UVSQ université PARIS-SACLAY

ISTY
Institut des Sciences et Techniques des Yvelines
CAMPUS DE MANTES EN YVELINES
CAMPUS DE SAINT-QUENTIN-EN-YVELINES

IATIC-4

AOB
Architecture des Ordinateurs - Bases
Rapport de Projet

Analyse des performances et Optimisation d'une Simulation 3D de Nbody

---

Réalisé Par :
Rochdi DARDOR

Encadré par :
M.William Jalby
M.Hugo Bolloré
M.Mohammed Ibnamar

Table des matières

1 Introduction

1.1 Contexte et objectifs du projet

L'optimisation des performances des programmes informatiques constitue un défi fondamental pour garantir l'efficacité des applications dans divers domaines scientifiques. Dans cette étude, nous nous sommes concentrés sur l'amélioration des performances d'un programme de simulation de particules, un domaine crucial pour la recherche scientifique, l'ingénierie et les applications industrielles.

Le but de cette étude était d'explorer et d'appliquer une série de techniques d'optimisation afin de réduire les temps d'exécution du programme de simulation de particules tout en maximisant l'utilisation des ressources matérielles disponibles. Notre approche s'est articulée autour de plusieurs stratégies d'optimisation, allant de la restructuration des données à l'utilisation de directives de compilation avancées et à la parallélisation du code.

Ce rapport présente une analyse rigoureuse des différentes étapes entreprises pour optimiser le programme, débutant par une évaluation détaillée de l'architecture matérielle sous-jacente et des spécifications techniques de la plateforme de test. Nous avons ensuite identifié les sections critiques du programme, analysé leur complexité et évalué l'impact potentiel de leur optimisation sur les performances globales.

Chaque stratégie d'optimisation a été soigneusement étudiée et mise en œuvre, accompagnée d'une analyse comparative de ses effets sur les performances du programme. Nous avons consigné les résultats obtenus, les limitations rencontrées et les recommandations découlant de ces analyses pour guider les futures améliorations des performances dans des contextes similaires.

Ce rapport constitue un compte rendu détaillé de notre méthodologie, des résultats obtenus et des implications pratiques de nos travaux en matière d'optimisation des performances des programmes de simulation de particules.

1.2 Structure du rapport

Le rapport est divisé en plusieurs sections pour présenter de manière détaillée les différentes étapes entreprises dans le processus d'optimisation du programme. Chaque section aborde une méthode spécifique d'optimisation, en mettant en évidence ses objectifs, ses implications et ses résultats.

2 Environnement et Configuration

2.1 Description de l'architecture cible

Le programme a été exécuté sur un HP EliteBook 840 G3, utilisant Ubuntu 22.04.2 LTS comme système d'exploitation avec un noyau Linux 5.15.0-91-generic. L'architecture est x86-64.

2.2 Informations sur les compilateurs et leurs versions

Trois compilateurs ont été utilisés pour ce projet :

— GCC : Version 11.4.0, faisant partie de l'environnement Ubuntu 22.04.2 LTS (Ubuntu 11.4.0-1ubuntu1 22.04).

— Clang : Version 14.0.0-1ubuntu1.1, fournissant un compilateur C compatible avec LLVM.

— Intel(R) oneAPI DPC++/C++ Compiler : Version 2024.0.2 (2024.0.2.20231213), spécialement dédié à la programmation parallèle pour les architectures Intel.

