Rapport du Projet de l'architecture parallèle

Yutai ZHAO

I. Introduction

a) Equation de Newton

En physique, la simulation à N-corps consiste vise à reproduire le comportement de N particules soumises à l'influence de forces telles que la gravité, le magnétisme, l'électromagnétisme, etc. Le code fourni implémente une simulation « simplifiée » de l'interaction Newtonienne de plusieurs particules dans un espace 3D en utilisant la loi gravitationnelle de Newton dont l'équation est la suivante :

$$F_{i,j} = \sum_{i \neq j}^{n} \frac{(p_i - p_j)}{(|(p_i - p_j)^3| + e)^{\frac{3}{2}}}$$

b) Objectif

Le projet consiste à analyser et optimiser ce code de simulation en adaptant le code à une machine cible.
Les optimisations seront faites à partir de deux approches : de la compilation et du code source. Le code source sera modifié, selon les optimisations présentées dans les sections «Optimisation». Quelques expériences supplémentaires seront ainsi effectuées pour obtenir la meilleure combinaison des optimisations et les décisions plus détaillées lors de ces optimisations (ex : le nombre de déroulage), ces expériences seront justifiées dans la section « Optimisation : Combinaison et Compilation ». Nous testerons en plus les performances de ces expériences en variant les compilateurs (GCC et Clang) ainsi que les flags d'optimisations (-O2. -O3, -Ofast). En fin, tous les résultats seront analysés dans la section « Comparaison » sous forme de graphe.

II. Table de matières

III. Environnement

Le codage est réalisé sur deux machines : la machine personnelle et le cluster. Mais les compilations et les exécutions sont effectuées que sur hsw01, dont les informations sont fournies dessous :

OS	**CPU**	**Cores/socket**	**CPU max MHz**	**L1 size**	**L2 size**	**L3 size**
Arch Linux x86_64	Intel Xeon E5-2680	12	3300	768 KiB	6Mib	60Mib

Les compilateurs utilisées et leurs versions sont :

Compiler	**Version**
GCC	13.2.1
Llvm-clang	16.0.6

Optimisations

Dans les sections « optimisations » suivantes, nous surlignerons les optimisations au niveau du code. Les codes seront ainsi compilés avec que GCC avec des flags proposés -O2, -O3, -Ofast. Les résultats seront présentés dans un tableau, parce qu'il n'y a que 3 données, c'est très lisible et en même temps nous connaissons les valeurs exactes.
Les optimisations au niveau de a compilation seront discutés plus tard dans la dernière section «Optimisation : Combinaison et Compilation ». Dans cette section nous étudierons plus précisément les influences des compilateurs et de leurs flags d'optimisations ainsi que le code assembleur. Nous nous donnons d'abord les performances de la version initiale sans aucune optimisation au niveau du code source :

Flags d'optimisations (de GCC)	-O2	-O3	-Ofast
Average performance (en GFLOP/s)	0.4 +- 0.0	0.6 +- 0.0	2.4 +- 0.0

IV. Traitements des opérations couteuses

➢ Introduction

❖ Objectif :

Cette optimisation consiste à modifier les opérations les plus couteuses comme les divisions, racines carrées, puissances en les transformant en addition et en multiplication.
L'idée initiale est de transformer la division en une autre expression et utiliser une autre définition mathématique de la norme pour éviter la racine carrée, la puissance sera transformée sous forme de multiplications.

❖ Transformation :

Mais vu la faisabilité et après avoir eu la confirmation auprès du professeur, les modifications effectuées sont plus simples et sont les suivantes :

- Remplacer la puissance pow() par les multiplications et sqrt()
- Remplacer la division par le produit avec une inverse
- NB : le nombre d'opérations de « Innermost loop » passe de 17 à 18

```
/*Initial code*/
const f32 d_3_over_2 = pow(d_2, 3.0 / 2.0);
fx += dx / d_3_over_2;
fy += dy / d_3_over_2;
fz += dz / d_3_over_2;
```

```
/*Optimal code*/
const f32 d_3_over_2 = 1/(d_2 * sqrt(d_2)); //sqrt() sera remplacé par sqrtf() ultérieurement
fx += dx * d_3_over_2;
fy += dy * d_3_over_2;
fz += dz * d_3_over_2;
```

➢ Résultats

Flags d'optimisations (de GCC)	-O2	-O3	-Ofast
Average performance (en GFLOP/s)	2.0 +- 0.0	2.0 +- 0.0	2.0 +- 0.0

➢ Conclusion

Nous obtenons une augmentation de 400% pour -O2, 233% pour -O3 et une baisse de 16.7% pour -Ofast par rapport à la version initiale.

V. Restructuration de données et alignement

➢ Introduction

❖ Objectif :

Cette optimisation consiste à restructurer les données pour bénéficier d'un accès mémoire linéaire, aligné, autrement-dit, il s'agit passer d'un AOS (Array of structure) à SOA(Structure of array). Cela permet de favoriser la vectorisation automatique par les compilateurs. En général, un accès mémoire linéaire permet d'augmenter les hits de cache, un accès mémoire aligne permet de réduire le temps d'accès mémoire vu que par exemple si une donnée n'est pas alignée, les octets les plus élevés et les plus bas d'une donnée ne se trouvent pas dans le même mot mémoire, l'ordinateur doit diviser l'accès à la donnée en plusieurs accès mémoire.

