OpenCL n-body

Vandevelde, Simon (simon.vandevelde@student.kuleuven.be)
Van Assche, Dylan (dylan.vanassche@student.kuleuven.be)

21 mei 2018

1 Testomgeving

1.1 Compilatieproblemen

De GLM package gaf bij compiletime een error wat ervoor zorgde dat de code niet gecompileerd kon worden in onze testomgeving. Een update heeft dit probleem gelukkig verholpen.

1.2 Hardware

- CPU: Intel(R) Core(TM) i7-4790 CPU @ 3.60Ghz
- GPU: Nvidia GTX 970
- RAM: 8GiB DDR3 @ 1866MHz
- OS: Fedora KDE Plasma

2 OpenCL aanpassingen

2.1 Eerste for-lus

De eerste for-lus, die we geparalleliseerd hebben, berekent de nieuwe snelheid van een lichaam. Hiervoor moet de host de positie en snelheden data overbrengen naar de GPU, wachten tot de GPU klaar is en dan de nieuwe snelheden ophalen van de GPU.

Listing 1: De eerste for-lus

```
for (int i = 0; i < length; ++i)
{
    for (int j = 0; j < length; ++j)
    {

        if (i == j)
            continue;
```

```
        cl_float3 pos_a = host_pos[i];
        cl_float3 pos_b = host_pos[j];

        float dist_x = (pos_a.s[0] - pos_b.s[0]) *
            distance_to_nearest_star;
        float dist_y = (pos_a.s[1] - pos_b.s[1]) *
            distance_to_nearest_star;
        float dist_z = (pos_a.s[2] - pos_b.s[2]) *
            distance_to_nearest_star;


        float distance = sqrt(
                dist_x * dist_x +
                dist_y * dist_y +
                dist_z * dist_z);

        float force_x = -mass_grav * dist_x / (distance * distance
            * distance);
        float force_y = -mass_grav * dist_y / (distance * distance
            * distance);
        float force_z = -mass_grav * dist_z / (distance * distance
            * distance);

        float acc_x = force_x / mass_of_sun;
        float acc_y = force_y / mass_of_sun;
        float acc_z = force_z / mass_of_sun;

        host_speed[i].s[0] += acc_x * delta_time;
        host_speed[i].s[1] += acc_y * delta_time;
        host_speed[i].s[2] += acc_z * delta_time;
    }

}
```

2.2 Tweede for-lus

De tweede for-lus, die we geparalleliseerd hebben, berekent de nieuwe positie van elk lichaam. Hiervoor zal de host eerst de positie en snelheden overdragen naar de GPU, wachten tot de GPU klaar is en dan de nieuwe positie data ophalen van de GPU.

Listing 2: De tweede for-lus

```
for(int i = 0; i < length; ++i)
{
    host_pos[i].s[0] += (host_speed[i].s[0] * delta_time) /
        distance_to_nearest_star;
    host_pos[i].s[1] += (host_speed[i].s[1] * delta_time) /
        distance_to_nearest_star;
    host_pos[i].s[2] += (host_speed[i].s[2] * delta_time) /
        distance_to_nearest_star;
}
```

2.3 Atomische variant van de eerste for-lus

Omdat het algoritme in for-lus 1 de afstand bepaalt tussen twee lichamen krijgen we een *race conditie*. Het probleem situeert zich bij het feit dat een GPU processor de positie van lichaam A en lichaam B moet lezen om de afstand van lichaam B tot lichaam A te vinden. De kans bestaat dat een andere GPU processor net de positie van lichaam A aan het wijzigen is terwijl deze net ingelezen wordt. Het gevolg hiervan is dat de waarden verschillen per keer dat we het programma uitvoeren.

We kunnen dit omzeilen door gebruik te maken van *atomische operaties*. Hierbij wacht een processor met het inlezen van een geheugenplaats als een andere processor net naar deze geheugenplaats aan het schrijven is en omgekeerd. Een nadeel van deze methode is dat de performantie zal dalen omdat de processor blijft wachten tot een andere processor klaar is met zijn dataoverdracht. OpenCL ondersteunt dit niet rechtstreeks maar via een omweg (via een `union`) kunnen we toch gebruik maken van *atomische operaties*.

2.4 Combinatie van beide for-lussen

Hierbij hebben we de twee bovenstaande for-lussen (atomische for-lus 1 en for-lus 2) gecombineerd tot één OpenCL programma. Beide lussen staan nog steeds in een aparte kernel. Indien we deze nog zouden combineren tot één kernel zouden we waarschijnlijk de performantie nog kunnen verbeteren (zie 4.1).

