Rapport Projet AP – N-BODY3D

Présenté par :

Hamza Amine DEMIGHA

Année universitaire : 2022/2023

Table des matières

I. Introduction :

Les simulations gravitationnelles à N-Body sont des calculs informatiques qui permettent de modéliser les mouvements et les interactions des corps célestes dans l'espace. Ces simulations utilisent des algorithmes pour simuler les forces gravitationnelles entre chaque corps, qui peuvent inclure des étoiles, des planètes, des comètes, des étoiles naines, etc. Les simulations N-Body sont utilisées pour étudier les systèmes stellaires, les galaxies et les amas de galaxies, ainsi que pour la prédiction des collisions potentielles entre corps célestes. Ces simulations sont très exigeantes en termes de puissance de calcul, car elles nécessitent de traiter des grandes quantités de données et de calculer de nombreuses interactions gravitationnelles en temps réel.

Dans ce rapport on va essayer optimiser le code du ce problème avec plusieurs versions.

II. Les Versions :

Afin d'optimiser le code, on a implémenté 7 versions :

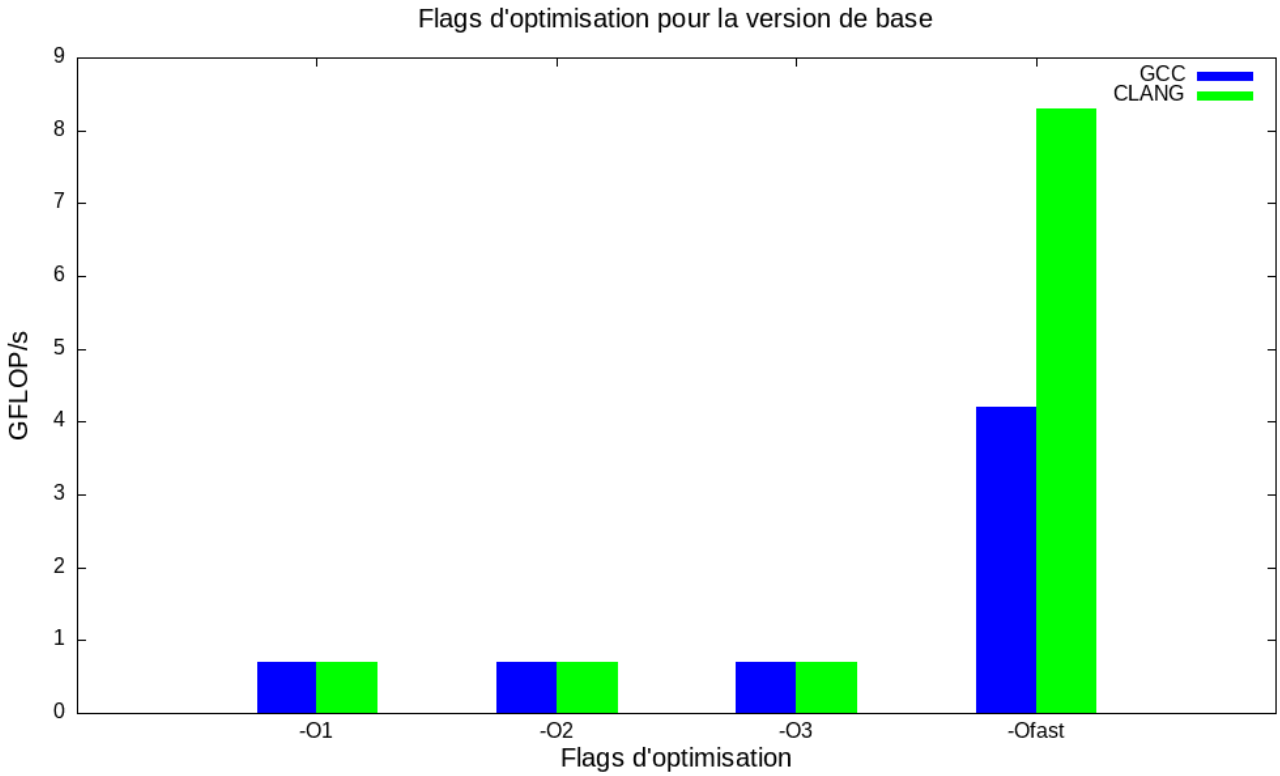
- Version 1 : Changer les flags d'optimisation pour la version de la base.
- Version 2 : Modifier la structure du programme AOS -> SOA.
- Version 3 : Changer l'alignement du mémoire.
- Version 4 : Modifier les instructions du calcul (sqrt, div, pow…).
- Version 5 : Ajouter le déroulage (Unroll).
- Version 6 : Vectorisation AVX.
- Version 7 : Vectorisation SSE.
- Version 8 : Parallélisation avec OpenMP.

