DIAS Nicolas

Rapport Architecture Parallèle

Introduction :

Un nbody est un programme qui permet de regarder le déplacement de différentes particules dans un espace donné en fonction des autres particules. Chaque particules va influencer le déplacement des autres car chacune d'entre elle attiré l'autre ce qui va modifier leur vitesse sur chaque axe.

```
void move_particles(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;

      //23 floating-point operations
      for (u64 j = 0; j < n; j++)
        {
          //Newton's law
          const f32 dx = p[j].x - p[i].x; //1
          const f32 dy = p[j].y - p[i].y; //2
          const f32 dz = p[j].z - p[i].z; //3
          const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening; //9
          const f32 d_3_over_2 = pow(d_2, 3.0 / 2.0); //11

          //Net force
          fx += dx / d_3_over_2; //13
          fy += dy / d_3_over_2; //15
          fz += dz / d_3_over_2; //17
        }

      //
      p[i].vx += dt * fx; //19
      p[i].vy += dt * fy; //21
      p[i].vz += dt * fz; //23
    }

  //3 floating-point operations
  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;
    }
    printf("val = %lf\n",p[0].x);
}
```

Cette fonction est notre fonction qui fait les différents calculs qui permettent de modifié la position et la vitesse de chaque particules. Tout d'abord on fait la soustraction des positions en x, y et z de chaque particule avec une autre. Cela nous permettra de savoir la force qu'exercent les particules les unes sur les autres et donc de déterminer la modification de leurs vitesses. Un fois nos vitesses calculé on peut calculer leur déplacement. Chacune de nos boucles vont de 0 à n, n étant le nombre de particules

Figure1 : fonction move_particles de base

Architecture du programme :

Le programme est donc un nbody en 3D, nous avons donc différentes particules qui ont une position en x, y et z et qui ont également une vitesse sur chacun de ces axes ce qui va modifier leurs positions. Chaque particule interagit avec toutes les autres, leur vitesse n'est donc pas constante.
Le programme utilise une structure de donnée « particle_t» qui permet de stocké la position et la vitesse d'une particule sur chacun des trois axes. Une fonction « init » permet d'initialiser la position et la vitesse de chaque particule, comme aucune « graine » n'est utilisée pour l'aléatoire alors les valeurs ne changent pas entre chaque lancement du programme. Nous avons ensuite une fonction qui nous permet de faire les calculs de déplacement des particules, cette fonction utilise une variable n qui correspond au nombre de particules que nous avons à gérer, La valeur de n peut être donné lors du lancement du programme ou alors une valeur par défaut sera rentré.

Version de base AOS (nbody.c):

La première version du code (la version original), le fichier nbody.c, est composé d'un tableau de structure de donnée (AOS = array of struct) qui va stocker la position en x, y et z et leur vitesse sur chaque axe pour chaque particule. Pour faire nos différents calculs nous utilisons donc un tableau permettant de stocker toutes nos structures. Cette version est peu optimisé car notre tableau de structure n'est pas aligné en mémoire, le compilateur ne va donc pas vectoriser le code par lui-même, tous les calculs étant fait en séquentiel il sera lent du fait que la fonction de calcul utilise deux boucles imbriqué qui sont toute deux de taille n (n le nombre de corps du programme) il y a ensuite un autre boucle elle aussi de taille n, la complexité de cette fonction est donc en O (n^2+n). Il y a également plusieurs endroits dans la fonction de calcul qui peuvent être amélioré pour éviter d'utiliser des fonctions qui utilisent trop de cycles, nous verrons ces améliorations un peu plus loin.

