCC et CLANG	7
3.2 Alignement en mémoire	8
3.2.1 Compilateur GCC	8
3.2.2 Compilateur CLANG	9
3.2.3 Comparaison entre GCC et CLANG	10
3.2.4 Comparaison entre le code initial et le code aligné	10
3.2.5 Conclusion	11
3.3 Vectorisation	11
3.3.1 Compilateur GCC	12
3.3.2 Compilateur CLANG	12
3.3.3 Compraison entre GCC et CLANG	13
3.3.4 Comparaison entre code initial et version vectorisée	13
3.3.5 Conclusion	14
3.4 Loop unrolling	14
3.4.1 Compilateur GCC	15
3.4.2 Compilateur CLANG	15
3.4.3 Comparaison ente GCC et CLANG	16

3.4.4	Comparaison entre code initial et déroulé	16
3.4.5	Conclusion	17
3.5	Version optimisée	17
3.5.1	Compilateur GCC	17
3.5.2	Compilateur CLANG	18
3.5.3	Comparaison entre GCC et CLANG	18
3.5.4	Comparaison entre le code initial et optimisé	19
3.5.5	Conclusion	20
3.6	Parallélisation : OpenMp	20
3.6.1	Compilateur GCC	21
3.6.2	Compilateur CLANG	21
3.6.3	Comparaison entre GCC et CLANG	22
3.6.4	Comparaison entre le code initial et parallélisé	22
3.7	Optimisation générale	23
3.7.1	Performances du code (GCC et CLANG)	23
3.7.2	Comparaison entre le code initial et final (GCC)	24
3.7.3	Conclusion	24

Table des figures

1	results for GCC compiler, length = 100	5
2	results for GCC compiler, length = 2800	6
3	results for CLANG compiler, length = 100	6
4	results for CLANG compiler, length = 2800	7
5	Comparaison des niveaux d'optimisation entre GCC et Clang, length = 100 . . .	7
6	Comparaison des niveaux d'optimisation entre GCC et Clang, length = 2800 . .	8
7	Histogrammes pour différents flags avec GCC	9
8	Histogrammes pour différents flags avec CLANG	9
9	Comparaison entre GCC et CLANG	10
10	Comparaison entre code initial et aligné pour GCC	10
11	Comparaison entre code initial et aligné pour Clang	11
12	version vectorisée du code (gcc compiler)	12
13	version vectorisée du code (clang compiler)	12
14	comparaison entre GCC et CLANG	13
15	comparaison entre code initial et vectorisé (GCC)	13
16	comparaison entre code initial et vectorisé (CLANG)	14
17	version déroulée du code (GCC)	15
18	version déroulée du code (CLANG)	15
19	Comparaison entre GCC et CLANG	16
20	Comparaison entre code initial et déroulé (GCC)	16
21	Comparaison entre code initial et déroulé (CLANG)	17
22	Version optimisée du code (GCC)	17
23	Version optimisée du code (CLANG)	18
24	Comparaison entre GCC et CLANG	18
25	Comparaison entre code initial et optimisé (GCC)	19
26	Comparaison entre code initial et optimisé (CLANG)	19
27	fonction hamming parallélisée	20
28	Version parallélisée (GCC)	21
29	Version parallélisée (CLANG)	21
30	Comparaison entre GCC et CLANG	22
31	Comparaison entre le code initial et parallélisé (GCC)	22
32	Comparaison entre le code initial et parallélisé (CLANG)	23
33	Version finale du code (GCC et CLANG)	23
34	Comparaison entre le code initial et final (GCC)	24
35	Comparaison entre le code initial et final (CLANG)	24

Introduction générale

1 Résumé de l'article

L'article se repose sur la construction d'une matrice stochastique basée sur le code génétique en utilisant la distance de Hamming et le code GRAY à 2 bits ($\{00, 01, 10, 11\}$) pour attribuer des valeurs aux bases génétiques $\{C, A, G, U\}$. Le domaine des mathématiques exprime cette relation par des fonctions utilisées pour coder les informations génétiques par des triplets de quatre caractères de l'ensemble $\{A, C, G, U\}$.

L'article nous montre trois types de tables de ce code génétique qui ont été révélées dans les différentes littératures et domaines grâce à leurs structures symétriques. La première table est basée sur le code Gray, la deuxième est appelée tableau bi-périodique et la troisième est générée à partir d'un arbre de 4-aires. Les chercheurs utilisent ces tables pour relier le code génétique avec les matrices en utilisant le code Gray, puis ils calculent la distance de Hamming pour chaque code Gray obtenu, puis ils génèrent des matrices numériques qu'ils doivent être stochastiques ou stochastiques doubles.

La distance de Levenshtein est plus sophistiquée. Elle est définie pour des chaînes de longueur arbitraire. Elle calcule les différences entre deux chaînes de caractères, alors que l'on compterait une différence non seulement quand les chaînes ont des caractères différents, mais aussi quand l'une a un caractère alors que l'autre n'en a pas.

2 Réalisation

2.1 Explication du programme "genseq.c"

Ce programme est un générateur de séquences d'ADN (A,C,G et T) d'une façon aléatoire.

En entrée le code prend le nom du fichier de sortie et la longueur souhaitée pour la chaîne de caractères d'ADN. La fonction `srand` est employée pour initialiser le générateur de nombres aléatoires en utilisant l'identifiant de processus (`getpid`) pour obtenir une séquence aléatoire différente chaque exécution (pour éviter les redondances). La fonction `randxy` a pour utilité la génération d'un nombre aléatoire entre 0 et 4 sera utilisé pour sélectionner un caractère des bases d'ADN (A, T, C ou G) pour l'ajouter ensuite à la séquence.

2.2 Explication du code "dist.c"

Ce programme calcule la distance de Hamming entre deux séquences d'ADN. La distance de Hamming est le nombre de positions où les deux séquences sont différentes (comme j'ai expliqué dans le résumé de l'article).

Le programme définit des structures de données pour les séquences d'ADN (`seq_t`) et des codes d'erreur pour les erreurs potentielles. Il utilise des fonctions pour charger les séquences d'un fichier donné, libérer de la mémoire utilisée pour les historiques et calculer la distance de Hamming entre les séquences une autre fois.

La fonction `load_seq()` est utilisée pour charger les séquences d'ADN à partir de fichiers en utilisant la fonction `stat()` pour obtenir la taille du fichier, puis en libérant de la mémoire pour stocker les séquences et en lisant les octets du fichier dans la mémoire allouée.

La fonction `release_seq()` est utilisée pour libérer la mémoire occupée allouée pour les séquences.

La fonction `hamming()` est utilisée pour calculer la distance de Hamming entre les séquences en utilisant la fonction `__builtin_popcount()` pour compter le nombre de bits différents entre les séquences en utilisant l'opérateur binaire XOR (renvoie 1 si les deux séquences sont différentes, et 0 si elles sont égales).

Dans la fonction `main()`, Le programme prend en entrée les noms des fichiers contenant les séquences d'ADN et utilise les fonctions précédentes pour charger les séquences afin de calculer la distance de Hamming et libérer la mémoire utilisée pour les séquences.

2.3 Fichier de type DNA

Un fichier de type ADN est utilisé pour stocker des informations génétiques sous forme de séquences d'ADN. Ces séquences d'ADN peuvent être utilisées pour une variété de domaines, notamment l'analyse génétique, médecine génétique, biotechnologie, conservation de la biodiversité,...

2.4 L’alignement mémoire

En C, l’alignement mémoire est utilisé pour améliorer les performances en accélérant l’accès aux données en mémoire de l’ordinateur. Les processeurs utilisent des caches pour accélérer la vitesse de l’accès aux données en mémoire. Les caches sont organisés en cache lines, qui sont généralement de la taille de quelques octets. Lorsque le processeur lit ou écrit des données en mémoire, il lit ou écrit généralement un cache line complet, même si seul un octet de données est nécessaire.

Lorsque les données sont alignées sur les limites des cache lines, le processeur peut accéder aux données plus rapidement, car il n’a pas besoin de lire ou d’écrire des données inutiles. Si les données ne sont pas alignées sur les limites des cache lines, le processeur doit lire ou écrire des données supplémentaires, ce qui ralentit l’accès aux données.

En utilisant l’alignement mémoire, on peut améliorer les performances en s’assurant que les données sont alignées sur les limites des cache lines. Il est à noter que cela n’est pas nécessaire pour tous les types de données, et que cela peut causer des problèmes de mémoire si les données sont trop grandes pour tenir dans une cache line.

Dans le code, on doit utiliser l’alignement mémoire car en fait la distance de Hamming acceptent les caractères de qui ont un nombre d’octets identique.

2.5 Remarque initiale du code

Sans alignement du code, on remarque que la distance de Hamming est égale à 0 et le temps de l’exécution du code se diffère suivant chaque flag d’optimisation.

2.6 Optimisation

L’utilisation de la fonction `popcount()` dans est optimale pour compter le nombre de bits qui sont différents entre deux octets (elle évite les duplication). Cette fonction est une intrinsic, ce qui signifie qu’elle est directement implémentée dans le compilateur et peut tirer parti de l’ensemble d’instructions de l’architecture cible pour effectuer l’opération efficacement. Cependant, dans certains cas, il peut être possible d’optimiser les performances de la fonction en utilisant un algorithme plus spécialisé, comme une table de recherche, qui est optimisé pour l’utilisation spécifique. Cela dépend des exigences et des contraintes spécifiques de l’application.

Vectorisation :

La directive `pragma omp simd` indique au compilateur de vectoriser la boucle, et la clause `reduction (+ : h)` spécifie que la variable `h` doit être traitée comme une variable de réduction, ce qui signifie qu’elle est cumulée sur toutes les itérations dans la boucle vectorisée.

3 Analyse des performances

3.1 Code initial

Avant tous, j'ai exécuté le code initial sans ajouter les options d'optimisation. Les schéma ci-dessous montre les performances du code :

3.1.1 Compilateur GCC

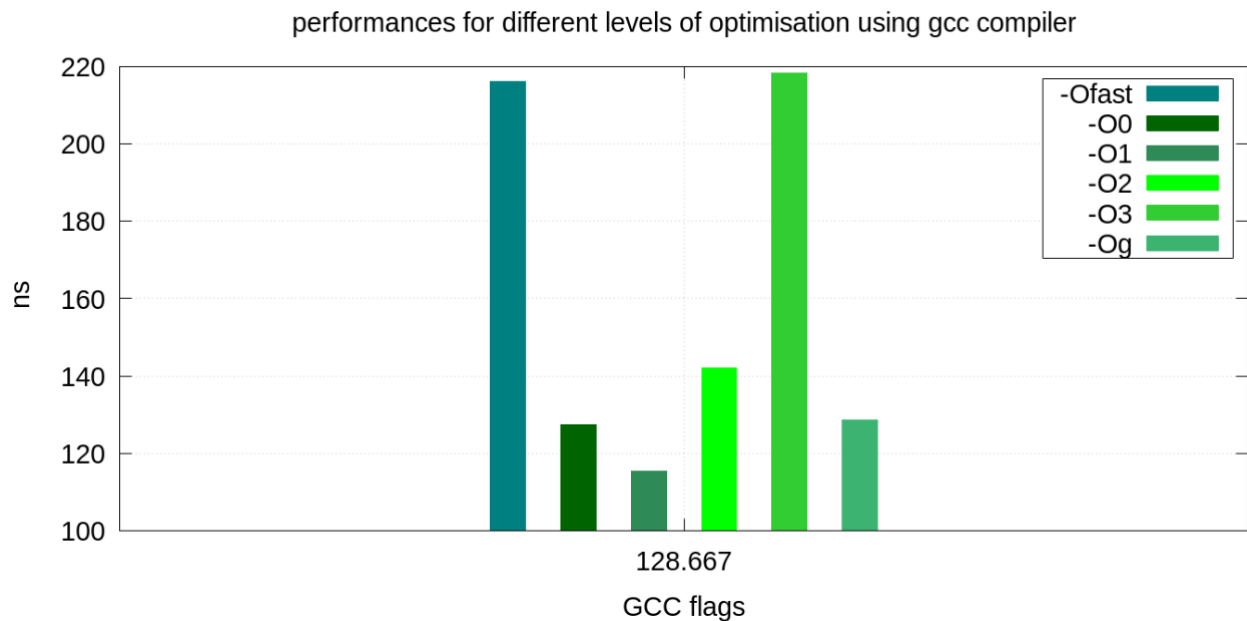


FIGURE 1 – results for GCC compiler, length =100

Ces histogrammes illustrent les différents niveaux d'optimisation en fonction de leur temps d'exécution (en (ns)) pour calculer la distance de hamming entre deux séquences d'ADN de longueur = 100.

On remarque que le niveau d'optimisation le plus performant (ou flag) est le niveau d'optimisation -O1. Le niveau d'optimisation le moins performant en utilisant GCC compiler est -O3 (d'après les résultats de ma machine).

