Hochschule für Technik und Wirtschaft Berlin

University of Applied Sciences

Master

Angewandte Informatik

Studienjahr - **SS18**

Aleksei Piatkin – 548482

Aray Karjauv – 562411

Semesterprojekt

N-Körper Simulation

Dokumentation

20. 06. 2018

Berlin

Inhaltverzeichnis

1. N-Körper Problem

Unter das N-Körper Problem in der Physik versteht man das Problem der Vorhersage der individuellen Bewegungen einer Gruppe von Himmelskörpern, die gravitativ miteinander interagieren. Leute haben versucht die Bewegungen der Sonne, des Mondes, der Planeten und der sichtbaren Sterne zu verstehen, dadurch wurde sie motiviert, die Lösung dieses Problems zu finden. Das N-Körper-Problem in der Allgemeinen Relativitätstheorie ist wesentlich schwieriger zu lösen.

Das klassische physikalische Problem kann grob wie folgt angegeben werden: Angesichts der quasistabilen Orbital-Eigenschaften (momentane Position, Geschwindigkeit und Zeit) einer Gruppe von Körpern (z.B. Himmelskörpern), sollen ihre interaktiven Kräfte prognostiziert werden - folglich, ihre Orbital-Bewegungen für alle zukünftigen Zeiten sollen auch vorhergesagt werden.

Zielformulierung

Es gibt eine Menge von n Körper - b_1, b_2, ..., b_n, wobei jeder Körper b_i hat eine Masse $\mathbf{m_i}$, eine Geschwindigkeit v_i und eine Position $\mathbf{r_i}$. Die Entfernung zwischen zwei Körpern b_i und b_j ist $\mathbf{r_{ij}}$, und die Gravitationskraft auf b_i als Ergebnis von b_j ist $\mathbf{f_{ij}}$.

$\mathbf{f_i}$ ist die gesamte Gravitationskraft auf einen Körper b_i. Für jede Iteration soll bei einem Zeitschritt **Δt** die neuen Positionen jedes Körpers berechnet werden. Dies kann in drei Phasen geschehen:

1) Zuerst wird Beschleunigung $\mathbf{f_{ij}}$ für alle Körperpaare berechnet:

$$f_{ij} = \gamma \sum_{j=1}^{N} \frac{m_j \vec{r}_{ij}}{(\|\vec{r}_{ij}\|^2 + \epsilon^2)^{\frac{3}{2}}}$$

   wobei Y - die Gravitationskonstante, e - ein softening factor.

2) Nachdem wird die Geschwindigkeit $\mathbf{v_i}$ und die Position $\mathbf{r_i}$ jedes Körpers mit Hilfe der klassischen Kraftgleichung F = ma aktualisieren:

$$\Delta v_i = f_i r_{ij}$$

$$v_i' = v_i + \Delta v_i$$

$$r_i' = r_i + v_i \Delta t + \frac{\Delta v_i}{2} \Delta t^2$$

Nun haben wir die aktualisierten Positionen in $\mathbf{r_i}$ und können für einen weiteren Zeitschritt **Δt** wiederholen.

2. Algorithmen

Mit der Computertechnologie erschien eine echte Gelegenheit, die Eigenschaften von Systemen gravitationsartiger Körper durch eine numerische Lösung des Bewegungsgleichungssystems zu untersuchen. Dazu wird beispielsweise die Runge-Kutta-Methode (vierte oder höhere Ordnung) verwendet.

Numerische Methoden konfrontieren mit den gleichen Problemen wie analytische Methoden - enge Körperbegegnungen müssen den Integrationsschritt reduzieren, und gleichzeitig nehmen numerische Fehler rasant zu. Darüber hinaus bei "direkter" Integration erhöht sich die Anzahl der Kraftberechnungen für jeden Schritt mit der Anzahl der Körper ungefähr wie N^2, was die Modellierung von Systemen, die aus Zehn- und Hunderttausenden von Körpern bestehen, praktisch unmöglich macht.

Direkte Methoden (**Particle-Particle**), die numerische Integration verwenden, erfordern in der Ordnung von $\frac{1}{2}n^2$ Berechnungen, um die potentielle Energie über alle Paare von Körpern auszuwerten, und haben somit eine Komplexität von $O(n^2)$. Für Simulationen mit vielen Körpern ist der $O(n^2)$ -Faktor besonders zeitaufwendig. Also ist bei Systemen mit mehr als 10^6 Körpern ineffizient.