2.5 Afstanden

Een extra uitbreiding aan het programma was het opvragen van de afstanden tussen elk lichaam en de andere lichamen:

- Gemiddelde afstand
- Minimale afstand
- Maximale afstand

3 Resultaten

Om onze resultaten te verwerken hebben we een klein Python script geschreven dat alle timestamps inleest en het gemiddelde neemt van de 100 eerste metingen per categorie. Deze gemiddelden worden dan uitgezet op een aantal grafieken welke hieronder zullen worden besproken.

Op de x-as van de volgende grafieken staat steeds het gebruikte programma en de gebruikte kernel.

0. De waarden gemeten bij de cpu. Hier is enkel de simulationtijd kunnen gemeten worden.
1. Deze vervangt de tweede for-lus, en doet dit niet atomisch.

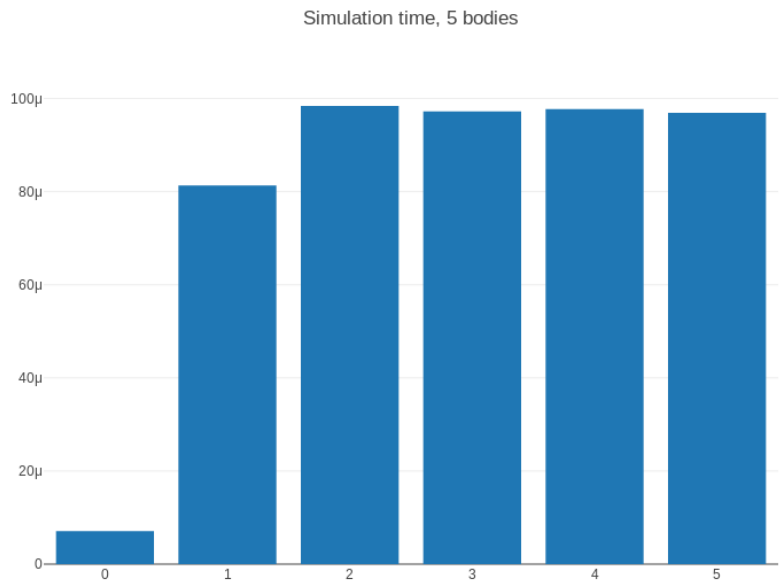
2. Deze vervangt de eerste for-lus, en doet dit ook niet atomisch. Dit zorgt voor racecondities waardoor dit programma eigenlijk niet betrouwbaar is

3. Deze vervangt de eerste for-lus, maar doet dit wel atomisch.

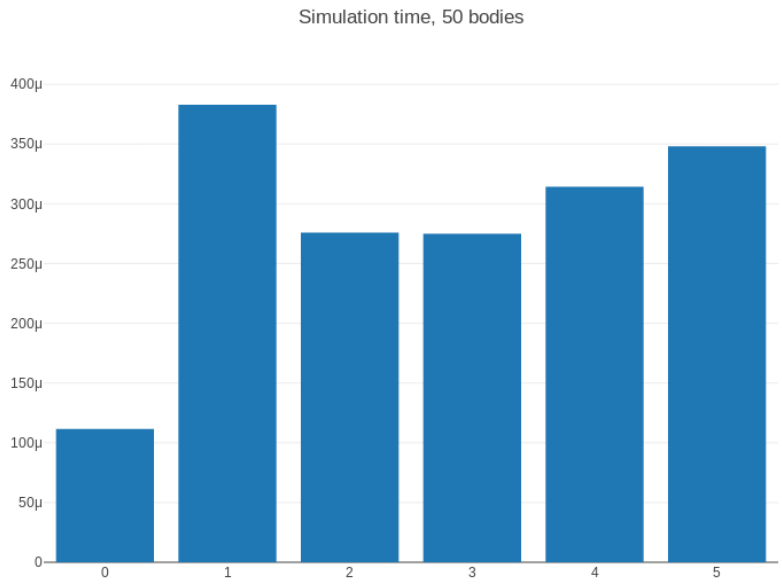
4. Deze vervangt de tweede for-lus door een atomische variant, hoewel dit helemaal niet nodig is aangezien er geen racecondities zijn in deze lus.