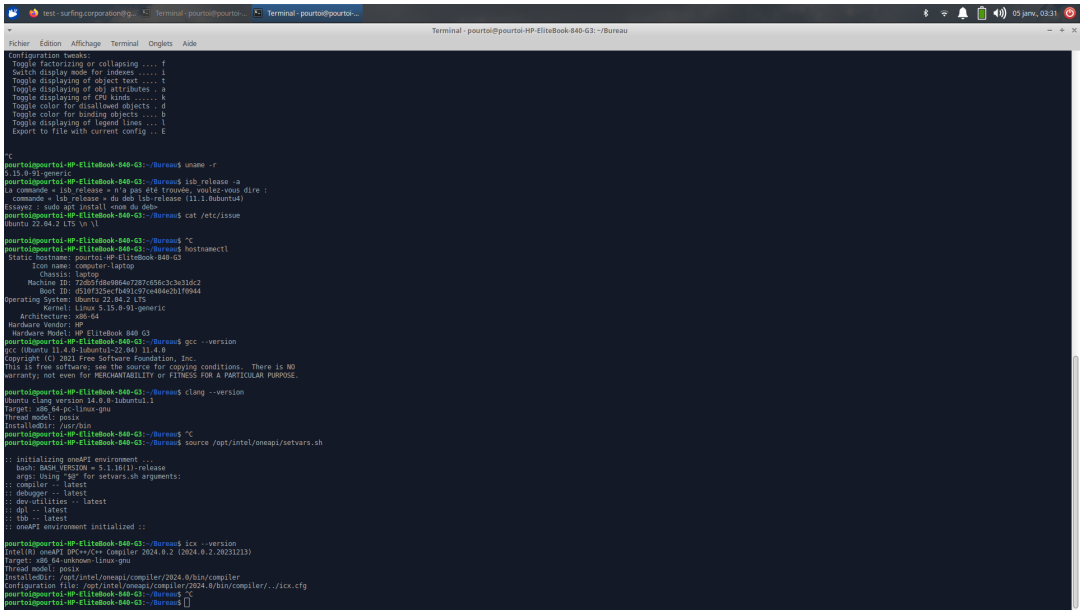

FIGURE 1 – version d'OS et compilateur

2.3 Détails sur le processeur et le cache

Le système est équipé d'un processeur Intel Core i7-6600U. Les spécifications du processeur incluent une architecture x86-64, avec une configuration comprenant 4 cœurs et 8 threads. Chaque cœur a une taille de cache L1d de 64 KiB (2 instances), L1i de 64 KiB (2 instances), L2 de 512 KiB (2 instances), et L3 de 4 MiB (1 instance).

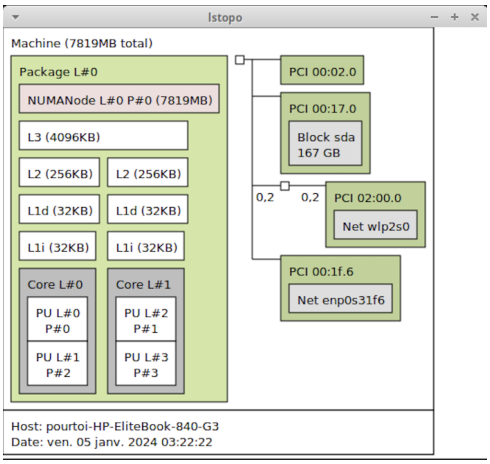

FIGURE 2 – lstopo command

2.4 Fréquences et gouvernance du CPU

Le processeur, un Intel(R) Core(TM) i7-6600U propose plusieurs fonctionnalités avancées telles que SSE2,AVX, AES, et bien d'autres. il opère dans une plage de fréquences allant de 400 MHz à 3,40 GHz. Initialement, la gouvernance des fréquences est réglée sur le mode "powersave". Cependant, lors des tests de performance, cette configuration sera ajustée vers le mode "performance" pour fixer la fréquence à son niveau maximal de 3,40 GHz. De plus, le programme sera assigné et exécuté sur un seul cœur spécifique (le cœur numéro 2) à l'aide de la commande taskset -c 2 ./nbody3D. Cette démarche ciblée vise à évaluer précisément les performances du programme dans un environnement contrôlé.

FIGURE 3 – lspu

2.5 Autres détails système

La machine est un ordinateur portable, modèle HP EliteBook 840 G3, fabriqué par HP. L'architecture est x86-64, et la configuration du processeur est un Intel(R) Core(TM) i7-6600U CPU @ 2.60GHz. De plus, le compilateur Intel oneAPI DPC++/C++ Compiler 2024.0.2 offre des fonctionnalités optimisées pour les architectures Intel, notamment pour la programmation parallèle.

Ces détails offrent une vue approfondie de l'environnement matériel et logiciel dans lequel le programme a été exécuté, intégrant les spécifications du processeur, les détails du cache et des informations sur la gouvernance des fréquences pour une compréhension plus complète du système.

3 Analyse du Programme

3.1 Vue d'ensemble du code

La simulation de particules est fondamentale dans de nombreuses disciplines scientifiques, exigeant une puissance de calcul considérable. Ce code spécifique simule le mouvement de particules sous l'influence des forces gravitationnelles.

Le programme Nbody simule le mouvement de différentes particules dans un espace tridimensionnel, où chaque particule interagit avec toutes les autres, modifiant ainsi leurs positions et vitesses sur les axes x, y et z. Cette fonction effectue divers calculs pour ajuster la position et la vitesse de chaque particule. Chaque cycle itératif, de 0 à n, correspondant au nombre de particules, soustrait les positions x, y et z de chaque particule, déterminant ainsi la force exercée par chaque particule sur les autres et leurs variations de vitesse.

3.2 Structure de Données :

Le programme utilise une structure de données nommée particles-t pour stocker les positions et vitesses des particules sur chaque axe (x, y, z). La fonction init-particles initialise aléatoirement ces valeurs pour chaque particule.

```
typedef struct particle_s {
  f32 x, y, z;
  f32 vx,vy,vz;
} particle_t;
```

}

3.4 Complexité algorithmique :

Le fonction présente une complexité quadratique de :

$$O(n^2 + n)$$

en raison de la double boucle parcourant l'ensemble des particules pour calculer les interactions. Cette complexité peut être un point critique lorsque le nombre de particules est important, entraînant une augmentation significative du temps de calcul.

3.5 Organisation Mémoire et Gestion des Données :

Le programme Nbody adopte une approche spécifique dans l'organisation de ses données en utilisant un tableau de structures (AOS - Array of Structures) pour stocker les informations relatives aux particules. La structure `particles_t` comprend des tableaux distincts pour les positions (x, y, z) et les vitesses (vx, vy, vz) de chaque particule.