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

 Step    Time, s Interact/s  GFLOP/s
val = -1.729782
    0  8.170e-01  3.285e+08      7.6 *
val = -4.054005
    1  8.165e-01  3.287e+08      7.6 *
val = -6.344671
    2  8.167e-01  3.287e+08      7.6 *
val = -8.621541
    3  8.168e-01  3.286e+08      7.6
val = -10.890901
    4  8.165e-01  3.288e+08      7.6
val = -13.155536
    5  8.167e-01  3.287e+08      7.6
val = -15.416925
    6  8.163e-01  3.288e+08      7.6
val = -17.675947
    7  8.166e-01  3.287e+08      7.6
val = -19.933167
    8  8.163e-01  3.288e+08      7.6
val = -22.188967
    9  8.165e-01  3.287e+08      7.6
-----------------------------------------------------
Average performance:          7.6 +- 0.0 GFLOP/s
-----------------------------------------------------
```

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

 Step    Time, s Interact/s  GFLOP/s
val = -1.729783
    0  1.079e+01  2.487e+07      0.6 *
val = -4.054006
    1  1.081e+01  2.482e+07      0.6 *
val = -6.344673
    2  1.080e+01  2.485e+07      0.6 *
val = -8.621544
    3  1.081e+01  2.482e+07      0.6
val = -10.890903
    4  1.083e+01  2.478e+07      0.6
val = -13.155539
    5  1.083e+01  2.478e+07      0.6
val = -15.416928
    6  1.081e+01  2.482e+07      0.6
val = -17.675951
    7  1.081e+01  2.483e+07      0.6
val = -19.933170
    8  1.081e+01  2.482e+07      0.6
val = -22.188971
    9  1.081e+01  2.482e+07      0.6
-----------------------------------------------------
Average performance:          0.6 +- 0.0 GFLOP/s
-----------------------------------------------------
```

Figure2 : programme de base avec icc Figure3 : programme de base avec gcc

On remarque qu'icc va beaucoup plus vite que la version avec gcc, cela est dû au fait que icc est beaucoup plus agressif sur l'optimisation du code il va lui-même essayer de vectoriser des parties du codes, icc est 12.6 fois plus rapide que gcc.

Version amélioré en SOA (nbodymieux.c):

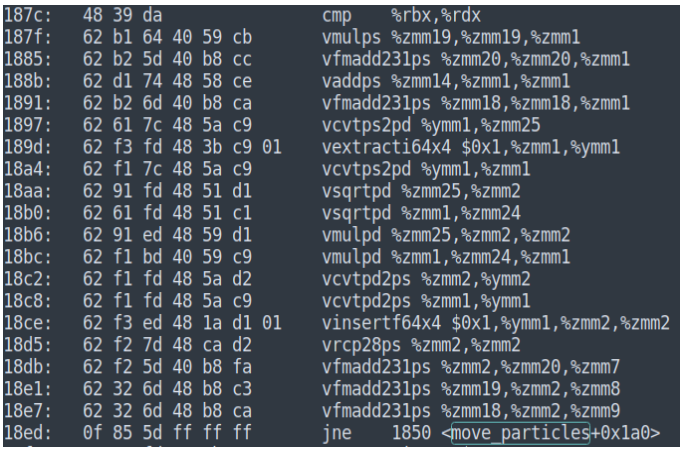
```c
typedef struct particle_s {

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

} particle_t;
```

```c
particle_t *p = malloc(sizeof(particle_t) * n);

p->x = aligned_alloc(64,sizeof(f32) * n);
p->y = aligned_alloc(64,sizeof(f32) * n);
p->z = aligned_alloc(64,sizeof(f32) * n);

p->vx = aligned_alloc(64,sizeof(f32) * n);
p->vy = aligned_alloc(64,sizeof(f32) * n);
p->vz = aligned_alloc(64,sizeof(f32) * n);
```

Figure 4 : modification apporté au programme

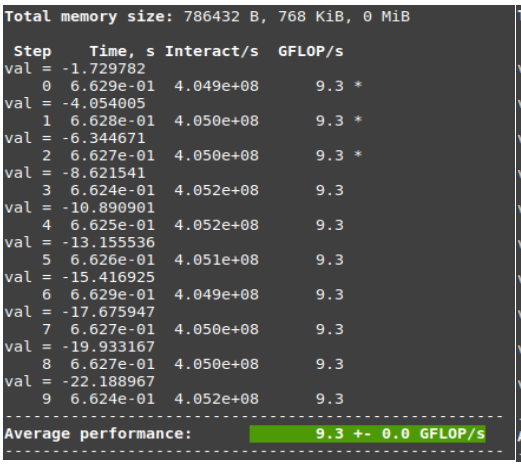
Comme nous l'avons vu précédemment le programme utilise un tableau de structure de données, ce rangement des données n'est pas très optimiser car on ne stockera pas toutes les positions et toutes les vitesses les unes à côté des autres en mémoire (toutes les positions en x à côté, toutes les positions y à côté, etc…) cela empêche donc le compilateur de vectoriser par lui-même et donc il ne fera pas d'optimisation. Pour régler ce problème nous avons donc utiliser une autre manière de ranger nos données qui est le rangement SOA (struct of array), cela consiste à ne garder qu'une seul structure de donnée au lieu de garder un tableau de structure. Dans cette unique structure nous allons faire des tableaux de position et de vitesse, comme ces tableaux seront alignés on aura tous nos x les uns à côté des autres, pareil pour nos autres positions et vitesses. Comme toutes les valeurs seront alignées en