Es wurde mehrere Methoden entwickelt, die die Zeitkomplexität relativ zu direkten Methoden reduzieren:

- **Tree code** Methode, wie beispielsweise eine Barnes-Hut-Simulation, sind Kollisionsfreie Methoden, wenn enge Begegnungen zwischen Körperpaaren nicht wichtig sind und entfernte Körpern nicht mit hoher Genauigkeit berechnet werden müssen. Das Potential einer entfernten Gruppe von Körpern wird unter Verwendung einer Multipolentwicklung des Potentials berechnet. Diese Approximation ermöglicht Reduktion der Komplexität auf $O(n \log n)$.
- **Fast multipole** Methode basiert auf dem Fakt, dass die multipol-expandierten Kräfte von fernen Körpern für nahe beieinanderliegende Körpern ähnlich sind. Es wird behauptet, dass diese weitere Approximation die Komplexität auf O(n) reduziert.
- **Particle mesh** Methode teilen den Simulationsraum in ein dreidimensionales Gitter auf, auf das die Massendichte den Körpern interpoliert wird. Dann wird das Berechnen des Potentials eine Frage des Lösens einer Poisson-Gleichung auf dem Gitter, die in O (n log n) -Zeit unter Verwendung von Techniken der schnellen Fourier-Transformation berechnet werden kann. Durch die Verwendung adaptiver

Gitterverfeinerung oder Mehrgitterverfahren kann die Komplexität der Methoden weiter reduziert werden.

- **P^{3}M** und **PM-tree** Methoden sind Hybridmethoden, die die Teilchengitter-Approximation für entfernte Körpern verwenden, aber genauere Methoden für nahe Körpern verwenden (innerhalb weniger Gitterintervalle). P^{3}M steht für Körpern - Körpern, Körpern - Netz und verwendet direkte Methoden mit erweichten Potentialen im Nahbereich. PM-Tree-Methoden verwenden stattdessen Baumcodes aus kurzer Entfernung. Wie bei Körpern - Netz Methoden können adaptive Netze die Recheneffizienz erhöhen.
- **Mean field** Methode approximieren das Körpersystem mit einer zeitabhängigen Boltzmann-Gleichung, die die Massendichte darstellt, die an eine selbstkonsistente Poisson-Gleichung gekoppelt ist, die das Potential darstellt. Es ist eine Art der Approximation der geglätteten Teilchenhydrodynamik, die für große Systeme geeignet ist.

Tree code Methode ist populär, aber das Projekt wird durch **Particle-Particle** Methode realisiert, weil die relativ simple ist und passt perfekt für kleine Menge den Körper ($<10^6$).

3. Parallelisierung

Der serielle Ansatz funktioniert gut für eine kleine Anzahl von Körpern, aber, wenn wir eine große Anzahl haben, wird es Zeit- und Ressource-aufwendig. Um das Problem zu lösen, brauchen wir einen parallelen Ansatz, indem wir die Computerressourcen nutzen, die wir haben. Für die Parallelisierung gibt es viele verschiedene Schnittstelle. In Rahmen dieser Arbeit werden wir zwei folgende betrachten:

3.1. OpenCL

Ein **OpenCL**-System besteht aus einem Host und einem oder mehreren OpenCL-Geräten. Ein Gerät besteht aus einer oder mehreren unabhängigen Recheneinheiten. Dies sind bei einem Mehrkernprozessor die verfügbaren Kerne, die zusammengefasst die CPU ergeben, und für die Grafikkarte die Shader. Die Compute Unit ist in ein oder mehrere ausführende Elemente (processing element) unterteilt. Der Host verteilt dabei die Kernel zur Laufzeit auf die verfügbaren Geräte.