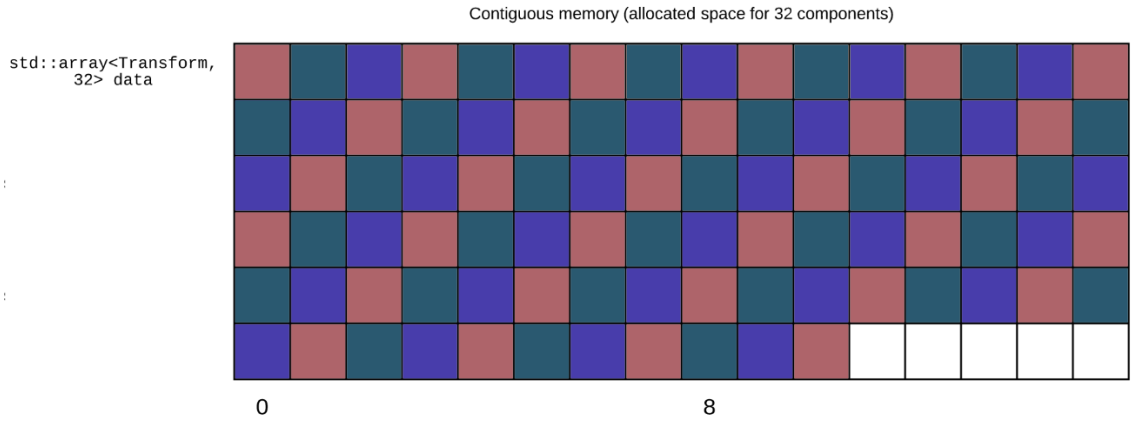

FIGURE 4 – Approche AOS

Cette organisation des données, bien que facilitant l'accès aux attributs spécifiques de chaque particule, induit des implications significatives en termes d'accès à la mémoire. Lorsque la fonction `move_particles` effectue des calculs pour évaluer les interactions entre les particules, les lectures séparées des positions et des vitesses de chaque particule entraînent des accès mémoire non contigus.

Ces accès discontinus peuvent générer une augmentation de la latence d'accès, impactant ainsi les performances globales du programme.

L'organisation en AOS peut engendrer une fragmentation de la mémoire, influencer de manière significative le fonctionnement du cache et augmenter les défauts de cache. Cette fragmentation peut impacter négativement l'efficacité du cache et compliquer la gestion des données, particulièrement lors des opérations d'allocation de mémoire. Cette

approche, bien que fonctionnelle, limite la capacité du compilateur à optimiser le code de manière efficace. Des optimisations potentielles, visant aussi à réduire le nombre de boucles, sont à envisager pour améliorer les performances globales du programme. La fonction de calcul comporte plusieurs possibilités d'amélioration afin d'éviter les fonctions qui utilisent trop de boucles, et nous étudierons ces améliorations un peu par la suite.

3.6 État initiale :

Évaluation des Performances avec Différents Compilateurs :
Les performances du programme nbody3D ont été évaluées lors de son exécution avec trois compilateurs différents : icx, gcc et clang. Chaque exécution a été mesurée en termes de temps, d'interactions par seconde et de GFLOP/s (Floating Point Operations Per Second).

— **Compilateur icx :** Lors de l'exécution avec le compilateur icx, le programme a démontré des performances stables après une période d'échauffement ("warm up"). Le temps d'exécution moyen par itération était de 0.925 secondes, atteignant une moyenne de 2.9e+08 interactions par seconde, avec une performance moyenne de 5.0 GFLOP/s.

— **Compilateur gcc :** En utilisant le compilateur gcc, les performances étaient légèrement inférieures par rapport à icx. Le temps d'exécution moyen par itération était de 7.478 secondes, avec une moyenne de 3.59e+07 interactions par seconde, correspondant à une performance moyenne de 0.6 GFLOP/s.

— **Compilateur clang :** Avec le compilateur clang, les performances se sont maintenues proches de celles obtenues avec gcc. Le temps d'exécution moyen par itération était de 8.001 secondes, avec une moyenne de 3.355e+07 interactions par seconde, atteignant une performance moyenne de 0.6 GFLOP/s.

Comparant les performances des compilateurs icx, gcc et clang sur le programme nbody3D, on remarque des résultats distincts. icx se distingue avec un temps moyen d'exécution de 0.925 secondes par itération, générant en moyenne 5.0 GFLOP/s. En revanche, gcc et clang ont montré des performances plus modestes, avec des temps d'exécution moyen respectifs de 7.478 et 8.001 secondes par itération, générant environ 0.6 GFLOP/s. Cette différence est attribuée aux optimisations plus agressives appliquées par le compilateur icx.