5. Deze zet beide atomische versies (for-lus 1 & for-lus 0) samen.

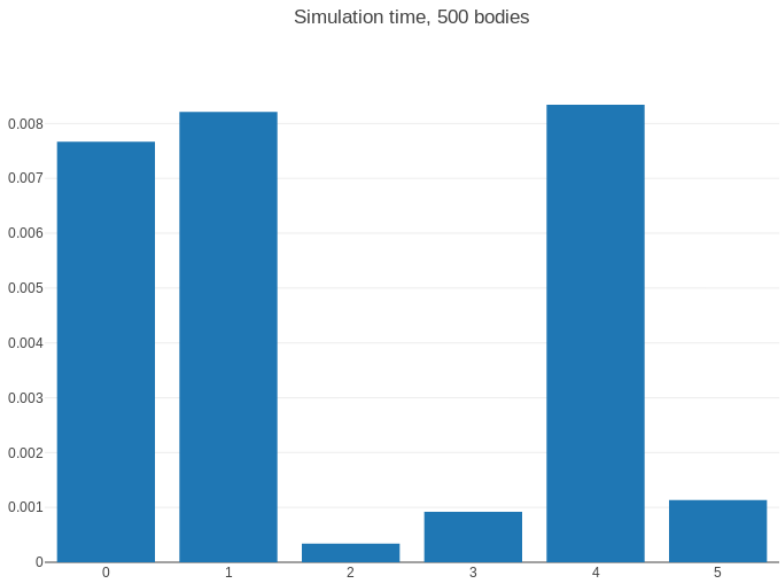
3.1 5 lichamen



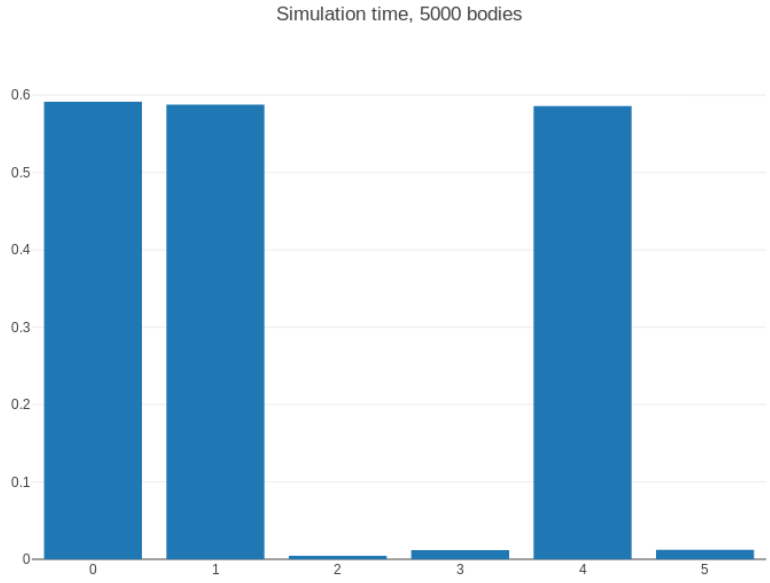
3.2 50 lichamen

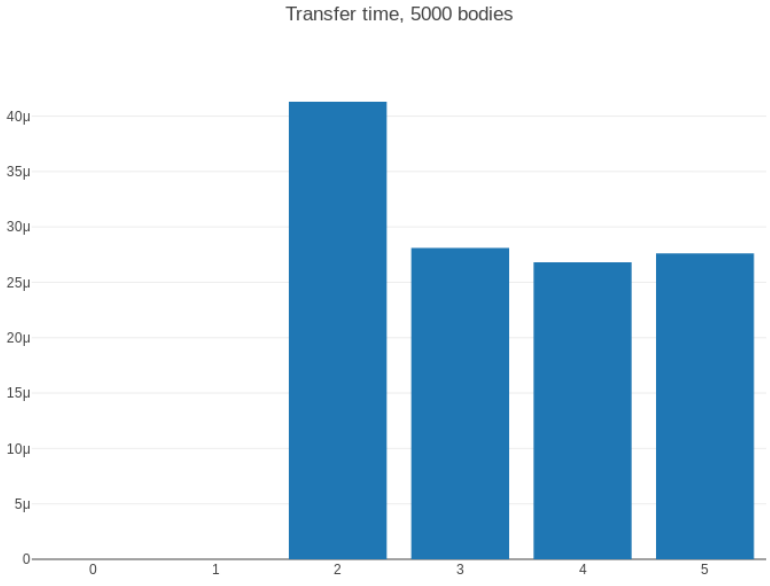


3.3 500 lichamen



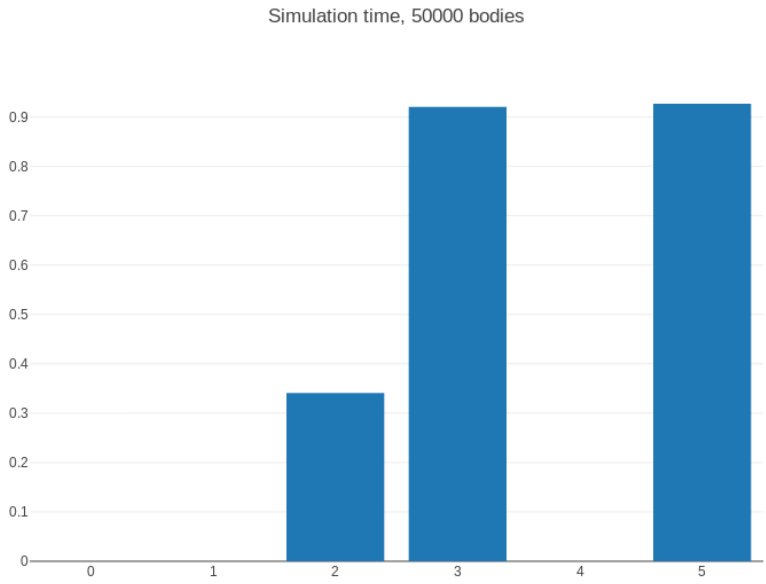
3.4 5 000 lichamen



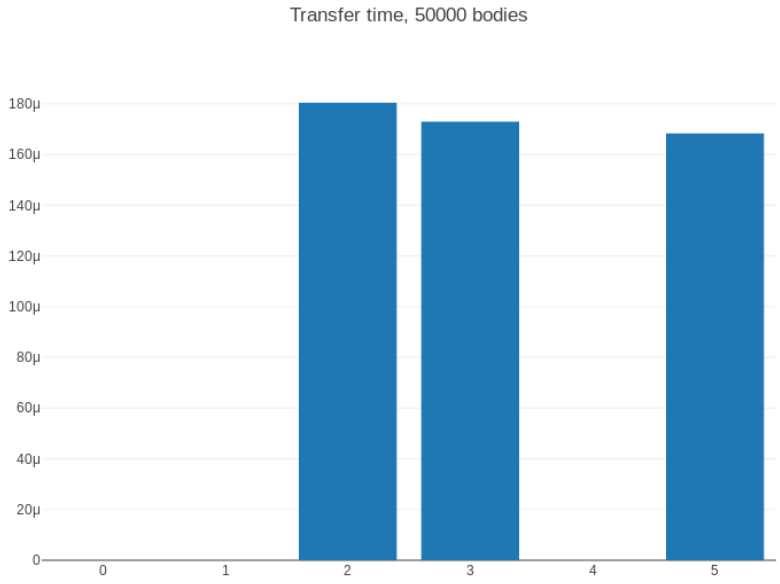


3.5 50 000 lichamen

opm: als de tijd gelijk is aan nul, betekent dit dat het programma weigerde te



starten.



Hier is heel duidelijk ook hoeveel impact de transfersnelheid heeft op het programma. Tussen 5000 en 50000 bodies is de transfertijd soms 5 keer groter.

3.6 500 000 lichamen

CPU faalt!

4 Mogelijke verbeteringen

Het is nog mogelijk om de laatste versie van het programma nog te versnellen door gebruik te maken van nog enkele OpenCL functies en ontwerppatronen. Deze werden niet geïmplementeerd wegens tijdsgebrek.

4.1 Minder dataoverdrachten

Het programma kopieert nu telkens per OpenCL kernel de data van de host naar de GPU en omgekeerd. Dit resulteert in vier dataoverdrachten, wat erg veel nutteloze overhead is, omdat beide for-lussen draaien als een aparte kernel.

We zouden de for-lussen kunnen overzetten in één kernel waardoor de dataoverdrachten worden gereduceerd tot slechts twee overdrachten.