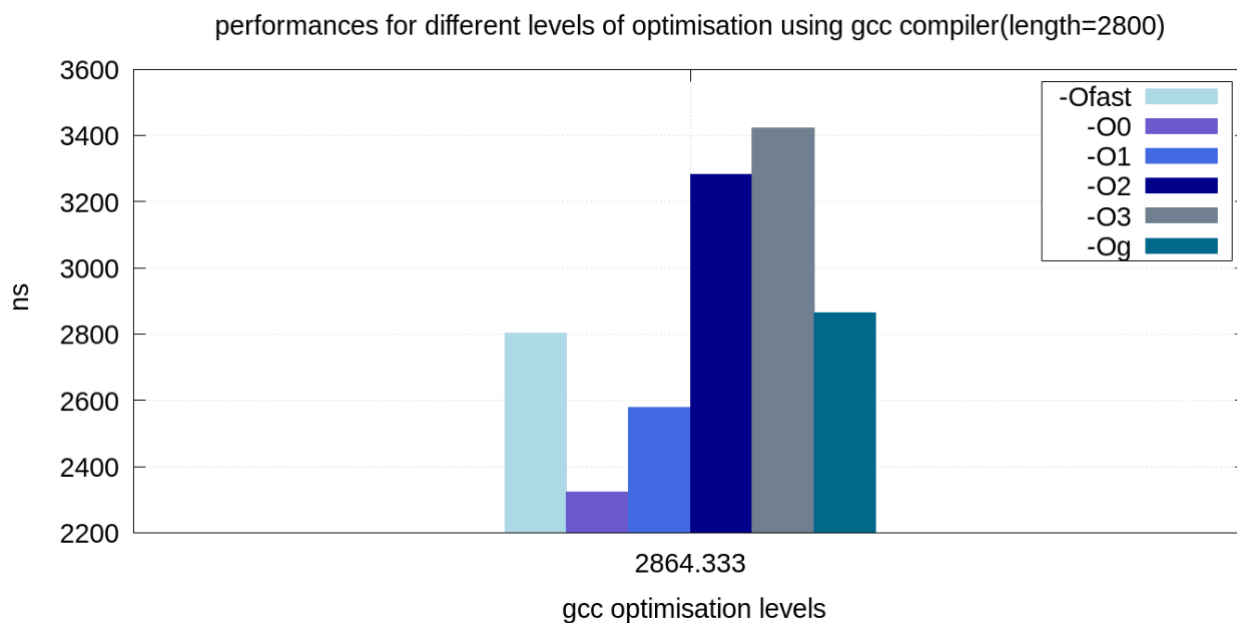


FIGURE 2 – results for GCC compiler, length = 2800

Pour une longueur de séquences égales à 2800 (supérieure à 1000), on remarque que le flag d'optimisation -O0 est le plus performant, suivi du flag -O1. Le plus faible en terme de performance est le niveau -O3 (même que pour une séquence de logueur inférieure ou égale à 100).

3.1.2 Compilateur CLANG

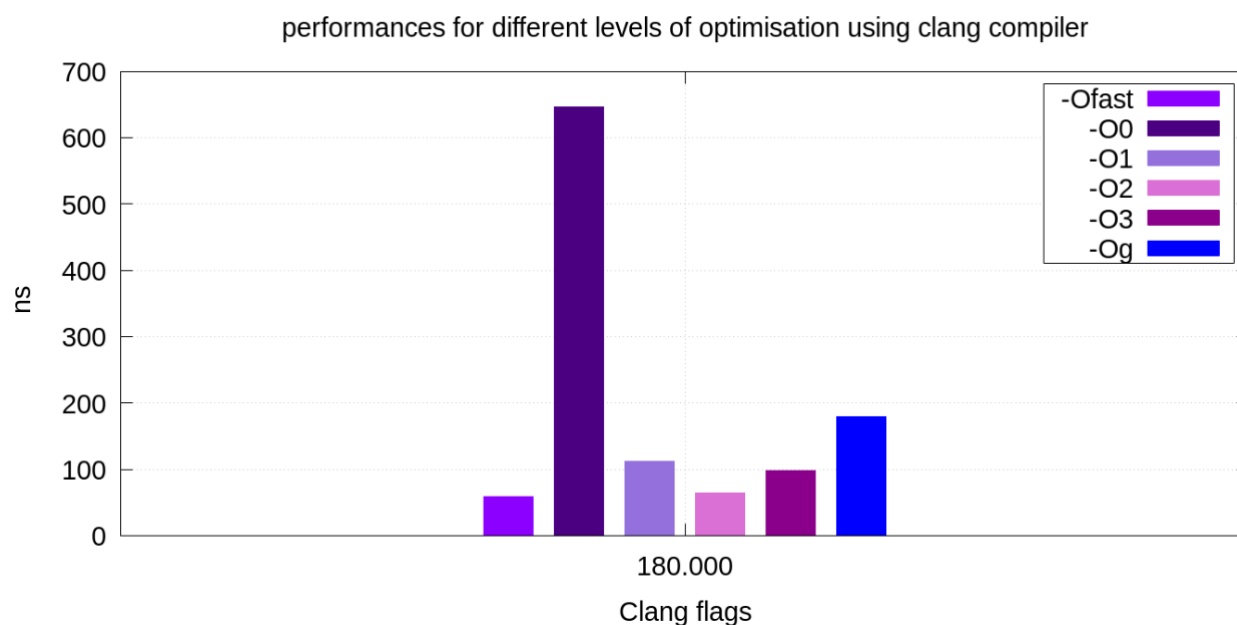


FIGURE 3 – results for CLANG compiler, length = 100

On remarque que pour le compilateur CLANG, le niveau d'optimisation le plus performant est -Ofast, et le moins performant (en fonction des ns) est le niveau -O0. Ce qui n'est pas le cas pour la compilateur GCC.

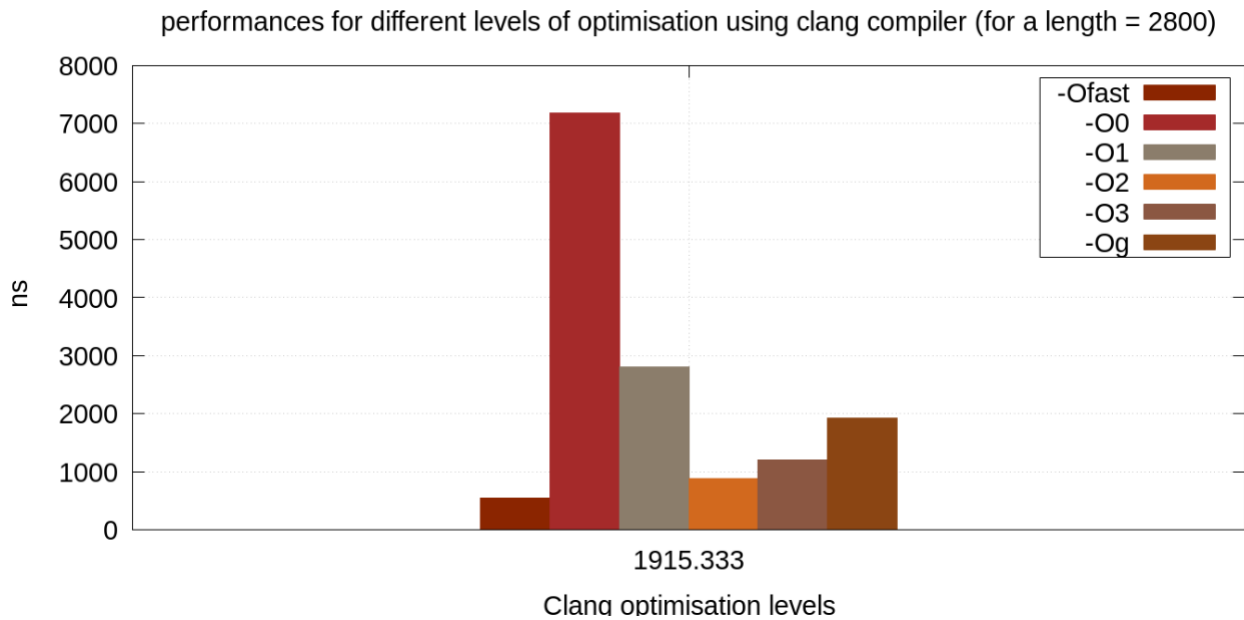


FIGURE 4 – results for CLANG compiler, length = 2800

Aussi que pour une séquence de logueur plus grand (2800), le niveau le plus performant est -Ofast, et le niveau le plus faible est -O0.

3.1.3 Comparaison entre GCC et CLANG

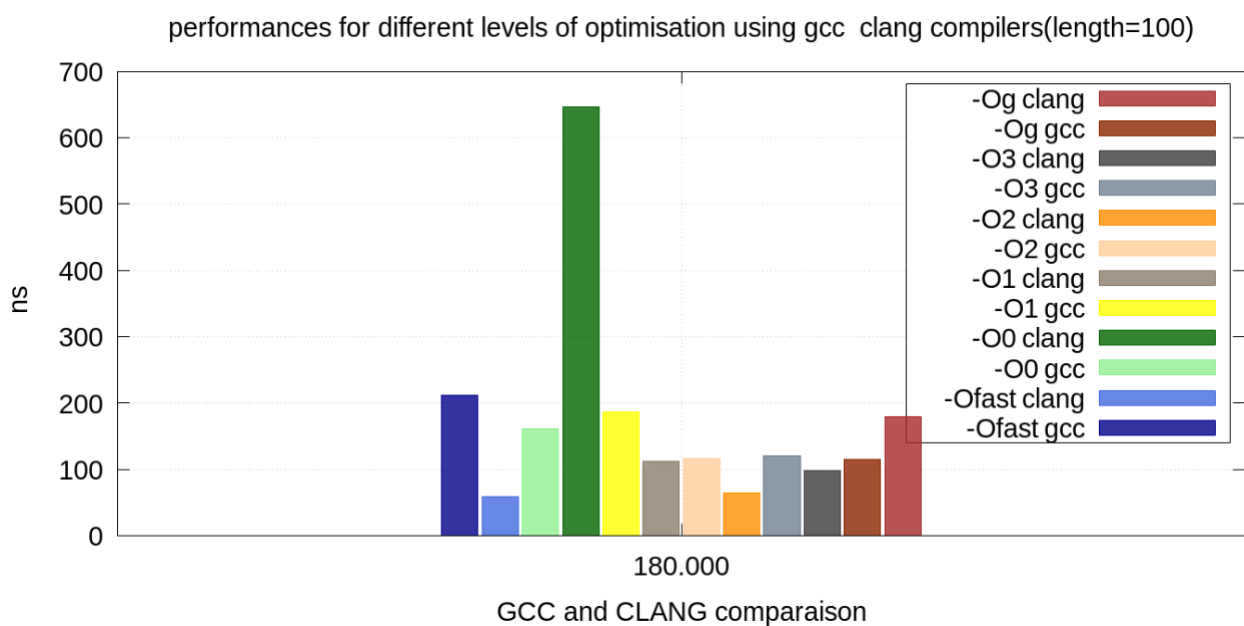


FIGURE 5 – Comparaison des niveaux d'optimisation entre GCC et Clang, length = 100

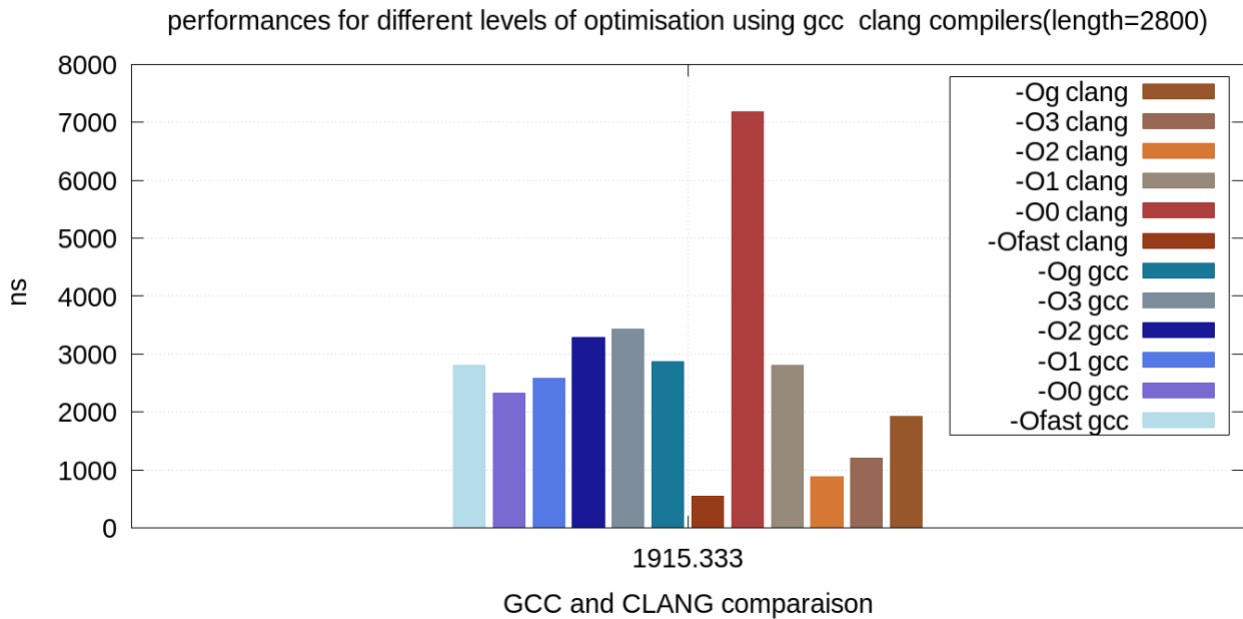


FIGURE 6 – Comparaison des niveaux d’optimisation entre GCC et Clang, length = 2800

On remarque -d’après les résultats issus de ma machine- que le compilateur développé par LLVM PROJECT, Clang, est le plus optimal en terme de performances (en fonction du temps écoulé en nano-secondes (ns)).

