Optimisation de Performances : Projet Architecture des Ordinateurs

Amayes Djermoune

January 8, 2024

1 Introduction

L'objectif principal de ce projet est de simuler les interactions gravitationnelles entre particules à l'aide de l'algorithme N-corps. Cette simulation est souvent utilisée dans le domaine de la modélisation astrophysique pour étudier le mouvement et les interactions des objets célestes tels que les étoiles, les planètes ou les galaxies.

2 Description de l'Architecture et l'Environnement

Processeur :

- Modèle : 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
- Famille du CPU : 6
- Nombre total de cœurs : 4
- Nombre total de processeurs logiques : 8 (indiquant la prise en charge de l'hyper-threading)
- Fréquence du CPU : Varie entre 1217.569 MHz et 1700.000 MHz

Cache :

- Taille totale du cache : 12288 KB (12 Mo)
- Configuration du cache :
  - L1 (64K instruction, 48K data)
  - L2 (1024K)
  - L3 (12288K)

Instructions et fonctionnalités :

- Prend en charge un large ensemble d'instructions et de fonctionnalités, y compris les jeux d'instructions avancés comme AVX512.
- Différentes optimisations matérielles telles que la virtualisation (VMX flags) et la gestion de l'alimentation sont prises en charge.

Bogomips :

- La valeur de Bogomips est d'environ 3379.20.

Système d'exploitation :

- Ubuntu 22.04

Autres informations :

- La taille des pages mémoire est de 4K.

3 Stabilisation

Maximisation de la Stabilité de l'Environnement

Pour maximiser la stabilité de l'environnement, il est d'important de penser à la configuration et gestion des ressources :

1. **Maximiser la fréquence des cœurs :**
   - S'assurer que la fréquence des cœurs du processeur est maximisée pour tirer pleinement parti des performances matérielles.
   - Vérification des paramètres du gestionnaire de puissance pour garantir que le processeur fonctionne à sa fréquence maximale lorsque nécessaire.
2. **Distribution des fichiers exécutables sur les cœurs actifs :**
   - Emploi de techniques d'affinité de processeur pour distribuer les fichiers exécutables sur des cœurs spécifiques.
   - Cela peut être réalisé en utilisant des paramètres d'affinité de processeur au niveau du système d'exploitation ou en utilisant des bibliothèques spécifiques à la programmation parallèle.

3. Couper la connexion pour limiter les erreurs :

- L'élimination de la connexion internet est nécessaire dans le but de limiter les bruits et s'assurer à ce que les mesures de performances soient effectués dans les meilleurs conditions possibles. Ajuster les règlages du pare-feu peut s'avérer utile pour empêcher les connexions non nécessaires .

4 Élimination/Remplacement des Opérations Coûteuses

4.1 Approche

Le code fourni de base est une implémentation parallèle de la simulation des interactions gravitationnelles entre particules utilisant l'algorithme N-corps. L'optimisation est réalisée à l'aide de la directive OpenMP, permettant la parallélisation de certaines parties du code pour exploiter les architectures multi-cœurs.

4.2 Points Clés de l'Optimisation

1. **Parallélisation avec OpenMP :** On utilise la directive OpenMP 'pragma omp parallel for' pour paralléliser la boucle externe, distribuant l'itération de la boucle sur plusieurs threads et exploitant les capacités multi-cœurs des processeurs modernes.

2. **Réduction pour le calcul des forces nettes :** Une directive OpenMP 'pragma omp parallel for reduction(+:fx, fy, fz)' est utilisée pour paralléliser le calcul des forces nettes. Cela assure que les variables 'fx', 'fy', et 'fz' sont partagées entre les threads. Ainsi, les résultats partiels sont correctement sommés à la fin de la boucle.

3. **Chauffage (warm-up) :** On exclut un certain nombre d'itérations initiales (définies par la variable 'warmup') lors du calcul des performances pour permettre au système de se "chauffer" et stabiliser les performances avant de mesurer les performances effectives.

4. **Mesure des Performances :** On mesure le temps d'exécution, le nombre d'interactions par seconde, et les opérations en virgule flottante par seconde (GFLOP/s) à chaque itération de la simulation. Les résultats sont ensuite affichés, et une moyenne des performances est calculée, excluant les itérations de chauffage.

5. **Mesure de la Taille Totale de la Mémoire Allouée :** On calcule et affiche la taille totale de la mémoire allouée pour stocker les particules, offrant une évaluation de l'impact sur la consommation mémoire du programme.