mémoire le compilateur pourra ranger plusieurs valeurs dans un seul registre qui sera assez grand pour stocké plusieurs valeurs, quand il fera ses calculs il pourra donc en faire un seul pour plusieurs valeur en même temps (il vectorise) ce qui va accélérer le programme car on fait moins d'opération que précédemment tout en obtenant le même résultat.



On voit effectivement que le compilateur a utiliser des instructions se terminant par ps, et a utilisé des registre ymm ce qui veux dire qu'il a vectoriser par lui-même notre boucle vu que stockage dans les caches est meilleur

Figure 5 : fichier compilé de la version SOA



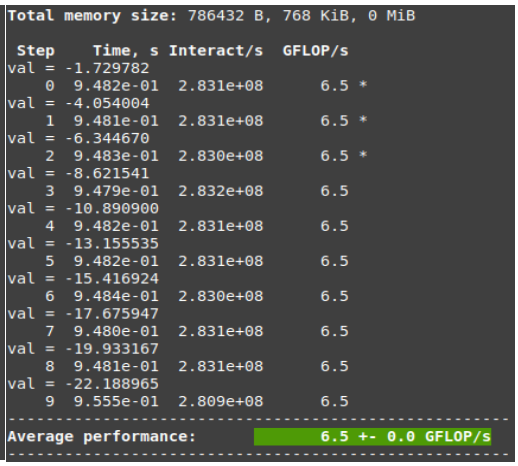


Figure 6 : programme SOA avec icc

Figure 7 : programme SOA avec gcc

On voit que sur cette version icc est également plus rapide que gcc, mais le gain est nettement plus élevé avec gcc qui va 10 fois plus vite que la version précédente alors qu'icc va environ 1.2 fois plus vite que la version précédente. Cela est dû au fait que comme icc a déjà fait plusieurs amélioration dans la version précédente il a moins de chose qu'il peut modifier et gagnera donc un pourcentage d'accélération plus faible, gcc par contre va 10 fois plus vite car il passe d'une version du programme en séquentiel à une version vectoriser ce qui va évidemment plus vite.

Version sans puissance (nbodymieux2.c):

Cette version utilise les améliorations précédente et en plus de cela nous avons retiré la fonction puissance qui est utilisé dans le programme par 2 multiplication ce qui revient au même résultat, mais qui est plus rapide parce que la fonction puissance utilise énormément de cycles pour s'exécuter alors que faire plusieurs multiplications coûtent beaucoup moins de cycles. De même nous ne faisons pas la multiplication de trois racines carrées mais nous faisons cette racine à l'avance et nous multiplions le

résultat. Cela permet au code de ne pas recalculé plusieurs fois la même racine carrée ce qui ralenti le programme.

```
const f32 dx = p->x[j] - p->x[i]; //1
const f32 dy = p->y[j] - p->y[i]; //2
const f32 dz = p->z[j] - p->z[i]; //3
const f32 d_2 = (dx * dx) + (dy * dy) + (dz * dz) + softening; //9
f32 val = sqrt(d_2);
const f32 d_3_over_2 = val * val * val; //11
```