❖ Transformation :

- Le variable P ne représente plus un tableau de structures contenant « x,y,z,vx,vy,vz », mais une structure contenant les tableaux de « x,y,z,vx,vy,vz » pour avoir un accès mémoire linéaire
- Ajout de mot clé « restrict » afin d'interdire tout accès ailleurs à la mémoire pointée par un pointeur spécifique
- Utiliser la fonction « aligned_alloc » pour allouer des mémoires alignés

```
/*Initial code*/
typedef struct particle_s {
  f32 x, y, z;
  f32 vx, vy, vz;
} particle_t;
//Allocation
particle_t *p = malloc(sizeof(particle_t) * n);
```

```
/*Optimal code*/
typedef struct particle_s {
  f32 *restrict x, *restrict y, *restrict z;
  f32 *restrict vx, *restrict vy, *restrict vz;
} particle_t;
//Allocations
u64 alignment = 32;
particle_t *restrict p = aligned_alloc(alignment,sizeof(particle_t));
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);
```

➢ Résultats

Flags d'optimisations (de GCC)	-O2	-O3	-Ofast
Average performance (en GFLOP/s)	2.1 +- 0.0	2.1 +- 0.0	4.0 +- 0.0

➢ Conclusion

Nous obtenons une augmentation de 5% pour -O2, 5% pour -O3 et 100% pour -Ofast par rapport à la version précédente. Nous obtenons une augmentation de 425% pour -O2, 250% pour -O3 et 66.7% pour -Ofast par rapport à la version initiale.

VI. Parallélisation OpenMP

➢ Introduction

❖ Objectif :

Cette optimisation consiste à faire paralléliser les calculs dans les boucles vu que peu importe quelles boucles, nous pouvons remarquer que tous les calculs sont indépendants, à savoir que les calculs qui se font à la 1ere itération sont indépendant des calculs de la 2eme itération. Le parallélisme est fait à l'aide de OpenMP, il permet de distribuer une partie de calculs sur un thread donc sur un processeur afin que les calculs ne se fassent pas tous sur qu'un seul processeur, mais sur plusieurs processeurs et parallèlement.

❖ Transformation :

Dans le cas ici, nous pouvons faire une parallélisation sur outer loop bouclé sur i ou sur inner loop bouclé sur j, nous avons choisi de faire une parallélisation sur outer loop, la raison est d'abord hypothétique puis empirique :

- Si la parallélisation se fait sur inner loop (avec tn threads et n itérations), chaque processeur est amené à faire :
  - n/tn itérations pour calculer la force selon la loi de Newton

  En fin n itérations sur qu'UN SEUL processeur pour mettre à jours « particle velocities : vx,vy,vz » :.
- Si la parallélisation se fait sur outer loop (avec tn threads et n itérations), chaque processeur est amené à faire :
  - n itérations pour calculer la force selon la loi de Newton
  - n/tn itérations pour mettre à jours « particle velocities : vx,vy,vz » .

Pensant que la différence entre (n+n/tn) = (1+1/tn)*n itérations sur un processeur et 1/tn*n itérations sur un processeur est vraiment négligeable ; sauf si tn est assez grande, ce qui n'est pas le cas ici. En plus, vu que dans le 1er cas nous avons encore n itérations à faire sur qu'un processeur pour la mise à jour des valeurs, nous pouvons faire l'hypothèse que la deuxième manière de parallélisation serait plus performante. Enfin, cela a été prouvé plus tard empiriquement par d'autres camarades en consultant leurs idées et leurs performances.

Ainsi, les modifications sont les suivantes :

- Ajout de « #pragma omp parallel proc_bind(spread) »: pour débuter le parallélisme en indiquant que les threads doivent être répartis de manière équilibrée sur les processeurs
- Ajout de « #pragma omp for nowait »: pour indiquer que les threads ne doivent pas attendre les uns les autres à la fin de la boucle parallèle

```
/*initial code*/
#pragma omp parallel proc_bind(spread)
for (u64 i = 0; i < n; i++)
    {…
   for (u64 j = 0; j < n; j++){…}
    …}
```

```
/*Optimal code*/
#pragma omp parallel proc_bind(spread)
{
#pragma omp for nowait
for (u64 i = 0; i < n; i++)
    {…
   for (u64 j = 0; j < n; j++){…}
    …}
}
```

➢ Résultats

Flags d'optimisations (de GCC)	-O2	-O3	-Ofast
Average performance (en GFLOP/s)	29.6 +- 2.7	35.5 +- 1.8	76.4 +- 1.8

NB : Dans la plupart des cas compilés avec GCC, nous obtenons au fait toujours un ensemble des résultats moins performants et un autre ensemble des résultats plus performants après la parallélisation, nous prenons toujours les plus performants !

➢ Conclusion

Nous obtenons une augmentation de 1309.5% pour -O2, 1590.5% pour -O3 et 1810% pour -Ofast par rapport à la version précédente. Nous obtenons une augmentation de 7300% pour -O2, 5816.7% pour -O3 et 3083.3% pour -Ofast par rapport à la version initiale.

