Université de Versailles Saint Quentin en Yvelines

Projet Architecture Parallèle
nBody3D Optimisation

Master Calcul Haute Performance Simulation (CHPS)

Rapport Projet Réalisé par:

KEDDAD Youcef
Faculté d'Informatique

Année Universitaire : 2022-2023

Introduction

Le problème Nbody consiste à utiliser des lois physiques pour produire le mouvement de N corps à partir de leur état initial. L'exemple le plus courant du problème Nbody est la modélisation du comportement des objets célestes sous l'effet de la gravité. Bien qu'aucune solution générale n'ait été trouvée pour N > 3, le mouvement de tous les Nbodies peut être simulé en effectuant continuellement de petits pas de temps. À chaque pas de temps, la force agissant sur un corps est recalculée, en tenant compte de l'influence gravitationnelle des autres corps du système, et sa vitesse et sa position sont également mises à jour en fonction de cette valeur.

Cette approche nécessite de considérer toutes les paires de corps dans le système pour trouver les forces totales, et donc la complexité de la simulation croît comme N2. La plupart des applications impliquent une très grande valeur de N, de sorte que le travail N2 ne peut être effectué dans un délai raisonnable.

Le programme qui nous a été donné représente une simulation simple du problème Nbody dans un espace à 3 dimensions, il affiche les performances de chaque itération puis la performance moyenne (avec l'erreur relative). Les 3 premières itérations sont considérées comme un échauffement et ne sont pas utilisées pour le calcul de la performance moyenne et de l'erreur.

Tous les résultats concernant la performance sont en millisecondes ($T_{tot} - T_{warm}$) et en GFLOPS (measure for the calculating speed of floating-point operations per second). Il faut optimiser le code en faisant plusieurs versions et les analyser et comparer. En utilisant les compilateurs gcc et clang.

Target architecture x86_64

Model	Cores	Max Frequency GHz	L1 size KiB	L2 size KiB	L3 size MiB	SIMD
AMD Ryzen 5 5500U	6	4.0	32	512	4	SSE, AVX2

1. Aos vs SoA

Array of Struct :
"Array of Struct" (AoS) stocke un tableau de structures, où chaque structure contient tous les champs pour un élément unique des données. Par exemple, un AoS stockerait un tableau de structures, où chaque structure contient des coordonnées x, y et z. Cela peut être utile lorsque les données sont accédées d'une manière qui itère sur l'ensemble de la structure à la fois, car cela permet un accès plus naturel et plus pratique aux données.

La structure AoS est plus simple et plus facile à utiliser dans les cas où la structure entière est accédée d'un coup.

Le programme qui nous a été donné impelemente une structure de stockage des coordonnées x,y,z et de leur vélocité vx,vy,vz en AoS.

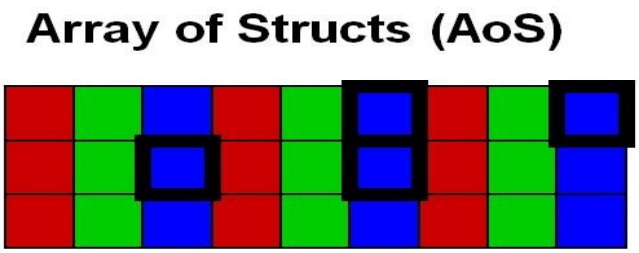

Figure 1: La structure AoS

Implémentation :

```
typedef struct particle_s {

  f32 x, y, z;
  f32 vx, vy, vz;

} particle_t;
```

Figure 2: Implementation (AoS)

Performance :

Compilateur	Time (s)	GFLOP/s
gcc	5.4	1.1
clang	5.7	1.1

Table 1: Performance AoS

flag d'optim: -march=native -g3 -O1 -fopenmp

Struct of Array:

"Struct of Array" (SoA), les données sont organisées sous forme d'un tableau de structures. Chaque structure dans le tableau contient un seul champ pour chaque élément des données. Par exemple, si vous avez une structure qui représente un point dans l'espace 3D, avec des champs pour les coordonnées x, y et z, une SoA stockerait toutes les coordonnées x dans un tableau unique, toutes les coordonnées y dans un autre tableau et toutes les coordonnées z dans un troisième tableau. Cela peut être utile lorsque les données sont accédées d'une manière qui itère sur un champ de la structure à la fois, car cela permet une meilleure localité de cache et peut entraîner une meilleure performance.