C++ Kernel

Als erstes wird der Kernel mit alle Körperparameter definiert:

```
__kernel void nbody_simple(
    __global float4* pos,
    __global float4* vel,
    __global float4* pos_new,
    __global float4* vel_new)
{
```

Entsprechende Konstanten werden definiert:

```
    const float STEP = 0.00027397260273972603; // 1/365 Tagen/10
    const float EPS = 0.000001; // Offset
    const float G = 0.03765; // Skalierte gravitation Konstant
    //Iteration Schritt:
    const float4 step = (float4)(STEP, STEP, STEP, 0.0f);
```

Hier bekommt man ID des Threads:

```
    int gti = get_global_id(0);
```

Und das Gesamtzahl den Körpern:

```
    int n = get_global_size(0);

    float4 p = pos[gti]; // Position des Körpers
    float4 v = vel[gti]; // Velocity des Körpers
```

```
    float4 a = (float4)(0.0f, 0.0f, 0.0f, 0.0f);
```

Körperparameter werden berechnet:

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

        // Den Effekt von selbst soll nicht berechnet sein
        if (j == gti)
            continue;
        float4 pj = pos[j];
        float4 diff = pj - p; // Die Masse wird ignoriert

        float invr_distance = rsqrt(diff.x*diff.x + diff.y*diff.y +
diff.z*diff.z + EPS); // umkehrende Formell, soll (1/sqrt) sein.
        a += pj.w * diff * invr_distance * invr_distance *
invr_distance;
    }
    a *= G; // multiplizieren mit dem Gravitation-Konstant.
```

Auf dem letzten Schritt werden die Daten im globalen Speicher aktualisiert:

```
    pos_new[gti] = p+step * v + 0.5f * step * step * a;
    vel_new[gti] = v + step * a;
}
```

Aktuelle Variablen **pos** und **vel** werden erst nachdem aktualisieren, wenn die Position und Velocity der allen Körpern in diesem Schritt berechnet werden. Das wird in folgendem Schritt implementiert:

Python Teil

```
# load data
pos_host, vel_host = load_2048(2)

# allocate and init memory on device
pos_dev = cl_array.to_device(queue, pos_host)
vel_dev = cl_array.to_device(queue, vel_host)
pos_new_dev = cl_array.empty_like(pos_dev)
vel_new_dev = cl_array.empty_like(vel_dev)

# loop over 365 days
for i in range(3650):
    # execute kernal with the size of work groups equals the number of
bodies
    nbody_simple(queue, (pos_host.shape[0],), None, pos_dev.data,
    vel_dev.data, pos_new_dev.data, vel_new_dev.data)

    # copy new position to host
    pos_host = pos_new_dev.get()

    # copy new position and velocity into old arrays on device
```

```
    cl.enqueue_copy(queue, pos_dev.data, pos_new_dev.data)
    cl.enqueue_copy(queue, vel_dev.data, vel_new_dev.data)
```

3.2. OpenMP

OpenMP (Open Multi-Processing) ist eine Programmierschnittstelle (API) für die Shared-Memory-Programmierung in C++ oder C auf Multiprozessor-Computern.

OpenMP parallelisiert Programme auf der Ebene von Schleifen, die in verschiedenen Threads ausgeführt werden, und unterscheidet sich dadurch von anderen Ansätzen, bei denen ganze Prozesse parallel laufen und durch Nachrichtenaustausch zusammenwirken.

Als erstes, werden alle notwendige Bibliotheken heruntergeladen.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <string.h>
#include <omp.h>    /* for OpenMP */

#include <iostream>
#include <vector>
#include <fstream>
#include <sstream>
#include <iterator>
#include <algorithm>
```

Entsprechende Konstanten werden definiert.

```
#define NUM_THS 4   // die Anzahl der Threads in OpenMP
#define NUM_BODY 22 // die Anzahl der Körper
#define ITERATION 500
#define DELTA_T 0.00027397260273972603 // Schritt der Position- und
Velocity-Änderung
#define EPS 0.000001
const double G_CONS=0.03765; // Skalierte gravitation Konstant
```

Körper-Structure wird durch 3 Hauptwerten initialisiert:

```
struct Body{
    double x,y,z; /* position */
    double vx,vy,vz; /* velocity */
    double w; /* weight */
    };
```

Hier werden drei zweidimensionale Array mit den Position- Velocity- und Masse-Werten für jeden Körper importiert. Die Werte sind reell und wurden von der NASA Webseiten genommen.