3.2 Alignement en mémoire

Pour que les processeurs de notre machine puissent accéder rapidement et d’une manière plus performante aux données d’un programme donné lors de son exécution, on utilise l’alignement en mémoire pour mieux optimiser les performances du code initial.

Pour je puisse aligner les données du code, j’ai utilisé : `'-falign-loops=1'` pour aligner les boucles dans les programmes en 1 octet pour que le processeur puisse accéder d’une manière plus flexibles aux données qui vont être alignées.

Ci-dessous, les histogrammes résultants des deux compilateurs GCC et Clang, pour une longueur de séquence égale à 2800, ainsi qu’une comparaison entre les performances du code initial et le code aligné.

3.2.1 Compilateur GCC

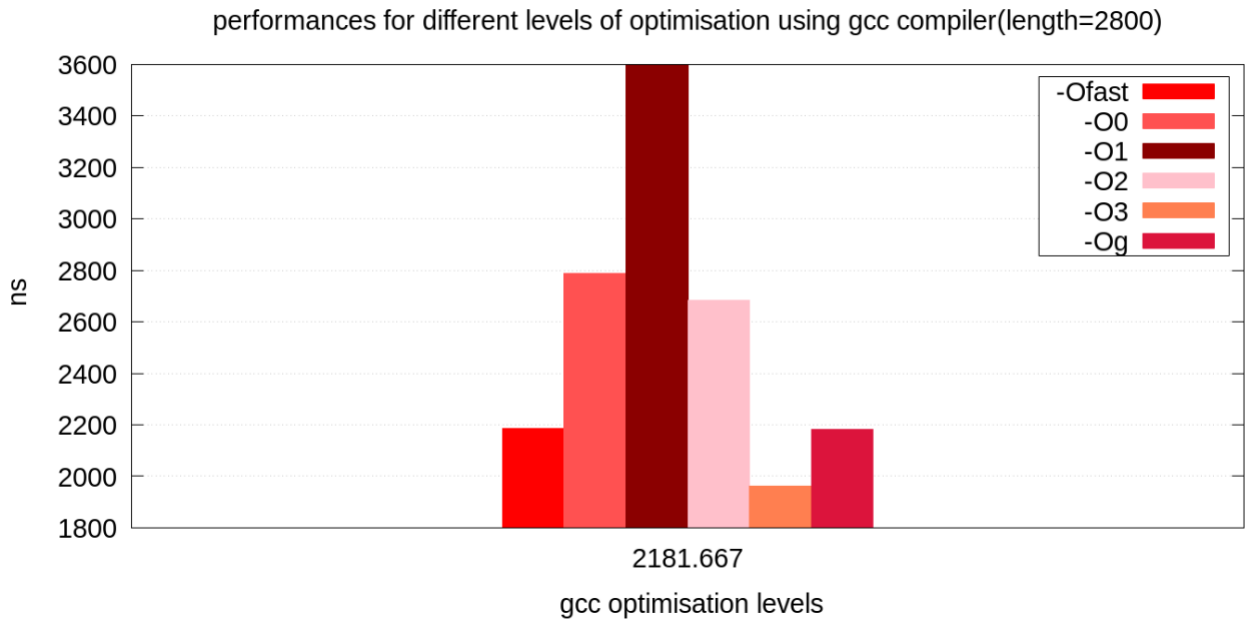


FIGURE 7 – Histogrammes pour différents flags avec GCC

D'après mes résultats, je remarque que le niveau le plus performant en utilisant le compilateur GCC est **-O3**.

3.2.2 Compilateur CLANG

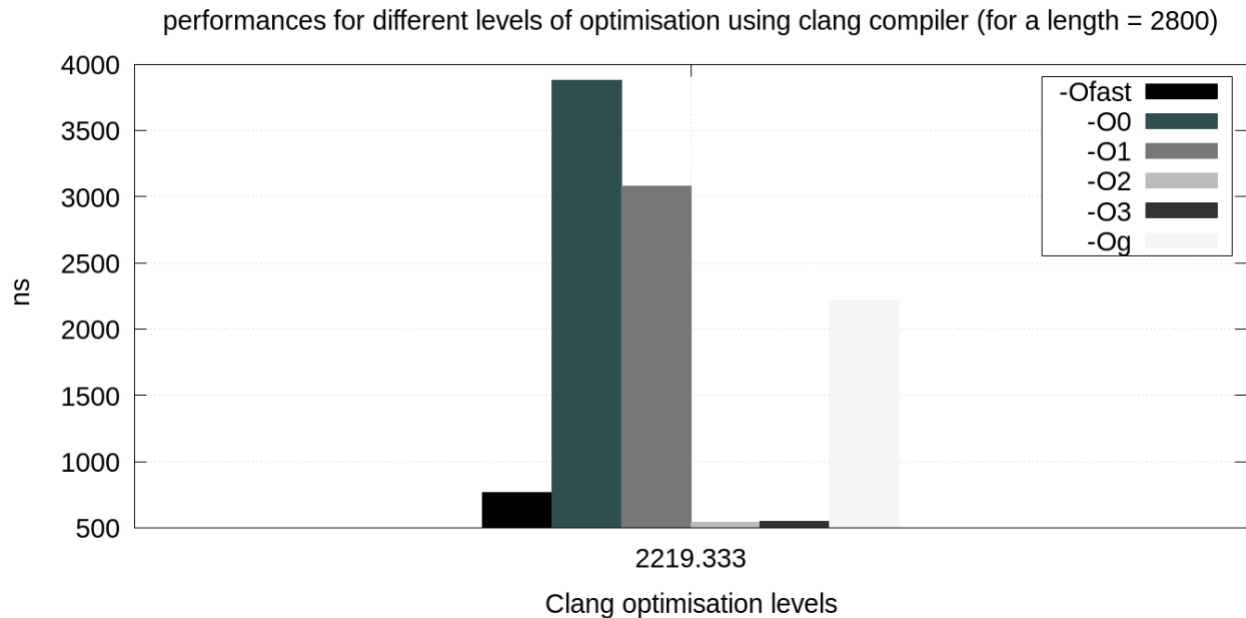


FIGURE 8 – Histogrammes pour différents flags avec CLANG

On remarque que le niveau le plus performant du compilateur CLANG est **-O2**.

3.2.3 Comparaison entre GCC et CLANG

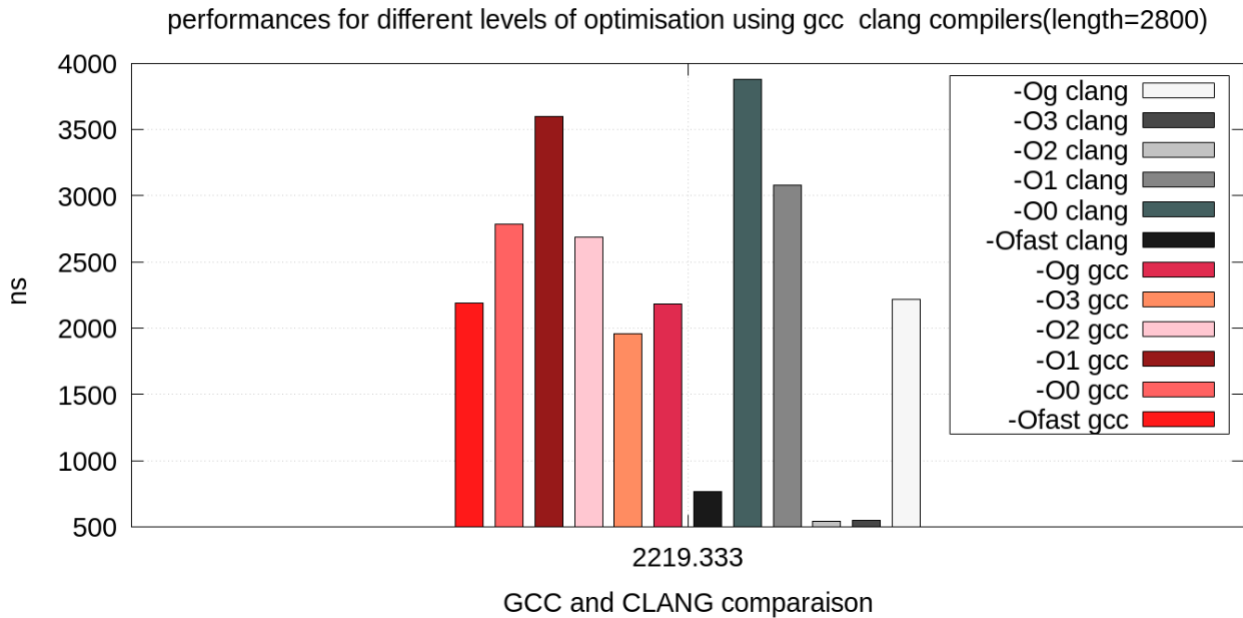


FIGURE 9 – Comparaison entre GCC et CLANG

3.2.4 Comparaison entre le code initial et le code aligné

a. Compilateur GCC

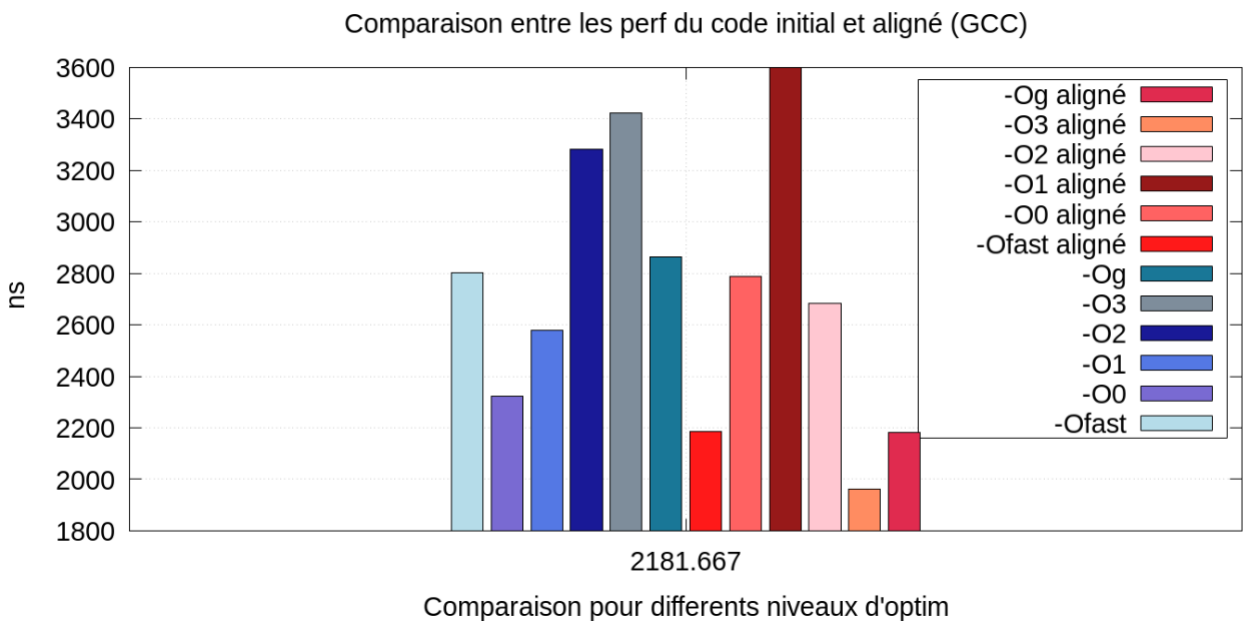


FIGURE 10 – Comparaison entre code initial et aligné pour GCC

On remarque que le code devient plus performant que le code initial.

b. Compilateur CLANG

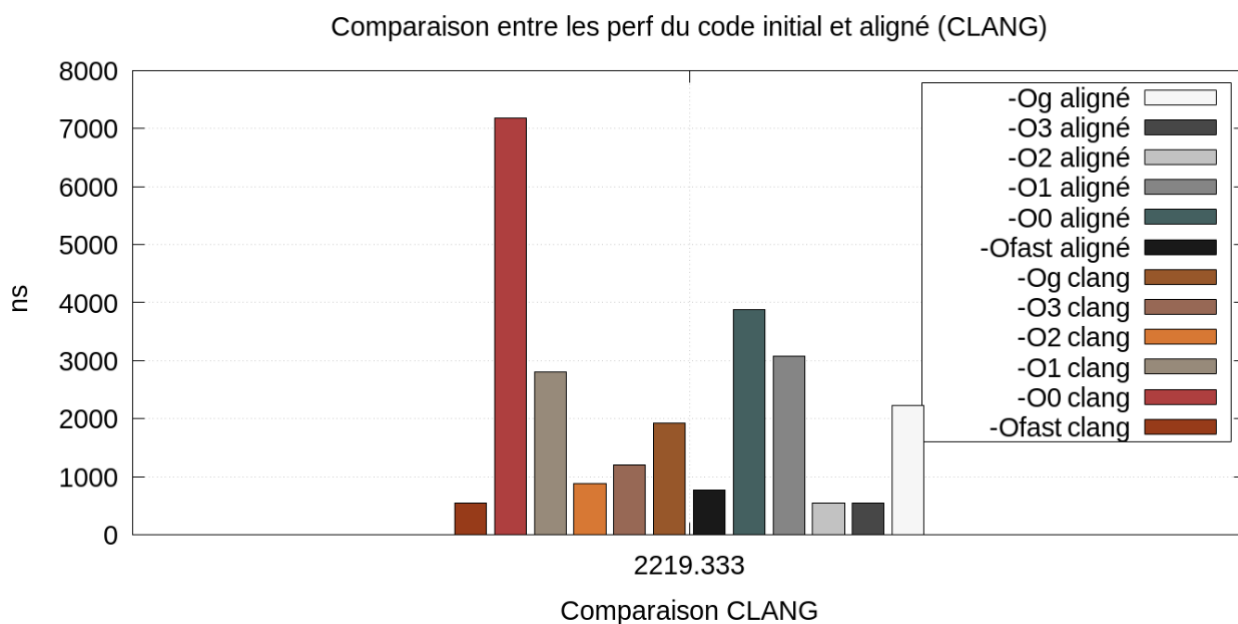


FIGURE 11 – Comparaison entre code initial et aligné pour Clang

On remarque que le code devient plus performant que le code initial pour CLANG aussi.