4.2 Efficiënt opsplitsen in werkgroepen

Beide for-lussen worden simpel weg op de GPU uitgevoerd zonder dat het algoritme geoptimaliseerd werd voor parrallelisme op de GPU. Dit resulteert in een slechter gebruik van het geheugen (globaal i.p.v. lokaal) waardoor de berekeningen trager kunnen verlopen door het geheugen bottleneck.

Elke processor kan aan de hele dataset. Beter gezegd: het geheugen is volledig gedeeld onder elke processor. Hierdoor staat dit globaal geheugen verder weg van de processor kern en duurt het dus langer vooraleer de data arriveert bij de processor.

Als we gebruik maken van het lokaal geheugen van een werkgroep kunnen we dit probleem voorkomen. Maar dit vereist tevens ook dat we de berekeningen zodanig kunnen opsplitsen zodat ze in een werkgroep passen.

4.3 Coalesced memory access

Indien we het programma zouden optimaliseren om zijn geheugentoegangen te limiteren tot zijn buren kunnen we een hogere snelheid halen omdat de naburige werkitems sneller toegankelijk zijn dan werkitems die aan de andere kant van de GPU liggen. Maar dit vereist opnieuw een ingreep op het algoritme zoals reeds besproken in 4.2, welke iets uitgebreider is omdat men rekening moet houden met de buren van elk workitem.

5 Conclusie