// für das Parsing (Trennung) in csv Datei

```
template<typename Out>
void split(const std::string &s, char delim, Out result) {
        std::stringstream ss(s);
        std::string item;
        while (std::getline(ss, item, delim)) {
                *(result++) = item;
        }
}

std::vector<std::string> split(const std::string &s, char delim) {
        std::vector<std::string> elems;
        split(s, delim, std::back_inserter(elems));
        return elems;
}
```

Hauptteil: main Funktion.

```
int main(int argc, char* argv[]){
```

// Hier werden alle Körperpositionen initialisiert. Daten wurden durch Telenet von der NASA Webseite herunterladen

```
std::ifstream dataFile("data.csv");
std::string line;

dataFile.unsetf(std::ios_base::skipws);
```

// Zählen die neuen Zeilen mit einem auf das Zählen spezialisierten Algorithmus:

```
unsigned NUM_BODY = std::count(
        std::istream_iterator<char>(dataFile),
        std::istream_iterator<char>(),
        '\n');
dataFile.clear();
dataFile.seekg(0, std::ios::beg);

int i,j,k,n;
struct Body Nbody[NUM_BODY];

std::cout << "loading data " << NUM_BODY << " bodies... ";
    i = 0;
        while (std::getline(dataFile, line)) {
                std::vector<std::string> pos = split(line, ',');
        Nbody[i].x = atof(pos[0].c_str());
        Nbody[i].y = atof(pos[1].c_str());
```

```
        Nbody[i].z = atof(pos[2].c_str());
        Nbody[i].w = atof(pos[3].c_str());

        Nbody[i].vx = 0;
        Nbody[i].vy = 0;
        Nbody[i].vz = 0;
        i++;
}
    int max_treads = omp_get_max_threads();
    std::cout << "done\n" << "max trheads: " << max_treads << "\n" ;
```

Variable für die Berechnung des neues Körper-Velocitys:

```
    double newVx[NUM_BODY];
    double newVy[NUM_BODY];
    double newVz[NUM_BODY];
```

Hier fängt man an die N-Körpern Problem zu simulieren. ITERATION-Zyklen werden ausgeführt.

```
    for(k=0;k<ITERATION;k++){
```

Die Hauptbestandteile von OpenMP sind Konstrukte zur Threaderzeugung, Lastverteilung auf mehrere Threads, Verwaltung des Gültigkeitsbereiches von Daten, Synchronisation, Laufzeitroutinen und Umgebungsvariablen.
Als erstes wird der Anzahl der Threads gesetzt.

```
omp_set_num_threads(NUM_THS);
```

omp parallel teilt das Programm (Originalthread) in mehrere Threads auf, so dass der vom Konstrukt eingeschlossene Programmteil parallel abgearbeitet wird. Der Originalthread wird als Master Thread bezeichnet und trägt die ID 0.
Der Bereich für die Parallelisierung von OpenMP muss zusätzlich zu j auf privat gesetzt werden, andere Teile sind alle lokale Variablen und werden automatisch auf privat gesetzt.

```
#pragma omp parallel private(j)
{
```

Die Parallelisierung von FOR wird verwendet, um die Position, Gravitationskraft und die neue Geschwindigkeit jedes Körpers in jeder Iteration zu berechnen. Die Parallelisierung von FOR wird wie folgt erklärt:

- Das **j** im FOR wird automatisch auf privat gesetzt, alle anderen sind lokale Variablen.

- Da jeder Körper die gleiche Menge der Berechnungen hat, werden statische Methoden verwendet, um jeden Thread gleichmäßig zuzuordnen.
- Für die Parallelisierung ist die letzte voreingestellte Barriere notwendig, da alle Körper berechnet werden müssen, bevor sie aktualisiert werden können.

```
    #pragma omp for schedule(static)
        for(i=0;i<NUM_BODY;i++){
            for(j=0;j<NUM_BODY;j++) {
```

Den Effekt von selbst soll nicht berechnet sein:

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

Die Entfernung zwischen zwei Körpern b_i und b_j wird berechnet:

```
            double delta_x=Nbody[j].x-Nbody[i].x;
            double delta_y=Nbody[j].y-Nbody[i].y;
            double delta_y=Nbody[j].z-Nbody[i].z;
            double distance=sqrt(pow(delta_x,2)+pow(delta_y,2)
+pow(delta_z,2)+EPS);
```

Wenn zwei Körper dieselbe Position haben, die Kraftberechnung wird übersprungen:

```
            if(distance==0){
                continue;
            }
```

Hier wird neue Geschwindigkeit jedes Körpers berechnet.

```
double force = G_CONS * Nbody[j].w * distance * distance * distance;
            newVx[i] = newVx[i] + force * delta_x;
            newVy[i] = newVy[i] + force * delta_y;
            newVz[i] = newVz[i] + force * delta_z;
        }
    }
```

Die Körperkoordinaten und Geschwindigkeitsinformationen werden in dem neuen Zyklus aktualisiert.