4 Optimisations

4.1 Optimisation 1 | Restructuration des Données en Structure de Tableaux (SOA) :

4.1.1 Objectif et justification :

Le programme utilise actuellement un tableau de structures, ce qui n'optimise pas l'organisation des données en mémoire. Les positions et vitesses ne sont pas stockées de manière contiguë, ce qui entrave la capacité du compilateur à effectuer la vectorisation automatiquement. Pour résoudre ce problème et exploiter la vectorisation, nous avons opté pour l'approche Structure of Arrays (SOA). Cette méthode consiste à regrouper les positions et vitesses dans des tableaux distincts au sein d'une unique structure, garantissant ainsi que les données similaires soient alignées en mémoire.

4.1.2 Méthodologie :

Pour remédier à ce problème, nous adoptons l'approche Structure of Arrays (SOA). Cette méthode consiste à regrouper les données similaires ensemble, en remplaçant le tableau de structures par une structure unique composée de tableaux distincts pour chaque type de données (positions et vitesses) :

```
typedef struct particle_s {
  f32 *x;
  f32 *y;
  f32 *z;
  f32 *vx;
  f32 *vy;
  f32 *vz;
} particle_t;
```

\\

Cette nouvelle structure regroupe les positions (x, y, z) et les vitesses (vx, vy, vz) dans des tableaux distincts, alignant ainsi les données de même type en mémoire de manière continue. Cette organisation améliore la possibilité d'exploiter les instructions vectorielles du processeur, permettant ainsi des opérations simultanées sur plusieurs données (vectorisation).

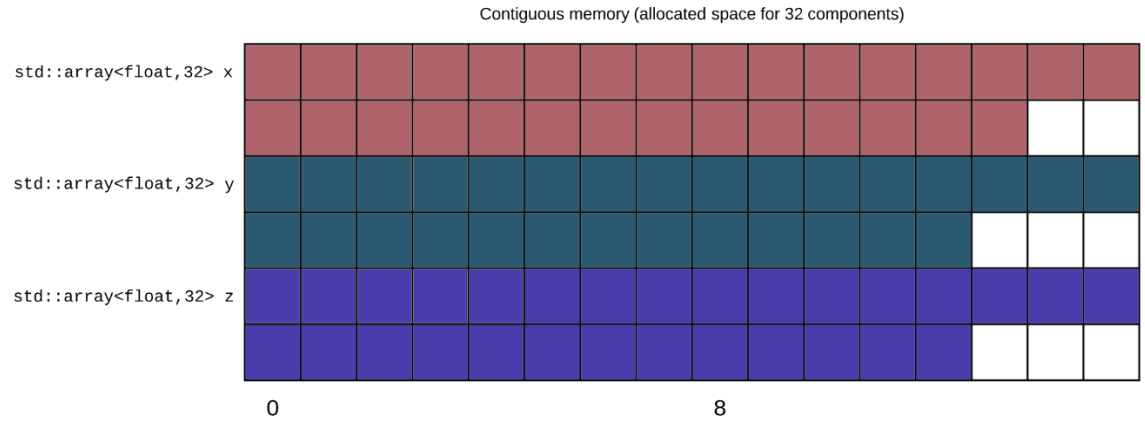

FIGURE 5 – Approche SOA

4.1.3 Résultats et Analyse :

Après avoir restructuré les données selon l'approche SOA (Structure of Arrays), l'expérience a exploré l'impact de cette optimisation en utilisant trois compilateurs différents : icx, gcc et clang. Chaque compilateur a été configuré avec des flags de compilation spécifiques tels que -xHost pour icx et -march=native pour gcc et clang.

Compilateur icx (Approche SOA)
- Moyenne GFLOP/s : 25.1 (± 0.0)
- Temps d'exécution moyen : 0.2468 s

Compilateur gcc (Approche SOA)
- Moyenne GFLOP/s : 6.5 (± 0.2)
- Temps d'exécution moyen : 0.925 s

Compilateur clang (Approche SOA)
- Moyenne GFLOP/s : 24.9 (± 0.1)
- Temps d'exécution moyen : 0.2472 s

4.1.4 Comparaison

La comparaison des performances entre les différentes versions du code, avec l'approche SOA et les compilateurs, met en évidence les avantages notables de cette optimisation. Icx a présenté une amélioration par rapport à gcc et clang, surpassant significativement leurs performances. Néanmoins, même avec l'optimisation SOA, gcc et clang ont affiché des performances relativement plus faibles par rapport à icx, soulignant ainsi la pertinence de l'optimisation spécifique réalisée avec icx.

4.1.5 Conclusion

L'application de l'approche SOA a généré une nette amélioration des performances du programme nbody3D sur tous les compilateurs étudiés. Cette restructuration des données a permis d'atteindre des GFLOP/s considérablement plus élevés par rapport à la version

de base du programme. Les temps d'exécution ont également été drastiquement réduits, confirmant l'efficacité de cette optimisation.