VII. Utilisation de sqrtf()

➢ Introduction

❖ Objectif et Transformation

Cette optimisation consiste à utiliser une autre fonction pour calculer la valeur de racine carrée en remplaçant sqrt() par sqrtf()

```
/*Previous code*/
const f32 d_3_over_2 = 1/(d_2 * sqrt(d_2));
fx += dx * d_3_over_2;
fy += dy * d_3_over_2;
fz += dz * d_3_over_2;
```

```
/*Optimal code*/
const f32 d_3_over_2 = 1/(d_2 * sqrtf(d_2));
fx += dx * d_3_over_2;
fy += dy * d_3_over_2;
fz += dz * d_3_over_2;
```

➢ Résultats

Flags d'optimisations (de GCC)	-O2	-O3	-Ofast
Average performance (en GFLOP/s)	69.8 +- 5.0	116.9 +- 1.1	310.3 +- 74.0

➢ Conclusion

Nous obtenons une augmentation de 135.9% pour -O2, 229.3% pour -O3 et 306.2% pour -Ofast par rapport à la version précédente. Nous obtenons une augmentation de 17350% pour -O2, 19383.3% pour -O3 et 12829.2% pour -Ofast par rapport à la version initiale.

VIII. Déroulage et Blocking

➢ Introduction

❖ Objectif :

Cette optimisation consiste à faire le blocking et le déroulage. Le blocking garantit que les données utilisées dans la boucle restent dans le cache jusqu'à ce qu'elles soient réutilisées. Quant au déroulage, il permet de remplir le pipeline par les instructions données ainsi d'augmenter la bande passante

Il y a 3 boucles dans notre fonction « move_particles », nous choisissons pour l'instant inner loop calculant la force pour appliquer ces optimisations. En effet, pour blocking en particulier, le choix est limité : il est inutile de faire le blocking pour la dernière boucle « Positions update » vu qu'il n'y pas une réutilisation de données dans la boucle. Nous étudierons ultérieurement dans « Optimisation : Combinaison et Compilation » d'autres choix de déroulage.

❖ Transformation : Blocking

- Restructurer la deuxième fois les données : « fx,fy,fz » sont les tableaux dans la structure
- Éliminer la plupart de créations de variables
- NB : le nombre d'opérations de « Innermost loop » passe de 18 à 36

```
/*Optimal code*/
typedef struct particle_s {
    f32 *restrict x, *restrict y, *restrict z;
    f32 *restrict vx, *restrict vy, *restrict vz;
    f32 *restrict fx, *restrict fy, *restrict fz;
```

```
} particle_t;

//Blocking
#pragma omp parallel proc_bind(spread)
    {
#pragma omp for nowait
        //For all particles
        for (u64 i = 0; i < n; i++)
        {
            for (u64 j = 0; j < n; j+=T)
            {
                for (u64 jj = j; jj < fmin(j+T,n); jj++){ //T=256
                    //Newton's law: 36 FLOPs (Floating-Point Operations) per iteration
                    const f32 d = 1/(( (p->x[jj] - p->x[i])* (p->x[jj] - p->x[i])) + ((p->y[jj] - p->y[i]) *
(p->y[jj] - p->y[i])) + (( p->z[jj] - p->z[i]) * ( p->z[jj] - p->z[i])) + softening * sqrtf(( (p->x[jj] -
p->x[i])* (p->x[jj] - p->x[i])) + ((p->y[jj] - p->y[i]) * (p->y[jj] - p->y[i])) + (( p->z[jj] - p->z[i]) *
( p->z[jj] - p->z[i])) + softening));

                    //Calculate net force: 6 FLOPs
                    p->fx[i] += (p->x[jj] - p->x[i]) * d;
                    p->fy[i] += (p->y[jj] - p->y[i]) * d;
                    p->fz[i] += (p->z[jj] - p->z[i]) * d;
                }
            }

        }
    }
```