1. Version 1 : Changer les flags d'optimisation pour la version de la base.

Pour les mesures des performances, on a utilisé 2 compilateurs GCC et CLANG.
Des flags d'optimisations ont été ajoutés : -Ofast au lieu de -O1.

```
minou@minou-ThinkPad-X250:~/Bureau/AP/nbody3D$ make run
taskset -c 3 ./nbody3D.g

Total memory size: 393216 B, 384 KiB, 0 MiB

 Step     Time, s  Interact/s  GFLOP/s
    0  1.435e+00  1.870e+08       4.3 *
    1  1.422e+00  1.888e+08       4.3 *
    2  1.431e+00  1.875e+08       4.3 *
    3  1.585e+00  1.694e+08       3.9
    4  1.550e+00  1.732e+08       4.0
    5  1.422e+00  1.887e+08       4.3
    6  1.433e+00  1.873e+08       4.3
    7  1.435e+00  1.870e+08       4.3
    8  1.495e+00  1.795e+08       4.1
    9  1.423e+00  1.887e+08       4.3
-----------------------------------------------------
Average performance:          4.2 +- 0.2 GFLOP/s
-----------------------------------------------------
taskset -c 3 ./nbody3D.cl

Total memory size: 393216 B, 384 KiB, 0 MiB

 Step     Time, s  Interact/s  GFLOP/s
    0  7.530e-01  3.564e+08       8.2 *
    1  7.598e-01  3.533e+08       8.1 *
    2  7.648e-01  3.510e+08       8.1 *
    3  7.499e-01  3.579e+08       8.2
    4  7.163e-01  3.747e+08       8.6
    5  7.972e-01  3.367e+08       7.7
    6  7.472e-01  3.592e+08       8.3
    7  7.212e-01  3.722e+08       8.6
    8  7.420e-01  3.618e+08       8.3
    9  7.236e-01  3.710e+08       8.5
-----------------------------------------------------
Average performance:          8.3 +- 0.3 GFLOP/s
-----------------------------------------------------
```

On remarque que la moyenne de performance a augmentée, pour GCC avec -O1 était 0.7 GFLOPS/S mais avec -Ofast ça devient 4.2 GFLOP/S. la même chose pour CLANG : 0.7 GFLOPS/S -> 8.3 GFLOPS/S.



2. Version 2 : Modifier la structure du programme AOS -> SOA.

La structure de tableau (SoA) est une méthode de stockage des données où les différents champs d'une structure sont stockés dans des tableaux séparés, plutôt que dans un seul tableau de structures. Cela permet d'accéder aux champs individuellement plus rapidement, car ils sont stockés contingemment en mémoire.

Le tableau de structures (AoS), en revanche, est une méthode de stockage des données où chaque élément d'un tableau est une structure contenant différents champs. Cela permet d'accéder aux différentes instances de la structure plus facilement, car elles sont stockées à des emplacements consécutifs en mémoire.

```
13  //
14  typedef struct particle_s {
15
16      f32 *x, *y, *z;
17      f32 *vx, *vy, *vz;
18  } particle_t;
19
```

```
102  //
103  particle_t p;
104  p.x = malloc(n * sizeof(f32));
105  p.y = malloc(n * sizeof(f32));
106  p.z = malloc(n * sizeof(f32));
107  p.vx = malloc(n * sizeof(f32));
108  p.vy = malloc(n * sizeof(f32));
109  p.vz = malloc(n * sizeof(f32));
110
```

Pour accéder aux éléments du notre structure, on change :
p[ ].x par **p.x[**].

```
        p.x[i] = sign * (f32)rand() / (f32)RAND_MAX;
        p.y[i] = (f32)rand() / (f32)RAND_MAX;
        p.z[i] = sign * (f32)rand() / (f32)RAND_MAX;

        //
        p.vx[i] = (f32)rand() / (f32)RAND_MAX;
        p.vy[i] = sign * (f32)rand() / (f32)RAND_MAX;
        p.vz[i] = (f32)rand() / (f32)RAND_MAX;
```