Figure 8 : modification de la puissance

On peut voir dans la figure 8 que nous avons modifié l'utilisation de la fonction pow précédemment utilisé « pow(d_2, 3.0 / 2.0) », on voit qu'ici on fait la puissance 3/2 or une puissance ½ correspond à la racine carrée donc nous la faisons à l'avance pour ne pas la refaire plusieurs fois (comme dit précédemment) et nous faisons donc plusieurs fois la multiplication pour éviter la fonction pow qui est plus lente que deux multiplication.

```
187c:   48 39 da                cmp    %rbx,%rdx
187f:   62 b1 64 40 59 cb       vmulps %zmm19,%zmm19,%zmm1
1885:   62 b2 5d 40 b8 cc       vfmadd231ps %zmm20,%zmm20,%zmm1
188b:   62 d1 74 48 58 cf       vaddps %zmm15,%zmm1,%zmm1
1891:   62 f1 7c 48 28 d1       vmovaps %zmm1,%zmm2
1897:   62 b2 6d 40 b8 d2       vfmadd231ps %zmm18,%zmm18,%zmm2
189d:   62 f2 7d 48 cc ca       vrsqrt28ps %zmm2,%zmm1
18a3:   62 f2 7d 48 ca c9       vrcp28ps %zmm1,%zmm1
18a9:   62 f1 74 48 59 ca       vmulps %zmm2,%zmm1,%zmm1
18af:   62 f2 7d 48 ca c9       vrcp28ps %zmm1,%zmm1
18b5:   62 72 5d 40 b8 c1       vfmadd231ps %zmm1,%zmm20,%zmm8
18bb:   62 32 75 48 b8 cb       vfmadd231ps %zmm19,%zmm1,%zmm9
18c1:   62 32 75 48 b8 d2       vfmadd231ps %zmm18,%zmm1,%zmm10
18c7:   75 87                   jne    1850 <move_particles+0x1a0>
```

Figure 9 : compiler de la version sans pow avec gcc

On voit effectivement que le programme ne fait moins d'instruction que dans la version précédente, il ne fait plus qu'une seule fois la racine carré, de plus on voit que le compilateur utilise une « rsqrt » qui est la racine carré inverse qui coute moins de cycle que la racine carré normale.

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

 Step     Time, s Interact/s  GFLOP/s
vit = -251.288086
val = -1.729782
    0  2.793e-01  9.609e+08     22.1 *
vit = -232.422272
val = -4.054004
    1  2.789e-01  9.625e+08     22.1 *
vit = -229.066620
val = -6.344670
    2  2.789e-01  9.625e+08     22.1 *
vit = -227.687042
val = -8.621541
    3  2.788e-01  9.627e+08     22.1
vit = -226.935913
val = -10.890900
    4  2.788e-01  9.628e+08     22.1
vit = -226.463501
val = -13.155535
    5  2.789e-01  9.623e+08     22.1
vit = -226.138947
val = -15.416924
    6  2.787e-01  9.630e+08     22.1
vit = -225.902206
val = -17.675947
    7  2.791e-01  9.618e+08     22.1
vit = -225.721878
val = -19.933167
    8  2.787e-01  9.632e+08     22.2
vit = -225.579926
val = -22.188965
    9  2.787e-01  9.632e+08     22.2
-----------------------------------------------------
Average performance:          22.1 +- 0.0 GFLOP/s
-----------------------------------------------------
```

Figure 10 : version sans pow avec gcc

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

 Step     Time, s Interact/s  GFLOP/s
vit = -251.288086
val = -1.729782
    0  2.752e-01  9.755e+08     22.4 *
vit = -232.422272
val = -4.054004
    1  2.748e-01  9.768e+08     22.5 *
vit = -229.066620
val = -6.344670
    2  2.748e-01  9.770e+08     22.5 *
vit = -227.687042
val = -8.621541
    3  2.748e-01  9.769e+08     22.5
vit = -226.935913
val = -10.890900
    4  2.748e-01  9.768e+08     22.5
vit = -226.463501
val = -13.155535
    5  2.747e-01  9.771e+08     22.5
vit = -226.138947
val = -15.416924
    6  2.748e-01  9.769e+08     22.5
vit = -225.902206
val = -17.675947
    7  2.751e-01  9.756e+08     22.4
vit = -225.721878
val = -19.933167
    8  2.748e-01  9.766e+08     22.5
vit = -225.579926
val = -22.188965
    9  2.748e-01  9.768e+08     22.5
-----------------------------------------------------
Average performance:          22.5 +- 0.0 GFLOP/s
-----------------------------------------------------
```

Figure 11 : version sans pow avec icc

On voit que pour cette version la version de gcc et la version de icc sont très proche au niveau des performance. On voit surtout que la version de gcc va environ 3.4 fois plus vite que la version

précédente de gcc et icc va 2.4 fois plus vite que la version précédente. On voit bien que faire la racine carrée à l'avance plus retirer la faonction « pow » améliore grandement les performances.