❖ **Transformation : Blocking**

- Éliminer la plupart de créations de variables
- Faire un déroulage de 7 fois
- NB : le nombre d'opérations de « Innermost loop » passe de 18 à 36

```
#pragma omp parallel proc_bind(spread)
    {
#pragma omp for nowait
        //For all particles
        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+=7)
            {
                //Newton's law: 36 FLOPs
```

```
            const f32 d1 = 1/(( (p->x[j] - p->x[i])* (p->x[j] - p->x[i])) + ((p->y[j] - p->y[i]) * (p->y[j] - p->y[i])) + (( p->z[j] - p->z[i]) * ( p->z[j] - p->z[i])) + softening * sqrtf(( (p->x[j] - p->x[i])* (p->x[j] - p->x[i])) + ((p->y[j] - p->y[i]) * (p->y[j] - p->y[i])) + (( p->z[j] - p->z[i]) * ( p->z[j] - p->z[i])) + softening)); //12

            //Calculate net force: 6 FLOPs
            fx += (p->x[j] - p->x[i]) * d1;
            fy += (p->y[j] - p->y[i]) * d1;
            fz += (p->z[j] - p->z[i]) * d1;

            const f32 d2 = 1/(( (p->x[j+1] - p->x[i])* (p->x[j+1] - p->x[i])) + ((p->y[j+1] - p->y[i]) * (p->y[j+1] - p->y[i])) + (( p->z[j+1] - p->z[i]) * ( p->z[j+1] - p->z[i])) + softening * sqrtf(( (p->x[j+1] - p->x[i])* (p->x[j+1] - p->x[i])) + ((p->y[j+1] - p->y[i]) * (p->y[j+1] - p->y[i])) + (( p->z[j+1] - p->z[i]) * ( p->z[j+1] - p->z[i])) + softening)); //12

            fx += (p->x[j+1] - p->x[i]) * d2;
            fy += (p->y[j+1] - p->y[i]) * d2;
            fz += (p->z[j+1] - p->z[i]) * d2;

            /*Analogiquement pour le reste*/

        }

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

- **Résultats : Blocking**

Flags d'optimisations (de GCC)	-O2	-O3	-Ofast
Average performance (en GFLOP/s)	89.0 +- 7.1	122.3 +- 16.3	124.8 +- 0.4

- **Conclusion**

Nous obtenons une augmentation de 27.5% pour -O2, 4.6% pour -O3 et une baisse de 59.8% pour -Ofast par rapport à la version précédente. Nous obtenons une augmentation de $2.2x10^4$ % pour -O2, $2x10^4$ % pour -O3 et $5x10^3$ % pour -Ofast par rapport à la version initiale.

- **Résultats : Déroulage**

Flags d'optimisations (de GCC)	-O2	-O3	-Ofast
Average performance (en GFLOP/s)	141.0 +- 1.8	250.7 +- 1.0	747.1 +- 195.9

➢ **Conclusion**

Nous obtenons une augmentation de 102% pour -O2, 114.5% pour -O3 et 140.8% pour -Ofast par rapport à la version précédente. Nous obtenons une augmentation de $3.5x10^4$ % pour -O2, $4.2x10^4$ % pour -O3 et $3.1x10^4$ % pour -Ofast par rapport à la version initiale.

IX. Combinaison et Compilation

➢ **Introduction**

❖ **Objectif :**

Grâce aux sections précédentes, nous pouvons remarquer qu'à priori la version la plus optimisée est celle avec le déroulage. Alors, dans cette section, déterminons-nous d'abord le meilleur nombre du déroulage pour la boucle « Innermost loop », puis si c'est intéressant de faire un autre déroulage pour la boucle « Positions update ».
