Univeristé Paris-Saclay

ARCHITECTURE PARALLÈLE

Optimisation de programme :

nBody 3D

Auteur :
LOZES Benjamin

3 avril 2022

Table des matières

0 Introduction

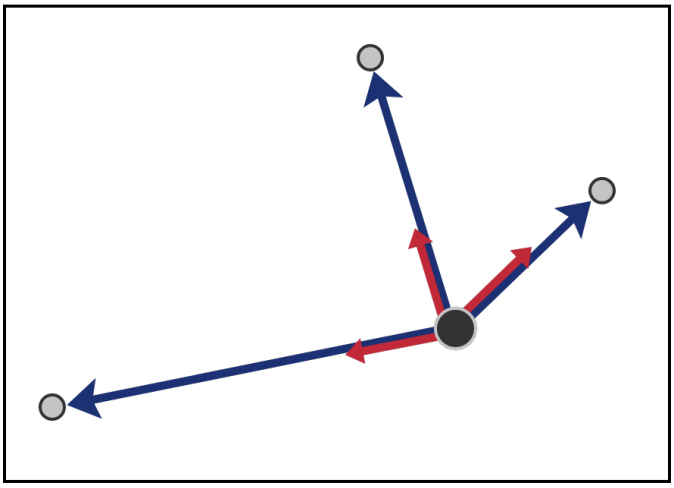

FIGURE 1 – Visualisation d'une itération nBody pour le point noir

Le programme qui nous a été donné consiste à simuler le déplacement de points dans un espace à 3 dimensions selon une seule règle physique :

"Chaque point est attiré par l'ensemble de ses voisins avec la même force"

Explication d'après 1 : L'objectif est de faire varier la position de chaque points en lui appliquant la somme des vecteurs normalisés (en rouge) de ce même point vers tous les autres de l'espace.

Ce programme sera implémenté et optimisé en C uniquement, bien qu'une majeure partie du travail d'optimisation consistera à l'analyse du code assembleur généré.

Tous les résultats concernant la performance qui suivront seront en milli-secondes ($temps_{total} - temps_{warmup}$) et en GFLOPS (avec adaptation du nombre d'opérations) et seront le résultat de la moyenne sur 10 itérations de la fonction de calcul. Toutes les mesures ont été réalisées sur une machines Knights Landing (KNL) 5.1 avec les compilateurs gcc et icc, les options seront également précisées en bas de la page.

1 Mémoire et caches

1.1 Array of Structures (AoS)

Le programme initial qui nous a été soumis implémentait une structure de stockage des points et de leur vélocité en AoS (cf 13).

Disponible ici.

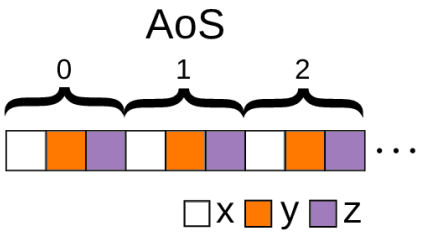

FIGURE 2 – Schématisation de la structure AoS [2]

Cette manière de structure la mémoire est certes très intuitive mais peut causer d'importantes latences pour les accès à la mémoire cache du processeur. En effet, chaque requête manquée (miss) entraînera un déplacement dans le cache processeur le plus proche la structure à l'indice désiré ainsi que nombre de ses voisins d'indice supérieur.

Cette structure pourrait alors nous avantager si nous cherchions à utiliser toutes les composantes de la structure {x, y, z, vz, vy, vz} avant de passer à la structure voisine. Or, ce n'est pas le cas pour ce programme ; nous utiliserons en majorité les 3 valeurs de position {x, y, z} pour tous les points (et donc tous les indices) avant d'écrire dans leurs vélocités {vx, vy, vz} respectives (cf 14).

En utilisant la structure AoS, nous déplaçons donc dans le cache les trois valeurs de vélocités qui ne seront pas utilisées, et ceux n^2 fois successivement.

	ms	gflops	page-faults
gcc [1]	66 300	0.7	180
icc [2]	4 361	9.9	2 330

TABLE 1 – Performance du programme initial (AoS)

Je n'arrive pas à expliquer la supériorité de icc sur cette version, en explorant l'assembleur généré, on voit que le programme est resté en AoS, il a seulement été vectorisé par endroits.

1. gcc -march=native -mavx2 -Ofast -lm
2. icc -xhost -Ofast -qmkl

1.2 Structure of Arrays (SoA)

Afin de résoudre le problème d'accès mémoire relevé à la section précédente, on se propose de ré-implémenter le programme avec une structure mémoire SoA (cf 15).