- Das **j** im FOR wird automatisch auf privat gesetzt, außerdem müssen keine anderen Variablen gesetzt werden.
- Da der Berechnungsaufwand für die Aktualisierung des Körpers gleich ist, wird er jedem Thread auf statische Weise zugewiesen.
- Für die Parallelisierung ist die letzte voreingestellte Barriere notwendig, da alle Körper aktualisiert werden müssen, bevor das Ergebnis gezeichnet werden kann.

```
    #pragma omp for schedule(static)
        for(i=0;i<NUM_BODY;i++){
          Nbody[i].x = Nbody[i].x + Nbody[i].vx * DELTA_T + 0.5 *
DELTA_T * DELTA_T * newVx[i];
          Nbody[i].y = Nbody[i].y + Nbody[i].vy * DELTA_T + 0.5 *
DELTA_T * DELTA_T * newVy[i];
          Nbody[i].z = Nbody[i].z + Nbody[i].vz * DELTA_T + 0.5 *
DELTA_T * DELTA_T * newVz[i];
          Nbody[i].vx = Nbody[i].vx + DELTA_T * newVx[i];
          Nbody[i].vy = Nbody[i].vy + DELTA_T * newVy[i];
          Nbody[i].vz = Nbody[i].vz + DELTA_T * newVz[i];
          newVx[i] = 0;
          newVy[i] = 0;
          newVz[i] = 0;
        }
    }
```

Hier wird das Programm beendet

```
    return 0;
}
```

4. Visualisierung

In diesem Projekt werden die Planeten aus dem Sonnensystem als Objekte für das N-Körper Problem benutzt. Die Planet-Parameter sind reell und wurden von der offiziellen NASA-Webseite heruntergeladen. Deswegen sollen sich die Planeten theoretisch auf den gleichen Flugbahnen wie in der Realität bewegen. Um das zu prüfen oder zu beweisen, wird im Projekt die Visualisierung verwendet.

4.1. Testzweck

Bei der Entwicklung von Prototypen ist es wichtig oft und gründlich eine Fehlererkennung durchzuführen und testen. Jupyter Notebook passt für diesen Zweck am besten, weil man schneller und bequemer den Code dadurch schreiben kann. So wurde der Visualisierung-Teil auf Python Programmiersprache geschrieben.

Als erstes sollen alle notwendige Bibliotheken importiert werden: numpy und matplot:

```
import numpy as np
import matplotlib.pyplot as plt
```

Weil dieser Projektteil in Python Programmiersprache programmiert wurde, werden alle Körper-Daten in Python gespeichert, so werden die Bahnen und Körper-Bewegungen automatisch visualisiert.

```
fig = plt.figure(figsize=(15,15))
plt.plot(bodies[:,9],bodies[:,10], marker='')
```

Die Erde soll offensichtlich eine Runde um die Sonne machen. Um zu prüfen ob das in dem Projekt auch stimmt, wurde das Programm nur mit zwei Körpern (die Sonne und die Erde) gestartet. Die Position-Ergebnisse wurden nachdem exportiert und verwendet. Das Visualisierungsergebnis zeigt die Abbildung 1a. Wie sieht die Körperinteraktion mit allen Planeten zeigt die Abbildung 1b.

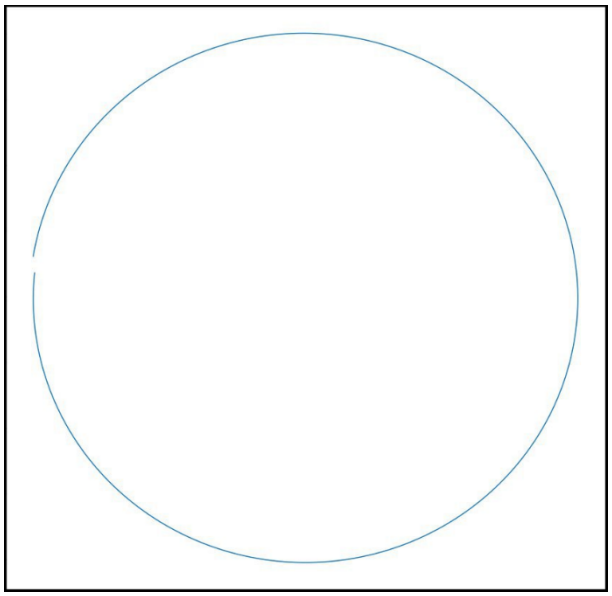

Abb. 1a. Visualisierung der Erdrotation.

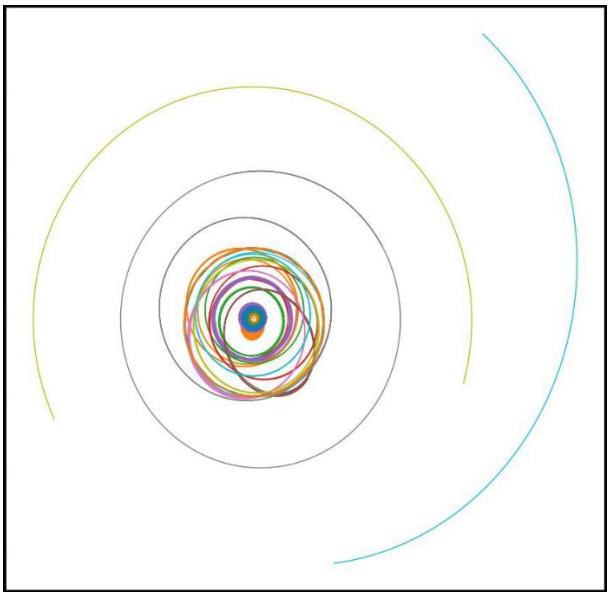

Abb. 1b. Visualisierung der Planetenbahnen.

4.2. Projektergebnisse

Die Visualisierung in Jupyter Notebook gilt nur für die Testphase. Um die Körperinteraktion klar und realistisch zu demonstrieren, werden ein Server auf Python Programmiersprache geschrieben, wo OpenCL Code ausgeführt wird und berechnete Daten über Web-Socket in Echtzeit geschickt werden.

Alle notwendigen Bibliotheken werden importiert:

```
import numpy as np
import asyncio
import websockets
import pandas as pd
import time
# scale factors to normalize one step to 1/10 day (3650 steps - one
year)
mass_cale = 18981.3
vel_scale = 365.25

def load_ephemeris():
    # load bodies' data taken from NASA telenet
    ephemeris = np.genfromtxt('ephemeris.csv', delimiter=',',
skip_header=1, dtype=np.float32, usecols=(1,3,4,5,6,7,8))
    # scale and reshape data
    mass = ephemeris[:,0]/mass_cale
    pos = np.concatenate((ephemeris[:,1:4],
mass.reshape(mass.shape[0],1)), axis=1)
    vel = np.zeros((ephemeris.shape[0], 4), dtype=np.float32)
    vel[:,:3] = ephemeris[:,4:]*vel_scale
```

```
    return pos, vel

def start_websocket(websocket, path):
    try:
        # initialize OpenCL
        ctx = cl.create_some_context()
        queue = cl.CommandQueue(ctx)
        # build kernal from file
        prg = cl.Program(ctx, open('kernal.cl').read()).build()

        # load solar system data into numpuy arrays on the host
        pos_host, vel_host = load_ephemeris()
        # create and initialize array on the device
        # copy initial postiton to device
        pos_dev = cl_array.to_device(queue, pos_host)
        # copy initial velocity to device
        vel_dev = cl_array.to_device(queue, vel_host)
        # allocate memory for new data on the devie
        pos_new_dev = cl_array.empty_like(pos_dev)
        vel_new_dev = cl_array.empty_like(vel_dev)

        # main loop
        while True:
            # run kernal with the work goup size of a number of bodies
            prg.nbody_simple(queue, (pos_host.shape[0],), None,
pos_dev.data, vel_dev.data, pos_new_dev.data, vel_new_dev.data)
            # copy new position into host np.array (device -> host)
            pos_host = pos_new_dev.get()
            # update position and velocity on the device (device ->
device)
            cl.enqueue_copy(queue, pos_dev.data, pos_new_dev.data)
            cl.enqueue_copy(queue, vel_dev.data, vel_new_dev.data)
            #send data to client
            yield from websocket.send(','.join(map(str,pos_host[:,
:3].flatten())))
    finally:
      yield from websocket.close()
```