Version intrinsic AVX2 (nbodymieuxfmasdd.c):

Dans cette version et celle d'après nous allons utiliser des intrinsic. Les fonctions intrinsic permettent de mettre dans le code compilé un équivalent de ces fonctions sans que le compilateur ajoute de modification (si on utilise une fonction add on aura un add dans le fichier compilé). Les fonctions intrinsic AVX2 utilise le type __m256 qui est de taille 256 bits soit 32 octet, des registre ymm seront donc utilisé pour faire nos calculs. Nous pouvons par conséquent ranger plusieurs flottant dans un seul registre, un flottant ayant une taille de 4 octet on peut en rager 8 dans un seul registre.

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

  __m256 x1, y1, z1, x2, y2, z2, vx, vy, vz;
  __m256 dx, dy, dz, dx2, dy2, dz2, sof, tmp;
  __m256 fx, fy, fz, d_2, d_3_over_2, mdt;

  sof[0] = softening;
  sof[1] = softening;
  sof[2] = softening;
  sof[3] = softening;
  sof[4] = softening;
  sof[5] = softening;
  sof[6] = softening;
  sof[7] = softening;

  mdt[0] = dt;
  mdt[1] = dt;
  mdt[2] = dt;
  mdt[3] = dt;
  mdt[4] = dt;
  mdt[5] = dt;
  mdt[6] = dt;
  mdt[7] = dt;

  //
  for (u64 i = 0; i < n; i+=8)
  {
      //
      fx =  _mm256_setzero_ps();
      fy =  _mm256_setzero_ps();
      fz =  _mm256_setzero_ps();

      x1 =  _mm256_loadu_ps(p->x+i);
      y1 =  _mm256_loadu_ps(p->y+i);
      z1 =  _mm256_loadu_ps(p->z+i);

      vx =  _mm256_loadu_ps(p->vx+i);
      vy =  _mm256_loadu_ps(p->vy+i);
      vz =  _mm256_loadu_ps(p->vz+i);

      //23 floating-point operations
    for (u64 j = 0; j < n; j++)
    {
      x2 =  _mm256_loadu_ps(p->x+j);
      y2 =  _mm256_loadu_ps(p->y+j);
      z2 =  _mm256_loadu_ps(p->z+j);

      //Newton's law
      dx =  _mm256_sub_ps(x2,x1);
      dy =  _mm256_sub_ps(y2,y1);
      dz =  _mm256_sub_ps(z2,z1);
```

```
      dx2 =  _mm256_mul_ps(dx,dx);
      dy2 =  _mm256_mul_ps(dy,dy);
      dz2 =  _mm256_mul_ps(dz,dz);

      d_2 =  _mm256_add_ps(dx2,dy2);
      d_2 =  _mm256_add_ps(d_2,dz2);
      d_2 =  _mm256_add_ps(d_2,sof);

      d_2 =  _mm256_rsqrt_ps(d_2);

      d_3_over_2 =  _mm256_mul_ps(d_2,d_2);
      d_3_over_2 =  _mm256_mul_ps(d_3_over_2,d_2);

      fx =  _mm256_fmadd_ps(dx,d_3_over_2,fx);
      fy =  _mm256_fmadd_ps(dy,d_3_over_2,fy);
      fz =  _mm256_fmadd_ps(dz,d_3_over_2,fz);
    }

    //
    vx =  _mm256_fmadd_ps(mdt,fx,vx);
    vy =  _mm256_fmadd_ps(mdt,fy,vy);
    vz =  _mm256_fmadd_ps(mdt,fz,vz);

     _mm256_storeu_ps(p->vx+i,vx);
     _mm256_storeu_ps(p->vy+i,vy);
     _mm256_storeu_ps(p->vz+i,vz);
  }
  printf("vit = %lf\n",p->vx[0]);

  //3 floating-point operations
  for (u64 i = 0; i < n; i+=8)
    {
      x1 =  _mm256_loadu_ps(p->x+i);
      y1 =  _mm256_loadu_ps(p->y+i);
      z1 =  _mm256_loadu_ps(p->z+i);

      vx =  _mm256_loadu_ps(p->vx+i);
      vy =  _mm256_loadu_ps(p->vy+i);
      vz =  _mm256_loadu_ps(p->vz+i);

      x1 =  _mm256_fmadd_ps(mdt,vx,x1);
      y1 =  _mm256_fmadd_ps(mdt,vy,y1);
      z1 =  _mm256_fmadd_ps(mdt,vz,z1);

       _mm256_storeu_ps(p->x+i,x1);
       _mm256_storeu_ps(p->y+i,y1);
       _mm256_storeu_ps(p->z+i,z1);
    }
    printf("val = %lf\n",p->x[0]);
```

Figure 12 : fonction move_particles en intrinsic (AVX2)

Chaque fonction est appelé avec « _mm256 » qui correspond au fait que l'on utilise des fonctions intrinsic AVX2 suivi du nom de ce que l'on souhaite faire, une addition, une multiplication, etc… suivi de « ps » pour dire que nous utilisons la version vectorisé. On met ensuite entre parenthèse les valeurs que l'on veut utiliser, (_mm256_add_ps (dx2,dy2) correspond à la soustraction de dx2 et dy2).

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

 Step     Time, s  Interact/s  GFLOP/s
vit = -251.321304
val = -1.730114
    0  3.772e-01  7.116e+08      16.4 *
vit = -232.445938
val = -4.054573
    1  3.771e-01  7.118e+08      16.4 *
vit = -229.088547
val = -6.345459
    2  3.771e-01  7.118e+08      16.4 *
vit = -227.707840
val = -8.622537
    3  3.769e-01  7.121e+08      16.4
vit = -226.956177
val = -10.892098
    4  3.770e-01  7.120e+08      16.4
vit = -226.483536
val = -13.156934
    5  3.775e-01  7.110e+08      16.4
vit = -226.158844
val = -15.418522
    6  3.774e-01  7.113e+08      16.4
vit = -225.922012
val = -17.677742
    7  3.775e-01  7.110e+08      16.4
vit = -225.741608
val = -19.935158
    8  3.769e-01  7.122e+08      16.4
vit = -225.599609
val = -22.191154
    9  3.771e-01  7.118e+08      16.4
-----------------------------------------------------
Average performance:          16.4 +- 0.0 GFLOP/s
-----------------------------------------------------
```

Figure 13 : version AVX2 avec icc

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

 Step     Time, s  Interact/s  GFLOP/s
vit = -251.321304
val = -1.730114
    0  3.925e-01  6.840e+08      15.7 *
vit = -232.445938
val = -4.0545573
    1  3.921e-01  6.846e+08      15.7 *
vit = -229.088547
val = -6.345459
    2  3.920e-01  6.847e+08      15.7 *
vit = -227.707840
val = -8.622537
    3  3.920e-01  6.848e+08      15.7
vit = -226.956177
val = -10.892098
    4  3.919e-01  6.849e+08      15.8
vit = -226.483536
val = -13.156934
    5  3.922e-01  6.844e+08      15.7
vit = -226.158844
val = -15.418522
    6  3.918e-01  6.850e+08      15.8
vit = -225.922012
val = -17.677742
    7  3.919e-01  6.850e+08      15.8
vit = -225.741608
val = -19.935158
    8  3.918e-01  6.850e+08      15.8
vit = -225.599609
val = -22.191154
    9  3.919e-01  6.849e+08      15.8
-----------------------------------------------------
Average performance:          15.8 +- 0.0 GFLOP/s
-----------------------------------------------------
```

Figure 14 : version AVX2 avec gcc

On voit que icc est plus efficace que gcc mais que ces deux versions son plus lente que les version précédente, en effet icc est 1,3 fois plus lente que la version précédente et gcc est 1,4 fois plus lent.

Cela est dû au fait que gcc va optimiser le code au mieux et son code dépend de la machine sur laquelle le code tourne. On peut voir avec le compiler.

```
187c:   48 39 da                cmp    %rbx,%rdx
187f:   62 b1 64 40 59 cb       vmulps %zmm19,%zmm19,%zmm1
1885:   62 b2 5d 40 b8 cc       vfmadd231ps %zmm20,%zmm20,%zmm1
188b:   62 d1 74 48 58 cf       vaddps %zmm15,%zmm1,%zmm1
1891:   62 f1 7c 48 28 d1       vmovaps %zmm1,%zmm2
1897:   62 b2 6d 40 b8 d2       vfmadd231ps %zmm18,%zmm18,%zmm2
189d:   62 f2 7d 48 cc ca       vrsqrt28ps %zmm2,%zmm1
18a3:   62 f2 7d 48 ca c9       vrcp28ps %zmm1,%zmm1
18a9:   62 f1 74 48 59 ca       vmulps %zmm2,%zmm1,%zmm1
18af:   62 f2 7d 48 ca c9       vrcp28ps %zmm1,%zmm1
18b5:   62 72 5d 40 b8 c1       vfmadd231ps %zmm1,%zmm20,%zmm8
18bb:   62 32 75 48 b8 cb       vfmadd231ps %zmm19,%zmm1,%zmm9
18c1:   62 32 75 48 b8 d2       vfmadd231ps %zmm18,%zmm1,%zmm10
18c7:   75 87                   jne    1850 <move_particles+0x1a0>
```

Figure 15 : compilé de la version SOA sans pow

Notre version AVX2 utilisera des registres ymm alors que dans la version SOA sans la fonction pow, le compilateur utilise des registres zmm, l'AVX512 est utilisé uniquement dans notre double boucle de calcul mais cette fonction étant celle où nous passons le plus de temps, il est normal que nous y gagnons autant de performance. La vectorisation de ce code étant meilleur, notre version AVX2 est donc plus lente. Si la version SOA sans pow est sur une machine sans AVX512 notre version AVX2 sera plus efficace.

Version intrinsic AVX512 (nbodymieuxfmasdd512.c):

Dans cette version nous allons également utiliser des fonctions intrinsic mais cette fois ci en AVX512, la différence principale est qu'en AVX512 nous utilisons des registres zmm qui sont encore plus grand, étant donné qu'ils sont de tailles 512 bits soit 64 octets. Comme ils sont deux fois plus grands notre programme fait donc le moins de calcul au final ce qui accélère le tout. Pour utiliser ces fonctions on utilise le type __m512 et les fonctions commenceront par _mm512.

```
for (u64 i = 0; i < n; i+=16)
{
    //
    fx = _mm512_setzero_ps();
    fy = _mm512_setzero_ps();
    fz = _mm512_setzero_ps();

    x1 = _mm512_loadu_ps(p->x+i);
    y1 = _mm512_loadu_ps(p->y+i);
    z1 = _mm512_loadu_ps(p->z+i);

    vx = _mm512_loadu_ps(p->vx+i);
    vy = _mm512_loadu_ps(p->vy+i);
    vz = _mm512_loadu_ps(p->vz+i);

    //23 floating-point operations
  for (u64 j = 0; j < n; j++)
  {
    x2 = _mm512_loadu_ps(p->x+j);
    y2 = _mm512_loadu_ps(p->y+j);
    z2 = _mm512_loadu_ps(p->z+j);

    //Newton's law
    dx = _mm512_sub_ps(x2,x1);
    dy = _mm512_sub_ps(y2,y1);
    dz = _mm512_sub_ps(z2,z1);

    dx2 = _mm512_mul_ps(dx,dx);
    dy2 = _mm512_mul_ps(dy,dy);
    dz2 = _mm512_mul_ps(dz,dz);

    d_2 = _mm512_add_ps(dx2,dy2);
    d_2 = _mm512_add_ps(d_2,dz2);
    d_2 = _mm512_add_ps(d_2,sof);

    #if __AVX512ER__
      d_2 = _mm512_rsqrt28_ps(d_2);
    #else
      d_2 = _mm512_rsqrt14_ps(d_2);
    #endif  //10

    d_3_over_2 = _mm512_mul_ps(d_2,d_2);
    d_3_over_2 = _mm512_mul_ps(d_3_over_2,d_2);

    fx = _mm512_fmadd_ps(dx,d_3_over_2,fx);
    fy = _mm512_fmadd_ps(dy,d_3_over_2,fy);
    fz = _mm512_fmadd_ps(dz,d_3_over_2,fz);
  }
  //
  vx = _mm512_fmadd_ps(mdt,fx,vx);
  vy = _mm512_fmadd_ps(mdt,fy,vy);
  vz = _mm512_fmadd_ps(mdt,fz,vz);
```

```
  _mm512_storeu_ps(p->vx+i,vx);
  _mm512_storeu_ps(p->vy+i,vy);
  _mm512_storeu_ps(p->vz+i,vz);
}
printf("vit = %lf\n",p->vx[0]);