Implémentation disponible ici.

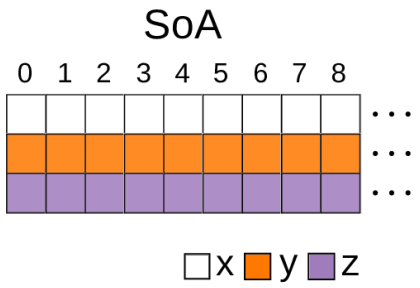

FIGURE 3 – Schématisation de la structure Soa [2]

	ms	gflops	page-faults
gcc [1]	6 231	6.9	179
icc [2]	4 103	10.5	2 330

TABLE 2 – Performance du programme initial (SoA)

Nous remarquons immédiatemment une nette amélioration pour le programme compilé avec gcc, le programme avec icc en revanche ne varie que très peu.

1. gcc -march=native -mavx2 -Ofast -lm
2. icc -xhost -Ofast -qmkl

1.3 Allocation

Afin de maximiser les chances d'avoir nos données en cache, il est capital d'adapter notre allocation mémoire. La méthode la plus efficace est de forcer l'alignement des données en mémoire, alignement qui sera fait selon les caractéristiques de notre machine.

Implémentation disponible ici.

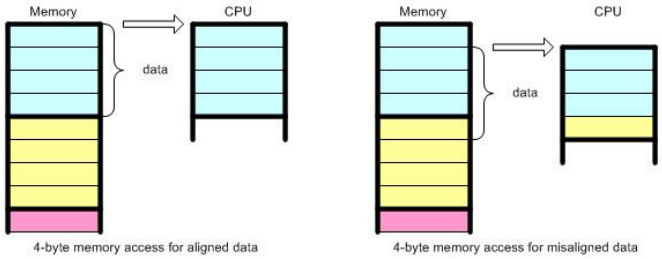

FIGURE 4 – Schématisation de l'alignement mémoire [1]

On se propose d'aligner nos tableaux en block de 64B (cf 5.1).

```
const u64 alignment = 64;

    particles_t p;
    p.x = aligned_alloc(alignment, sizeof(f32) * n);
    p.y = aligned_alloc(alignment, sizeof(f32) * n);
    p.z = aligned_alloc(alignment, sizeof(f32) * n);

    p.vx = aligned_alloc(alignment, sizeof(f32) * n);
    p.vy = aligned_alloc(alignment, sizeof(f32) * n);
    p.vz = aligned_alloc(alignment, sizeof(f32) * n);
```

FIGURE 5 – Initialisation alignée des données

	ms	gflops	page-faults
gcc [3]	5 906	7.3	123
icc [4]	3 919	11.0	2 333

TABLE 3 – Performance du programme après alignement mémoire (64B)

On constate une légère amélioration de la performance, notamment grâce au passage en instruction de chargement vectorielle alignées (*loadups* $\longrightarrow$ *loadps*).

3. gcc -march=native -mavx2 -Ofast -lm
4. icc -xhost -Ofast -qmkl

2 Algorithmie

2.1 Typage

Dans cette sous-section, nous allons nous concentrer sur une vérification manuelle des calculs réalisés et des instructions qui le composent.
Implémentation disponible ici.

```
// const f32 softening = 1e-20;
const f32 softening = 1e-20f;
```

FIGURE 6 – Suppresion du cast de f64 à f32

```
// const f32 d_3_over_2 = pow(d_2, 3.0 / 2.0);
const f32 d_3_over_2 = powf(d_2, 3.0f / 2.0f);
```

FIGURE 7 – Suppresion du cast de f64 à f32 + Utilisation de pow() pour un type f32

	ms	gflops
gcc [1]	1 756	24.6
icc [2]	1 436	30.1

TABLE 4 – Performance après correction des types

Augmentation des performances entre 150% et 200%, simplement en lisant la documentation C.

1. gcc -march=native -mavx2 -Ofast -lm
2. icc -xhost -Ofast -qmkl

2.2 Adaptation des calculs

Nous allons chercher à voir si certains calculs ou fonctions ne serait pas réalisés de façon incorrecte.
Implémentation disponible ici.

```
// const f32 d_3_over_2 = powf(d_2, 3.0f / 2.0f);
const f32 d_3_over_2 = d_2 * sqrtf(d_2);
```

FIGURE 8 – $a^{\frac{3}{2}} \equiv a * \sqrt{a}$

	ms	gflops
gcc [3]	1 333	32.4
icc [4]	1 073	40.3

TABLE 5 – Performance après correction des types

Cette transformation vise surtout à faciliter la vectorisation pour le compilateur, en effet il est non-seulement plus rapide d'invoquer une racine carrée qui est encodée dans le processeur contrairement à la puissance, mais elle est aussi largement vectorisable.