4.2 Optimisation 2 | Utilisation de la directive 'restrict'

4.2.1 Objectif et justification :

Dans le cadre de cette optimisation, l'utilisation de la directive `restrict` (C99) vise à informer le compilateur que certains pointeurs ne seront jamais utilisés pour référencer une variable référencée par un autre pointeur. Cette directive est particulièrement recommandée pour les paramètres de tableaux. Son objectif principal est de permettre au compilateur de générer un code plus performant en éliminant les possibilités de dépendances entre les pointeurs, ce qui peut potentiellement entraver l'optimisation du code.

4.2.2 Méthodologie :

Pour implémenter cette optimisation, la directive `restrict` a été utilisée sur les pointeurs appropriés dans le code source.

```
typedef struct particle_s {
  f32 *restrict x;
  f32 *restrict y;
  f32 *restrict z;
  f32 *restrict vx;
  f32 *restrict vy;
  f32 *restrict vz;
} particle_t;
```

4.2.3 Résultats et analyse :

L'intégration de la directive `restrict` n'a pas entraîné une augmentation significative des performances en termes de GFLOP/s pour chacun des compilateurs testés. Toutefois, cette optimisation a considérablement amélioré la stabilité des performances du code. Elle a assuré des résultats plus constants et prévisibles, indépendamment des variations possibles dans les conditions d'exécution.

- **Compilateur gcc :** La directive `restrict` a stabilisé les performances à environ 6.7 ($\pm$ 0.0) GFLOP/s, témoignant d'une stabilité accrue malgré l'absence d'une amélioration significative en termes de GFLOP/s.
- **Compilateur icx :** Les performances ont maintenu une stabilité autour de 25.0 ($\pm$ 0.1) GFLOP/s, soulignant l'impact limité sur les GFLOP/s, mais la stabilité remarquable obtenue.
- **Compilateur clang :** Les performances ont été maintenues autour de 24.9 ($\pm$ 0.2) GFLOP/s, illustrant une stabilité similaire sans un gain substantiel de performances brutes.

par seconde ont également connu une légère hausse après cette optimisation.

— **Compilateur Clang :** Avant l'optimisation, les performances du programme étaient stables autour de 24.9 GFLOP/s (± 0.2). Après l'optimisation, une nette amélioration est survenue, les performances passant à une moyenne de 45.2 GFLOP/s. Cela a été accompagné d'une augmentation des interactions par seconde et d'une amélioration globale du temps d'exécution.

— **Compilateur Intel icx :** Avant l'optimisation, les performances étaient établies à 25.0 GFLOP/s (± 0.1). Suite à l'optimisation, une amélioration significative a été observée, les performances augmentant à une moyenne de 49.4 GFLOP/s. Cela a été accompagné par une augmentation notable des interactions par seconde et une amélioration globale du temps d'exécution.

4.3.4 Comparaison des sorties d'assembleur générées par GCC :

L'examen des sorties d'assembleur des deux versions du code via l'outil Godbolt révèle une distinction notable. Dans la deuxième version, suite à l'optimisation, le nombre d'instructions a été significativement réduit. Cette réduction provient de la consolidation du calcul de la racine carrée en une seule opération, ce qui a contribué à minimiser le nombre de cycles d'exécution, entraînant une nette amélioration des performances du programme.

```
// https://godbolt.org/

-----> Version de Base :
pxor    xmm2, xmm2
        cvtss2sd        xmm2, DWORD PTR [rbp-60]
        movq    rax, xmm2
        movq    xmm0, rax
        call    sqrt
        cvtsd2ss        xmm0, xmm0
        movss   DWORD PTR [rbp-64], xmm0

-----> Deuxieme Version :
pxor    xmm2, xmm2
        cvtss2sd        xmm2, DWORD PTR [rbp-60]
        movq    rax, xmm2
        movq    xmm0, rax
        call    sqrt
        cvtsd2ss        xmm0, xmm0
        movss   DWORD PTR [rbp-64], xmm0
        movss   xmm0, DWORD PTR [rbp-64]
        mulss   xmm0, xmm0
        movss   xmm1, DWORD PTR [rbp-64]
        mulss   xmm0, xmm1
        movss   DWORD PTR [rbp-68], xmm0
```

```
        const f32 sqrt_d_2_3 = sqrt(d_2_4);

      . . .

        fx += dx1 / d_3_over_2_1;
        fy += dy1 / d_3_over_2_1;
        fz += dz1 / d_3_over_2_1;

        fx += dx2 / d_3_over_2_2;
        fy += dy2 / d_3_over_2_2;
        fz += dz2 / d_3_over_2_2;

        fx += dx3 / d_3_over_2_3;
        fy += dy3 / d_3_over_2_3;
        fz += dz3 / d_3_over_2_3;

        fx += dx4 / d_3_over_2_4;
        fy += dy4 / d_3_over_2_4;
        fz += dz4 / d_3_over_2_4;
    }
. . .
```

4.6.4 Résultats et analyse :

Les résultats de l'optimisation par déroulage de boucle sont les suivants :

GCC : Les performances moyennes ont été mesurées à 15.7 GFLOP/s. Comparé à l'optimisation antérieure de 19.6 GFLOP/s, il est notable qu'il y a eu une diminution des performances avec cette méthode.