Grâce à cette optimisation, on a amélioré la performance.

```
taskset -c 3 ./nbody3D1.g

Total memory size: 786432 B, 768 KiB, 0 MiB

 Step     Time, s Interact/s  GFLOP/s
    0  6.807e-01  3.943e+08      9.1 *
    1  6.963e-01  3.855e+08      8.9 *
    2  6.786e-01  3.955e+08      9.1 *
    3  6.833e-01  3.928e+08      9.0
    4  7.014e-01  3.827e+08      8.8
    5  7.063e-01  3.800e+08      8.7
    6  6.979e-01  3.846e+08      8.8
    7  7.150e-01  3.754e+08      8.6
    8  6.825e-01  3.933e+08      9.0
    9  7.394e-01  3.630e+08      8.3
-----------------------------------------------------
Average performance:          8.8 +- 0.2 GFLOP/s
-----------------------------------------------------
taskset -c 3 ./nbody3D1.cl

Total memory size: 786432 B, 768 KiB, 0 MiB

 Step     Time, s Interact/s  GFLOP/s
    0  6.767e-01  3.967e+08      9.1 *
    1  6.700e-01  4.006e+08      9.2 *
    2  6.799e-01  3.948e+08      9.1 *
    3  6.753e-01  3.975e+08      9.1
    4  6.647e-01  4.038e+08      9.3
    5  6.824e-01  3.933e+08      9.0
    6  6.810e-01  3.941e+08      9.1
    7  6.657e-01  4.032e+08      9.3
    8  6.899e-01  3.890e+08      8.9
    9  6.881e-01  3.901e+08      9.0
-----------------------------------------------------
Average performance:          9.1 +- 0.1 GFLOP/s
-----------------------------------------------------
```

3. Version 3 : Changer l’alignement du mémoire.

La fonction aligned_alloc() permet d'allouer de la mémoire avec un alignement souhaité en nombre de octets, pour notre cas 64.

Le passage de la fonction d'allocation mémoire dynamique malloc() a la fonction aligned_ alloc() permet d'indiquer au compilateur d'allouer de la place dans des zones mémoires alignées les unes aux autres. Cela permet également d'accélérer l'exécution du programme, car lorsque le processeur récupère des informations en mémoire, il les récupère par blocs de données.

```
104     particle_t p;
105     const u64 al = 64;
106     p.x = aligned_alloc(al, n * sizeof(f32));
107     p.y = aligned_alloc(al, n * sizeof(f32));
108     p.z = aligned_alloc(al, n * sizeof(f32));
109     p.vx = aligned_alloc(al, n * sizeof(f32));
110     p.vy = aligned_alloc(al, n * sizeof(f32));
111     p.vz = aligned_alloc(al, n * sizeof(f32));
112
```

Grâce à cet alignement, on a amélioré un peu plus la performance.

```
taskset -c 3 ./nbody3D2.g

Total memory size: 786432 B, 768 KiB, 0 MiB

 Step    Time, s Interact/s  GFLOP/s
    0  6.735e-01  3.985e+08      9.2 *
    1  6.761e-01  3.970e+08      9.1 *
    2  6.778e-01  3.960e+08      9.1 *
    3  6.876e-01  3.904e+08      9.0
    4  6.753e-01  3.975e+08      9.1
    5  6.759e-01  3.971e+08      9.1
    6  6.799e-01  3.948e+08      9.1
    7  6.868e-01  3.908e+08      9.0
    8  6.701e-01  4.006e+08      9.2
    9  6.661e-01  4.030e+08      9.3
-------------------------------------------------------
Average performance:          9.1 +- 0.1 GFLOP/s
-------------------------------------------------------
taskset -c 3 ./nbody3D2.cl

Total memory size: 786432 B, 768 KiB, 0 MiB

 Step    Time, s Interact/s  GFLOP/s
    0  6.780e-01  3.959e+08      9.1 *
    1  6.766e-01  3.967e+08      9.1 *
    2  7.023e-01  3.822e+08      8.8 *
    3  6.897e-01  3.892e+08      9.0
    4  6.752e-01  3.976e+08      9.1
    5  6.741e-01  3.982e+08      9.2
    6  6.773e-01  3.963e+08      9.1
    7  6.809e-01  3.942e+08      9.1
    8  6.748e-01  3.978e+08      9.1
    9  6.944e-01  3.865e+08      8.9
-------------------------------------------------------
Average performance:          9.1 +- 0.1 GFLOP/s
-------------------------------------------------------
```