3. gcc -march=native -mavx2 -Ofast -lm
4. icc -xhost -Ofast -qmkl

En explorant la documentation sur les instruction assembleurs de l'AVX512, on trouve qu'il existe rsqrt[ps-pd] qui calculerait l'inverse d'une racine carée. Sachant que nous appliquons une division à cette même racine dans notre programme, il pourrait être intéressant d'adapter nos calculs.

Implémentation disponible ici.

```
// const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening;
// const f32 d_3_over_2 = d_2 * sqrtf(d_2);

// // Net force
// fx += dx / d_3_over_2;
// fy += dy / d_3_over_2;
// fz += dz / d_3_over_2;

const f32 d_2 =
    1.0f / sqrtf((dx * dx) + (dy * dy) + (dz * dz) + softening);
const f32 d_3_over_2 = d_2 * d_2 * d_2;

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

FIGURE 9 – $\frac{x}{a} \equiv x * \frac{1}{a}$

	ms	gflops
gcc [1]	1 331	34.0
icc [2]	1 073	42.0

TABLE 6 – Performance du programme calcul de la racine inverse

On note une très légère amélioration, en vérifiant sur CompilerExplorer, voit très clairement que la racine a été parfaitement vectorisée par l'instruction $vrsqrt28ps$ qui calcule une l'inverse d'une racine carrée avec 28 chiffres décimaux de précision. Notons qu'ajouter l'option $-fimf-precision = low$ ne nous permet pas d'accéder à l'instruction $vrsqrt14ps$ (14 chiffres décimaux) qui aurait été plus adaptée pour du calcul de float.

1. gcc -march=native -mavx2 -Ofast -lm
2. icc -xhost -Ofast -qmkl

3 Génération du programme

3.1 Inline

L'inlining consiste a copier le code interne d'une fonction dans une autre afin de ne pas avoir à effectuer des instructions de type *jump*.
Pour ce faire, nous pouvons soit utiliser les mots clés *static inline* qui sont censés être reconnus par la plupart des compilateurs, sinon l'option pragma forceinline recursive nous permet de forcer le compilateur à déplcaer le code la fonction selectionnée et toutes ses fonctions appelées en interne.

Implémentation disponible ici.

```
static inline void move_particles(...) {
    // ...
}

//
int main() {
    // ...

    for (...) {
        // ...

        #pragma forceinline recursive
        move_particles(...);

        // ...
    }
    // ...
}
```

FIGURE 10 – Inlining de *move_particles* (naturelle et forcée)

	ms	gflops
gcc [3]	1 290	35.0
icc [4]	1 073	42.0

TABLE 7 – Performance avec alignement forcé de la fonction *move_particles*

On note une très légère amélioration avec gcc (2 GFLOPS), icc quant-à-lui appliquait déjà l'inlining sans qu'on le lui spécifie.

3. gcc -march=native -mavx2 -Ofast -lm
4. icc -xhost -Ofast -qmkl

3.2 Déroulage de boucle

Le déroulage de boucles consiste a dupliquer une section interne à une boucle N fois en augmentant de façon statique l'indice. Ainsi les section {n, n+1, ..., n+N} s'effectuent de façon contigues sans appel à un quelconque *jump* ou *cmp*.

Implémentation disponible ici.

```
static inline void move_particles(...) {
    // ...
    #pragma unroll
    for (...) {
        // ...

        #pragma unroll
        for (...) {
            // ...
        }

        // ...
    }

    #pragma unroll
    for (...) {
        // ...
    }
}
```

FIGURE 11 – Déroulage des boucles de *move_particles*

	ms	gflops
gcc [1]	1 232	36.6
icc [2]	1 033	43.6

TABLE 8 – Performance avec déroulage des boucles de la fonction *move_particles*

Selon l'architecture, le compilateur va dérouler nos boucles d'un facteur adapté, ici il utilise un déroulage de 16, qui est le facteur maximal utilisable avec les directives *pragmaunroll*.
Nb : la directive *pragma unroll_and_jam* donne des résultats qui sont au mieux identiques et au pire légèrement inférieurs (env .5 GFLOPS) à cause du fait que les instructions de (dé-)chargement vectorielles ne peuvent pas être ré-organisées pour s'effectuer de manière concurrente.

1. gcc -march=native -mavx2 -Ofast -lm
2. icc -xhost -Ofast -qmkl

3.3 Découpage en block

Une autre méthode pour optimiser nos boucles est le découpage en block afin d'améliorer la réutilisation des données en cache. Néanmoins, d'après la documentation Intel, notre programme serait peu adapté à un passage en block à cause des instructions entre nos deux boucles i et j qui empêcherait de correctement découper le code.

Essayons tout de même !

Ne sachant pas tout à fait comment décider de la taille des blocks, j'ai décidé d'essayer avec des tailles multiples de 1024 jusqu'à arriver à un résultat satisfaisant.

Implémentation disponible ici.