Clang : Les performances ont également diminué , mais de façon moins marquée, avec une moyenne de 74.6 GFLOP/s contre 78.7 GFLOP/s dans l'optimisation précédente, montrant une légère réduction des performances mais restant significativement améliorées par rapport aux résultats antérieurs.

Intel icx : Les performances moyennes ont fortement augmenté pour atteindre 102.9 GFLOP/s, comparées aux 75 GFLOP/s obtenus avec l'optimisation précédente. Cette amélioration s'accompagne également d'une augmentation de la précision.

4.6.5 Comparaison entre les Compilateurs :

En comparant les performances obtenues avec cette nouvelle optimisation pour les trois compilateurs, on observe une tendance différente. Clang continue de dépasser les autres compilateurs avec une moyenne de 74.6 GFLOP/s. Intel icx, cependant, a montré une amélioration significative, passant de 75 GFLOP/s à 102.9 GFLOP/s, dépassant ainsi largement ses résultats antérieurs. À l'inverse, GCC a montré une réduction significative, passant de 19.6 GFLOP/s à 15.7 GFLOP/s avec cette méthode.

En conclusion, le déroulage de boucle a eu un impact variable sur les performances selon les compilateurs. Clang a manifesté une nette amélioration, montrant une bonne adaptation à cette optimisation. Cependant, il est crucial de noter que pour certains com-

pilateurs comme GCC, l'efficacité de la vectorisation automatique peut être compromise, ce qui a pu limiter les gains de performances.

4.6.6 Conclusion :

En conclusion, le déroulage de boucle a eu un impact variable sur les performances selon les compilateurs. Clang a manifesté une nette amélioration, montrant une bonne adaptation à cette optimisation. Cependant, il est crucial de noter que pour certains compilateurs comme GCC, l'efficacité de la vectorisation automatique peut être compromise, ce qui a pu limiter les gains de performances.

4.7 Optimisation 7 | Cache Blocking / Tiling :

4.7.1 Objectif et justification :

L'objectif de cette section est d'intégrer le blocage des boucles, également appelé tuilage, pour améliorer les performances du programme. Le blocage des boucles est une optimisation bien connue qui vise à exploiter efficacement le cache en découpant l'espace d'itération en tuiles plus petites. En maximisant la réutilisation des données dans chaque tuile, cette technique favorise le maintien de l'ensemble de travail dans le cache de données, réduisant ainsi les incohérences de cache et permettant un traitement plus efficace et parallèle des particules.

4.7.2 Méthodologie - Changements au niveau de Code :

Pour intégrer le blocage des boucles dans le code, j'ai ajouté une constante `tile_size` définissant la taille des tuiles. À l'aide de directives OpenMP, j'ai par la suite réorganisé les boucles pour diviser l'espace d'itération en tuiles. Chaque itération du premier passage parcourt les tuiles allouées à chaque thread, définissant ainsi une zone de travail spécifique pour chaque thread.

```
. . .
void move_particles(particles_t *p, const f32 dt, u64 n) {
  const f32 softening = 1e-20;
  const int tile_size= 64 ;
  // Using Parallelization for i loop
    #pragma omp parallel
{

    const int num_threads = omp_get_num_threads();
    const int thread_id = omp_get_thread_num();
    const int tiles_per_thread = (n + tile_size - 1) / tile_size / num_thre
    const int start_tile = thread_id * tiles_per_thread;
    const int end_tile = (thread_id + 1) * tiles_per_thread;

   for (int tile = start_tile; tile < end_tile; tile++) {
      const int tile_start_index = tile * tile_size;
      const int tile_end_index = fmin((tile + 1) * tile_size, n);
   // Travailler avec les particules dans la tuile
```

5 Comparaison des Performances

L'évaluation des différentes optimisations entreprises dans le cadre de ce projet a révélé des tendances distinctes pour chaque compilation. Les efforts pour améliorer les performances du programme ont abouti à des résultats variables selon les optimisations et les compilateurs. GCC a montré une progression graduelle mais constante, avec des améliorations notables après chaque itération d'optimisation, culminant avec une augmentation significative lors de l'Optimisation 8. En parallèle, Clang et Intel icx a présenté des améliorations progressives, avec une croissance continue des performances d'une optimisation à l'autre, aboutissant à des gains significatifs après l'Optimisation 7.