La structure SoA est plus adaptée au cache et peut entraîner une meilleure performance dans les cas où les éléments du même champ sont accédés ensemble.

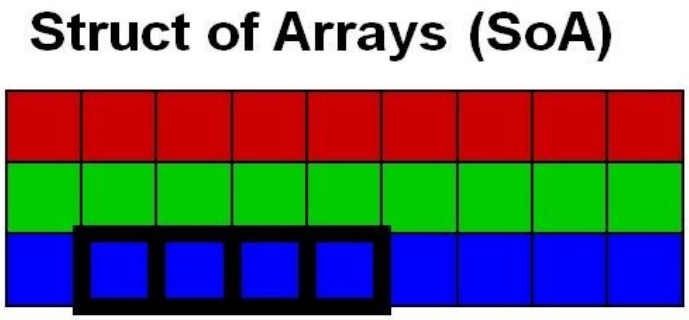

Figure 3: La structure SoA

Implémentation :

```
typedef struct particles_s {

  f32 *x, *y, *z;
  f32 *vx, *vy, *vz;

} particles_t;
```

Figure 4: Implementation (SoA)

Performance :

Compilateur	Time (s)	GFLOP/s
gcc	3.396e-01	18.1
clang	3.256e-01	19.0

Table 2: Performance SoA

On constate une amélioration pour le programme compilé avec gcc et clang.
flag d'optim: -march=native -g3 -mtune=native -Ofast -fopenmp

2. Memory alignment

L'alignement mémoire est la pratique de stocker les variables à des adresses mémoire spécifiques pour améliorer les performances de l'accès à la mémoire.

Les processeurs utilisent des bus de mémoire pour accéder aux données en mémoire. La plupart des processeurs modernes ont des bus de mémoire de 64 bits ou plus. Cela signifie qu'ils peuvent accéder à 8 octets (64 bits) de données en une seule opération. Comme on a fait dans cette version on a aligné nos tableaux en block de 64 bits.

Implémentation :

```
const u64 boundary = 64;
//
 p->x = aligned_alloc(boundary, sizeof(f32) * n);
 p->y = aligned_alloc(boundary, sizeof(f32) * n);
 p->z = aligned_alloc(boundary, sizeof(f32) * n);

 p->vx = aligned_alloc(boundary, sizeof(f32) * n);
 p->vy = aligned_alloc(boundary, sizeof(f32) * n);
 p->vz = aligned_alloc(boundary, sizeof(f32) * n);
```

Figure 5: Memory alignment Implementation

Performance :

Compilateur	Time (s)	GFLOP/s
gcc	3.394e-01	18.2
clang	3.250e-01	19.0

Table 3: Performance Memory alignment

On remarque une légère amélioration par rapport à la version SoA pour le programme compilé avec gcc.

flag d'optim: -march=native -g3 -mtune=native -Ofast -finline-functions -falign-functions -fopenmp

3. Instructions: sqrt, pow, division

Dans cette partie, on concentre à supprimer ou remplacer les instructions qui coûtent beaucoup d'opérations de calculs.

3.1 nbody_pow

remplacer pow() par une multiplication et sqrt() par sqrtf() ; $X^{3/2} = X * sqrtf(X)$

Implémentation :

```
const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening; //9 (mul, add)
// const f32 d_3_over_2 = pow(d_2, 3.0 / 2.0);
const f32 d_3_over_2 = d_2 * sqrtf(d_2);  //11 (mul, sqrt)
```

Figure 6: Implementation remove pow & sqrt

Performance :

Compilateur	Time (s)	GFLOP/s
gcc	2.184e-01	28.3
clang	1.506e-01	41.0

Table 4: Performance pow

3.2 nbody_div

remplacer les divisions par des multiplications par l'inverse.

