

Labo Parallele

Robbe Goovaerts, Paul Leroy

14/05/2018

1 Varianten v/h programma

1.1 CPU

Het berekenen van de positie en snelheid gebeurt op basis van 2 geneste for-loops. Het nadeel hiervan is dat elke berekening serieel gebeurt waardoor de snelheid lager is.

```
1  for (int i = 0; i < length; ++i)
2  {
3      for (int j = 0; j < length; ++j)
4      {
5
6          if (i == j)
7              continue;
8
9          cl_float3 pos_a = host_pos[i];
10         cl_float3 pos_b = host_pos[j];
11
12         float dist_x = (pos_a.s[0] - pos_b.s[0]) *
distance_to_nearest_star;
13         float dist_y = (pos_a.s[1] - pos_b.s[1]) *
distance_to_nearest_star;
14         float dist_z = (pos_a.s[2] - pos_b.s[2]) *
distance_to_nearest_star;
15
16
17         float distance = sqrt(
18             dist_x * dist_x +
19             dist_y * dist_y +
20             dist_z * dist_z);
21
22         float force_x = -mass_grav * dist_x / (distance * distance *
distance);
23         float force_y = -mass_grav * dist_y / (distance * distance *
distance);
24         float force_z = -mass_grav * dist_z / (distance * distance *
distance);
25
26         float acc_x = force_x / mass_of_sun;
27         float acc_y = force_y / mass_of_sun;
28         float acc_z = force_z / mass_of_sun;
29
30         host_speed[i].s[0] += acc_x * delta_time;
```

```

31         host_speed[i].s[1] += acc_y * delta_time;
32         host_speed[i].s[2] += acc_z * delta_time;
33
34
35     }
36 }
37
38

```

1.2 V1: parallellisatie (kernel.cl)

Hierbij hebben we de buitenste for-loop geparallelliseerd. In de kernel file staat de integer *i* gedefinieerd die steeds automatisch zal incrementeren telkens wanneer de kernel wordt opgeroepen. De GPU kan meerdere threads tegelijk berekenen. Bij meerdere threads zal er een aanzienlijk snelheidsverschil te merken zijn.

```

1     if(i>=length) {
2         return;
3     }
4
5     for (int j = 0; j < length; ++j)
6     {
7         if (i == j)
8             continue;
9
10        float3 pos_a = host_pos[i];
11        float3 pos_b = host_pos[j];
12
13        float dist_x = (pos_a.s0 - pos_b.s0) * distance_to_nearest_star;
14        float dist_y = (pos_a.s1 - pos_b.s1) * distance_to_nearest_star;
15        float dist_z = (pos_a.s2 - pos_b.s2) * distance_to_nearest_star;
16
17
18        float distance = sqrt(
19            dist_x * dist_x +
20            dist_y * dist_y +
21            dist_z * dist_z);
22
23        float force_x = -mass_grav * dist_x / (distance * distance *
24            distance);
25        float force_y = -mass_grav * dist_y / (distance * distance *
26            distance);
27        float force_z = -mass_grav * dist_z / (distance * distance *
28            distance);
29
30        float acc_x = force_x / mass_of_sun;
31        float acc_y = force_y / mass_of_sun;
32        float acc_z = force_z / mass_of_sun;
33
34        host_speed[i].s0 += acc_x * delta_time;
35        host_speed[i].s1 += acc_y * delta_time;
36        host_speed[i].s2 += acc_z * delta_time;
37    }
38

```

```

38
39     host_pos[i].s0 += (host_speed[i].s0 * delta_time) /
distance_to_nearest_star;
40     host_pos[i].s1 += (host_speed[i].s1 * delta_time) /
distance_to_nearest_star;
41     host_pos[i].s2 += (host_speed[i].s2 * delta_time) /
distance_to_nearest_star;
42
43 }
44

```

1.3 V2: Atomische operaties toevoegen (kernel2.cl)

In deze code maken we gebruik van de functie `atomic_add()`. Met de CPU moeten de standaard Read,Modify en Write operaties uitgevoerd worden. Met de `atomic_add()` functie maak je gebruik van speciale hardware die de GPU heeft ingebouwd. Hierdoor verzekeren we dat steeds 1 thread RMW uitvoert zodat er geen fouten ontstaan.

```

1     typedef union
2     {
3         float3 vec;
4         float arr[3];
5     } float3_;
6
7     __kernel void simulate_gravity( __global float3 *host_pos, __global float3_
*host_speed, const int length)
8     {
9         const int i = get_global_id(0);
10
11         const float delta_time = 1.f;
12         // const float grav_constant = 6.67428e-11;
13         const float grav_constant = 1;
14         const float mass_of_sun = 2;
15         const float mass_grav = grav_constant * mass_of_sun * mass_of_sun;
16         const float distance_to_nearest_star = 50;
17
18
19         if(i>=length) {
20             return;
21         }
22
23         for (int j = 0; j < length; ++j)
24         {
25             if (i == j)
26                 continue;
27
28             float3 pos_a = host_pos[i];
29             float3 pos_b = host_pos[j];
30
31             float dist_x = (pos_a.s0 - pos_b.s0) * distance_to_nearest_star;
32             float dist_y = (pos_a.s1 - pos_b.s1) * distance_to_nearest_star;
33             float dist_z = (pos_a.s2 - pos_b.s2) * distance_to_nearest_star;
34
35
36             float distance = sqrt(
37                 dist_x * dist_x +

```

```

38         dist_y * dist_y +
39         dist_z * dist_z);
40
41         float force_x = -mass_grav * dist_x / (distance * distance *
42 distance);
43         float force_y = -mass_grav * dist_y / (distance * distance *
44 distance);
45         float force_z = -mass_grav * dist_z / (distance * distance *
46 distance);
47
48         float acc_x = force_x / mass_of_sun;
49         float acc_y = force_y / mass_of_sun;
50         float acc_z = force_z / mass_of_sun;
51
52         // host_speed[i].s0 += acc_x * delta_time;
53         // host_speed[i].s1 += acc_y * delta_time;
54         // host_speed[i].s2 += acc_z * delta_time;
55
56         AtomicAdd(&host_speed[i].arr[0], (float)(acc_x * delta_time));
57         AtomicAdd(&host_speed[i].arr[1], (float)acc_y * delta_time);
58         AtomicAdd(&host_speed[i].arr[2], (float)acc_z * delta_time);
59
60     }
61
62     host_pos[i].s0 += (host_speed[i].vec.s0 * delta_time) /
63 distance_to_nearest_star;
64     host_pos[i].s1 += (host_speed[i].vec.s1 * delta_time) /
65 distance_to_nearest_star;
66     host_pos[i].s2 += (host_speed[i].vec.s2 * delta_time) /
67 distance_to_nearest_star;
68 }

```

2 Vergelijking

In de tabel weergegeven in figuur 1 ziet u de resultaten van onze metingen. Deze resultaten hebben we ook gevisualiseerd in een grafiek. Hier ziet u heel duidelijk dat hoe meer punten er berekend moeten worden, hoe groter het belang van parallelisatie is.

Figuur 1: Resultaten

	V1	CPU	V2
10	0,153335	0,148368	0,143848
100	0,147805	0,149218	1,151004
1000	0,143767	0,303834	0,151156
10000	0,35235	22,948522	1,823096

Figuur 2: Grafiek
Vergelijking