//3 floating-point operations
for (u64 i = 0; i < n; i+=8)
  {
    x1 = _mm512_loadu_ps(p->x+i);
    y1 = _mm512_loadu_ps(p->y+i);
    z1 = _mm512_loadu_ps(p->z+i);

    vx = _mm512_loadu_ps(p->vx+i);
    vy = _mm512_loadu_ps(p->vy+i);
    vz = _mm512_loadu_ps(p->vz+i);

    x1 = _mm512_fmadd_ps(mdt,vx,x1);
    y1 = _mm512_fmadd_ps(mdt,vy,y1);
    z1 = _mm512_fmadd_ps(mdt,vz,z1);

    _mm512_storeu_ps(p->x+i,x1);
    _mm512_storeu_ps(p->y+i,y1);
    _mm512_storeu_ps(p->z+i,z1);
  }
  printf("val = %lf\n",p->x[0]);
```

Figure 15 : fonction move_particles en intrinsic (AVX512)

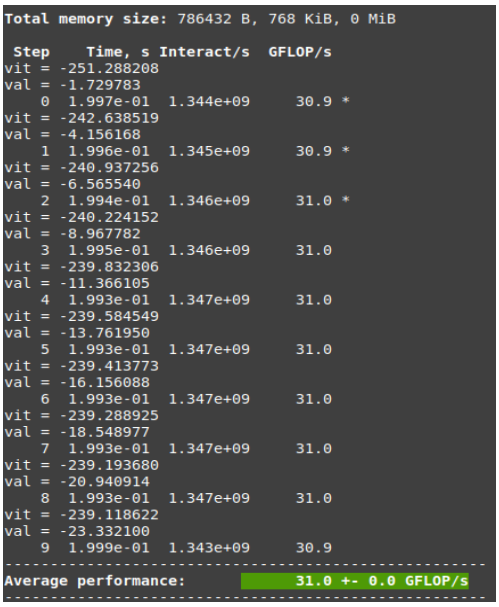


Figure 16 : version AVX512 avec icc

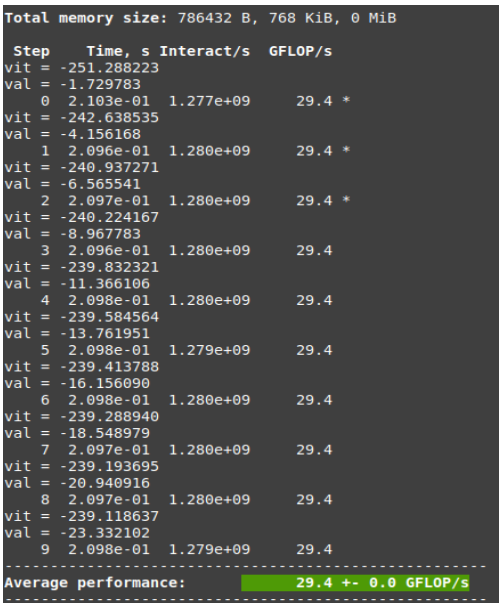


Figure 17 : version AVX512 avec gcc

On remarque encore une fois que la version icc est plus rapide que la version de gcc. Le compilateur icc est 1,3 fois plus rapide que la version sans intrinsic et la version gcc est 1.3 fois plus rapide que la version sans intrinsic. Ces version sons plus rapide car même si l'autre version utilise de l'AVX512, elle ne le fait pas sur l'intégralité de la fonction alors qu'ici la totalité de la fonction est vectorisé avec de l'AVX512.

On voit également que notre version AVX512 avec gcc est 50 fois plus rapide que notre code de base.

Conclusion :

On remarque donc que notre code peut-être optimisé de différentes manières, tout dépend du compilateur que l'on utilise, de la machine que l'on utilise ou comment on adapte notre code afin de faire le moins de calcul possible et en vectorisant au maximum notre code. On voit bien qu'avec les changements que nous avons faits nous avons un code qui s'exécute 50 fois plus rapidement que notre code de base. Sur un code qui s'exécute de base très vite cela peut paraitre insignifiant mais sur des programmes qui doivent tourner pendant très longtemps la différence est très importante et notre temps de calcul sera largement réduit (nous permettant de laisser notre machine allumé en mode performance moins longtemps). Une autre possibilité pour améliorer notre code serait de faire du parallélisme ce qui peut augmenter drastiquement les performances.