3.2.5 Conclusion

Pour le code aligné, le compilateur CLANG est mieux en général que GCC pour tous les niveaux d'optimisation utilisés.

3.3 Vectorisation

La vectorisation est parmi les outils d'optimisation d'un programme donné, elle consiste à traiter plusieurs données à la fois. On va utiliser la vectorisation dans ce code pour optimiser et nettoyer notre programme car il contient des calculs mathématiques (calculs de la distance de hamming à partir des matrices stochastiques ou double stochastiques), donc on peut effectuer plusieurs opérations mathématiques à la fois à l'aide des instructions vectorielles.

Pour vectoriser le code, j'ai ajouté l'option **"-ftree-vectorize"** à l'option **CFLAGS** dans le Makefile, pour que le processeur cherche dans le programmes les boucles qui peuvent être vectorisées, et donc il peut utiliser directement des instructions vectorielles pour exécuter le code et optimiser à la fois le temps de la compilation.

3.3.1 Compilateur GCC

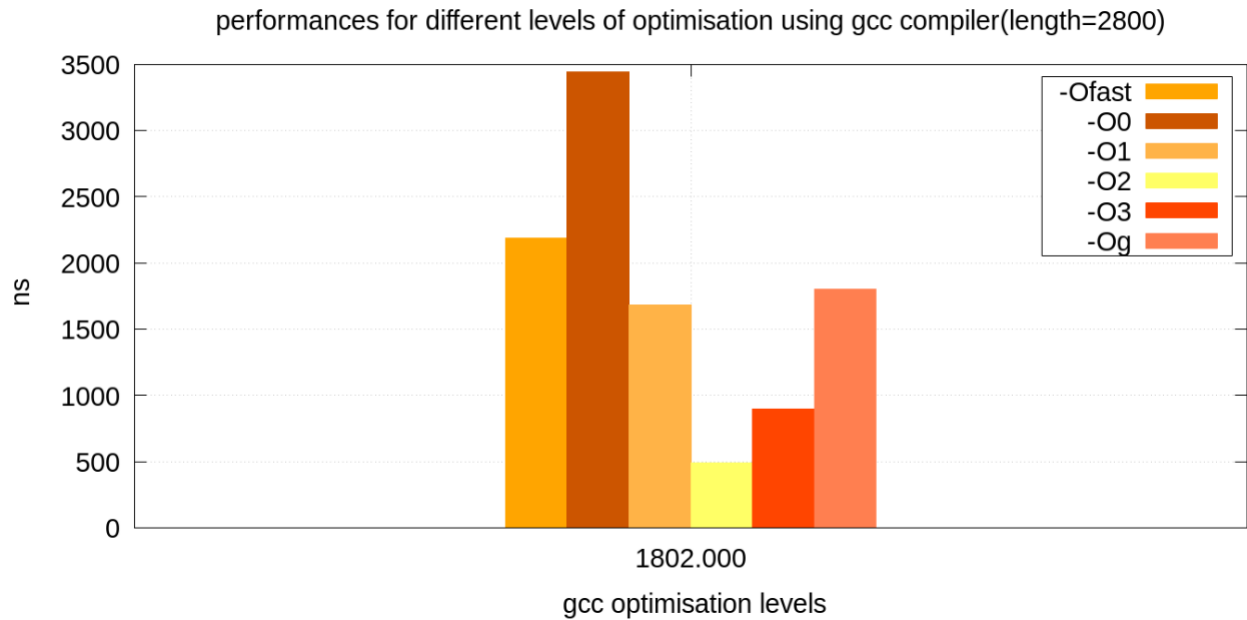


FIGURE 12 – version vectorisée du code (gcc compiler)

On remarque que le niveau d'optimisation **-O2** est le plus performant si on utilise le compilateur GCC.

3.3.2 Compilateur CLANG

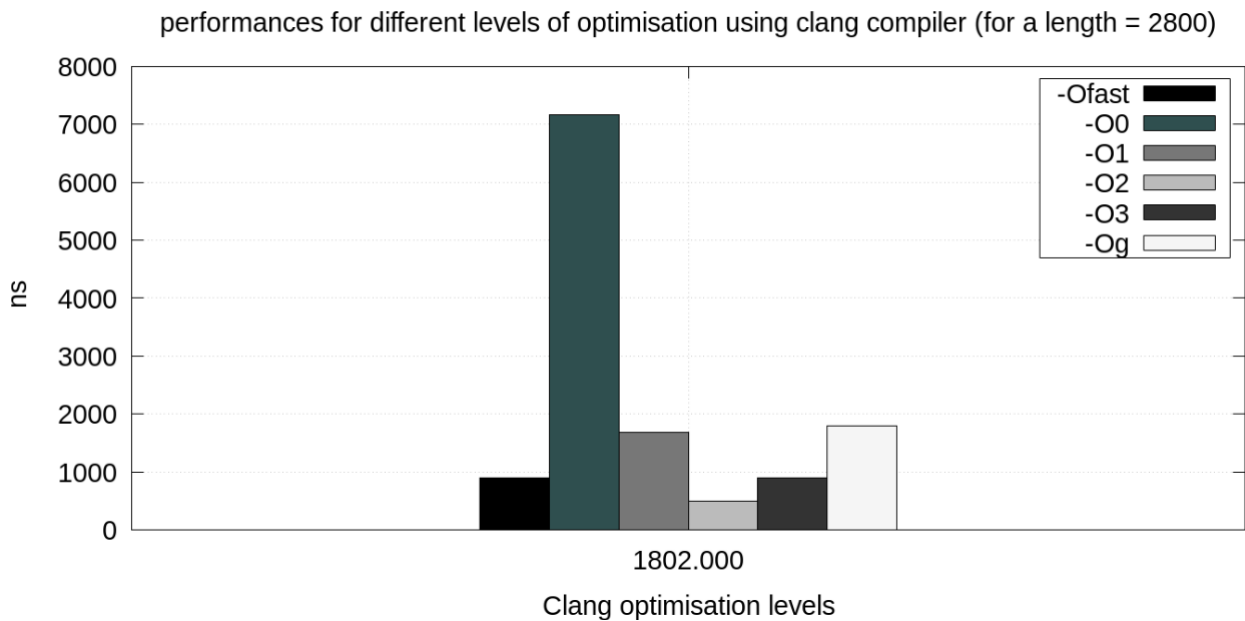


FIGURE 13 – version vectorisée du code (clang compiler)

Pour le compilateur Clang, **-O2** est le flag optimum.

3.3.3 Comparaison entre GCC et CLANG

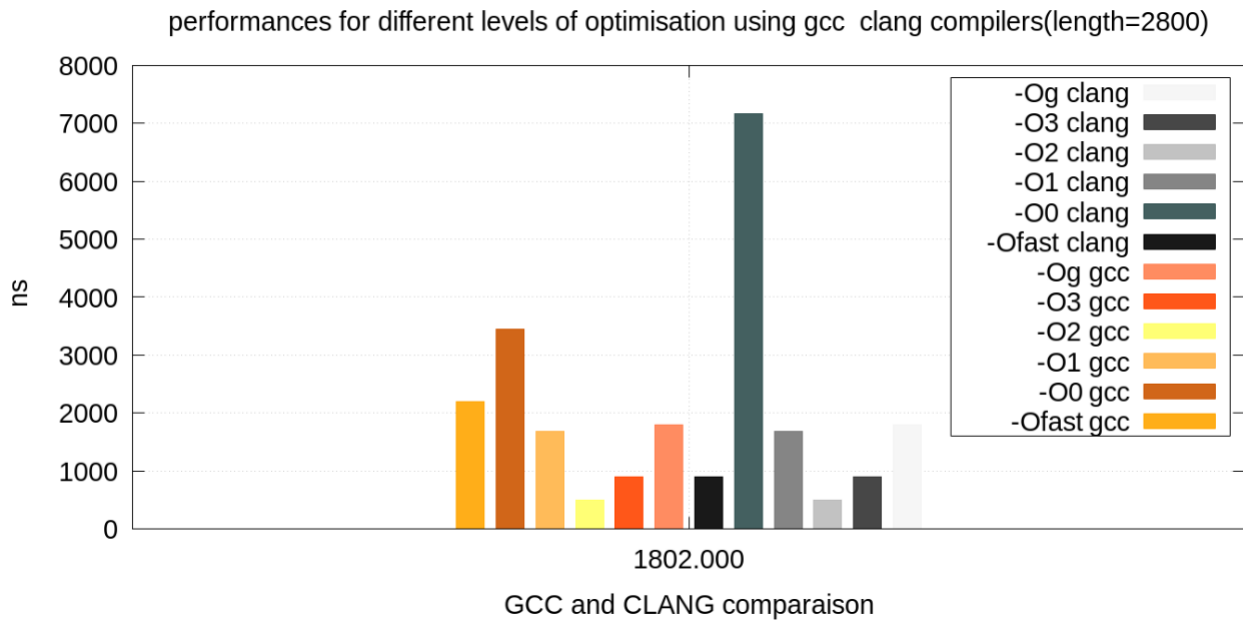


FIGURE 14 – comparaison entre GCC et CLANG

On remarque qu'il n'y a pas une grande différence entre les deux compilateurs en termes de performances en fonction du temps écoulé en (ns), sauf au niveau **-O0**.

3.3.4 Comparaison entre code initial et version vectorisée

a. Compilateur GCC

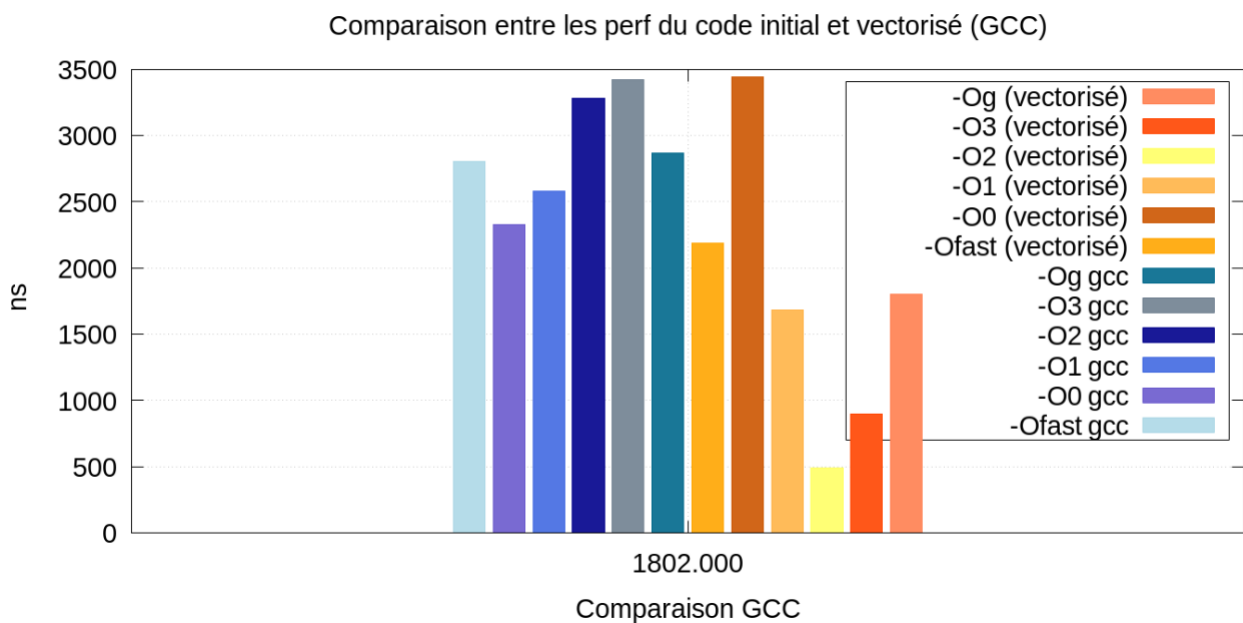


FIGURE 15 – comparaison entre code initial et vectorisé (GCC)