```
#define BLOCK_FACTOR 8192

static inline void move_particles(...) {
    // ...

    #pragma block_loop factor(BLOCK_FACTOR) level(1)
    #pragma block_loop factor(BLOCK_FACTOR*2) level(2)
    for (u64 i = 0; i < n; i++) {
        // ...

        for (u64 j = 0; j < n; j++) {
            // ...
        }

        // ...
    }

    // ...
}
```

FIGURE 12 – Découpage en block des deux boucles de *move_particles*

	ms	gflops
gcc [3]	1 232	36.6
icc [4]	960	47.0

TABLE 9 – Performance avec déroulage des boucles de la fonction *move_particles*

Avec une taille de block de 8192 (soit $1024 * 16$), on arrive à une légère amélioration avec icc, gcc quant à lui ne varie absolument pas.
On passe également sous la barre de la seconde avec intel !

3. gcc -march=native -mavx2 -Ofast -lm
4. icc -xhost -Ofast -qmkl

3.4 Entraînement du compilateur

La dernière piste d'optimisation "traditionnelle" que je vais présenter consiste simplement à entraîner notre compilateur sur notre programme, en le lançant plusieurs fois et en générant des rapports, afin que le compilateur puisse par la suite voir quelles branchements étaient inutiles ou simplement mal-appréhendés.

L'option nous permettant de dire à notre programme de générer des rapports est :
— gcc : *-fprofile-gen=nom_du_profil*
— icc : *-prof-gen*
L'option nous permettant de dire au compilateur de tenir compte dans rapports est :
— gcc : *-fprofile-use=nom_du_profil*
— icc : *-prof-use*
On se propose d'essayer ces options sur notre meilleure version actuelle (cf 12) et d'entraîner nos programmes 5 fois chacun.
Implémentation disponible ici.

	ms	gflops
gcc [1]	1 193	37.8
icc [2]	960	47.0

TABLE 10 – Performance avec profilage

On note une petite amélioration pour gcc, néanmoins intel ne varie pas du tout.

1. gcc -march=native -fprofile-use=profile.gcc -Ofast -lm
2. icc -xhost -Ofast -prof-use -qmkl

4 Bonus

4.1 Fast Inverse Square Root

La fonction Fast Inverse Square Root (ou fisqrt) pourrait être utilisée, elle permettrait de gagner en performance sur un programme purement scalaire. Néanmoins, notre précision numérique serait réduite à néant au bout de 6 itérations, elle ne sera donc pas présentée ici.
Plus d'informations sur ce sujet ici.

Implémentation disponible ici.

4.2 Double interaction

On se propose d'optimiser légèrement les calculs en ne calculant qu'une seule fois la distance entre deux points. Réduisant ainsi la complexité de la première double-boucle de *move_particles* de n^2 à $n\frac{n-1}{2}$ mais doublant le nombre d'accès mémoire sur les vélocités.
Implémentation disponible ici.

	ms	gflops
gcc [3]	1 451	38.8
icc [4]	1 313	42.9

Table 11 – Performance avec la double interaction

La performance s'avère donc être plutôt correcte grâce à une gestion des caches appropriées. Il serait néanmoins sans doute utile de transformer les deux premières boucles en blocks manuellement car le compilateur n'en est pas capable.
nb : *-qopt-prefetch=5* à été utilisée pour icc afin d'améliorer cette importante dépendance aux caches.

4.3 Vectorisation manuelle (AVX512)

J'ai cherché à vectoriser manuellement ce programme avec les intrinsèques Intel pour AVX512.
Le résultat était prévisible, je suis moins doué en vectorisation que le compilateur mais je tiens tout de même à signaler que ce travail me força à explorer la documentation Intel et me fit découvrir énormément de concepts et méthodes que je ne connaissais pas du tout.
Implémentation disponible ici.

	ms	gflops
gcc [5]	1 840	33.5
icc [6]	1 222	35.4

Table 12 – Performance avec les Instrisics __mm512

3. gcc -march=native -mavx2 -Ofast -lm
4. icc -xhost -Ofast -qopt-prefetch=5 -qmkl
5. gcc -march=native -mfma -mavx512f -mavx512er -mavx512cd -Ofast -ftree-vectorize -funroll-all-loops -fassociative-math -lm
6. icc -xhost -Ofast -qmkl

4.4 Rotation de vecteur normalisé

L'idée était de changer les calculs en ne normalisant plus aucun vecteur mais en appliquant une rotation à un vecteur pré-normalisé afin d'en calculer les composants {x, y, z}.
Cette méthode aurait certes l'avantage de supprimer le calcul de la racine carrée, mais serait selon toute probabilité moins efficace car nous devrions calculer une rotation, donc utiliser une matrice 3x3 de sinus et de cosinus, fonctions qui sont peu ou pas du tout vectorisables, et surtout plus nombre qu'une unique racine carrée.

Je n'ai eu cette idée que très récemment et n'ai pas eu le temps de l'implémenter.

5 Spécifications

5.1 Cluster

— Serveur knl03 : Intel Knights Landing

Modèle : Intel(R) Genuine Intel(R) CPU 0000 @ 1.30GHz
Architecture : x86_64
Fréquence : 1400.00 MHz
Taille totale du cache : 1024 KB
Taille cache L1d : 2MiB ⟶ 2097 KB
Taille cache L2 : 32MiB ⟶ 33 554 KB
Attributs importants : sse, sse2, ssse3, fma, sse4, avx, avx2, avx512f, avx512pf, avx512er, avx512cd
Alignement du cache : 64

6 Bibliographie

Références

Ahn, S. H. (2005-2012). *Data alignment.* Consulté sur https://www.songho.ca/misc/alignment/dataalign.html

Tang, Y.-H., & Karniadakis, G. (2013, 11). Accelerating dissipative particle dynamics simulations on gpus : Algorithms, numerics and applications. *Computer Physics Communications*, *185*. doi: 10.1016/j.cpc.2014.06.015

7 Annexes

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

} particle_t;
```

FIGURE 13 – Programme initial (AoS) : Structure

```c
void move_particle (particle_t *p, const f32 dt, u64 n) {
    const f32 softening = 1e-20;

    for (u64 i = 0; i < n; i++) {
        f32 fx = 0.0;
        f32 fy = 0.0;
        f32 fz = 0.0;

        for (u64 j = 0; j < n; j++) {
            // Newton's law
            const f32 dx = p[j].x - p[i].x;
            const f32 dy = p[j].y - p[i].y;
            const f32 dz = p[j].z - p[i].z;
            const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening;
            const f32 d_3_over_2 = pow(d_2, 3.0 / 2.0);

            // Net force
            fx += dx / d_3_over_2;
            fy += dy / d_3_over_2;
            fz += dz / d_3_over_2;
        }


        p[i].vx += dt * fx;
        p[i].vy += dt * fy;
        p[i].vz += dt * fz;
    }

    for (u64 i = 0; i < n; i++) {
        p[i].x += dt * p[i].vx;
        p[i].y += dt * p[i].vy;
        p[i].z += dt * p[i].vz;
    }
}
```

FIGURE 14 – Programme initial (AoS) : Calcul

```c
typedef struct particles_s {
    f32 *x, *y, *z;
    f32 *vx, *vy, *vz;
} particles_t;
```

FIGURE 15 – Programme initial (SoA) : Structure

```c
void move_particle (particles_t p, const f32 dt, u64 n) {
    const f32 softening = 1e-20;

    for (u64 i = 0; i < n; i++) {
        f32 fx = 0.0;
        f32 fy = 0.0;
        f32 fz = 0.0;

        for (u64 j = 0; j < n; j++) {
            // Newton's
            const f32 dx = p.x[j] - p.x[i];
            const f32 dy = p.y[j] - p.y[i];
            const f32 dz = p.z[j] - p.z[i];
            const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening;
            const f32 d_3_over_2 = pow(d_2, 3.0 / 2.0);

            // Net
            fx += dx * d_3_over_2;
            fy += dy * d_3_over_2;
            fz += dz * d_3_over_2;
        }

        p.vx[i] += dt * fx;
        p.vy[i] += dt * fy;
        p.vz[i] += dt * fz;
    }

    for (u64 i = 0; i < n; i++) {
        p.x[i] += dt * p.vx[i];
        p.y[i] += dt * p.vy[i];
        p.z[i] += dt * p.vz[i];
    }
}
```

FIGURE 16 – Programme initial (SoA) : Calcul