Nous analysons en particulier, le code assembleur dans cette partie, vu que nous intéressons aux effets des compilateurs et de leurs flags.

❖ **Transformation**

- Essayer le nombre du déroulage = 4,6,7,8 pour la boucle « Innermost loop »
- Essayer de faire un déroulage x8 pour la boucle « Positions update »
- En fin, compiler ces versions avec GCC et Clang avec comme flags proposés -O2, -O3, -Ofast

➢ **Résultats : Déroulage x4 sur « Innermost loop » SANS Déroulage sur « Positions update »**

Flags d'optimisations (de GCC)	-O2	-O3	-Ofast
Average performance (en GFLOP/s)	225.6 +- 0.8	236.1 +- 3.3	610.0 +- 148.9
Flags d'optimisations (de Clang)	-O2	-O3	-Ofast
Average performance (en GFLOP/s)	79.1 +- 6.4	60.2 +- 4.4	223.7 +- 18.7

➢ **Conclusion : Déroulage x4 sur « Innermost loop » SANS Déroulage sur « Positions update »**

Analyse code assembleur :
GCC :
-O2 (https://godbolt.org/z/6fqrav693) : Pas de déroulage (dans .L4 : pas de répétition d'instructions et jb .L4) et Vectorisation de 4 éléments (xmm = 128 bits = 4x32bits et SIMD : movups, addps, hufps)
-O3 (https://godbolt.org/z/76M91Y991) : Déroulage x4 (Répétition 4 fois des mêmes instructions) et Vectorisation (SIMD : movups, addps, hufps)
-Ofast (https://godbolt.org/z/759G7zdzK) : Déroulage x4, Vectorisation et optimisations maths (par ex, selon llvm-mca(trunk) : pour -O2 : latence de divps + sqrtps = 11 +15 =26, or ici on a rsqrtps = 3 !!!)

Clang :
-O2 (https://godbolt.org/z/aW8W3Wxb3) : Déroulage x4 et pas de vectorisation (addss, divss, movss)
-O3 (https://godbolt.org/z/PPYa3E9x6) : Déroulage x4 et pas de vectorisation
-Ofast (https://godbolt.org/z/s9WsdoWEP) : Déroulage x4 et Vectorisation (SIMD : addps, movaps, mulps)

Pour les versions compilées avec GCC, nous obtenons une augmentation de $5.6x10^4$ % pour -O2, $3.9x10^4$ % pour -O3 et $2.5x10^4$ % pour -Ofast par rapport à la version initiale. Pour les versions compilées avec Clang, nous obtenons une augmentation de $2.0x10^4$ % pour -O2, $9.9x10^3$ % pour -O3 et $9.2x10^3$ % pour -Ofast par rapport à la version initiale.

- **Résultats : Déroulage x6 sur « Innermost loop » SANS Déroulage sur « Positions update »**

Flags d'optimisations (de GCC)	-O2	-O3	-Ofast
Average performance (en GFLOP/s)	216.4 +- 0.7	285.0 +- 1.8	622.0 +- 121.9
Flags d'optimisations (de Clang)	-O2	-O3	-Ofast
Average performance (en GFLOP/s)	62.0 +- 5.1	80.7 +- 5.8	136.2 +- 12.0

- **Conclusion : Déroulage x6 sur « Innermost loop » SANS Déroulage sur « Positions update »**

Analyse code assembleur : Analogique à la version précédente Déroulage x4 avec la même taille de vectorisation mais avec un déroulage de x6 fois quand cela a lieu.
GCC :
-O2 (https://godbolt.org/z/afod1nsxq),
-O3 (https://godbolt.org/z/1T9sh5T9f),