Das wird letztendlich als websocket Server auf den entsprechenden Host (localhost) und Port (8778) gestartet.

```
start_server = websockets.serve(start_websocket, '0.0.0.0', 8778)
asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

5. Vergleich der Parallelisierungsverfahren

Das Projekt simuliert die Interaktion den Körpern im Sonnensystem, nämlich 22 Körpern. Die Zahl ist nicht groß genug um eine Unterschied zwischen verschiedene Parallelisierungsverfahren in dieser Arbeit zu zeigen. Darüber hinaus ist es schwer der Vorteil der Mehrkernprozessoren gegenüber den 1-Kern-CPU zu beweisen. Deswegen wurde für den Test die Interaktion der 2048 Körpern berechnet. Das Ergebnis zeigt Abbildung 2.

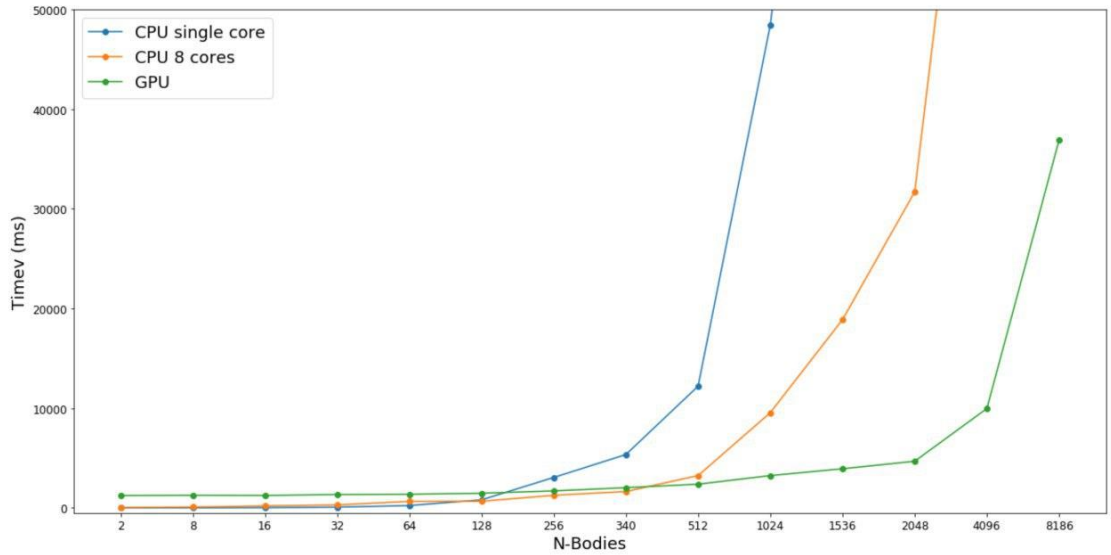

Abb. 2. Vergleich der 1-Kern, 8-Kern CPU und GPU Prozessorleistung.

Wie erwartet, die besten Leistungsergebnisse zeigt GPU und Mehrkernprozessor.

Abb. 2 demonstriert auch, dass der obergenannte Vorteil sinnvoll verwendet werden soll. So wenn in dem Interaktion-Prozess ungefähr bis zu 200 Körper teilnehmen, wird 1-Kern Prozessor rationaler verwendet werden.

Die Zeitergebnisse für verschiedene Parallelisierung-Verfahren werden folgendes erklärt: in **OpenCL** Realisation verbraucht man viele Zeit für die Memory-Kopierung vom Device zum Gerät und zurück. Darüber hinaus wird in dem Python-Teil Code eine äußere FOR-Schleife realisiert, die Realisation ist auch zeitaufwendig.