4. Version 4 : Modifier les instructions du calcul (sqrt, div, pow…).

Dans cette étape, on élémine la puissance et les division et on les remplace par la racine carrée et la multiplication car sont moins complexes.

```
65      //const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening; //9 (mul, add)
66      const f32 d_2 = 1.0 / sqrtf((dx * dx) + (dy * dy) + (dz * dz) + softening); //11 (mul,add,div,sqrtf)
67
68      //const f32 d_3_over_2 = pow(d_2, 3.0 / 2.0); //11 (pow, div)
69      //const f32 d_3_over_2 = d_2 * sqrtf(d_2); //11 (sqrt, mult)
70
71      const f32 d_3_over_2 = d_2 * d_2 * d_2; //13 (mult)
72
73      //Net force
74      //fx += dx / d_3_over_2; //13 (add, div)
75      //fy += dy / d_3_over_2; //15 (add, div)
76      //fz += dz / d_3_over_2; //17 (add, div)
77
78      fx += dx * d_3_over_2; //15 (add, mul)
79      fy += dy * d_3_over_2; //17 (add, mul)
80      fz += dz * d_3_over_2; //19 (add, mul)
81
```

Grâce à l'élimination ces calcules complexe, on a amélioré beaucoup la performance.

```
taskset -c 3 ./nbody3D3.g

Total memory size: 786432 B, 768 KiB, 0 MiB

 Step     Time, s Interact/s  GFLOP/s
    0  1.725e-01  1.556e+09     35.8 *
    1  1.675e-01  1.603e+09     36.9 *
    2  1.732e-01  1.550e+09     35.6 *
    3  1.664e-01  1.613e+09     37.1
    4  1.881e-01  1.427e+09     32.8
    5  1.707e-01  1.572e+09     36.2
    6  1.744e-01  1.539e+09     35.4
    7  1.732e-01  1.550e+09     35.6
    8  1.745e-01  1.538e+09     35.4
    9  1.747e-01  1.536e+09     35.3
-------------------------------------------------------
Average performance:          35.4 +- 1.2 GFLOP/s
-------------------------------------------------------
taskset -c 3 ./nbody3D3.cl

Total memory size: 786432 B, 768 KiB, 0 MiB

 Step     Time, s Interact/s  GFLOP/s
    0  1.714e-01  1.566e+09     36.0 *
    1  1.714e-01  1.566e+09     36.0 *
    2  1.708e-01  1.571e+09     36.1 *
    3  1.654e-01  1.623e+09     37.3
    4  1.661e-01  1.616e+09     37.2
    5  1.651e-01  1.626e+09     37.4
    6  1.725e-01  1.556e+09     35.8
    7  1.689e-01  1.589e+09     36.6
    8  1.910e-01  1.405e+09     32.3
    9  1.727e-01  1.554e+09     35.7
-------------------------------------------------------
Average performance:          36.0 +- 1.6 GFLOP/s
-------------------------------------------------------
```

5. Version 5 : Ajouter le déroulage (Unroll).

Le déroulage de boucles consiste à dupliquer une section interne à une boucle N fois en augmentant de façon statique l'indice.

'#pragma unroll' est une directive utilisée pour indiquer au compilateur qu'une boucle devrait être déroulée, c'est-à-dire que les instructions de la boucle devraient être copiées plusieurs fois à la place de la boucle elle-même. Cela peut aider à améliorer les performances en réduisant le nombre d'opérations de boucle nécessaires.

On ajoute le flag **'-funroll-loops'** à la compilation.

```
#pragma unroll
  for (u64 i = 0; i < n; i++)
    {
      //
      f32 fx = 0.0;
      f32 fy = 0.0;
      f32 fz = 0.0;

      //23 floating-point operations
    #pragma unroll
      for (u64 j = 0; j < n; j++){
        //Newton's law
```

```
  #pragma unroll
  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];
    }
}
```

