Uniwersytet Jagielloński
Wydział Matematyki i Informatyki
Zespół Katedr i Zakładów Informatyki Matematycznej

Wielowątkowa symulacja N ciał z implementacją w architekturze CUDA

Autor
Damian Stachura

Opiekun
dr Maciej Ślusarek

August 15, 2018

Contents

1 Przedstawienie problemu symulacji N ciał

Symulacja N ciał jest zagadnieniem z mechaniki klasycznej, które polega na wyznaczeniu toru ruchów wszystkich ciał danego układu o danych masach, prędkościach i położeniach początkowych w oparciu o prawa ruchu i założenie, że ciała oddziałują ze sobą zgodnie z prawem grawitacji Newtona.

1.1 Szczególne przypadki

Problem wyznaczenia dokładnego ruchu dowolnej liczby ciał jest trudny, więc można znaleźć wiele prac skupiających się jedynie na ustalonej, małej liczbe ciał.

1.1.1 Problem dwóch ciał

Problem dla dwóch ciał podlegających prawom klasycznej dynamiki Newtona i przyciągających się zgodnie z newtonowskim prawem powszechnego ciążenia został rozstrzygnięty przez J. Bernoulliego przy założeniu, że masa obiektu koncentruje się w jego środku. [Rog71]
Obydwa ciała poruszają się po krzywych stożkowych, których rodzaj zależy od całkowitej energii układu. Przykładowo, gdy energia jest mała, to ciała nie mogą się od siebie uwolnić, więc krążą wokół siebie po elipsach.

1.1.2 Problem trzech ciał

Problem trzech ciał wciąż nie został rozwiązany w ogólności. Jednakże istnieją rozwiązania dla szczególnych przypadków, jak na przykład [Ala00; GLo03]. Inną wariacją tego problemu jest system, w którym masa jednego z ciał jest zaniedbywalnie mała, jest to tak zwany ograniczony problem trzech ciał - przedstawiony przez J. L. Lagrange'a w XVIII wieku. Badał on układ Słońce-Ziemia-Księżyc.

1.2 Zastosowania

Symulacje N ciał są szeroko wykorzystywanymi narzędziami w fizyce oraz astronomii. Problemem, w którym symulacje są użyteczne jest na przykład dynamika systemu z kilkoma ciałami jak układ Słońce-Ziemia-Księżyc [JEi], co może pomóc w zrozumieniu działania olbrzymich systemów we wszechświecie. [Heg91] W kosmologii symulacje są wykorzystywane do studiowania procesów tworzenia nieliniowych struktur jak galaktyczne halo z wpływem ciemnej materii [EDO02]. Z kolei, bezpośrednie symulacje N ciał są wykorzystywane na przykład do studiowania dynamicznej ewolucji klastrów gwiazd lub do symulacji dynamiki planetozymali. [DL98]. Symulacje są wykorzystywane, również w innych dziedzinach, chociażby w algorytmach rysowania grafu skierowanego siłą [DEg15]

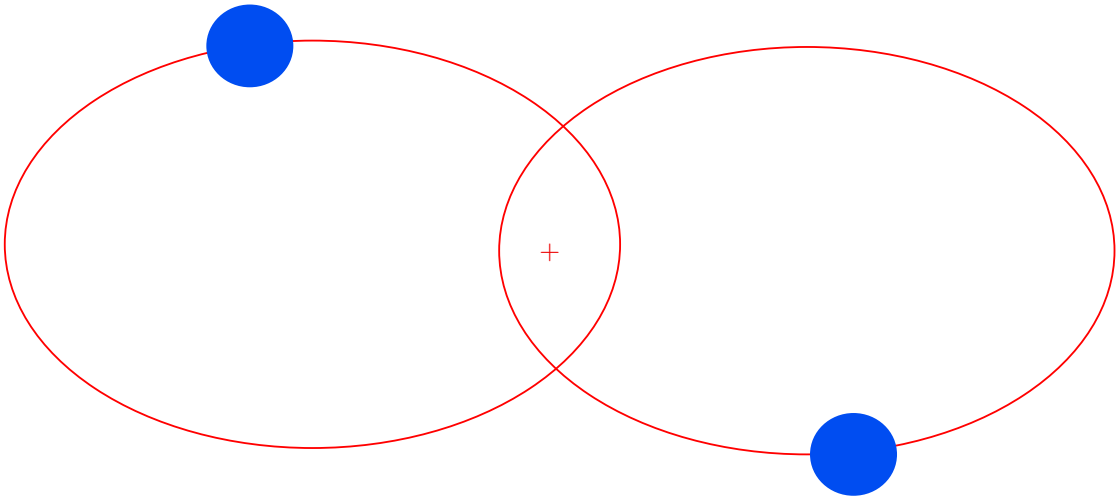

Figure 1: Symulacja dwóch ciał poruszających się po elipsach

1.3 Implementacja i wykorzystane technologie

W pierwszej części mojej pracy przedstawię równoległą implementację naiwnego algorytmu symulacji N ciał. W każdym kroku algorytm bezpośrednio wyznacza siły oddziałujące wzajemnie pomiędzy każdymi dwoma ciałami w systemie, czyli wyznacza siłę jaką pozostałe obiekty działają na wybrany. W drugiej części zaimplementuję wielowątkowo algorytm Barnes Hut'a, który korzysta z drzew ósemkowych.
Repozytorium jest dostępne pod tym linkiem. Całość została zaimplementowana w C++. Innymi technologiami wykorzystanymi w pracy są OpenGL, CUDA czy Thrust, które zastosowanie zostanie wspomniane później.
Pełna instalacja niezbędnego oprogramowania do uruchomienia symulacji jest zawarta w repozytorium (dla linuxa Ubuntu).

2 Pierwsze podejście implementacyjne

2.1 Sformułowanie problemu

W celu przedstawienia ogólnego sformułowania problemu potrzebujemy przytoczyć trzy prawa dynamiki sformułowane przez Isaaca Newtona [Rog71]

Prawo 1. *Każde ciało pozostaje w stanie spoczynku lub ruchu jednostajnego w linii prostej, chyba że jest zmuszone zmienić ten stan przez zewnętrzne oddziaływanie z innymi ciałami, czyli każde ciało jest w układzie inercjalnym.*

Prawo 2. *Szybkość zmiany pędu jest proporcjonalna do siły wywieranej i znajduje się w tym samym kierunku co siła.*

Co oznacza, że w inercjalnym układzie odniesienia zachodzi równość $F = ma$, gdzie F jest wektorem sum sił działających na obiekt, m to masa obiektu, a to jego przyśpieszenie.

Prawo 3. *Każdej akcji towarzyszy reakcja równa co do wartości i kierunku, lecz przeciwnie zwrócona.*

Co oznacza, że jeśli ciało A działa na ciało B siłą F (akcja), to ciało B działa na ciało A siłą (reakcja) o takiej samej wartości i kierunku, lecz o przeciwnym zwrocie.

Niezbędne jest również przytoczenie prawa powszechnego ciążenia Newtona [Rog71]

Prawo 4. *Każdy obiekt przyciąga każdy inny obiekt z siłą, która jest wprost proporcjonalna do iloczynu ich mas i odwrotnie proporcjonalna do kwadratu odległości między ich środkami.*

Czyli między dowolną parą ciał posiadających masy pojawia się siła przyciągająca, która działa na linii łączącej ich środki, a jej wartość rośnie z iloczynem ich mas i maleje z kwadratem odległości.

Aplikując to prawo do symulacji N ciał, uzyskujemy że na każde i-te ciało działa siła F_i zdefiniowana następująco:

$$F_i = -G \cdot m_i \sum_{j=1, j\neq i}^{N} \frac{m_j(r_i - r_j)}{|r_i - r_j|^3},$$

gdzie G to stała grawitacji, m_i masa ciała na które oddziałują inne ciała, m_j masa ciała oddziałującego na i-te ciało, $r_i - r_j$ to różnica wektorów pozycji dwóch ciał, $|r_i - r_j|$ to dystans między ciałami.

Z wykorzystaniem powyższych praw możemy podać następującą definicję

Symulacja N ciał — Dla N ciał mających ustalone masy oraz początkowe położenie i prędkość, ruch każdego obiektu jest symulowany z wykorzystaniem prawa powszechnego ciążenia oraz poprzez wyznaczenie przyspieszenia obiektu korzystając z drugiego prawa dynamiki Newtona.

Potrzebujemy zdefiniować jeszcze jedno pojęcie, jakim jest **ruch jednostajnie przyśpieszony prostoliniowy**. Konkretniej będziemy potrzebować wyprowadzonych wzorów na zmianę prędkości i pokonaną drogę przez obiekt w danym odcinku czasu.

$$v_k = v_p + a \cdot t,$$

gdzie v_k jest prędkością po wykonaniu kroku symulacji, v_p jest prędkością początkową, a oznacza wektora przyśpieszenia, a t to delta time.
Drugim wzorem jest:

$$s = v_p \cdot t + \frac{a \cdot t^2}{2},$$

gdzie s oznacza drogą przebytą w jednym kroku, a pozostałe oznaczenia są identyczne jak powyżej. Zrodlo???

2.2 Jednowątkowa wersja naiwnego algorytmu

Każde ciało na początku symulacji ma pseudolosową pozycję, prędkość oraz masę. W mojej symulacji odległości są wyrażone w metrach, masy ciał są podane w kilogramach, a jednostką prędkości jest metr na sekundę.

Prosty pseudokod dla algorytmu symulującego problem N ciał może wyglądać tak:

Listing 1: Pseudokod naiwnego algorytmu

```
1 ustaw mase oraz początkową pozycję i prędkość dla każdego ciała
2 while(true):
3    for i in {1...N}:
4           for j in {1...N}:
5                  if (i!=j):
6                         Force[i] += SiłaPomiedzyCiałami(i, j)
7    for i in {1...N}:
8           UaktualnijPozycjeCiała(i)
```

Są dostosowane do wzoru na siłe, która dla i-tego ciała, jak już wyżej wspomniałem, wyraża się wzorem:

$$F_i = -G \cdot m_i \sum_{j=1, j\neq i}^{N} \frac{m_j(r_i - r_j)}{|r_i - r_j|^3},$$

gdzie stała grawitacji G wynosi

$$G = 6,67408(31) \cdot 10^{-11} \frac{m^3}{kgs^2}.$$

W liniach 3-6 tego algorytmu liczymy bezpośrednio siłę z jaką dwa ciała oddziałują na siebie dla każdej możliwej pary ciał. Jest to najbardziej kosztowna operacja w tym algorytmie, której złożoność to $\mathcal{O}(N^2)$. W linii 7 uaktualniamy pozycję każdego ciało uwzględniając całkowitą siłę, która na nie działa. Siła ta wynika z powyżej przytoczonego wzoru.

Twierdzenie 1. *Jednowątkowy naiwny algorytm dla symulacji N ciał złożoność obliczeniową* $\mathcal{O}(N^2)$.

Dowód: Dla każdego ciała najpierw musimy wyznaczyć siłę działającą na nie poprzez interakcję z innymi ciałami, czyli dla każdego z N obiektów musimy policzyć siłę oddziałującą nań z każdym innym obiektem, więc musimy policzyć wartość wzoru wynikającego z prawa 4 $N \cdot (N-1)$ razy. Z tego wynika, że złożoność tej podoperacji $\mathcal{O}(N^2)$. Następnie dla każdego obiektu musimy wyznaczyć jego przyśpieszenie oraz nową pozycję i prędkość, co jesteśmy w stanie zrobić w czasie $\mathcal{O}(N)$. Poprzez zsumowanie złożoności obu podoperacji, widzimy że złożoność obliczeniowa jednego kroku symulacji naiwnego algorytmu wynosi $\mathcal{O}(N^2)$. □

2.3 Paralelizacja naiwnego algorytmu

Jednakże ten algorytm jest zbyt wolny dla dużej liczby ciał. W tym podrozdziale zoptymalizujemy go poprzez zrównoleglenie obliczeń. Siła oddziałująca na pewne ciało oraz wyznaczenie mu nowej pozycji jest niezależne od takich samych obliczeń dla innych ciał. A to oznacza, że pojedynczy krok algorytmu możemy policzyć równolegle dla każdego obiektu.

2.3.1 Architektura CUDA

W tym celu wykorzystamy architekturę CUDA. Jest to uniwersalna architektura procesorów wielordzeniowych (głównie kart graficznych) umożliwiająca zaimplementowanie ich mocy obliczeniowej w wielu problemach, które mogą się wykonywać zarówno sekwencyjne i wielowątkowo. Wykorzystałem CUDĘ w wersji v9.1.85 z compute compability 3.0 i wyżej. W dzisiejszych czasach poza zastosowaniem w renderowaniu grafiki, jest również często używana do masywnych obliczeń nawet na tysiącach wątków jednocześnie.
Wątki są pogrupowane w bloki. W compute compability 3.0 każdy blok może mieć do 1024 wątków. Z kolei bloki są ułożone w gridzie, który może być nawet trójwymiarowy. W wymiarze x może być aż $2^{31} - 1$ wątków, w dwóch kolejnych 65535. [Cora] Dodatkowo wątki są grupowane w mniejsze, liczące 32 wątki, grupy niż bloki, zwane warpami. Wątki w jednym warpie są uruchamiane jednocześnie i zarządzane przez warp scheduler. [Cora]

Warto, także wspomnieć o podziale pamięci w programach pisanych w tej architekturze. Możemy wyróżnić trzy rodzaje pamięci :

- lokalna - osobna dla każdego wątku, czyli wątki mogą mieć zmienne lokalne na swój użytek,
- współdzielona - pamięć dzielona przez wszystkie wątki w bloku(ale jest jej tylko 48kb [Cora]),
- globalna - najwolniejsza ze wszystkich rodzajów pamięci, ale wspólna dla wszystkich bloków.

2.3.2 Thrust

Thrust jest szablonową biblioteką dla CUDA bazująca na bibliotece STL z C++. Thrust umożliwia implementację aplikacji wielowątkowych za pośrednictwem interfejsu wysokiego poziomu, który jest w pełni zgodny z CUDA C. Korzystałem z wersji v9.2.88.

2.3.3 Implementacja wielowątkowa

W implementacji wykorzystuje dwie szablonowe struktury z biblioteki Thrust. Host_vector jest odpowiednikiem std::vectora. Rezyduje w pamięci hosta powiązanego z równoległym devicem. Device_vector różni się tym, że pamięc związana z nim jest w pamięci równoległego device. Implementacje podzielimy na dwie funkcje.

Funkcja NaiveSimBridge, przedstawiona w listingu 2, przyjmuje jako argument host_vector z pozycjami wszystkich elementów symulacji, a następnie kopiuje go do device_vectora dla odowiedzialnego za transport pozycji ciał do pamięci device'a.
Następnie w liniach 3-5 konwertuje device_vectory dla pozycji, prędkości i masy do raw pointerów. Wykorzystywane są w 6 linii, w której wywołujemy kernel NaiveSim. W potrójnych nawiasach specyfikujemy liczbę bloków oraz liczbę wątków w każdym bloku(jest to składnia z CUDA Runtime API). W linii 7 kopiuje nowe pozycje z device'a do hosta.

Listing 2: Bridge pomiędzy główną pętlą a kernelem

```
1 void NaiveSimBridge(host_vector &pos, int numberOfBodies, float dt) {
2   thrust::device_vector<float> posD = pos;
3   float *d_positions = thrust::raw_pointer_cast(posD.data());
4   float *d_velocities = thrust::raw_pointer_cast(veloD.data());
5   float *d_weights = thrust::raw_pointer_cast(weightsD.data());
6   NaiveSim<<<numberOfBlocks, threadsPerBlock>>>(d_positions,
7         d_velocities, d_weights, numberOfBodies, dt);
8   pos = posD;
9 }
```

Kernelem nazywamy funkcję, którą możemy uruchomić dla wielu wątków w pamięci device'a. W tym przypadku kernel NaiveSim, ukazany w listingu 3, jest odpowiedzialny za wyznaczenie nowej pozycji dla każdego ciała.

W pierwszej linii mamy stałą G, której wartość oznacza wcześniej zdefiniowaną stałą grawitacji. W sygnaturze kernela widzimy, że przyjmuje trzy raw_pointery do odpowiednio pozycji, prędkości oraz mas. A poza tym dostaje również delta time, czyli odcinek czasu, w którym wykonywany jest dany krok oraz całkowitą liczbę ciał w symulacji. W 6 linii wyznaczamy indeks obiektu, dla którego będziemy prowadzić obliczenia. Jako że korzystamy tylko z jednowymiarowrgo schematu bloków, to wystarczy wymnożyć id bloku z rozmiare pojedynczego bloku oraz dodac id wątku w bloku. W linii 7 mamy zabezpiecznie na wypadek gdyby obiekt o takim indeksie nie istniał, gdyż czasami liczba wywołanych wątków jest większa niz rzeczywista liczba obiektów.

W liniach 8-10 tworzymy zmienne lokalne dla dotychczasowych pozycji oraz masy naszego ciała, żeby zdecydowanie ograniczyć odwołania do droższej pamięci globalnej oraz tworzymy tablicę, w której będziemy liczyć siły działające na ciało.

Dalej mamy część, dla której równoleglimy nasz algorytm. Najpierw w liniach 12-23 mamy pętlę, w której liczymy siłę działającą na obiekt o id, któ©ej jest równe wcześniej wyliczonemu thid(threadId) poprzez każdy inny obiekt w symulacji. Postępujemy zgodnie ze wzorem z Prawa 4. W liniach 15-16 wyliczamy wektory odległości między naszym obiektem a każdym innym. W następnych dwóch liniach wyliczamy odległość między tymi dwoma obiektami i podnosimy ją do potęgi trzeciej, jak we wzorze. Z kolei w ostatnich liniach tej pętli podstawiamy wszystkie wartości do wzoru, aby obliczyć siłę działającą na ciało.

Mając policzoną siłę oddziałującą na obiekt, przechodzimy do wyliczenia nowej pozycji dla obiektu. Jako, że obiekt porusza się ruchem jednostajnym przyśpieszonym, to możemy w tym celu wykorzystać wcześniej wprowadzone wzory.
Najpierw w linii 25, korzystając z prawa 2, wyliczamy wektor przyśpieszenia ciała. W linii 26 do aktualnej pozycji dodajemy wektor drogi, o którą przesuwamy ciało po obecnym kroku. I w końcu, linia 27 aktualizuje wektor prędkości.

Listing 3: Kernel NaiveSim

```
1  const double G = 6.674 * (1e-11);
2  template <typename T>
3  __global__
4  void NaiveSim(T *pos, T *velo, T *weigh, int numberOfBodies, double dt)
5  {
6    int thid = blockIdx.x * blockDim.x + threadIdx.x;
7    if(thid>=numberOfBodies) return;
8    double pos[3] = {pos[thid*3], pos[thid*3+1], pos[thid*3+2]};
9    double weighI = weigh[thid];
10   double force[3] = {0.0, 0.0, 0.0};
11
12   for (int j = 0; j < numberOfBodies; j++) {
13     if (j != thid) {
14       double d[3];
15       for(int k=0; k<3; k++)
16           d[k] = pos[j*3 + k] - pos[k];
17       float dist = (d[0]*d[0] + d[1]*d[1] + d[2]*d[2]);
18       dist = dist*sqrt(dist);
19       float F = G * (weighI * weigh[j]);
20       for(int k=0; k<3; k++)
21                 force[k] += F * d[k] / dist;
22     }
23   }
24   for(int k=0; k<3; k++) {
25     float acc = force[k] / weighI;
26     pos[thid*3+k] += velo[thid*3+k]*dt + acc*dt*dt/2;
27     velo[thid*3+k] += acc*dt;
28   }
29 }
```

Wykorzystując architekturę CUDA, mogliśmy przyśpieszyć nasz algorytm. Jednakże, symulacja implementowana tym algorytmem ma jeszcze jeden duży problem. Gdy obiekty są bardzo blisko siebie, to znaczy odległość między nimi jest bliska zera, to wtedy siła, którą na siebie działają jest ogromna. Wtedy ciała wyraźnie się oddalają od siebie, co jest nienaturalne. W dwóch następnych podrozdziałach podam dwie optymalizacje, które ograniczają skutki takich sytuacji.

2.4 Softening

Pierwszą optymalizacją jest, tak zwany softening. Polega na modyfikacji wzoru wprowadzonego w Prawie 4, do następującej postaci:

$$F_i = -G \cdot m_i \sum_{j=1, j\neq i}^{N} \frac{m_j \cdot (r_i - r_j)}{(|r_i - r_j|^2 + \epsilon^2)^{\frac{3}{2}}},$$

gdzie wprowadzony ϵ ma za zadanie nie dopuścić do bardzo małych odległośći w mianowniku wzoru. W mojej pracy przyjąłem, że $\epsilon = 0.01$.

2.5 Aproksymacja kroku

3 Drugie podejście

3.1 Algorytm Barnesa-Huta z pseudokodem

3.2 Zrównoleglenie algorytmu Barnesa-Huta

sudo apt-get install texlive-full

3.3 Implementacja

$http://www.deltami.edu.pl/temat/fizyka/mechanika/2015/11/26/Problem_dwoch_cial/$ apt-

get install texlive-lang-polish
Random citation embeddeed in text.s
sudo apt-get install texlive-bibtex-extra
sudo apt-get install texlive-bibtex-extra biber
biber Praca
https://www.sharelatex.com/learn/Bibliography_management_with_biblatex
inkscape -D -z –file=drawing.svg –export-pdf=draw.pdf –export-latex

4 Wizualizacja

4.1 OpenGL

OpenGL jest API do tworzenia grafiki. Skorzystałem z OpenGL3, w celu zwizualizowania symulacji w 3D.
Masa słońca jest definiowana następująco

$$\mathrm{M}_\odot = 1.9884{\cdot}10^{30}$$

Symulowane gwiazdy mają wagi z zakresu $[0.5, 10]$ $\mathrm{M}_\odot$.
- jednostką odległości jest parsek, czyli odległość, dla której paralaksa roczna położenia Ziemi widzianej prostopadle do płaszczyzny orbity wynosi 1 sekundę łuku. W przeliczeniu na metry i po zaokrągleniu jest to

$$1 \text{ pc} \approx 3,2616 \text{ roku świetlnego} \approx 3,086 \cdot 10^{16} \text{ m}$$

W astronomii stała grawitacji jest wyrażana jako

$$G = 4,3 \cdot 10^{-3} \frac{pc}{M_\odot} \frac{km^2}{s^2}$$

5 Podsumowanie

References

[Aar03] Sverre J. Aarseth. *Gravitional N-Body Simulations. Tools and Algorithms 1 edition.* Cambridge University Press, 2003.

[Ala00] Richard Montgomery Alain Chenciner. "A remarkable periodic solution of the three-body problem in the case of equal masses". In: *Annals of Mathematics* 152 (2000), pp. 881–901.

[Cora] NVIDIA Corporation. *CUDA C Programming Guide.* v9.1.85, 2018. URL: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 03/05/2018).

[Corb] NVIDIA Corporation. *NVIDIA CUDA Runtime API.* v9.1.85, 2018. URL: http://docs.nvidia.com/cuda/cuda-runtime-api/index.html (visited on 03/05/2018).

[DEg15] A.Bezgodov D.Egorov. *Improved Force-Directed Method of Graph Layout Generation with Adaptive Step Length.* 2015.

[DL98] J. Stadel D. Richardson T. Quinn and G. Lake. *Direct Large-Scale N-Body Simulations of Planetesimal Dynamics.* 1998.

[EDO02] G.Chincarini E.D'Onghia C.Firmani. *The Halo Density Profiles with Non-Standard N-body Simulations.* 2002.

[GLo03] M.Kramer G.Lodge J. A. Walsh. "A Trilinear Three-Body Problem". In: *International Journal of Bifurcation and Chaos* 13 (2003), pp. 2141–2155.

[GOO] GOOGLE? *THRUST.* v9.2.88, 2018. URL: https://docs.nvidia.com/cuda/thrust/index.html (visited on 05/15/2018).

[Gro17] Khronos Group. *OpenGL API, OpenGL Shading Lanugage and GLX Specifications.* OpenGL 4.6. 2017. URL: https://www.khronos.org/registry/OpenGL/index_gl.php (visited on 07/30/2017).

[Heg91] