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.

Bestanden: `n-body-2.c`, `kernel-2.cl`

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.

Bestanden: `n-body-1.c`, `kernel-1.cl`

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 2 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*.

Bestanden: `n-body-3.c`, `kernel-3.cl`

2.4 Atomische variant van de tweede for-lus

Ondanks dat for-lus 2 geen *race condities* bevat hebben we toch eens de invloed van atomische operaties op deze for-lus getest.

Bestanden: `n-body-4.c`, `kernel-4.cl`

2.5 Combinatie van beide for-lussen

Hierbij hebben we de 2 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).

Bestanden: `n-body-5.c`, `kernel-5.cl`

2.6 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

Ook hier hebben we opnieuw last van *race condities* waardoor we deze ook atomisch zouden moeten uitvoeren of door elk lichaam een volledig array te

geven om zijn data in op te slaan. Daarna kan er in een aparte kernel dan al deze data samengevoegd worden.

Helaas door tijdsgebrek hebben we deze niet volledig kunnen afwerken, daarom hebben we deze ook niet in onze testresultaten weergegeven.

Bestanden: `n-body-6.c`, `kernel-6.cl`

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.

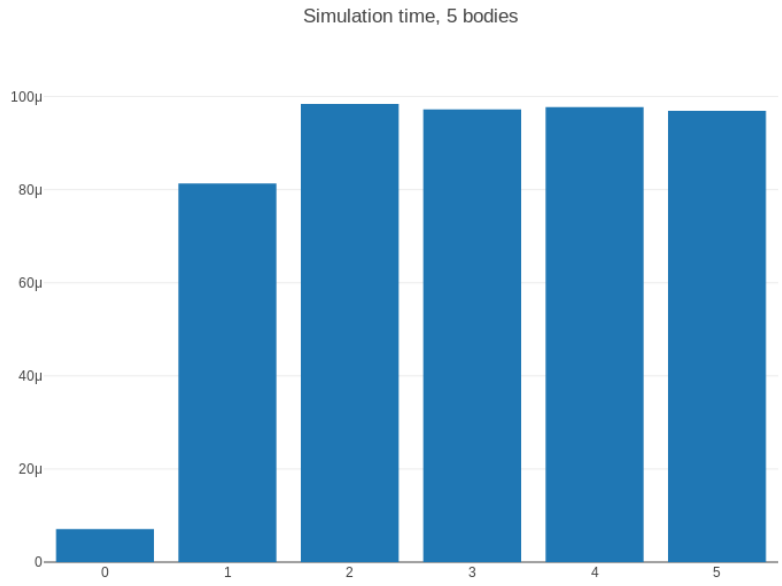
Op de x-as van de volgende grafieken staat steeds het gebruikte programma en de gebruikte kernel.

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.

Opmerking: Als de tijd gelijk is aan nul, betekent dit de tijd zo klein is dat deze niet gemeten kon worden of dat het programma niet kon starten door een te hoog aantal lichamen.

3.1 5 lichamen

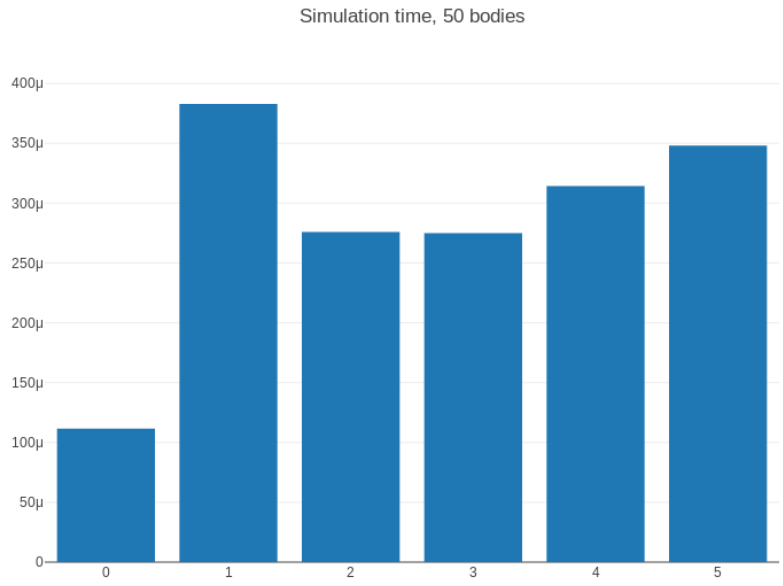
Hier zien we duidelijk dat de CPU veel sneller is dan de GPU bij het berekenen van slechts 5 lichamen. De oorzaak is vrij duidelijk: de overhead bij het kopiëren van de data van de host naar de GPU en terug zorgt ervoor dat de GPU hier tot 10 maal trager is dan de CPU. Ook het opstarten van de OpenCL interface zorgt voor enige overhead.