-Ofast (https://godbolt.org/z/bz7fP1s15)
Clang :
-O2 (https://godbolt.org/z/6jvvqsEKT),
-O3 (https://godbolt.org/z/3jf7M638a),
-Ofast (https://godbolt.org/z/Y34efcYrE)

Pour les versions compilées avec GCC, nous obtenons une augmentation de $5.4x10^4$ % pour -O2, $4.7x10^4$ % pour -O3 et $2.6x10^4$ % pour -Ofast par rapport à la version initiale. Pour les versions compilées avec Clang, nous obtenons une augmentation de $1.5x10^4$ % pour -O2, $1.3x10^4$ % pour -O3 et $5.6x10^3$ % pour -Ofast par rapport à la version initiale.

- **Résultats : Déroulage x7 sur « Innermost loop » SANS Déroulage sur « Positions update »**

Flags d'optimisations (de GCC)	-O2	-O3	-Ofast
Average performance (en GFLOP/s)	141.0 +- 1.8	250.7 +- 1.0	747.1 +- 195.9
Flags d'optimisations (de Clang)	-O2	-O3	-Ofast
Average performance (en GFLOP/s)	60.9 +- 4.6	80.3 +- 7.0	133.1 +- 16.0

- **Conclusion : Déroulage x7 sur « Innermost loop » SANS Déroulage sur « Positions update »**

Analyse code assembleur :
GCC :
-O2 (https://godbolt.org/z/86qTbEhx8) : Déroulage x7 (répétition d'instructions 7 fois) et pas de vectorisation
-O3 (https://godbolt.org/z/4as5n13ef) , -Ofast (https://godbolt.org/z/9heqGTdzx) : Analogique à la version Déroulage x4 avec la même taille de vectorisation mais avec un déroulage de x7 fois quand cela a lieu.

Clang : Analogique à la version Déroulage x4 avec la même taille de vectorisation mais avec un déroulage de x7 fois quand cela a lieu.
-O2 (https://godbolt.org/z/vhP1fjM3q),
-O3 (https://godbolt.org/z/M79hPofde),
-Ofast (https://godbolt.org/z/xbrTM8YK4)

Pour les versions compilées avec GCC, nous obtenons une augmentation de $3.5x10^4$ % pour -O2, $4.2x10^4$ % pour -O3 et $3.1x10^4$ % pour -Ofast par rapport à la version initiale. Pour les versions compilées avec Clang, nous obtenons une augmentation de $1.5x10^4$ % pour -O2, $1.3x10^4$ % pour -O3 et $5.4x10^3$ % pour -Ofast par rapport à la version initiale.

- **Résultats : Déroulage x8 sur « Innermost loop » SANS Déroulage sur « Positions update »**

Flags d'optimisations (de GCC)	-O2	-O3	-Ofast
Average performance (en GFLOP/s)	229.4 +- 3.1	229.0 +- 2.7	625.6 +- 126.8
Flags d'optimisations (de Clang)	-O2	-O3	-Ofast
Average performance (en GFLOP/s)	62.0 +- 5.0	80.2 +- 6.3	180.0 +- 0.5

- **Conclusion : Déroulage x8 sur « Innermost loop » SANS Déroulage sur « Positions update »**

Analyse code assembleur :
GCC :
-O2 (https://godbolt.org/z/3bGTj4E43) : Déroulage x2 (répétition d'instructions 2 fois) et Vectorisation
-O3 (https://godbolt.org/z/PW4vzehab), -Ofast (https://godbolt.org/z/ad1f8qvoc) : Analogique à la version précédente Déroulage x4 avec la même taille de vectorisation mais avec un déroulage de x8 fois quand cela a lieu.