Pour résumer, la performance initiale était relativement basse, autour de 0.6 GFLOP/s pour les compilateurs GCC et Clang, et de 5.0 GFLOP/s pour le compilateur icx. Au fur et à mesure des optimisations successives, ces performances ont été améliorées de manière significative. L'Optimisation 1 a marqué une amélioration notable, en particulier pour

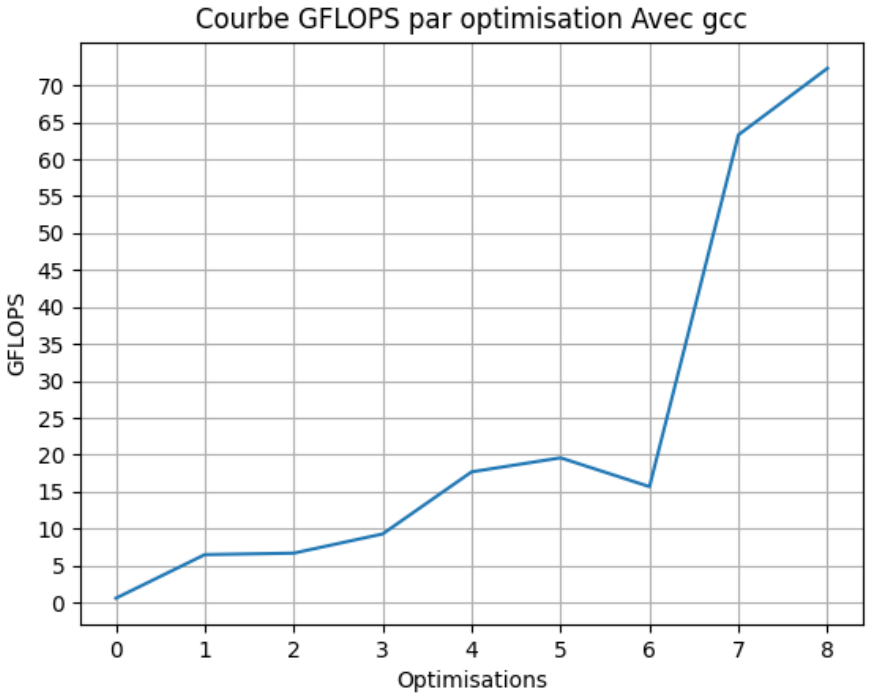

FIGURE 6 – Courbe de gain Avec gcc

GCC, augmentant les GFLOP/s de 6.5 à 9.3 en moyenne. En revanche, l'Optimisation 2 a maintenu la stabilité des performances sans apporter de gains significatifs en termes de GFLOP/s pour les trois compilateurs. L'Optimisation 3 a engendré des améliorations pour tous les compilateurs, avec une augmentation significative pour icx, passant de 25.0 à 49.4 GFLOP/s. L'Optimisation5 a entraîné des progrès pour GCC et Clang, mais a légèrement impacté négativement les performances d'icx.,À l'inverse, l'Optimisation 6 a diminué les performances de GCC et Clang, tandis qu'icx a largement progressé pour atteindre 102.9 GFLOP/s. L'Optimisation 7 a été remarquable, générant des améliorations significatives pour GCC et Clang, mais a marqué un recul pour icx. Enfin, l'Optimisation 8 a montré une augmentation pour GCC, une légère diminution pour Clang et une baisse significative pour icx.

— Les améliorations les plus notables ont été constatées lors de l'OPT4, où toutes les configurations ont enregistré une augmentation significative des performances, en particulier avec icx qui a augmenté de 49.4 à 82.4 GFLOP/s.

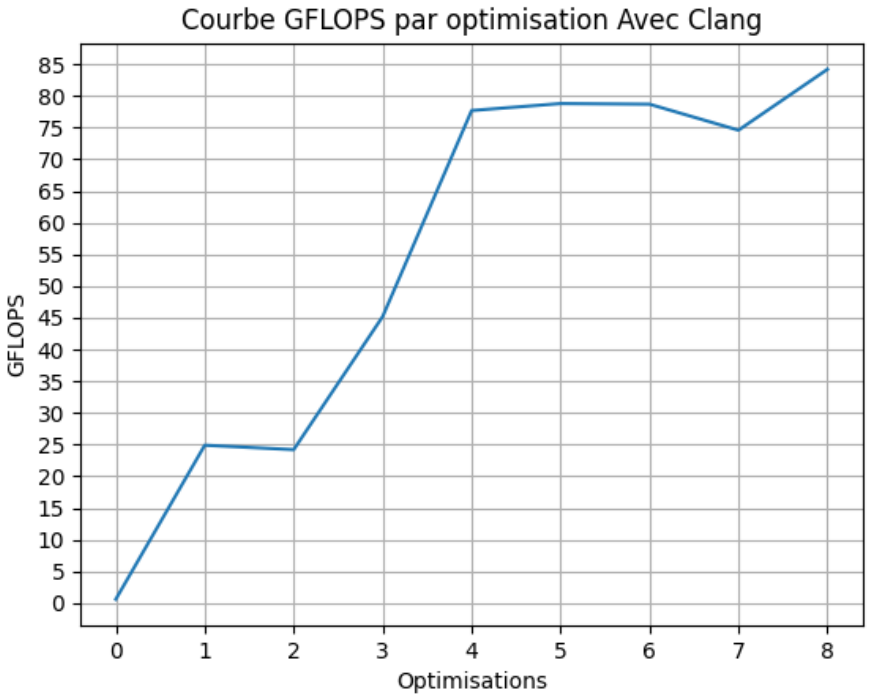

FIGURE 7 – Courbe de gain Avec Clang

— L'Optimisation 7 de Cache Blocking a présenté des performances exceptionnelles sur les trois compilateurs, avec des améliorations pour GCC et Intel icx.
— Intel icx a souvent bénéficié des optimisations, mais l'Optimisation 8 a montré une légère baisse, indiquant une spécificité matérielle pour certaines optimisations.

