1 problem description

The main step of the MD model is the *evolve* function.The *evolve* function is a loop with the follow steps as bellow:

1. set the viscosity term in the force calculation for each dimension:

$$f[i] = -vis[i] \times velo[i], i = 0, 1, \cdots, N-1 \tag{1}$$

2. add the wind term in the force calculation for each dimension:

$$f[i] = f[i] - vis[i] \times b, i = 0, 1, \cdots, N-1 \tag{2}$$

3. calculate distance from central mass:

$$r[i] = \sqrt{pos[0][i]^2 + pos[1][i]^2 + pos[2][i]^2} \tag{3}$$

4. calculate central force for each dimension:

$$f[i] = f[i] - \frac{G \times mass[i] \times M\_central \times pos[i]}{r[i]^3} \tag{4}$$

5. calculate pairwise separation of the particles for each dimension:$delta\_pos$

6. calculate norm of separation vector:

$$delta\_r[k] = \sqrt{delta\_pos[0][k]^2 + delta\_pos[1][k]^2 + delta\_pos[2][k]^2}$$

7. add pairwise forces

8. update positions $pos[j][i] = pos[j][i] + dt * velo[j][i]$

9. update velocities:$velo[j][i] = velo[j][i] + dt * (f[j][i]/mass[i])$

2 performance profile

Before do the optimization, we need to profile the MD application.We use Intel advisor to profile the performance.The roofline of the application is given as bellow:

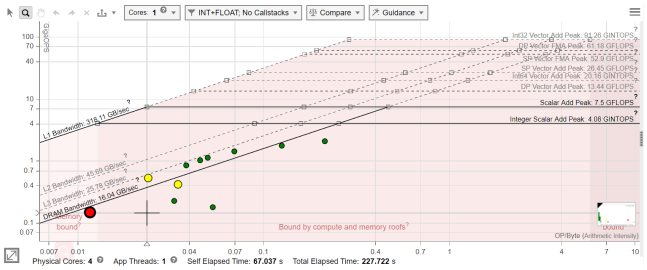
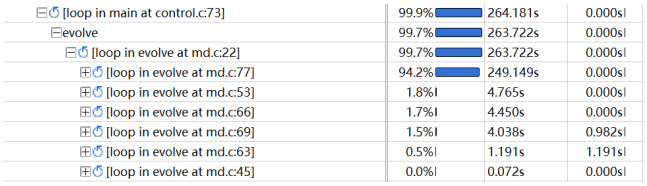


Figure 1: roofline of the MD application

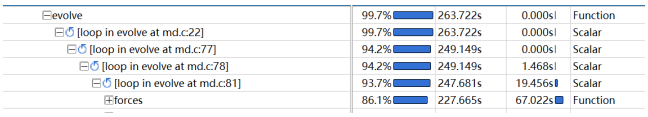
from the roofline figure, we can get that the bottle-neck of the application is bounded by memory. we have get that the main process of the application is the *evolve* function. The elapsed time of the code in *evolve* function is given as bellow (descending order):



[loop in main at control.c:73]	99.9%	264.181s	0.000s
evolve	99.7%	263.722s	0.000s
[loop in evolve at md.c:22]	99.7%	263.722s	0.000s
[loop in evolve at md.c:77]	94.2%	249.149s	0.000s
[loop in evolve at md.c:53]	1.8%	4.765s	0.000s
[loop in evolve at md.c:66]	1.7%	4.450s	0.000s
[loop in evolve at md.c:69]	1.5%	4.038s	0.982s
[loop in evolve at md.c:63]	0.5%	1.191s	1.191s
[loop in evolve at md.c:45]	0.0%	0.072s	0.000s

Figure 2: *evolve* function elapsed time diftribution (descending order)

From Fig.2 we can get that the bottle-neck of the application is the code in *md.c* line 77, Then in order to get more details, we get the figure from *intel advisor* and given as bellow:



evolve	99.7%	263.722s	0.000s	Function
[loop in evolve at md.c:22]	99.7%	263.722s	0.000s	Scalar
[loop in evolve at md.c:77]	94.2%	249.149s	0.000s	Scalar
[loop in evolve at md.c:78]	94.2%	249.149s	1.468s	Scalar
[loop in evolve at md.c:81]	93.7%	247.681s	19.456s	Scalar
forces	86.1%	227.665s	67.022s	Function

Figure 3: the bottle-neck of the code is the function $force$

3 optimization

From the performace profile part, we can get that the application is bounded by memory, and the $force$ function is the bottle-neck of the application. So our optimization will include the tips bellow:

1. focus on bottle-neck then do other optimization
2. merge some operations into one loop to reduce memory read and write
3. change the order of the loop to increase memory cache hit
4. optimize the force function.
5. reduce logical branches jump time.
6. reduce repeat operation

3.1 bottle-neck optimization

First the $force$ function is

Listing 1: force function

```
double forces(double Wv, double deltav, double rv){
  return Wv*deltav/(pow(rv,3.0));
}
```

We can get that the operation $pow(rv, 3.0)$ is much slower than the operation $rv * rv * rv$, so the new $force$ function is

Listing 2: modified force function

```
double forces(double Wv, double deltav, double rv){
  return Wv*deltav/(rv*rv*rv);
}
```

3.1.1 reduce logical branches and reduce Redundant operation

From Fig.2 we can get that the bottle-neck of the application is the code in *md.c* line 77, the code is given as bellow:

```
for(i=0;i<Nbody;i++)
{
  for(j=i+1;j<Nbody;j++)
  {
    size = radius[i] + radius[j];
    have_collided=0;
    for(l=0;l<Ndim;l++)
    {
      /* flip force if close in */
      if( delta_r[k] >= size )
      {
        f[l][i] = f[l][i] -
        forces(G*mass[i]*mass[j],delta_pos[l][k],delta_r[k]);
        f[l][j] = f[l][j] +
        forces(G*mass[i]*mass[j],delta_pos[l][k],delta_r[k]);
      }
      else
      {
        f[l][i] = f[l][i] +
        forces(G*mass[i]*mass[j],delta_pos[l][k],delta_r[k]);
        f[l][j] = f[l][j] -
        forces(G*mass[i]*mass[j],delta_pos[l][k],delta_r[k]);
        have_collided=1;
      }
    }
    if( have_collided == 1 )
    collisions++;
    k = k + 1;
  }
}
```

From the code we can get that the logical branch for $if(delta\_r[k] >= size)$ can put outside the third loop. And as for the $collisions++$ operation, we can remove the $if(have_collided == 1)$ branch, and add it into the $else$ branch of the code. Then as to the $force$ operation in the branch, its all the same operation, so we can just do it only once.

Then as to $forces(G * mass[i] * mass[j], delta_pos[l][k], delta_r[k])$, we can get that each inner loop only change is $delta_pos[l][k]$, so we can spilt the operation into two steps

Listing 3: split force into two steps

```
double G_mass_r3 = G * mass[i] * mass[j] / (delta_r[k] * delta_r[k] *
    delta_r[k]);
gravitation = G_mass_r3*delta_pos[l][k];
```

So we can get that for the bottle-neck of this part after optimization is :

Listing 4: split force into two steps

```
double gravitation;
for(i=0;i<Nbody;i++)
{
  for(j=i+1;j<Nbody;j++)
  {
    size = radius[i] + radius[j];
    double G_mass_r3 = G * mass[i] * mass[j] / (delta_r[k] *
        delta_r[k] * delta_r[k]);
    if (delta_r[k] >= size)
    {
      for (l = 0; l < Ndim; l++)
      {
        gravitation = delta_pos[l][k] * G_mass_r3;
        f[l][i] -= gravitation;
        f[l][j] += gravitation;
      }
    }
    else
    {
      for (l = 0; l < Ndim; l++)
      {
        gravitation = delta_pos[l][k] * G_mass_r3;
        f[l][i] += gravitation;
        f[l][j] -= gravitation;
      }
    collisions++;}
    k = k + 1;}}
```

3.2 merge operation and unroll loop

From the equation eq.(1) eq.(2) eq.(3) eq.(4), we can get that the 4 steps will visit the array f many times, but if we use unroll and merge the 4 equations into one, will just need to visit f only once. and for the distance variable r, we nolonger need to visit the r array, we just use a temp variable replace. So the 4 equations code after optimization is given as bellow:

Listing 5: loop unroll and merge the 4 equations

```
for (i = 0; i < Nbody; i++)
{
   double tmp = sqrt(pos[0][i] * pos[0][i] + pos[1][i] * pos[1][i] +
       pos[2][i] * pos[2][i]);
   tmp = G * mass[i] * M_central / (tmp*tmp*tmp);
   f[0][i] = -vis[i] * (velo[0][i] + wind[0]) - tmp* pos[0][i];
   f[1][i] = -vis[i] * (velo[1][i] + wind[1]) - tmp* pos[1][i];
   f[2][i] = -vis[i] * (velo[2][i] + wind[2]) - tmp* pos[2][i];
}
```

the steps 5,6,7 in section 1 is compute *delta_pos* and $delta_r$, then use *delta_pos* and $delta_r$ to compute pairwise forces. so we can also merge the three steps into one. So we can merge the steps 5,6 and the code in *Listing* 4 into one loop. the optimized code is given as bellow:

Listing 6: merge steps 5,6,7 into one step

```
k = 0;
double delta_r_k;
double gravitation;
double tmp_pos[Ndim];
double tmp_fli[Ndim];
for(i=0;i<Nbody;i++)
{

   for (l = 0; l < Ndim; l++)
   {
      tmp_pos[l] = pos[l][i];
      tmp_fli[l] = f[l][i];
   }
   for(j=i+1;j<Nbody;j++)
   {
      delta_pos[0][k] = tmp_pos[0] - pos[0][j];
      delta_pos[1][k] = tmp_pos[1] - pos[1][j];
      delta_pos[2][k] = tmp_pos[2] - pos[2][j];

      delta_r_k = sqrt(delta_pos[0][k] * delta_pos[0][k] +
      delta_pos[1][k] * delta_pos[1][k] +
      delta_pos[2][k] * delta_pos[2][k]);
      size = radius[i] + radius[j];
```

```
    double G_mass_r3 = G * mass[i] * mass[j] / (delta_r_k * delta_r_k
        * delta_r_k);
    if (delta_r_k >= size)
    {
      for (l = 0; l < Ndim; l++)
      {
        gravitation = delta_pos[l][k] * G_mass_r3;
        tmp_fli[l] -= gravitation;
        f[l][j] += gravitation;
      }
    }
    else
    {
      for (l = 0; l < Ndim; l++)
      {
        gravitation = delta_pos[l][k] * G_mass_r3;
        tmp_fli[l] += gravitation;
        f[l][j] -= gravitation;
      }
      collisions++;
    }
    k = k + 1;
  }
  for (l = 0; l < Ndim; l++)
  f[l][i] = tmp_fli[l];
}
```

From the code, we can get that in-order to reduce the visit time of $pos[l][i]$ and $f[l][i]$, we store them in two tmp array tmp_pos and tmp_fli

3.3 merge update positions and velocities into one loop

for the positions and velocities can be compute in one loop (steps 8,9), then we merge them into one loop and in order to reduce cache hit, we use unroll. the optimized code is given as bellow:

Listing 7: merge steps 8,9 into one step

```
  for(i=0;i<Nbody;i++)
  {
    pos[0][i] = pos [0][i] + dt * velo[0][i];
    velo[0][i] = velo[0][i] + dt * (f[0][i] / mass[i]);

    pos[1][i] = pos[1][i] + dt * velo[1][i];
    velo[1][i] = velo[1][i] + dt * (f[1][i] / mass[i]);

    pos[2][i] = pos[2][i] + dt * velo[2][i];
    velo[2][i] = velo[2][i] + dt * (f[2][i] / mass[i]);
  }
```

4 correctness check and result

4.1 correctness check

for the original application is so slow, when we do the correctness check, we set $Nsteps = 50, Nsave = 5$ to do the correctness check. We use $diff - output$ to compare the output data. We can find that the original application result and the optimized application result is all the same.

4.2 summary

We can get that the original application includes 9 steps from section 1. But after the optimization, the application only includes 3 steps

1. calculate the central force for each dimension
2. add pairwise forces
3. update positions and velocities

To optimize the code, the skill we use is unroll loop, merge operations, reduce logical branches, reduce memory access, change the order of loop, use multiply to replace *pow* operation.

4.3 compare the performance

1. 250 timesteps took 21.814722 seconds for **optimized code**
2. 250 timesteps took 995.058564 secondss for **original code**

So we can get that the optimized code is 50 faster than the original code.

4.4 test platform

1. compile tool : gcc
2. cpu : i7-8550U CPU @ 1.80GHz
3. system: ubuntu
4. profile tool: intel advisor