Les résultats des tests d'optimisation avec GCC et Clang pour la restructuration des structures de données à différents niveaux d'optimisation (0, 2, 3, fast) sont présentés ci-dessous :

Avec GCC :

Niveau d'Optimisation 0

- Average performance: 0.3 ± 0.0 GFLOP/s

Niveau d'Optimisation 2

- Average performance: 0.7 ± 0.0 GFLOP/s

Niveau d'Optimisation 3

- Average performance: 0.8 ± 0.0 GFLOP/s

Niveau d'Optimisation Fast

- Average performance: 0.7 ± 0.0 GFLOP/s

Avec Clang :

Niveau d'Optimisation 0

- Average performance: 0.3 ± 0.0 GFLOP/s

Niveau d'Optimisation 2

- Average performance: 0.7 ± 0.0 GFLOP/s

Niveau d'Optimisation 3

- Average performance: 0.7 ± 0.0 GFLOP/s

Niveau d'Optimisation Fast

- Average performance: 0.7 ± 0.0 GFLOP/s

4.3 Analyse Comparative :

- Les performances sont généralement améliorées avec l'augmentation du niveau d'optimisation.
- Le niveau d'optimisation 3 avec GCC a donné la meilleure performance avec 0.8 GFLOP/s.
- Les performances entre GCC et Clang sont comparables, montrant une légère compétitivité entre les deux compilateurs.

5 Restructuration des Structures de Données et Déroulage, Blocking Manuel

5.1 Approche d'Optimisation

Une autre idée d'optimisation consiste à effectuer une implémentation parallèle de la simulation des interactions gravitationnelles entre particules, également basée sur l'algorithme N-corps. L'optimisation est réalisée en utilisant la directive OpenMP pour exploiter les capacités de parallélisme offertes par les architectures multi-cœurs.

5.2 Points Clés de l'Optimisation

1. **Allocation Dynamique avec Unrolling :** Les tableaux dynamiques nécessaires aux calculs sont alloués une seule fois, et des techniques de déroulage de boucle sont utilisées pour améliorer l'efficacité de la manipulation des données en mémoire ainsi qu'une exploitation efficace de cette dernière.

2. **Pré-Calcule des Valeurs :** Certains calculs, tels que les différences de positions, les distances au carré, et les inverses des distances au cube, sont pré-calculés pour éviter des opérations répétitives à l'intérieur des boucles principales et les calculs inutiles à chaque itération.

3. **Boucle Déroulée avec Blocking Manuel :** La boucle principale de calcul est déroulée avec un blocage manuel pour permettre un meilleur accès aux données en mémoire et améliorer le parallélisme sur les architectures vectorielles.

4. **Utilisation de `#pragma omp simd` :** Pour améliorer le parallélisme, la directive OpenMP `#pragma omp simd` est utilisée pour appliquer des instructions SIMD (Single Instruction, Multiple Data) à certaines parties du code, optimisant ainsi l'utilisation des unités de calcul vectoriel.

5. **Exclusion des Itérations Initiales (Warm-Up) :** Les performances du système sont mesurées après une période initiale d'échauffement (warm-up) pour permettre au système de stabiliser ses performances avant les mesures effectives.

6. **Affichage Clair des Résultats de mesure :** Les résultats sont présentés de manière claire avec des informations telles que le temps d'exécution, le nombre d'interactions par seconde, et les GFLOP/s, facilitant le suivi de l'évolution des performances au fil des itérations.

7. **Mesure de la Taille Totale de la Mémoire Allouée :** La taille totale de la mémoire allouée pour stocker les particules est calculée et affichée, offrant une évaluation de l'impact sur la consommation mémoire du programme.

5.3 Analyse des Performances avec GCC et Clang

L'analyse des résultats pour GCC et Clang, basée sur les performances en termes de GFLOP/s (gigaflops par seconde) dans différents niveaux d'optimisation (0, 2, 3, fast), révèle quelques points assez intéressants :

GCC :

- **Niveau d'optimisation 0 (Aucune optimisation) :**
  - **Performance :** 0.3 GFLOP/s en moyenne.
  - **Observations :** C'est le niveau d'optimisation le plus bas, et la performance est relativement faible, ce qui est attendu.
- **Niveau d'optimisation 2 :**
  - **Performance :** 0.7 GFLOP/s en moyenne.
  - **Observations :** Une amélioration significative par rapport au niveau d'optimisation 0. Le compilateur GCC montre sa capacité à optimiser le code source.
- **Niveau d'optimisation 3 :**
  - **Performance :** 0.8 GFLOP/s en moyenne.
  - **Observations :** Une nouvelle amélioration notable par rapport au niveau 2. Cela suggère que des optimisations plus agressives ont été appliquées.
- **Niveau d'optimisation Fast :**
  - **Performance :** 0.7 GFLOP/s en moyenne.
  - **Observations :** Bien que le niveau soit appelé "Fast", la performance moyenne n'est pas significativement meilleure que le niveau 2. Cela peut s'expliquer par le compromis entre vitesse de compilation et optimisations agressives.