b. Compilateur CLANG

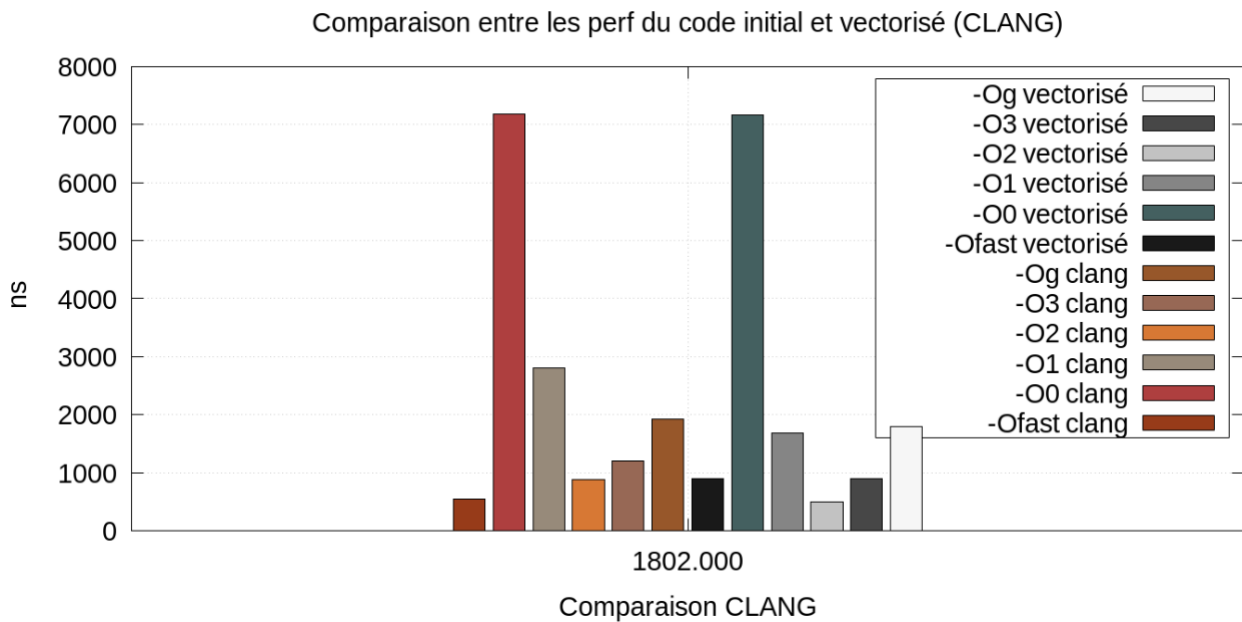


FIGURE 16 – comparaison entre code initial et vectorisé (CLANG)

3.3.5 Conclusion

On remarque qu'il n'y a pas un changement énorme entre les performances du code initial et vectorisé en utilisant le compilateur GCC. Cependant, l'exécution avec CLANG a montré un changement positif au niveau de la vitesse de compilation du code, sauf au flag **-O0** (les performances ont resté constantes dans les deux versions du code).

3.4 Loop unrolling

Le déroulage de boucle (Loop unrolling) nous permet d'exécuter toutes les itérations d'une boucle en utilisant une seule instruction qui exécute toutes les itérations à la fois. Cette technique améliore les performances de notre code.

Pour utiliser "loop unrolling", j'ai modifié le programme principal **'hamming'** dans le code **'dist.c'** et j'ai ajouté l'option **'UNROLL'** pour exécuter **a[i]** et **a[i+1]** à la fois.

Dans le **Makefile** j'ai ajouté l'option **'-funroll-loops'** dans les deux cibles **dist** et **genseq** dans l'option d'optimisation **OFLAGS**, donc lors de la compilation des codes en utilisant la commande **make**, les boucles du code seront déroulées.

3.4.1 Compilateur GCC

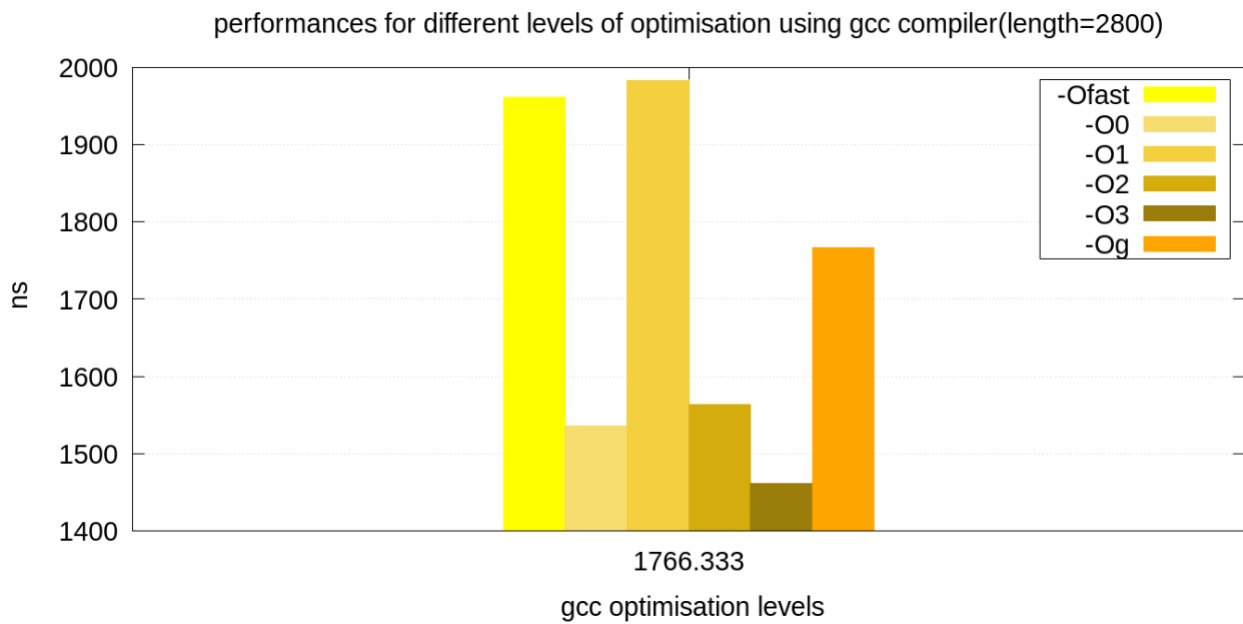


FIGURE 17 – version déroulée du code (GCC)

Le flag le plus performant est **-O3**.

3.4.2 Compilateur CLANG

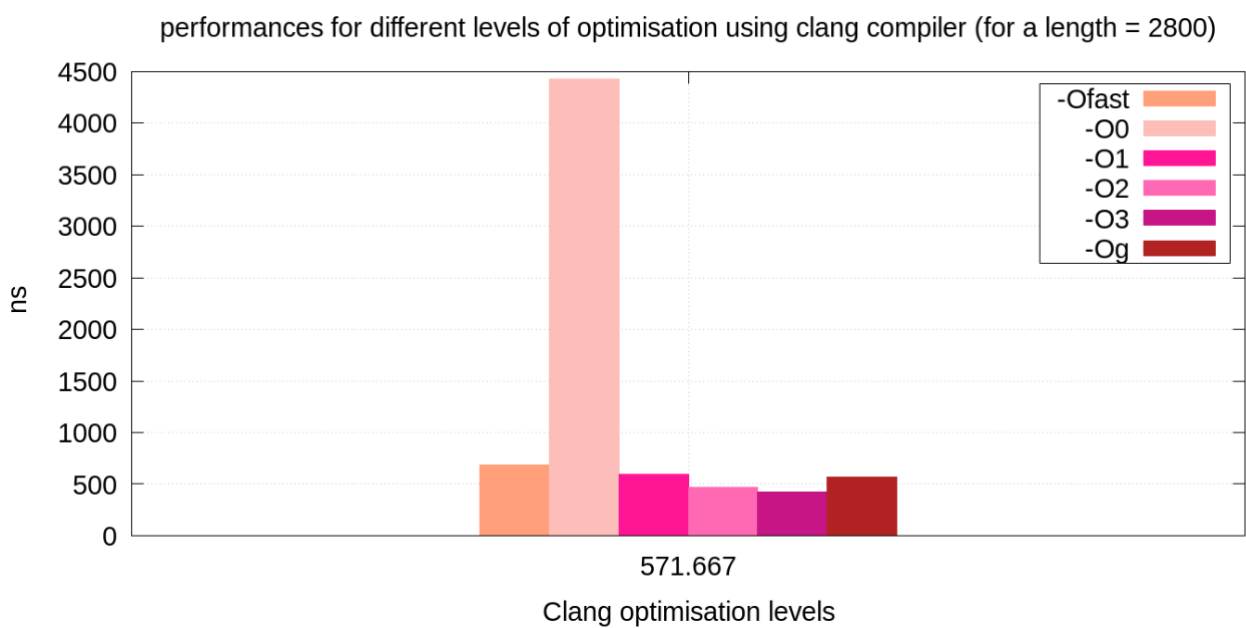


FIGURE 18 – version déroulée du code (CLANG)

3.4.3 Comparaison entre GCC et CLANG

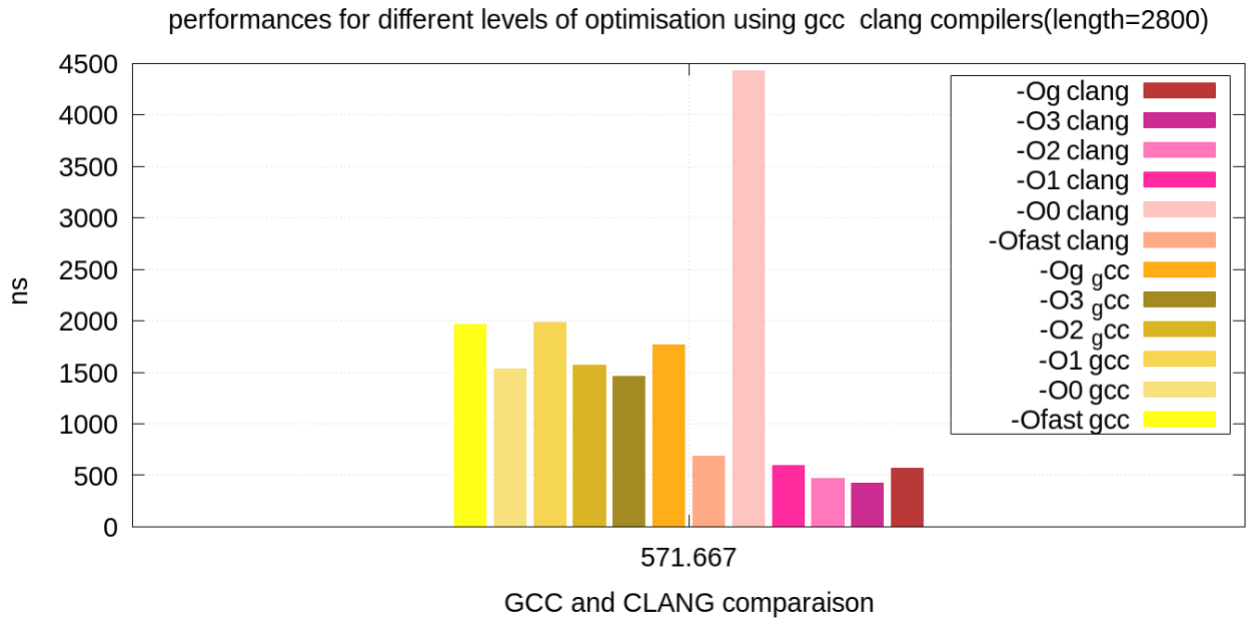


FIGURE 19 – Comparaison entre GCC et CLANG

En général, on remarque que CLANG est plus optimal que GCC.