Figuur 1: Simulatietijd 5 lichamen

3.2 50 lichamen

Vanaf 50 lichamen is de CPU nog steeds 2 - 3 keer sneller dan de GPU. Het aantal lichamen stijgt waardoor de overhead ten opzichte van de hoeveelheid data is verminderd. De verschillen tussen de OpenCL varianten zijn niet erg groot op dit moment. Maar dit zal veranderen als het aantal lichamen nog stijgt.

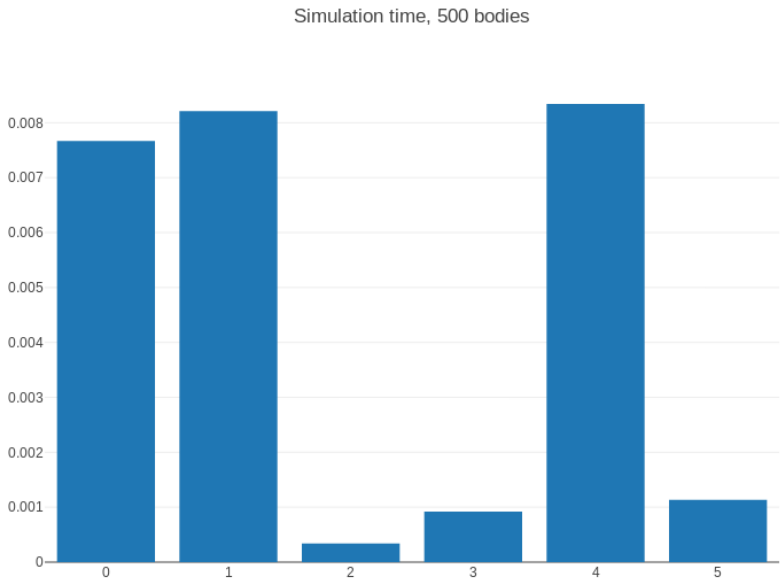


Figuur 2: Simulatietijd 50 lichamen

3.3 500 lichamen

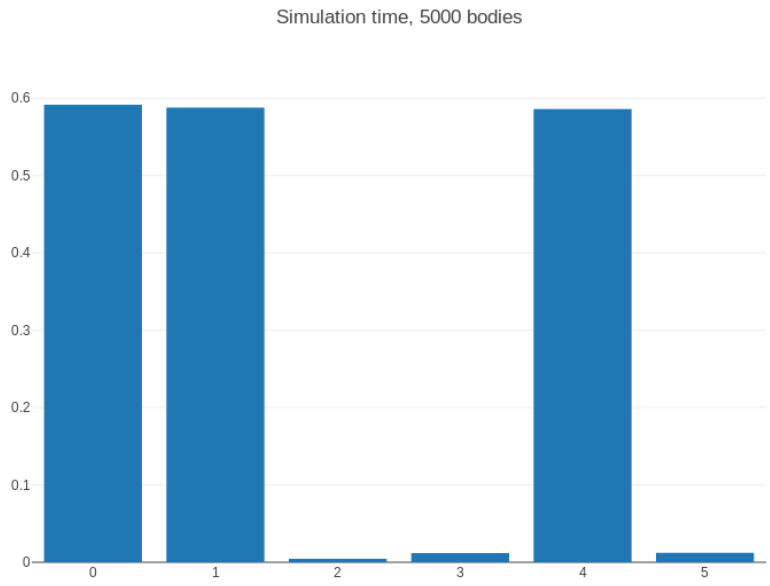
Vanaf 500 lichamen bereiken we het breekpunt van de CPU, vanaf nu is de GPU sneller om de gravitatie te simuleren dan de CPU.

- Kernel 1 en 2 zijn niet-atomisch uitgevoerd (zie 2.1 en 2.2) waardoor kernel 2 zijn performantie hoger is dan zijn atomische variant. Maar kernel 2 heeft last van *race condities* wat dus foutieve resultaten oplevert.
- Kernel 3 lost de *race condities* op van kernel 2, hierdoor is deze wel trager maar levert hij wel de juiste resultaten op.
- Kernel 4 is gelijkaardig aan kernel 1, de oorzaak hiervan is dat we de atomische operaties ook getest hebben op for-lus 2 (zie 2.4). Maar aangezien deze geen *race condities* bevat, zien we ook geen verschil in performantie. Enkel de kleine for-lus 2 op de GPU draaien heeft dus weinig zin.
- Kernel 5 presteert duidelijk beter als zijn voorgangers. Door de combinatie te maken van beide for-lussen (zie 2.5) verkrijgen we een betere performantie omdat alles op de GPU berekent werd. Deze kernel combineert eigenlijk kernel 3 en 4 waardoor het aantal dataoverdrachten stijgt. Deze zouden we nog kunnen optimaliseren, hierover later meer in paragraaf 4.1.



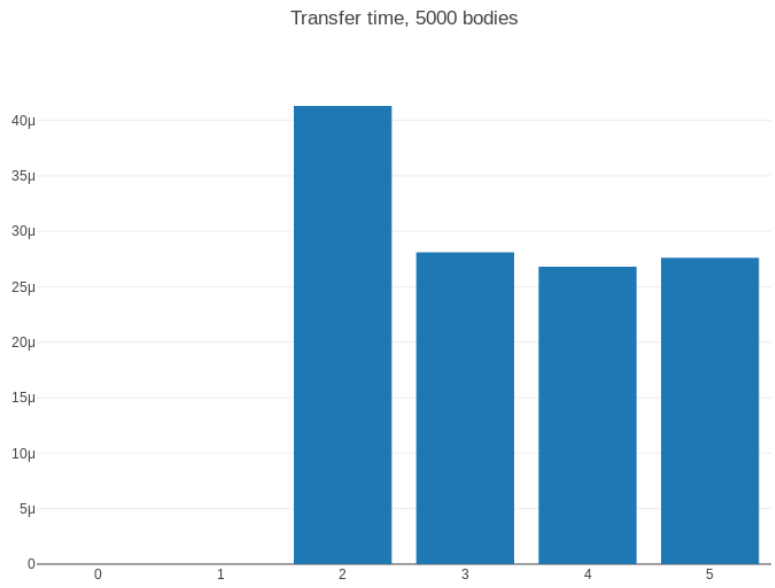
Figuur 3: Simulatietijd 500 lichamen

3.4 5 000 lichamen



Figuur 4: Simulatietijd 5 000 lichamen

Hier zien we dezelfde trend als we reeds besproken hebben bij 500 lichamen (paragraaf 3.3). De kernels 2, 3 en 5 zijn beduidend sneller dan de CPU versie en kernels 1 en 4. Kernel 2 mogen we niet rekenen als sneller omdat deze foutieve resultaten oplevert. Kernel 1 en 4 parallelliseren de kleine for-lus 2 wat niet eg veel verbetering met zich meebrengt op vlak van performantie omdat de bewerking te kort is.

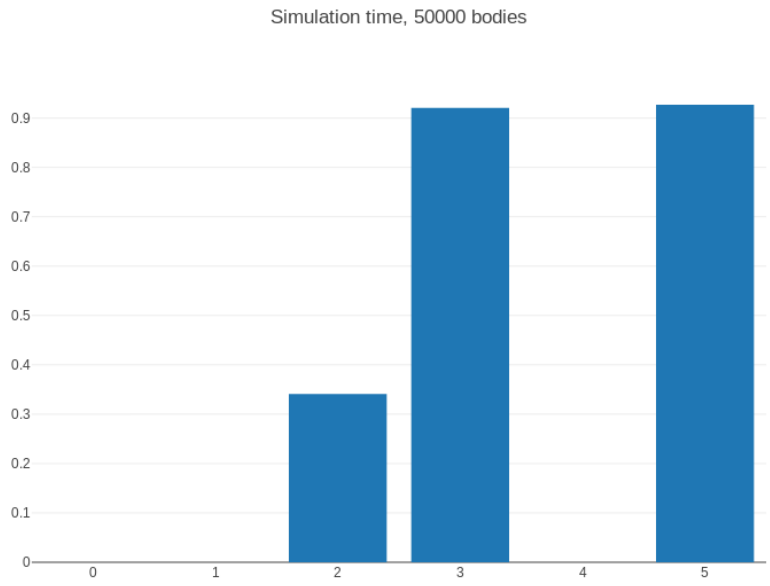


Figuur 5: Transfertijd 5 000 lichamen

We zien duidelijk dat de transfertijd van kernel 4 veel meer is dan kernel 1.

Opmerking: Er is geen transfertijd gemeten bij de CPU versie omdat er geen overdracht plaats vindt.

3.5 50 000 lichamen

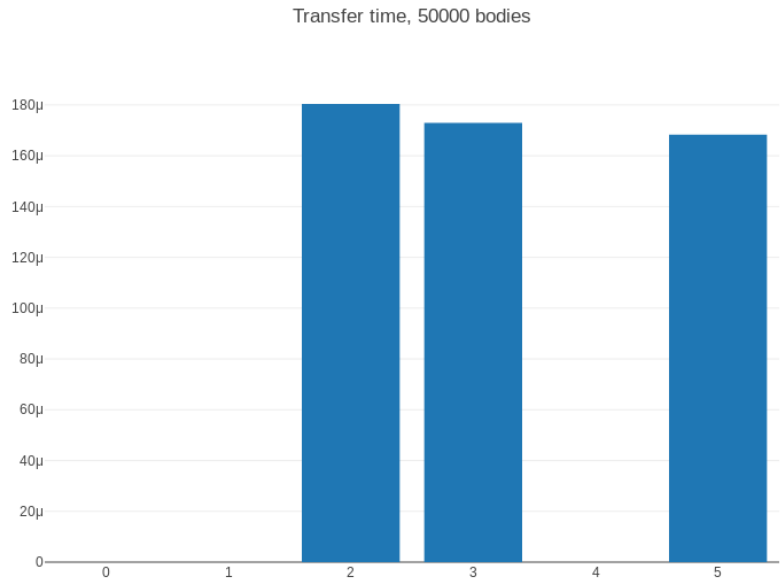


Figuur 6: Simulatietijd 50 000 lichamen

Vanaf 50 000 lichamen kunnen de CPU versie 0 en kernels 1 en 4 niet meer opstarten. Kernels 3 en 5 leveren daarentegen nog steeds een uitstekende performantie, op slechts 0.9 seconden kunnen zij de 50 000 lichamen berekenen. Dit is wel 100 maal trager dan bij 5 000 lichamen, de oorzaak hiervan is de GPU hardware zelf. Deze was op dat moment 100% in gebruik. Het gevolg hiervan is dat de berekeningen in wachtrij worden gezet tot de GPU klaar is voor de volgende taak.

Deze kernels leveren echter wel een groot verschil op in simulatietijd ten opzichte van de CPU versie die bij 5 000 lichamen er al reeds 0.6 s over deed. Zij doen er slechts 0.9 s erover om 50 000 lichamen uit te rekenen. Het aantal lichamen vertienvoudigt maar de simulatietijd stijgt slechts met 50% ten opzichte van de CPU versie van 5 000 lichamen.

Bij 500 000 lichamen geven deze kernels het helaas ook op en vereisen deze ook nog ingrijpende aanpassingen om de performantie te verhogen zoals we zullen bespreken in 4.



Figuur 7: Transfertijd 50 000 lichamen

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 doordat er mee data van en naar GPU gekopieerd moet worden.

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 4 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 data-overdrachten worden gereduceerd tot slechts 2 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