Clang :

- **Niveau d'optimisation 0 (Aucune optimisation) :**
  - **Performance :** 0.3 GFLOP/s en moyenne.
  - **Observations :** Comme pour GCC, le niveau d'optimisation le plus bas a une performance relativement basse.
- **Niveau d'optimisation 2 :**
  - **Performance :** 0.7 GFLOP/s en moyenne.

  - **Observations :** Des performances similaires à GCC au niveau 2, montrant une capacité comparable à générer un code optimisé.

- **Niveau d'optimisation 3 :**
  - **Performance :** 0.7 GFLOP/s en moyenne.
  - **Observations :** Les performances au niveau 3 sont similaires à celles de GCC au niveau 2. Cela peut varier en fonction de la nature du code source.

- **Niveau d'optimisation Fast :**
  - **Performance :** 0.6 GFLOP/s en moyenne.
  - **Observations :** Contrairement à GCC, le niveau "Fast" montre une légère baisse de performance par rapport au niveau 2. Cela peut être dû à la priorité donnée à la rapidité de compilation plutôt qu'à des optimisations plus agressives.

Comparaison Générale :

- GCC semble montrer une amélioration minuscule des performances globales par rapport à Clang, en particulier au niveau d'optimisation 3.
- Clang, quant à lui, est souvent apprécié pour ses temps de compilation plus rapides et ses messages d'erreur plus clairs, mais dans ce cas particulier, les performances sont généralement similaires à celles de GCC.

Analyse Détaillée :

- **Temps d'exécution :** - GCC semble avoir des temps d'exécution légèrement plus longs que Clang. Cela pourrait être lié à la manière dont chaque compilateur génère le code optimisé.
- **Interactions par seconde :** - Les interactions par seconde sont similaires entre GCC et Clang, montrant que malgré quelques variations dans les temps d'exécution, le nombre d'interactions par seconde reste cohérent.
- **GFLOP/s :** - Les résultats indiquent que le nombre moyen de GFLOP/s généré par GCC est légèrement supérieur à celui de Clang.

Conclusion :

- Les deux compilateurs génèrent des résultats cohérents et stables en termes de performances pour ce code spécifique.
- GCC semble avoir une légère avance en termes de GFLOP/s moyens, bien que la différence soit relativement petite.

- La sélection entre GCC et Clang peut également dépendre d'autres considérations, telles que la facilité d'utilisation, la compatibilité avec d'autres bibliothèques.

6 Algorithme Barnes-Hut

L'algorithme de Barnes-Hut est une technique fondamentale utilisée pour accélérer le calcul des interactions gravitationnelles entre particules dans un système, notamment dans des simulations astrophysiques, particulièrement dans le contexte des simulations N-corps. L'objectif principal de cet algorithme est de réduire la complexité algorithmique de $O(N^2)$à$O(NlogN)$.

Fonctionnement de l'algorithme :

1. **Construction de l'arbre octal (Octree) :** Les particules de l'espace sont regroupées dans une structure de données appelée arbre octal (Octree). Cet arbre subdivise l'espace tridimensionnel de manière récursive. Chaque nœud de l'arbre représente une région de l'espace, et chaque nœud peut avoir jusqu'à huit sous-régions (nœuds enfants) qui sont un résultat de la subdivision.

2. **Calcul du centre de masse pour les nœuds de l'arbre :** Pour chaque nœud de l'arbre, le centre de masse ainsi que la masse totale des particules contenues dans ce nœud sont calculés. Cette étape permet de représenter plusieurs particules par un seul point (le centre de masse) lorsque la subdivision de l'espace atteint une certaine profondeur.

3. **Approximation des interactions lointaines :** L'idée clé de l'algorithme de Barnes-Hut réside dans l'approximation des interactions gravitationnelles entre groupes éloignés de particules. Si un nœud de l'arbre est suffisamment éloigné d'un point spécifique, l'influence gravitationnelle de toutes les particules contenues dans ce nœud est approximée par un seul point de masse égale à la masse totale du nœud, situé au centre de masse du nœud.