3.4.4 Comparaison entre code initial et déroulé

a. Compilateur GCC

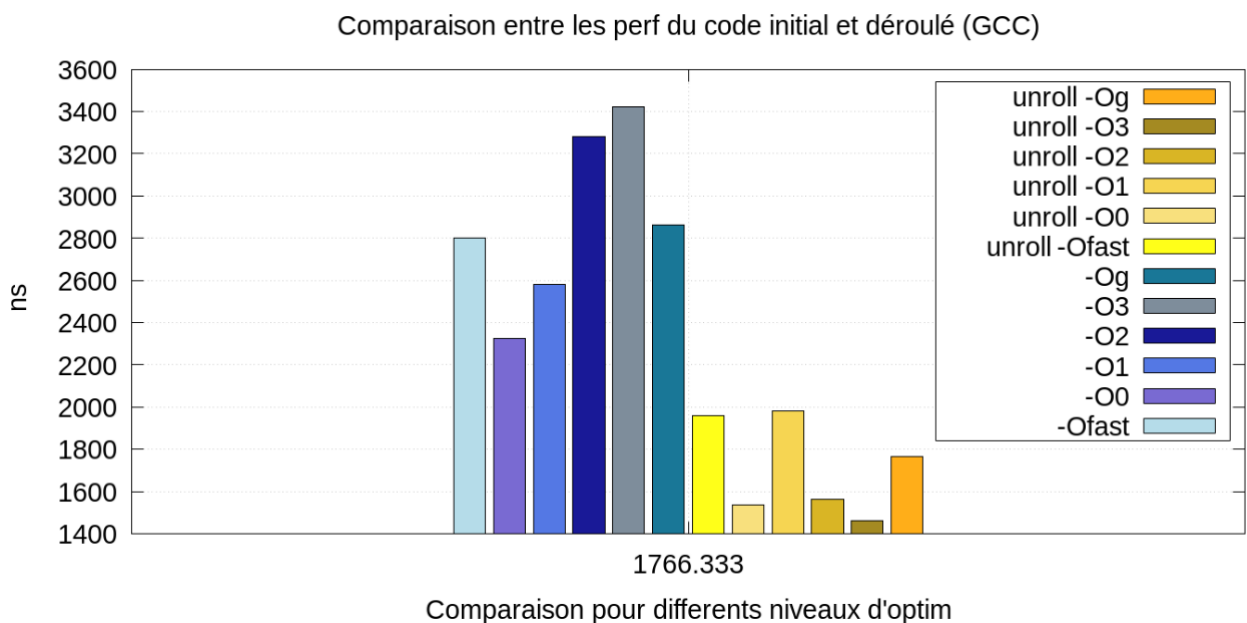


FIGURE 20 – Comparaison entre code initial et déroulé (GCC)

b. Compilateur CLANG

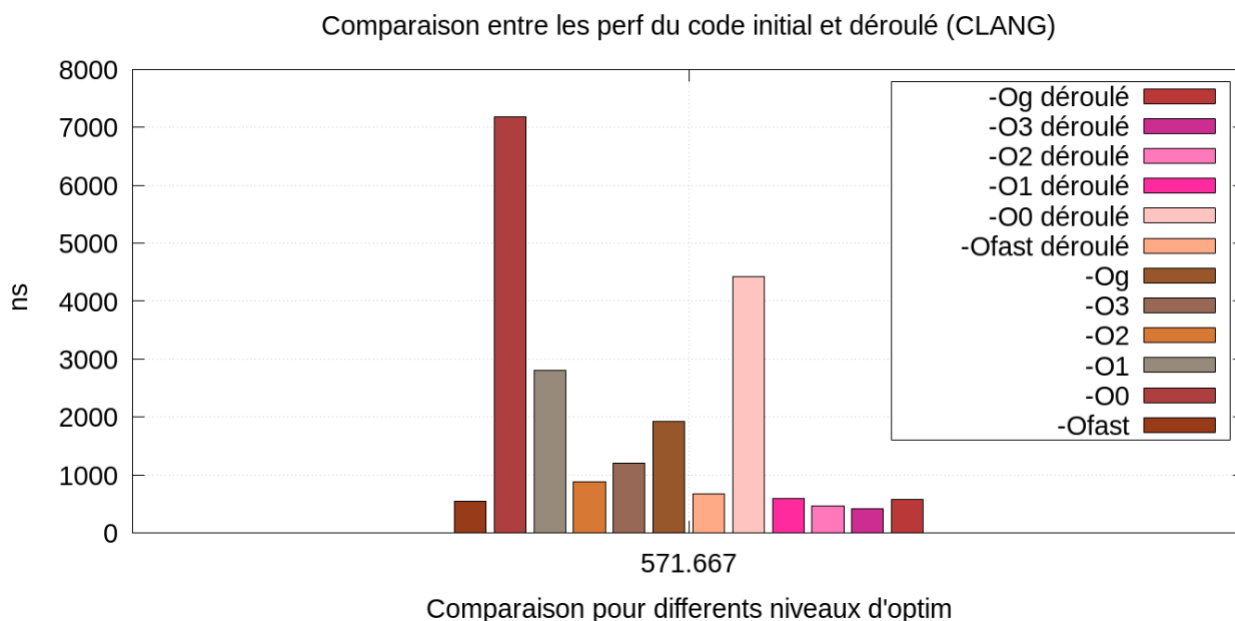


FIGURE 21 – Comparaison entre code initial et déroulé (CLANG)

3.4.5 Conclusion

D'après mes résultats, je remarque que la version déroulée du code est plus performante vu que le code contient plusieurs boucles, ainsi le compilateur CLANG est plus optimal que GCC.

3.5 Version optimisée

Pour cette version (**nom du dossier optim sur github**), j'ai ajouté l'option '**-foptimize**' dans la cible **dist** pour optimiser et accélérer l'exécution du code principal '**dist**'.

3.5.1 Compilateur GCC

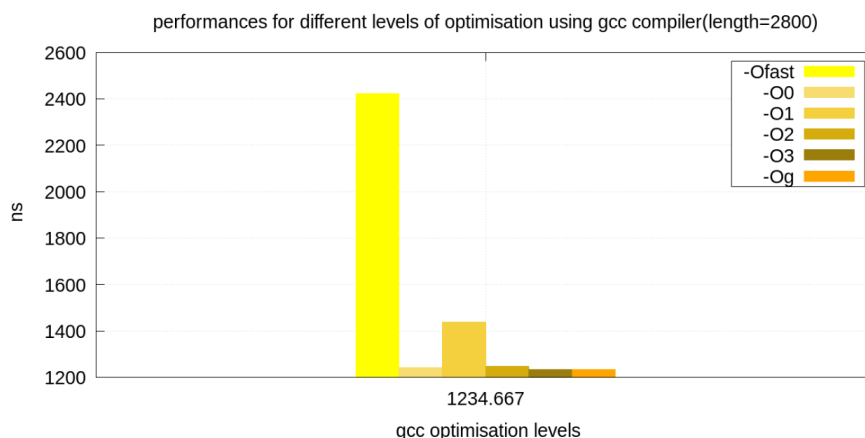


FIGURE 22 – Version optimisée du code (GCC)

On remarque que les deux flags **-O3** et **-Og** sont les niveaux les plus performants en utilisant le compilateur GCC.

3.5.2 Compilateur CLANG

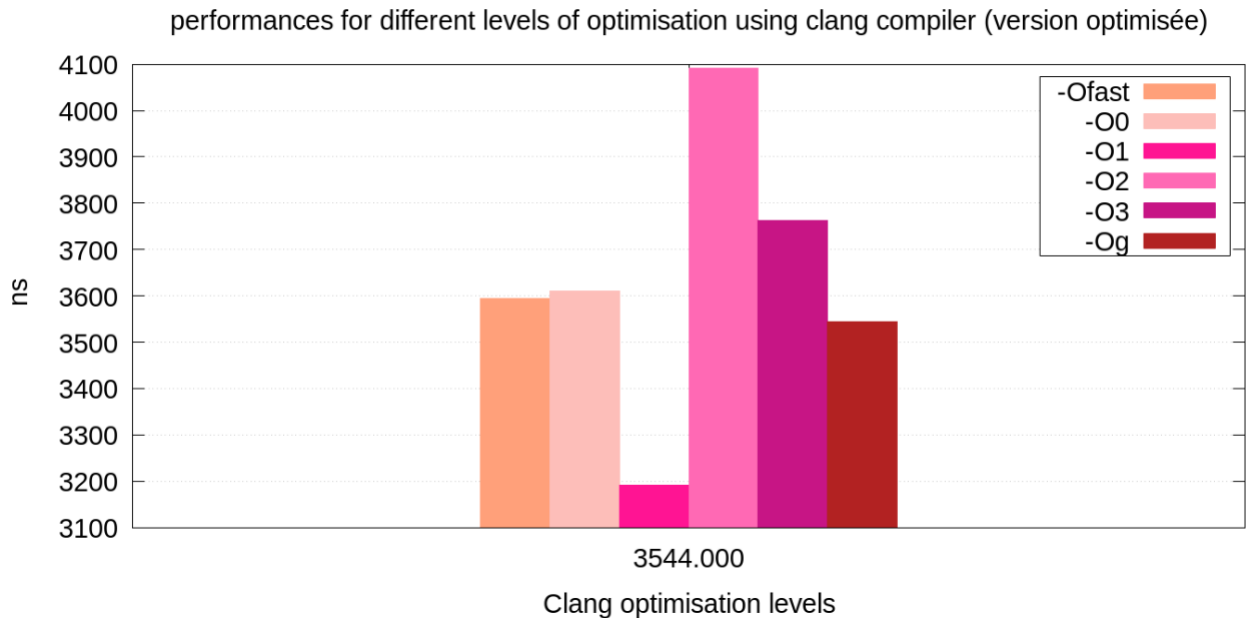


FIGURE 23 – Version optimisée du code (CLANG)

3.5.3 Comparaison entre GCC et CLANG

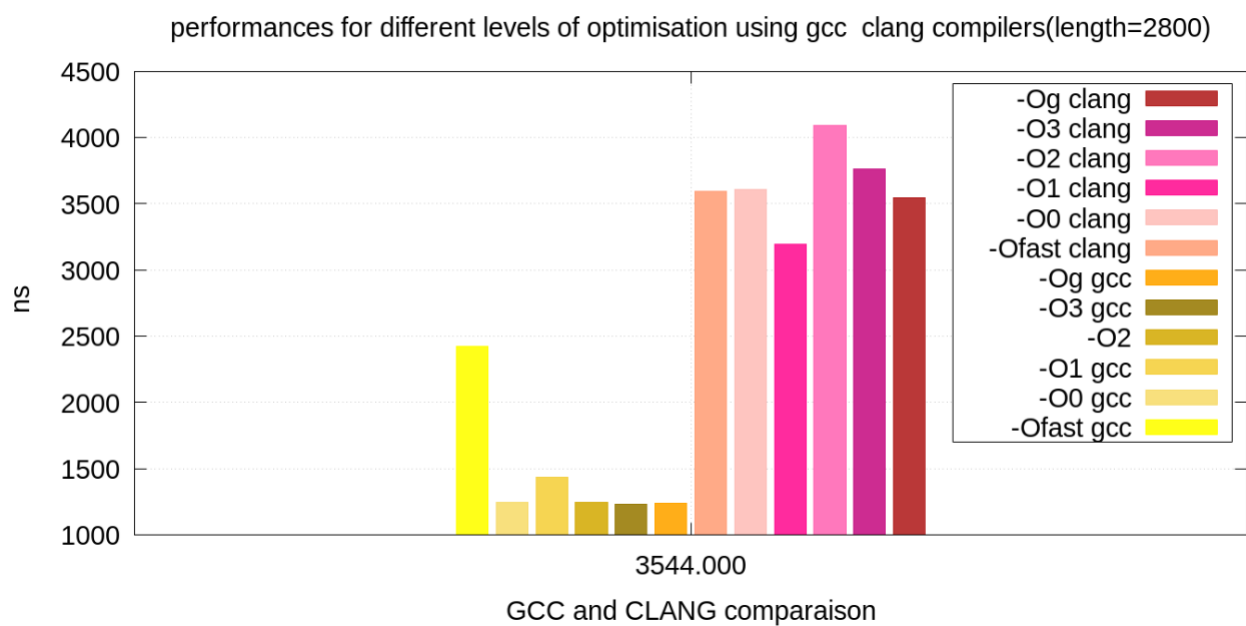


FIGURE 24 – Comparaison entre GCC et CLANG

On remarque que le compilateur GCC est plus optimal pour cette version.