Grâce à le déroulage (Unroll), on a amélioré un peu plus la performance.

```
taskset -c 3 ./nbody3D4.g

Total memory size: 786432 B, 768 KiB, 0 MiB

 Step     Time, s Interact/s  GFLOP/s
    0  1.686e-01  1.592e+09     36.6 *
    1  1.670e-01  1.608e+09     37.0 *
    2  1.921e-01  1.397e+09     32.1 *
    3  1.670e-01  1.607e+09     37.0
    4  1.664e-01  1.613e+09     37.1
    5  1.628e-01  1.649e+09     37.9
    6  1.641e-01  1.635e+09     37.6
    7  1.642e-01  1.634e+09     37.6
    8  1.775e-01  1.512e+09     34.8
    9  1.691e-01  1.587e+09     36.5
-------------------------------------------------------
Average performance:          36.9 +- 1.0 GFLOP/s
-------------------------------------------------------
taskset -c 3 ./nbody3D4.cl

Total memory size: 786432 B, 768 KiB, 0 MiB

 Step     Time, s Interact/s  GFLOP/s
    0  1.729e-01  1.552e+09     35.7 *
    1  1.710e-01  1.569e+09     36.1 *
    2  1.724e-01  1.557e+09     35.8 *
    3  1.720e-01  1.561e+09     35.9
    4  1.721e-01  1.559e+09     35.9
    5  1.723e-01  1.558e+09     35.8
    6  1.679e-01  1.599e+09     36.8
    7  1.663e-01  1.614e+09     37.1
    8  1.678e-01  1.600e+09     36.8
    9  1.660e-01  1.617e+09     37.2
-------------------------------------------------------
Average performance:          36.5 +- 0.6 GFLOP/s
-------------------------------------------------------
```

6. Version 6 : Vectorisation AVX.

Pour la vectorisation AVX, on a changé le code en utilisant les fonctions intrinsèques x86.

La plupart des fonctions sont contenues dans des bibliothèques, mais certaines sont intégrées au compilateur. Au lieu d'écrire le code en assembleur avec __asm__ __volatile__ on utilise les fonctions intrinsèques.

On a ajoutée **'static inline'** pour indiquer que la fonction doit être **inline.**

On ajoute le flag **'-mavx'** et **'-finline-functions** à la compilation.

```
45  static inline void move_particles(particle_t p, const f32 dt,const u64 n)
46  {
47    //
48    const f32 softening = 1e-20;
49
50    //set les elements en AVX
51    __m256 softening_mm = _mm256_set1_ps(softening);
52    __m256 dt_mm = _mm256_set1_ps(dt);
53
54    //
55  #pragma unroll
56    for (u64 i = 0; i < n; i++)
57      {
58        //
59        // f32 fx = 0.0;
60        // f32 fy = 0.0;
61        // f32 fz = 0.0;
62
63        __m256 fx = _mm256_setzero_ps();
64        __m256 fy = _mm256_setzero_ps();
65        __m256 fz = _mm256_setzero_ps();
66        __m256 dx_mm = _mm256_set1_ps(p.x[i]);
67        __m256 dy_mm = _mm256_set1_ps(p.y[i]);
68        __m256 dz_mm = _mm256_set1_ps(p.z[i]);
69
```

```
134   #pragma unroll
135   for (u64 i = 0; i < n; i=i+8)
136     {
137       // p.x[i] += dt *  p.vx[i];
138       // p.y[i] += dt *  p.vy[i];
139       // p.z[i] += dt *  p.vz[i];
140
141     __m256 xx = _mm256_fmadd_ps(dt_mm,_mm256_loadu_ps(p.vx+i),_mm256_loadu_ps(p.x+i));
142     __m256 yy = _mm256_fmadd_ps(dt_mm,_mm256_loadu_ps(p.vy+i),_mm256_loadu_ps(p.y+i));
143     __m256 zz = _mm256_fmadd_ps(dt_mm,_mm256_loadu_ps(p.vz+i),_mm256_loadu_ps(p.z+i));
144
145     _mm256_storeu_ps(p.x + i, xx);
146     _mm256_storeu_ps(p.y + i, yy);
147     _mm256_storeu_ps(p.z + i, zz);
148
```

Grâce à la vectorisation AVX, on a amélioré la performance.

```
taskset -c 3 ./nbody3D5.g

Total memory size: 786432 B, 768 KiB, 0 MiB

 Step    Time, s Interact/s  GFLOP/s
    0  1.090e-01  2.462e+09     56.6 *
    1  1.126e-01  2.384e+09     54.8 *
    2  1.133e-01  2.368e+09     54.5 *
    3  1.140e-01  2.355e+09     54.2
    4  1.137e-01  2.362e+09     54.3
    5  1.136e-01  2.364e+09     54.4
    6  1.155e-01  2.325e+09     53.5
    7  1.133e-01  2.369e+09     54.5
    8  1.151e-01  2.331e+09     53.6
    9  1.248e-01  2.152e+09     49.5
-----------------------------------------------------
Average performance:         53.4 +- 1.6 GFLOP/s
-----------------------------------------------------
taskset -c 3 ./nbody3D5.cl

Total memory size: 786432 B, 768 KiB, 0 MiB

 Step    Time, s Interact/s  GFLOP/s
    0  1.366e-01  1.964e+09     45.2 *
    1  1.161e-01  2.313e+09     53.2 *
    2  1.128e-01  2.380e+09     54.7 *
    3  1.142e-01  2.349e+09     54.0
    4  1.147e-01  2.340e+09     53.8
    5  1.127e-01  2.381e+09     54.8
    6  1.161e-01  2.313e+09     53.2
    7  1.142e-01  2.350e+09     54.1
    8  1.189e-01  2.258e+09     51.9
    9  1.164e-01  2.307e+09     53.1
-----------------------------------------------------
Average performance:         53.6 +- 0.8 GFLOP/s
-----------------------------------------------------
```