Ces résultats soulignent la diversité des réponses des compilateurs aux optimisations, mettant en évidence l'importance d'adapter les stratégies d'optimisation pour tirer le meilleur parti des spécificités de chaque compilateur et maximiser les performances du programme.

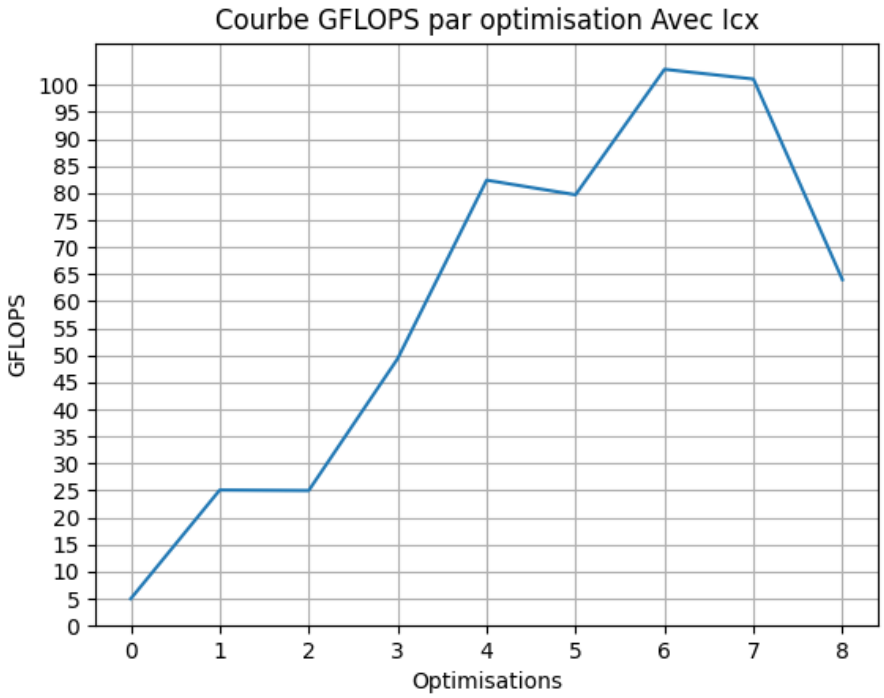

FIGURE 8 – Courbe de gain Avec Icx

6 Conclusion

Le processus d'optimisation du programme de simulation de particules a été une exploration approfondie des diverses stratégies visant à améliorer les performances du code.Cette entreprise a révélé une dynamique riche et variée, mettant en lumière l'impact différencié des optimisations sur les compilateurs GCC, Clang et Intel icx.

L 'exploration des diverses optimisations pour améliorer les performances du programme de simulation de particules a été une démarche complexe et instructive. Notre parcours a dévoilé une véritable diversité dans les réponses des compilateurs GCC, Clang et Intel icx aux différentes stratégies d'optimisation mises en œuvre.

Initialement, les performances modestes autour de 0.6 GFLOP/s pour GCC et Clang, et de 5.0 GFLOP/s pour icx, ont tracé le point de départ de notre évaluation. Chaque optimisation a offert une vision précise de l'impact variable de ces modifications sur les performances. Certaines optimisations ont été marquées par des améliorations significatives, tandis que d'autres ont montré des gains plus modestes, voire nuls. L'Optimisation 4, axée sur la parallélisation, a été particulièrement fructueuse, affichant des améliorations notables pour toutes les configurations, surtout pour icx. De même, l'Optimisation 7, le Cache Blocking, a livré des performances exceptionnelles pour GCC et icx, soulignant la spécificité des résultats en fonction des techniques employées. Toutefois, l'Optimisation 8 a souligné la complexité des performances matérielle-dépendantes, avec une baisse pour icx malgré les efforts consentis.

Cette expérience a démontré l'importance cruciale d'une approche adaptative et minutieuse pour choisir les stratégies d'optimisation des spécificités du matériel et des compilateurs utilisés.. La variabilité des réponses des compilateurs souligne la nécessité de bien cerner les spécificités matérielles pour obtenir les meilleurs résultats.

References

[1] *Array of Structs vs Struct of Arrays.* URL: https://medium.com/@savas/nomad-game-engine-part-4-3-aos-vs-soa-storage-5bec879aa38c.

[2] *Gcc Options That Control Optimization.* URL: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html.

[3] *High Performance Computing Serial optimizations and vectorization.* URL: http://www.metz.supelec.fr/metz/personnel/vialle/course/CS-2A-SG6-HPC/notes-de-cours-specifiques/SG6-02-OptimSerielle-Vecto-2spp.pdf.

[4] *What is Vectorisation ?* URL: https://chryswoods.com/vector_c++/vectorisation.html.