3.5.4 Comparaison entre le code initial et optimisé

a. Compilateur GCC

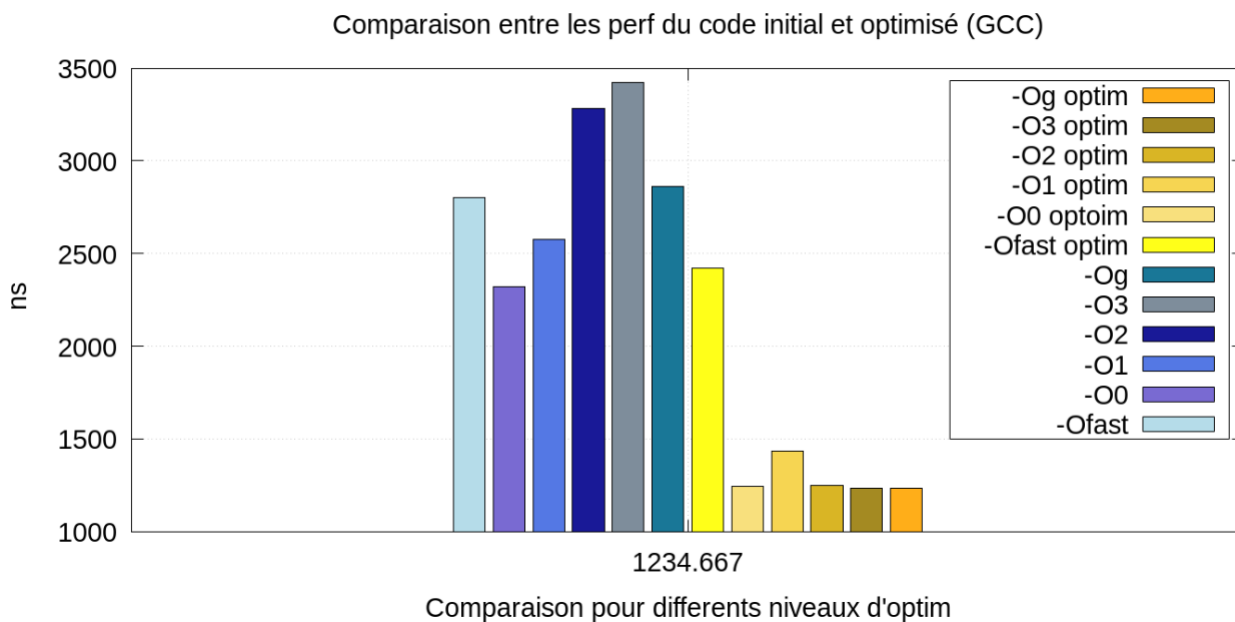


FIGURE 25 – Comparaison entre code initial et optimisé (GCC)

b. Compilateur CLANG

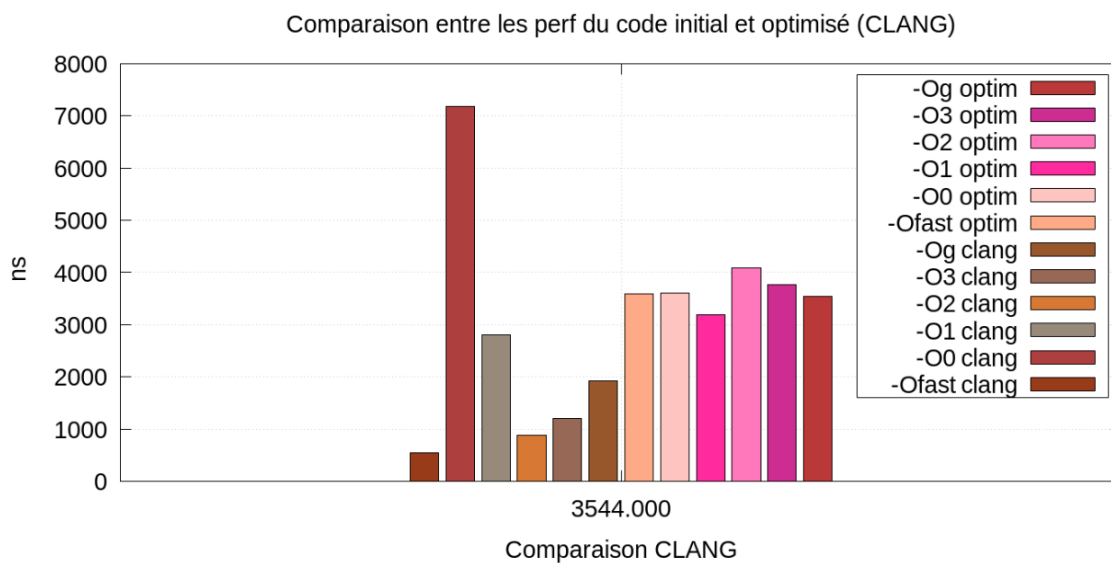


FIGURE 26 – Comparaison entre code initial et optimisé (CLANG)

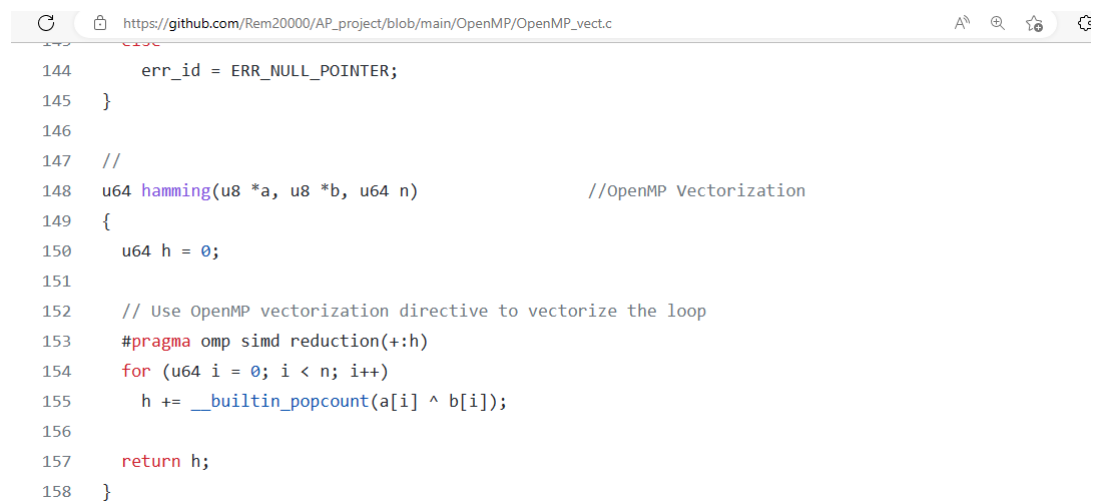
3.5.5 Conclusion

GCC est le compilateur le plus optimal en utilisant cette version du code.

3.6 Parallélisation : OpenMp

OpenMp est la bibliothèque qui permet de paralléliser le code en C. pour utiliser cette version, il faut d'abord inclure le fichier '**omp.h**' qui contient les fonctions nécessaires pour utiliser la bibliothèque OpenMp au sein de notre code.

Pour paralléliser le code principal '**dist.c**', j'ai modifié la fonction principale '**hamming**' de la manière suivante :



```
144     err_id = ERR_NULL_POINTER;
145 }
146
147 //
148 u64 hamming(u8 *a, u8 *b, u64 n)           //OpenMP Vectorization
149 {
150     u64 h = 0;
151
152     // Use OpenMP vectorization directive to vectorize the loop
153     #pragma omp simd reduction(+:h)
154     for (u64 i = 0; i < n; i++)
155         h += __builtin_popcount(a[i] ^ b[i]);
156
157     return h;
158 }
```

FIGURE 27 – fonction hamming parallélisée

J'ai utilisé la directive :

' #pragma omp simd reduction(+ :h)'

C'est une extension de '**OpenMp**' pour vectoriser les boucles du programme en utilisant les instructions **SIMD**, et exécuter en parallèle les itérations sur plusieurs processeurs (ou CPU) de la machine.

L'option ajoutée "**reduction (+ : h)**" indique au compilateur d'additionner les résultats de toutes les itérations de la boucle 'for' pour obtenir la valeur finale de la distance de hamming, notée '**h**' dans le code.

Dans le **Makefile** j'ai ajouté l'option '**gomp**' dans les deux cibles (dist et genseq) liés à l'exécution des codes '**OpenMP_vect.c**' et '**genseq.c**' pour bien paralléliser le programme à l'aide de la bibliothèque OpenMp.

3.6.1 Compilateur GCC

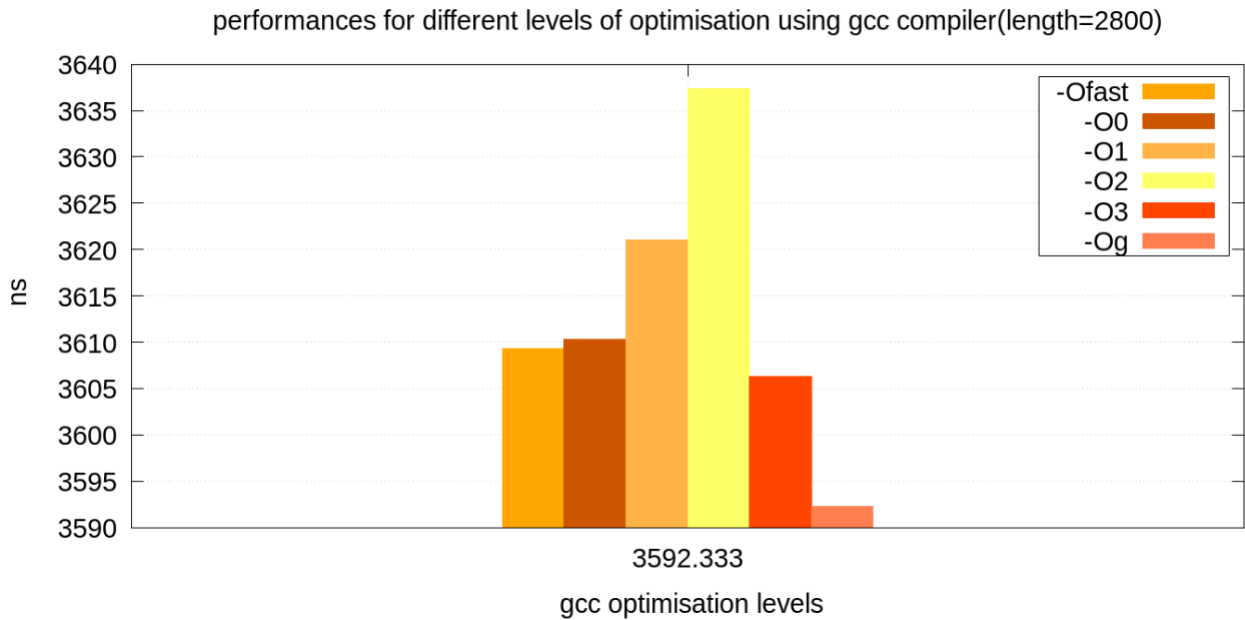


FIGURE 28 – Version parallélisée (GCC)

Le flag **-Og** est le plus performant.

3.6.2 Compilateur CLANG

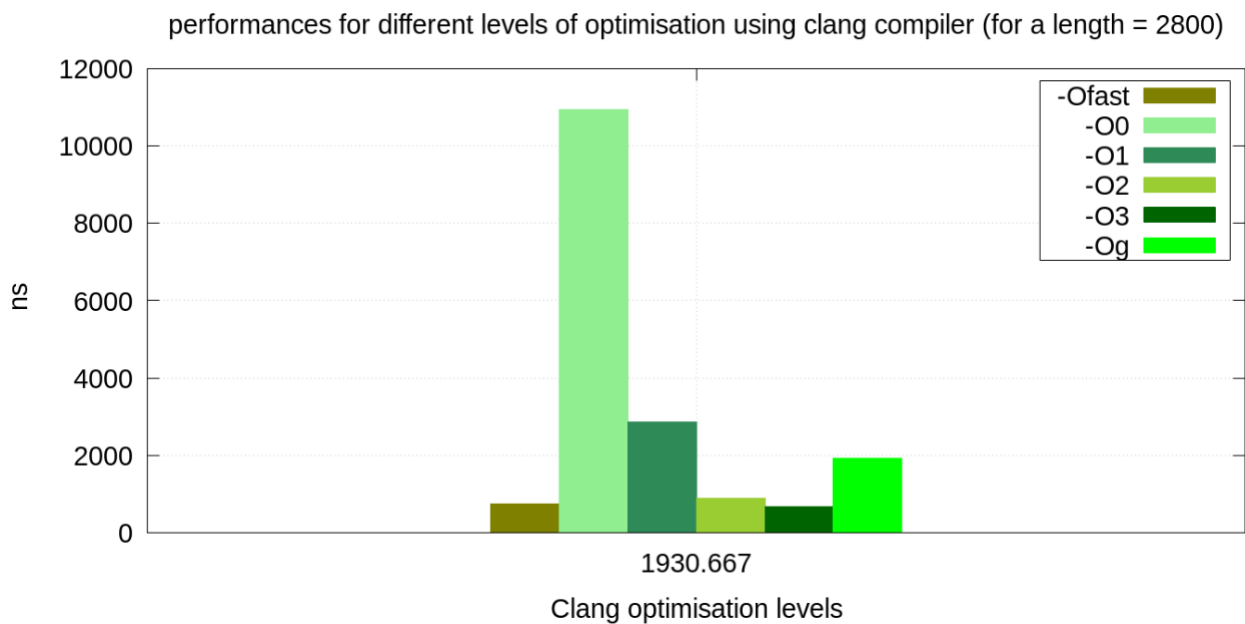


FIGURE 29 – Version parallélisée (CLANG)

Le flag **-O3** est le plus performant.