Clang : Analogique à la version précédente Déroulage x4 avec la même taille de vectorisation mais avec un déroulage de x8 fois quand cela a lieu.
-O2 (https://godbolt.org/z/PGWoK74nq),
-O3 (https://godbolt.org/z/Ye9jYav39),
-Ofast (https://godbolt.org/z/TqfMzvoKT)

Pour les versions compilées avec GCC, nous obtenons une augmentation de $5.7x10^4$ % pour -O2, $3.8x10^4$ % pour -O3 et $2.6x10^4$ % pour -Ofast par rapport à la version initiale. Pour les versions compilées avec Clang, nous obtenons une augmentation de $1.5x10^4$ % pour -O2, $1.3x10^4$ % pour -O3 et $7.4x10^3$ % pour -Ofast par rapport à la version initiale.

- **Résultats : Déroulage x7 sur « Innermost loop » AVEC Déroulage x8 sur « Positions update »**

Flags d'optimisations (de GCC)	-O2	-O3	-Ofast
Average performance (en GFLOP/s)	150.1 +- 6.6	254.2 +- 0.9	732.3 +- 191.7
Flags d'optimisations (de Clang)	-O2	-O3	-Ofast
Average performance (en GFLOP/s)	60.7 +- 4.5	81.2 +- 6.8	135.9 +- 18.9

- **Conclusion : Déroulage x7 sur « Innermost loop » AVEC Déroulage x8 sur « Positions update »**

Analyse code assembleur :
GCC :
-O2 (https://godbolt.org/z/q979M9Kje) : Déroulage x7 et pas de vectorisation, pas de déroulage pour « Positions update »
-O3 (https://godbolt.org/z/nq3jd6fPr) : Déroulage x7 et vectorisation, pas de déroulage pour « Positions update »
-Ofast (https://godbolt.org/z/1s786Kooe) : : Déroulage x7 et vectorisation, pas de déroulage pour « Positions update », optimisations maths
Clang :
-O2 (https://godbolt.org/z/svTxP4KMa) : Déroulage x7 pour « Innermost loop », pas de vectorisation, Déroulage x8 pour « Positions update »
-O3 (https://godbolt.org/z/rKEf9as7c) : Déroulage x7 pour « Innermost loop », pas de vectorisation, Déroulage x8 pour « Positions update »
-Ofast (https://godbolt.org/z/veMh33boK) : Pas de déroulage pour « Innermost loop », vectorisation, Déroulage x8 pour « Positions update »

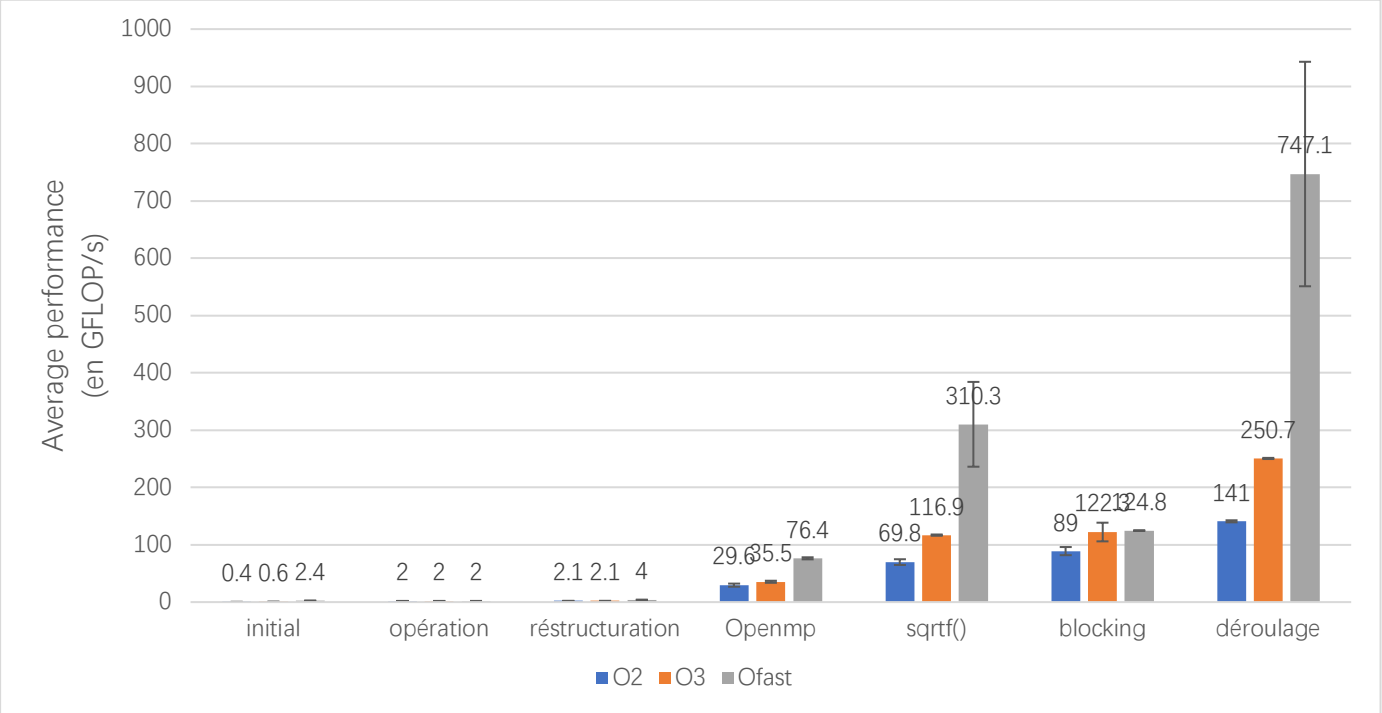
Pour les versions compilées avec GCC, nous obtenons une augmentation de $3.7x10^4$ % pour -O2, $4.2x10^4$ % pour -O3 et $3x10^4$ % pour -Ofast par rapport à la version initiale. Pour les versions compilées avec Clang, nous obtenons une augmentation de $1.5x10^4$ % pour -O2, $1.3x10^4$ % pour -O3 et $5.6x10^3$ % pour -Ofast par rapport à la version initiale.

X. Comparaison

Comme dans les sections d'optimisations, les comparaisons se font également par 2 parties, une comparaison avec toutes les optimisations faites jusqu'au déroulage, une autre comparaison qui se fait avec les différents choix de déroulage.

a) Comparaison entre différents types d'optimisations

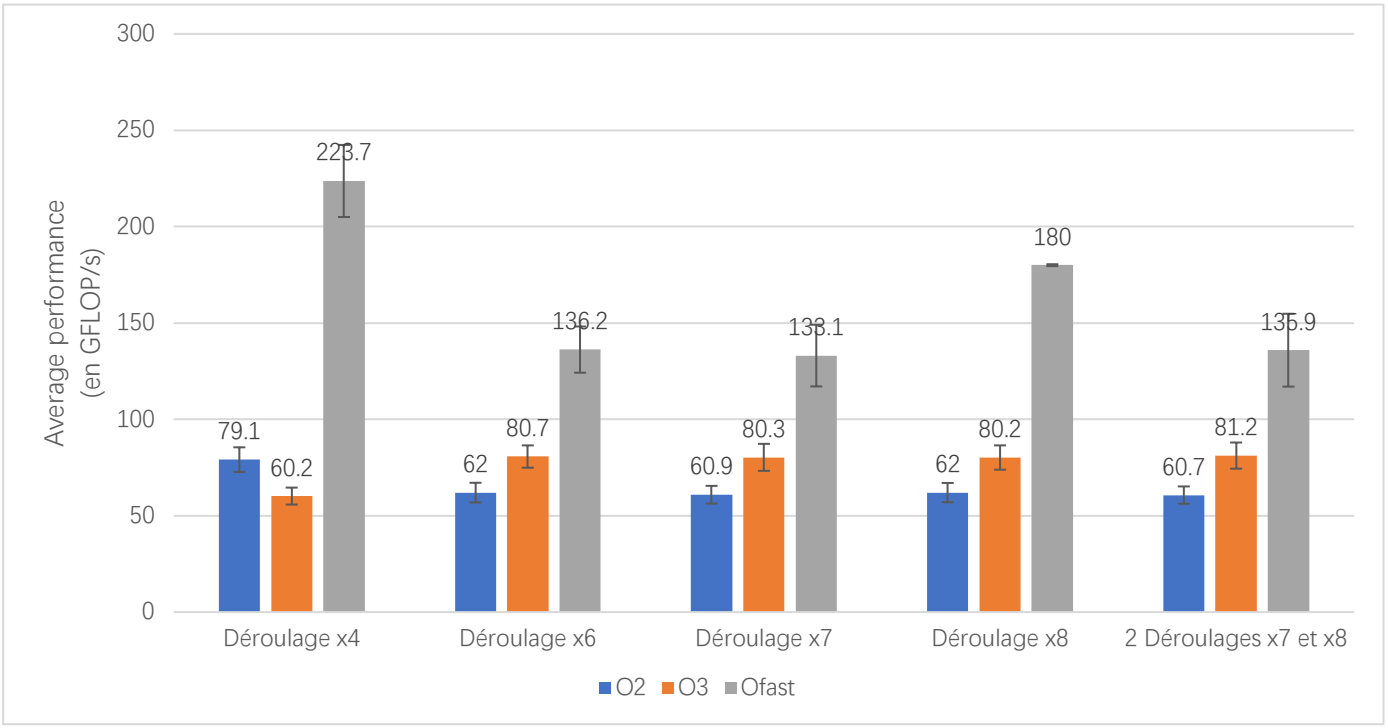
Différentes versions d'optimisations
avec une accumulation d'optimisations avant le blocking et déroulage



	initial	opération	réstructuration	Openmp	sqrtf()	blocking	déroulage