4. **Calcul récursif des interactions :** L'algorithme de Barnes-Hut utilise la récursivité pour calculer efficacement les interactions gravitationnelles. À chaque niveau de l'arbre, la décision est prise d'approximer ou de détailler l'influence des particules en fonction de la distance par rapport au point d'intérêt. Si la distance est suffisamment grande, l'interaction est approximée à l'aide du point de masse unique du nœud. Sinon, l'algorithme descend dans les nœuds enfants pour effectuer des calculs plus détaillés.

5. **Mise à jour des positions et des vitesses :** Une fois que toutes les interactions ont été calculées, les positions et les vitesses des particules sont mises à jour en fonction des forces résultantes.

L'avantage significatif de l'algorithme de Barnes-Hut réside dans sa capacité à réduire considérablement le nombre de calculs nécessaires pour simuler les interactions gravitationnelles dans un système composé de N particules.

7 Comparaison des Performances entre GCC et Clang

Résultats avec GCC :

Compilation Niveau 0 :

- Temps d'exécution moyen : 0.00434 secondes - Interactions par seconde : 61.78 milliards - GFLOP/s : 1050.3

Compilation Niveau 2 :

- Temps d'exécution moyen : 0.00431 secondes - Interactions par seconde : 62.22 milliards - GFLOP/s : 1057.8

Compilation Niveau 3 :

- Temps d'exécution moyen : 0.0043 secondes - Interactions par seconde : 62.44 milliards - GFLOP/s : 1061.5

Compilation Niveau fast :

- Temps d'exécution moyen : 0.00446 secondes - Interactions par seconde : 60.17 milliards - GFLOP/s : 1022.9

Moyenne des performances :

- GFLOP/s : 1048.5 $\pm$ 19.6

Résultats avec Clang :

Compilation Niveau 0 :

- Temps d'exécution moyen : 0.00419 secondes - Interactions par seconde : 64.1 milliards - GFLOP/s : 1089.7

Compilation Niveau 2 :

- Temps d'exécution moyen : 0.00351 secondes - Interactions par seconde : 76.48 milliards - GFLOP/s : 1300.2

Compilation Niveau 3 :

- Temps d'exécution moyen : 0.00283 secondes - Interactions par seconde : 95.41 milliards - GFLOP/s : 1622.1

Compilation Niveau :

- Temps d'exécution moyen : 0.00274 secondes - Interactions par seconde : 98.87 milliards - GFLOP/s : 1680.8

Moyenne des performances :

- GFLOP/s : 1499.4 $\pm$ 93.9

Analyse Comparative :

1. **Temps d'exécution :** - En général, Clang semble avoir des temps d'exécution inférieurs à ceux de GCC à tous les niveaux de compilation.

2. **Interactions par seconde :** - Clang montre une tendance à augmenter le nombre d'interactions par seconde à mesure que le niveau de compilation augmente.

3. **GFLOP/s :** - Clang surpasse significativement GCC en termes de GFLOP/s à tous les niveaux de compilation, avec une moyenne de 1499.4 comparé à 1048.5 pour GCC.

Conclusion :

Les résultats suggèrent que, dans ce cas spécifique, Clang a des performances supérieures à GCC en termes de GFLOP/s, même si le temps d'exécution peut varier. Il est important de noter que les performances peuvent dépendre du type de code, des optimisations possibles, des options de compilation spécifiques, et des caractéristiques de l'architecture matérielle utilisée.

8 Conclusion

8.1 Choix entre GCC et Clang

Le choix entre GCC et Clang dépend souvent du contexte spécifique du projet, des préférences personnelles et des besoins particuliers en matière de compilation. Les deux compilateurs sont puissants, largement utilisés et ont leurs propres avantages. Voici quelques points à considérer :

GCC :

- **Historique :** GCC (GNU Compiler Collection) a une longue histoire et est utilisé depuis des décennies dans le développement logiciel.
- **Compatibilité :** GCC est souvent considéré comme très stable et offre une compatibilité élevée avec les normes du langage.
- **Communauté :** Il bénéficie d'une grande communauté d'utilisateurs et de contributeurs, ce qui signifie un support important.