3.6.3 Comparaison entre GCC et CLANG

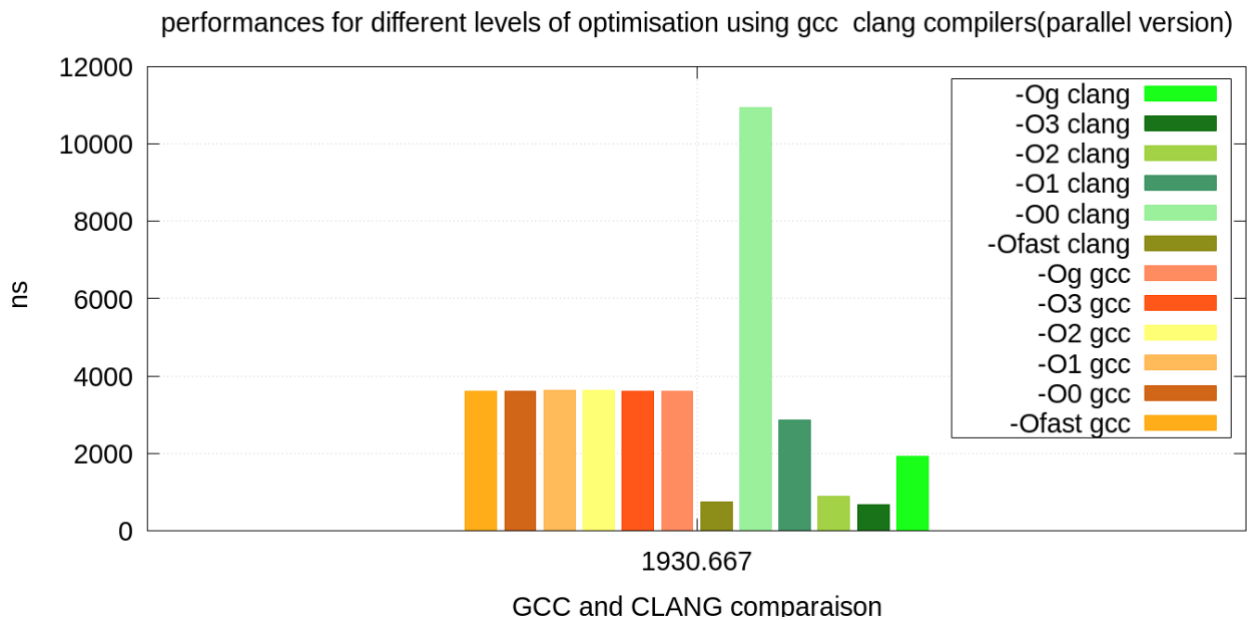


FIGURE 30 – Comparaison entre GCC et CLANG

On remarque que les performances des flags de GCC sont proches, ainsi que le compilateur qui me parait plus optimal est CLANG.

3.6.4 Comparaison entre le code initial et parallélisé

a. Compilateur GCC

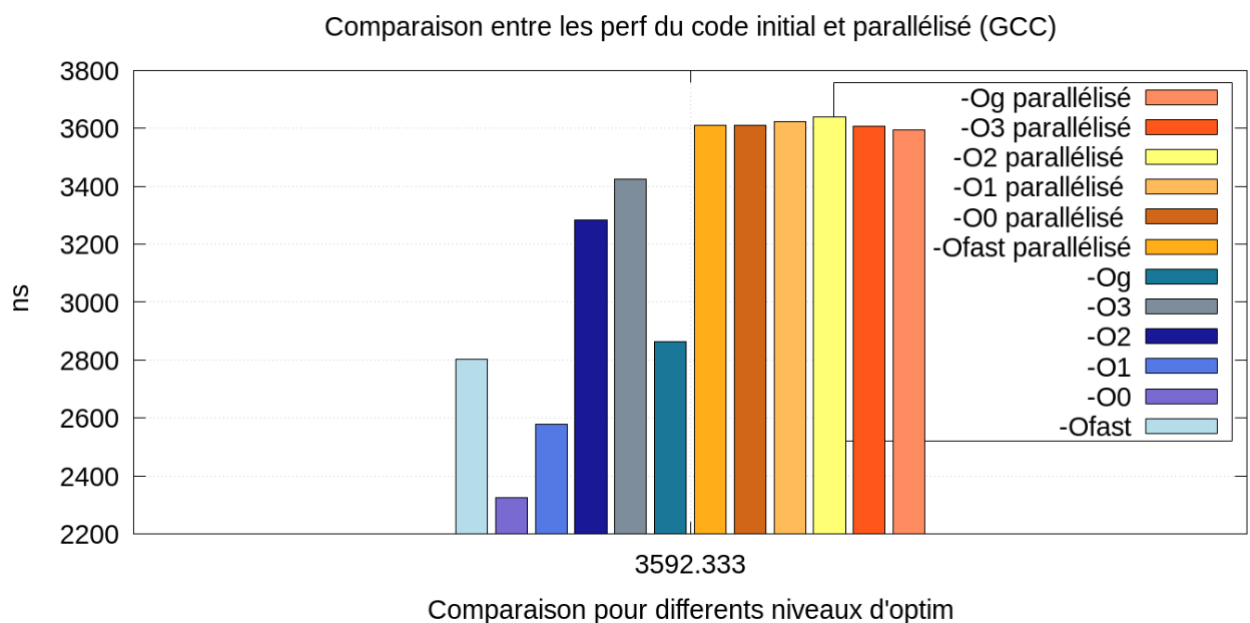


FIGURE 31 – Comparaison entre le code initial et parallélisé (GCC)

b. Compilateur CLANG

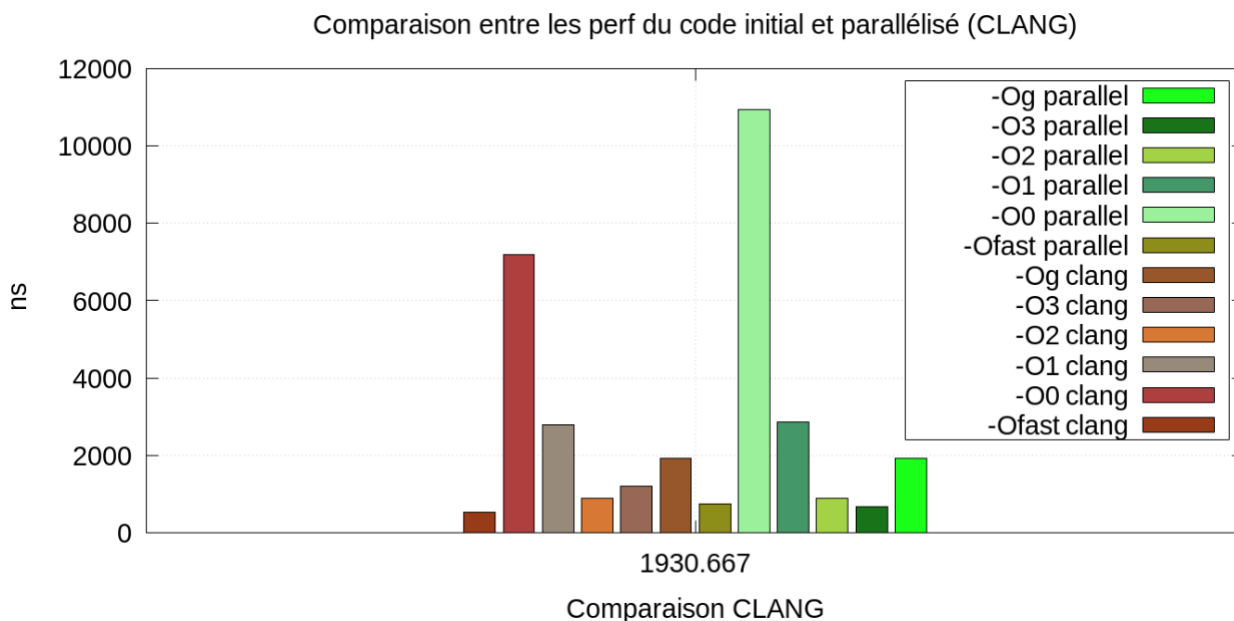


FIGURE 32 – Comparaison entre le code initial et parallélisé (CLANG)

3.7 Optimisation générale

Dans cette version du code, j'ai utilisé toutes les options précédente dans le **Makefile**, donc le code sera aligné, déroulé, vectorisé et parallélisé.

Ci-dessous les performances du code en utilisant les compilateurs GCC et CLANG, ainsi qu'une comparaison entre le code initial et final.

Le dossier '**my_optimum**'

3.7.1 Performances du code (GCC et CLANG)

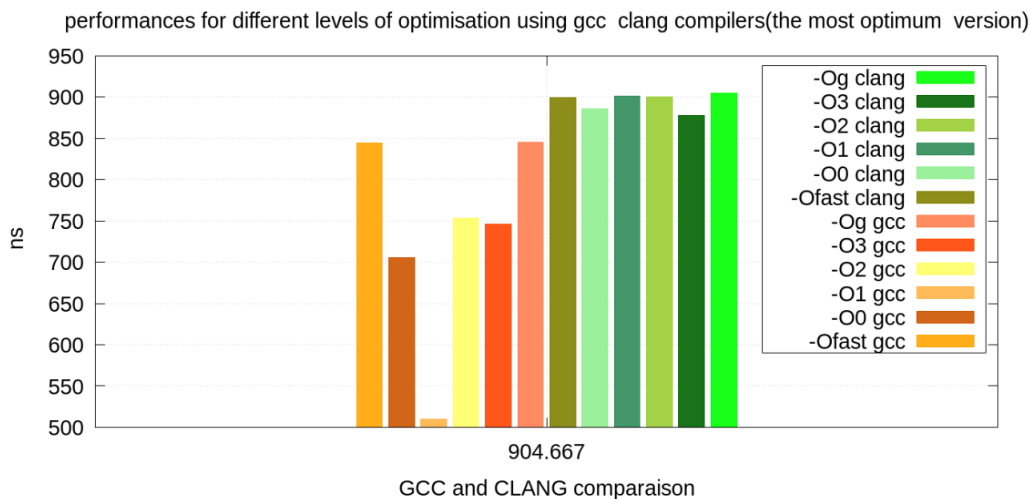


FIGURE 33 – Version finale du code (GCC et CLANG)

Le flag **-O0** est le plus optimal en utilisant le compilateur GCC, et **-O3** est le plus performant en utilisant CLANG.

3.7.2 Comparaison entre le code initial et final (GCC)

a. Compilateur GCC

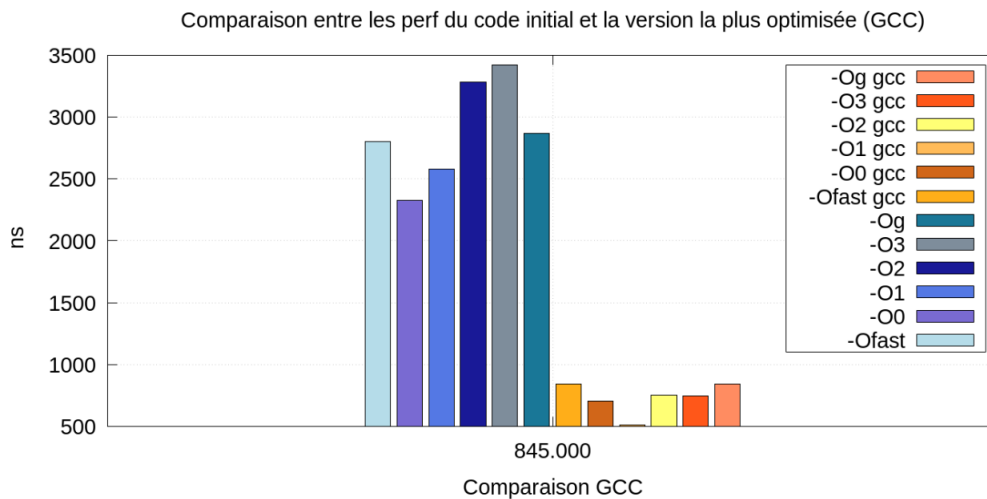


FIGURE 34 – Comparaison entre le code initial et final (GCC)

b. Compilateur CLANG

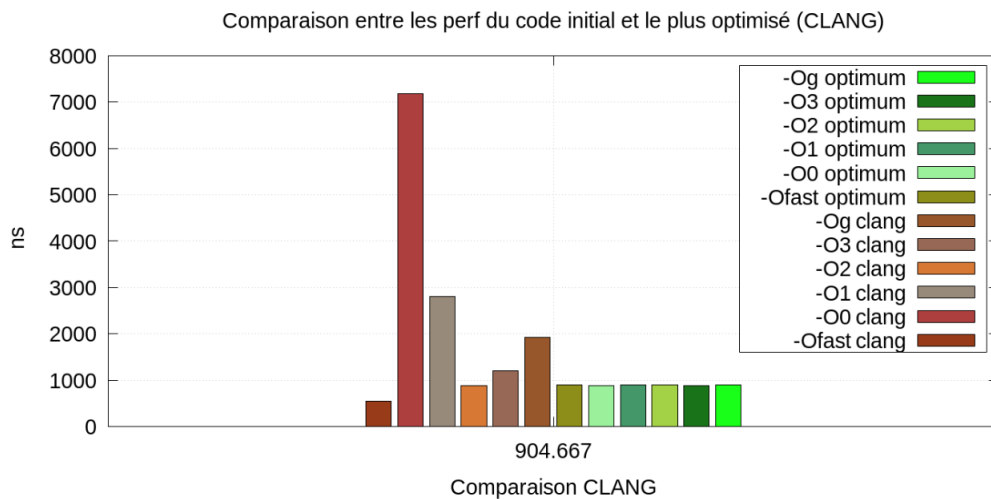


FIGURE 35 – Comparaison entre le code initial et final (CLANG)

3.7.3 Conclusion

La version finale est bien plus performante en terme du temps de l'exécution du code entier en (ns), donc les différentes options utilisées sont très importantes pour optimiser le code en général.