Implémentation :

```
const f32 d_2 = 1.0f / sqrtf((dx * dx) + (dy * dy) + (dz * dz) + softening); // 10 (mul, add, div)
const f32 d_3_over_2 = d_2 * d_2 * d_2; // 12 (mul)


// Net force
fx += dx * d_3_over_2;  //14
fy += dy * d_3_over_2;  //16
fz += dz * d_3_over_2;  //18
```

Figure 7: Implementation remove divs

Performance :

Compilateur	Time (s)	GFLOP/s
gcc	1.400e-01	46.0
clang	1.483e-01	43.3

Table 5: Performance div

On remarque une augmentation des performances de 150 % par rapport au SoA pour les deux compilateurs, car les divisions coûtent plus que les multiplications.

flag d'optim: -march=native -g3 -mtune=native -Ofast -fopenmp

4. Loop unrolling

En utilisant #pragma unroll qui est une directive de préprocesseur utilisée pour indiquer à l'optimiseur de boucle de dérouler une boucle particulière. Cela signifie que l'optimiseur de boucle va remplacer la boucle avec une série d'instructions répétées statiquement, éliminant ainsi les opérations de boucle telles que l'incrémentation du compteur de boucle et la vérification de la condition de sortie. Cela peut améliorer les performances en réduisant le nombre d'opérations de saut et en permettant une optimisation plus efficace des instructions à l'intérieur de la boucle.

#pragma simd pour générer des instructions SIMD (Single Instruction, Multiple Data) pour une boucle particulière. Les instructions SIMD permettent d'exécuter plusieurs opérations arithmétiques ou logiques sur des données simultanément en utilisant un seul appel d'instruction.

Implémentation :

```
#pragma GCC unroll 8
#pragma GCC ivdep
        //24 floating-point operations
        for (u64 j = 0; j < n; ++j) {
            //Newton's law
            #pragma simd
            const f32 dx = p->x[j] - dxi; //1 (sub)
            const f32 dy = p->y[j] - dyi; //2 (sub)
            const f32 dz = p->z[j] - dzi; //3 (sub)

            const f32 d_2 = 1.0f / sqrtf((dx * dx) + (dy * dy) + (dz * dz) + softening); // 10 (mul, add, div)
            const f32 d_3_over_2 = d_2 * d_2 * d_2; // 12 (mul)
```

Figure 8: Implementation unroll

Performance :

Compilateur	Time (s)	GFLOP/s
gcc	1.403e-01	45.9
clang	4.170e-01	15.4

Table 6: Performance Unroll

flag d'optim: -march=native -g3 -mtune=native -Ofast -fopenmp -msse2 -funroll-all-loops -floop-interchange

5. Vectorization (AVX)

On a vectorisé manuellement le programme avec les instruction AVX2.

Performance :

Compilateur	Time (s)	GFLOP/s
gcc	9.272e-02	57.9
clang	9.211e-02	58.3

Table 7: Performance AVX

On constate une meilleur amélioration pour l'instant du programme compilé avec gcc et clang.
flag d'optim: -march=native -g3 -mtune=native -Ofast -fopenmp -mavx2 -ftree-vectorize -ftree-loop-vectorize

6. Parallélisation

En parallélisant notre programme par la directive OpenMP **"#pragma omp parallel for"** qui indique au compilateur de paralléliser la boucle for qui la suit.

Implémentation :

```
#pragma omp parallel for
for (u64 i = 0; i < n; i++)
  {
    // ..........

  }
```

Figure 9: Parallel Implementation

Performance : Deux versions ont été élaboré l'une pour le code nbody_div et l'autre pour le code vectorisé en AVX2

Compilateur	Time (s)	GFLOP/s
gcc	2.504e-02	260.1
clang	2.468e-02	263.9

Table 8: Performance Parallel

Compilateur	Time (s)	GFLOP/s
gcc	1.663e-02	336.7
clang	1.675e-02	336.0

Table 9: Performance Parallel AVX

On remarque une augmentation des performances 6 fois mieux que les versions avant parallélisation pour les deux compilateurs, car les itérations de la boucle seront exécutées en parallèle par différents threads, ce qui améliore fortement les performances du programme. Les threads sont créés par la bibliothèque d'exécution OpenMP et sont gérés par le compilateur.