7. Version 7 : Vectorisation SSE.

Pour la vectorisation SSE, on fait la même chose que la vectorisation précédente AVX, on change le code en utilisant les fonctions intrinsèques x86.

On ajoute le flag **‘-msse’** à la compilation.

```
137         // p.x[i] += dt *  p.vx[i];
138         // p.y[i] += dt *  p.vy[i];
139         // p.z[i] += dt *  p.vz[i];
140
141         __m128 xx = _mm_fmadd_ps(dt_mm,_mm_loadu_ps(p.vx+i),_mm_loadu_ps(p.x+i));
142         __m128 yy = _mm_fmadd_ps(dt_mm,_mm_loadu_ps(p.vy+i),_mm_loadu_ps(p.y+i));
143         __m128 zz = _mm_fmadd_ps(dt_mm,_mm_loadu_ps(p.vz+i),_mm_loadu_ps(p.z+i));
144
145         _mm_storeu_ps(p.x + i, xx);
146         _mm_storeu_ps(p.y + i, yy);
147         _mm_storeu_ps(p.z + i, zz);
```

Grâce à la vectorisation SSE, on a amélioré la performance.

```
taskset -c 3 ./nbody3D6.g

Total memory size: 786432 B, 768 KiB, 0 MiB

 Step    Time, s Interact/s  GFLOP/s
    0  1.040e-01  2.582e+09     59.4 *
    1  1.016e-01  2.642e+09     60.8 *
    2  1.047e-01  2.563e+09     58.9 *
    3  1.134e-01  2.367e+09     54.4
    4  1.044e-01  2.571e+09     59.1
    5  1.030e-01  2.606e+09     59.9
    6  1.024e-01  2.620e+09     60.3
    7  1.041e-01  2.579e+09     59.3
    8  1.017e-01  2.639e+09     60.7
    9  1.032e-01  2.601e+09     59.8
-----------------------------------------------------
Average performance:         59.1 +- 2.0 GFLOP/s
-----------------------------------------------------
taskset -c 3 ./nbody3D6.cl

Total memory size: 786432 B, 768 KiB, 0 MiB

 Step    Time, s Interact/s  GFLOP/s
    0  1.050e-01  2.555e+09     58.8 *
    1  1.001e-01  2.683e+09     61.7 *
    2  9.818e-02  2.734e+09     62.9 *
    3  1.002e-01  2.679e+09     61.6
    4  9.669e-02  2.776e+09     63.8
    5  9.869e-02  2.720e+09     62.6
    6  9.756e-02  2.751e+09     63.3
    7  1.003e-01  2.676e+09     61.6
    8  1.012e-01  2.652e+09     61.0
    9  1.040e-01  2.581e+09     59.4
-----------------------------------------------------
Average performance:         61.9 +- 1.4 GFLOP/s
-----------------------------------------------------
```

8. **Version 8 : Parallélisation avec OpenMP**.

Pour cette version, on a utilisé OpenMP pour le parallélisme, à l'aide du mot clé **'#pragma omp parallel for'** qui indique au compilateur de générer du code pour une boucle for parallèle. La boucle sera exécutée en parallèle par plusieurs threads, chaque thread traitant une itération différente de la boucle. Cela peut grandement améliorer les performances.

```
#pragma omp parallel for
  for (u64 i = 0; i < n; i++)
    {
      //
      // f32 fx = 0.0;
      // f32 fy = 0.0;
      // f32 fz = 0.0;
```

```
#pragma omp parallel for
  for (u64 i = 0; i < n; i=i+8)
    {
      // p.x[i] += dt *  p.vx[i];
      // p.y[i] += dt *  p.vy[i];
      // p.z[i] += dt *  p.vz[i];
```

Grâce à la parallélisation avec OpenMP, on a amélioré beaucoup la performance.

```
./nbody3D7.g

Total memory size: 786432 B, 768 KiB, 0 MiB

 Step    Time, s Interact/s  GFLOP/s
    0  5.214e-02  5.148e+09    118.4 *
    1  5.281e-02  5.082e+09    116.9 *
    2  5.505e-02  4.876e+09    112.1 *
    3  5.118e-02  5.245e+09    120.6
    4  5.205e-02  5.157e+09    118.6
    5  5.074e-02  5.290e+09    121.7
    6  5.114e-02  5.249e+09    120.7
    7  5.531e-02  4.853e+09    111.6
    8  5.271e-02  5.092e+09    117.1
    9  5.104e-02  5.259e+09    121.0
-----------------------------------------------------
Average performance:         118.8 +- 3.3 GFLOP/s
-----------------------------------------------------
./nbody3D7.cl

Total memory size: 786432 B, 768 KiB, 0 MiB

 Step    Time, s Interact/s  GFLOP/s
    0  5.301e-02  5.064e+09    116.5 *
    1  5.638e-02  4.761e+09    109.5 *
    2  5.195e-02  5.166e+09    118.8 *
    3  5.179e-02  5.183e+09    119.2
    4  5.422e-02  4.950e+09    113.9
    5  5.187e-02  5.175e+09    119.0
    6  5.278e-02  5.086e+09    117.0
    7  5.375e-02  4.994e+09    114.9
    8  5.550e-02  4.836e+09    111.2
    9  5.104e-02  5.259e+09    121.0
-----------------------------------------------------
Average performance:         116.6 +- 3.2 GFLOP/s
-----------------------------------------------------
```

.

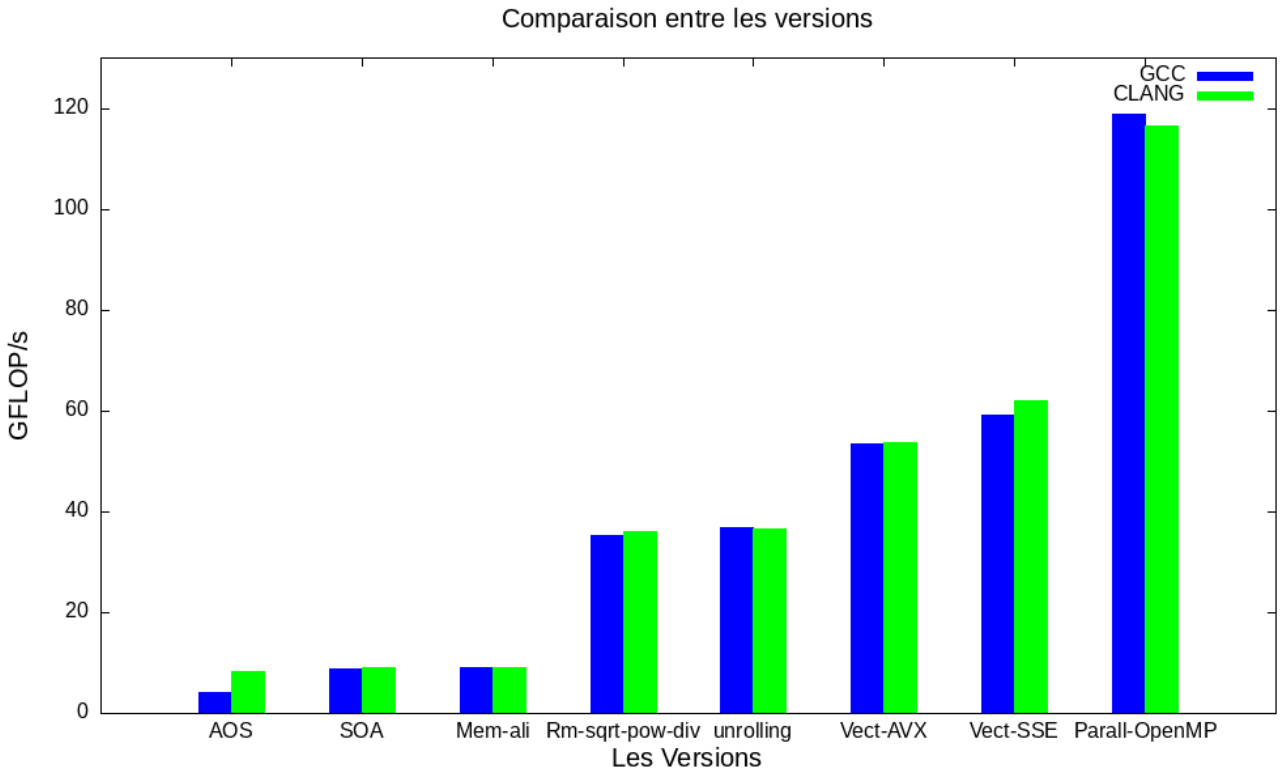
III. Performances :

Dans cette section, nous allons discuter les mesures de performances des différentes versions d'optimisations vues précédemment.

Nous allons premièrement comparer les performances des versions avec GCC et CLANG.

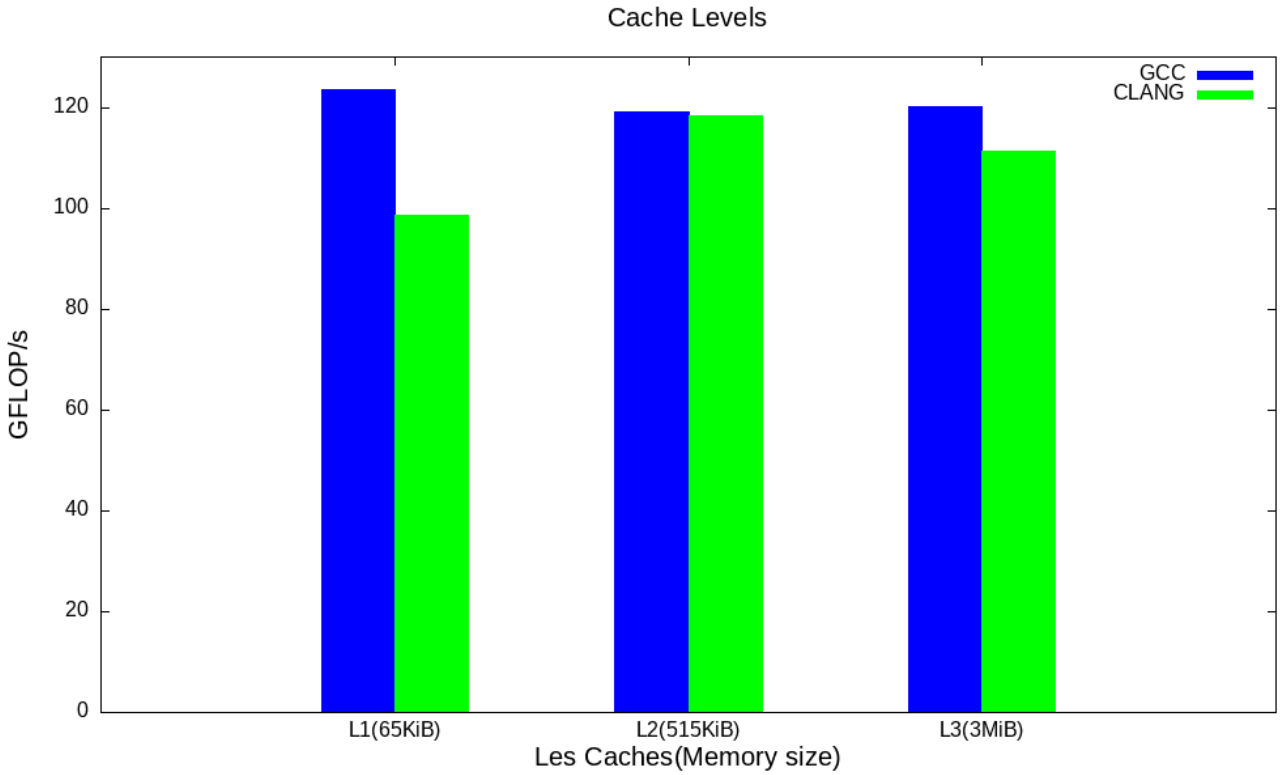
Ensuite, on va comparer les niveaux de cache L1, L2 et L3.

1. Comparaison entre toutes les versions :



<u>Discussion</u> : On peut conclure que le compilateur CLANG donne de meilleures performances en générale par rapport à GCC. Et la vectorisation et le parallélisme augmentent beaucoup la performance.

2. Performances en les niveaux de cache :



IV. Conclusion :

Pour conclure, afin d’optimiser le programme N-Body3D, on a implémenté 8 versions. Chaque version nous permettons d’améliorer encore plus la performance, jusqu’à nous avons eu la meilleure performance atteinte ≈118 GFLOPS/S avec GCC et ≈116 GFLOPS/S avec CLANG grâce à la vectorisation (AVX, SSE) et le parallélisme avec OpenMP.