Clang :

- **Clarté des messages d'erreur :** Clang est souvent apprécié pour ses messages d'erreur plus clairs et faciles à comprendre, facilitant le processus de débogage.
- **Temps de compilation :** Clang est souvent vanté pour ses temps de compilation plus rapides dans certaines situations.
- **Modules :** Clang a introduit la prise en charge des modules C++, une fonctionnalité qui peut améliorer la gestion des dépendances dans les grands projets.

En fin de compte, il n'y a pas de réponse universelle à la question de savoir lequel est "meilleur", car cela dépend des besoins spécifiques du projet et des préférences de l'équipe de développement. Il peut être utile d'essayer les deux compilateurs nous offrent: Une mesure des performances et Considération de la qualité des diagnostics avant la prise de décision.

8.2 Pratiques et Leçons Tirées

À la lumière des résultats obtenus dans ce projet, voici quelques pratiques et leçons à tirer :

1. **Comparaison de Compilateurs :** - Il est essentiel de comparer les performances de différents compilateurs pour un projet donné. Les performances peuvent varier en fonction du type de code, des optimisations possibles, et des caractéristiques de l'architecture matérielle.

2. **Optimisation du Code :** - Investir du temps dans l'optimisation du code peut conduire à des améliorations significatives des performances. Des changements subtils dans l'écriture du code peuvent avoir un impact sur la manière dont les compilateurs optimisent le programme. Une simple ligne de code peut tout changer.

3. **Choix du Compilateur en Fonction des Besoins :** - Le choix entre GCC et Clang (ou d'autres compilateurs) dépend des besoins spécifiques du projet. Dans ce cas, Clang a montré des performances supérieures en termes de GFLOP/s, mais il peut y avoir des cas où GCC est plus adapté et flexible.

4. **Analyse Fine des Résultats :** - Il est crucial de ne pas se fier uniquement à une métrique, comme le temps d'exécution. Puisque dans certains projets, y'aura des moments ou pour deux codes différents, on aura le meme résultat suivant une métrique particulière, ce qui nous pousse à changer de métrique pour plus de précision. L'analyse fine des résultats, y compris le nombre d'interactions par seconde et les GFLOP/s, permet d'obtenir une compréhension plus complète des performances.

5. **Compréhension des Options de Compilation :** - Les résultats peuvent être fortement influencés par les options de compilation. Il est important de comprendre comment ces options affectent le code généré et d'expérimenter avec différentes combinaisons pour tirer les meilleures performances.

6. **Adaptation aux Caractéristiques Matérielles :** - Les performances peuvent varier en fonction de l'architecture matérielle. Il peut être utile de tester le code sur différentes architectures pour s'assurer qu'il est optimisé pour le matériel cible. Bien évidemment, les résultats varient d'une architecture à une autre et aussi de la façon dont chaque coeur du CPU se comporte.

7. **Surveillance des Mises à Jour des Compilateurs :** - Les compilateurs sont continuellement mis à jour avec des améliorations de performances. Surveiller les mises à jour des compilateurs et effectuer des tests réguliers est recommandé.

Pour finir, l'optimisation des performances nécessite une approche holistique, impliquant à la fois l'optimisation du code source et la compréhension des caractéristiques des compilateurs et de l'architecture matérielle. D'ou je tiens à remercier Mr.Bolloré de m'avoir donné l'opportunité de travailler sur ce projet que j'ai trouvé assez instructif et utile pour ma carrière en tant que futur ingénieur de l'ISTY :) .

Références

- OpenMP Architecture Review Board. (2015). *OpenMP Application Program Interface Version 4.5.* Retrieved from https://www.openmp.org/specifications/
- Aarseth, S. J. (2003). *Gravitational N-Body Simulations: Tools and Algorithms.* Cambridge University Press.
- Barnes, J., & Hut, P. (1986). *A Hierarchical O(N log N) Force-Calculation Algorithm.* Nature, 324, 446–449.
- Quinn, T., Katz, N., & Stadel, J. (1997). *Particle Mesh Ewald: An N-log(N) method for Ewald sums in large systems.* The Astrophysical Journal, 486(2), 43–55.
- Godbolt, M. (n.d.). Compiler Explorer. Retrieved from https://godbolt.org/
- Intel. (n.d.). *Intel Intrinsics Guide.* Retrieved from https://www.intel.com/content/
- Agner Fog. (2021). *Optimizing software in C++: An optimization guide for Windows, Linux, and Mac platforms.* Retrieved from https://www.agner.org/optimize/
- ChatGPT - Recherches sur l'Algorithme de Barnes-Hut