flag d'optim: -march=native -g3 -mtune=native -Ofast -fopenmp -mavx2 -funroll-loops -ftree-vectorize -ftree-loop-vectorize

Performance profiling

Dans cette section on va présenter les résultats de performances des versions qui ont des changement remarquable au niveau du Gflops en utilisant les différents caches L1, L2, L3 et compilés avec GCC.

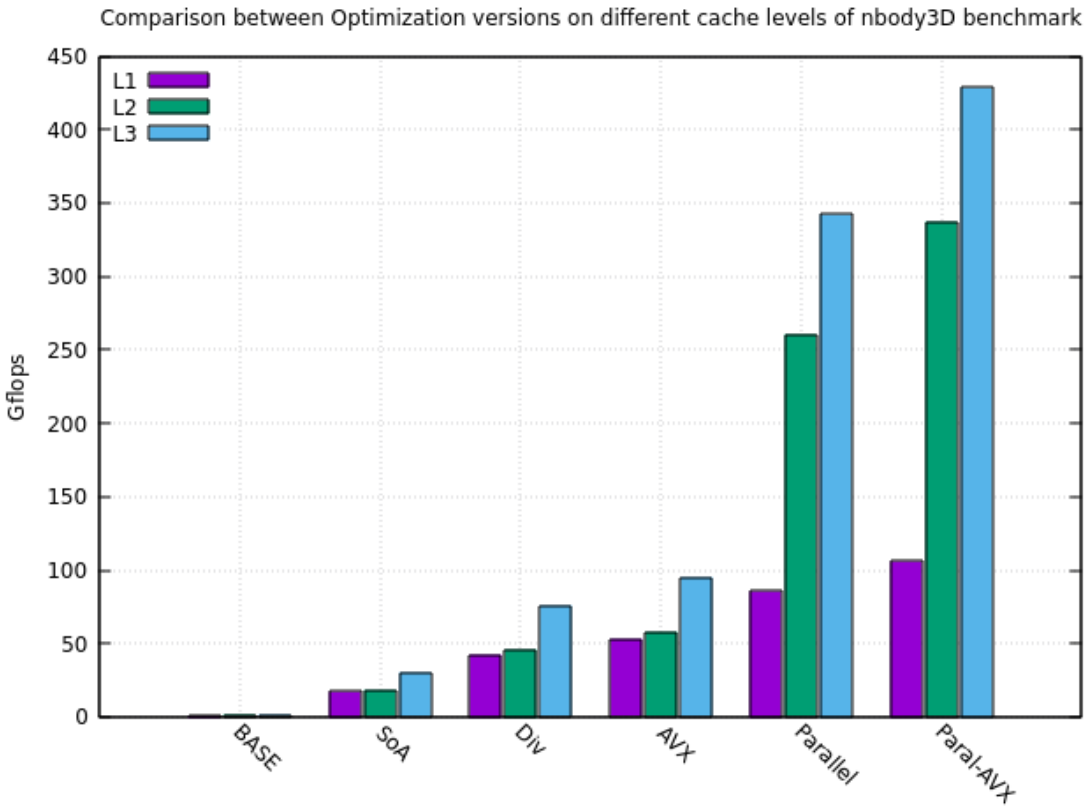

Figure 10: Comparaison entre les optimisations sur des différents niveaux de cache

De ce que nous observons sur le graphique de performance (exprimé en nombre de Gflops) ci-dessus, les optimisations les plus efficaces sont celles de la parallélisation. Notez que ces valeurs de performance ont été obtenues avec le compilateur GCC en utilisant le flag d'optimisation -Ofast sauf pour la version de base où nous avons utilisé le flag -O1. En second lieu, nous trouvons la version en assembleur AVX écrite à la main.