GCC -O2	0.4 +- 0.0	2.0 +- 0.0	2.1 +- 0.0	29.6 +- 2.7	69.8 +- 5.0	89.0 +- 7.1	141.0 +- 1.8
GCC -O3	0.6 +- 0.0	2.0 +- 0.0	2.1 +- 0.0	35.5 +- 1.8	116.9 +- 1.1	122.3 +- 16.3	250.7 +- 1.0
GCC -Ofast	2.4 +- 0.0	2.0 +- 0.0	4.0 +- 0.0	76.4 +- 1.8	310.3 +- 74.0	124.8 +- 0.4	747.1 +- 195.9

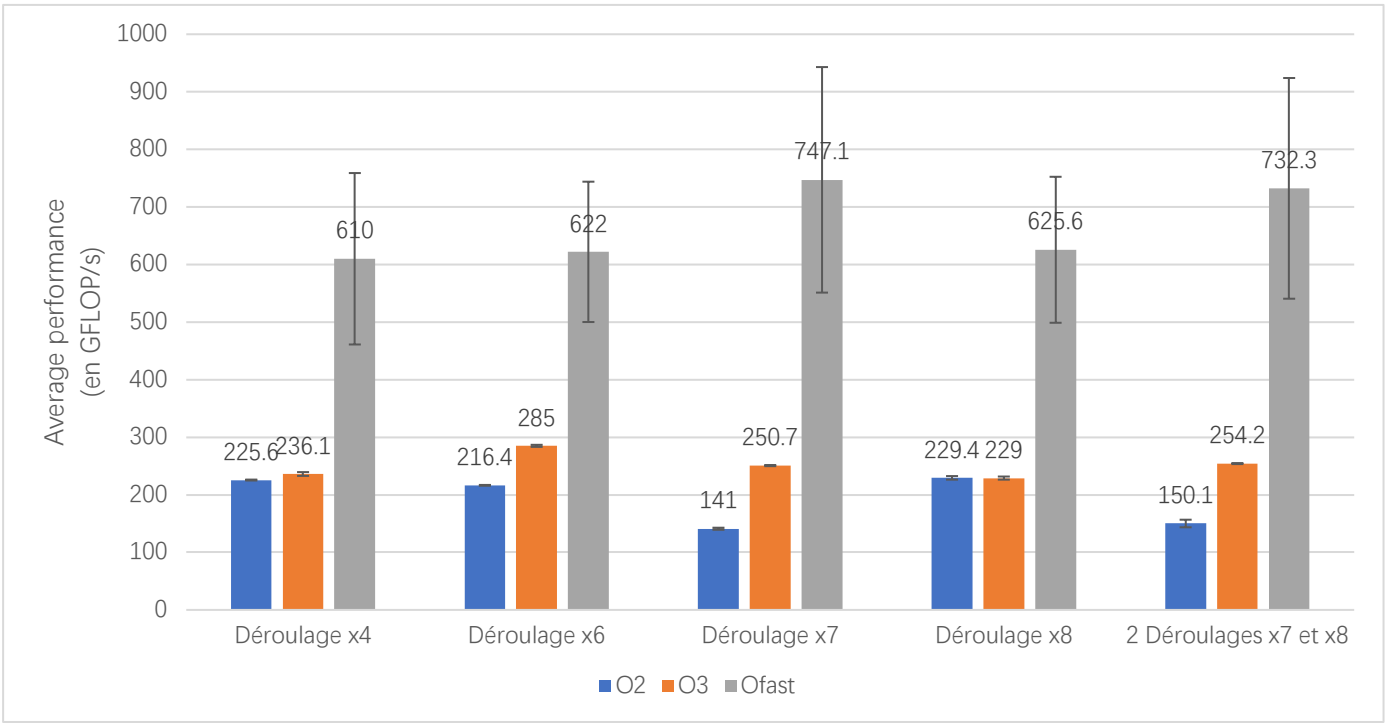
b) Comparaison entre différents choix de déroulage

Différentes versions de déroulages compilées par Clang



	Déroulage x4	Déroulage x6	Déroulage x7	Déroulage x8	2 Déroulages x7 et x8
Clang-O2	79.1 +- 6.4	62.0 +- 5.1	60.9 +- 4.6	62.0 +- 5.0	60.7 +- 4.5
Clang -O3	60.2 +- 4.4	80.7 +- 5.8	80.3 +- 7.0	80.2 +- 6.3	81.2 +- 6.8
Clang -Ofast	223.7 +- 18.7	136.2 +- 12.0	133.1 +- 16.0	180.0 +- 0.5	135.9 +- 18.9

Différentes versions de déroulages compilées par GCC



	Déroulage x4	Déroulage x6	Déroulage x7	Déroulage x8	2 Déroulages x7 et x8
GCC -O2	225.6 +- 0.8	216.4 +- 0.7	141.0 +- 1.8	229.4 +- 3.1	150.1 +- 6.6
GCC -O3	236.1 +- 3.3	285.0 +- 1.8	250.7 +- 1.0	229.0 +- 2.7	254.2 +- 0.9
GCC -Ofast	610.0 +- 148.9	622.0 +- 121.9	747.1 +- 195.9	625.6 +- 126.8	732.3 +- 191.7

XI. Conclusion

Nous avons d'abord commencé par l'étude de l'algorithme de Barnes-Hut comme à priori c'est l'optimisation la plus compliquée, mais après 2,3 jours d'essai et ne pas avoir pu l'implémenter, nous avons abandonné cette implémentation et nous sommes repartis sur l'optimisation des opérations coûteuses.

Nous avons suivi toutes les étapes proposées avant le déroulage et blocking comme il y a plusieurs possibilités de les faire, nous avons donc fait la parallélisation OpenMP d'abord vu que nous avions qu'un choix raisonnable dont l'explication est donnée dans la section correspondante. En fin nous avons remarqué que la version déroulée, parallélisée, restructurée et alignée et optimisée au niveau des opérations couteuses est la version la plus performante.

Il y a d'autres expériences réalisées au niveau du code source qui ne sont pas présentées dont les codes sources sont fournis, par exemple : utiliser 2 fois Openmp / tester d'autre nombre de threads pour la parallélisation / appliquer la nouvelle structure permettant le blocking au déroulage / faire un blocking sur la boucle « Position update » pour valider l'hypothèse disant qu'il est inutile de faire un blocking sur une telle boucle.

En testant d'autres choix de déroulages, nous pouvons déduire que la version déroulée x7 compilée par GCC -Ofast donne la meilleure performance et que ce n'est pas la peine de faire un déroulage manuel sur la boucle « Position update », comme nous retrouvons les performances très similaires.

En fin, tous les codes sources sont fournis dans les répertoires correspondants, tous les données brutes des versions présentées dans ce rapport sont fournies dans les répertoires « res » des répertoires de code.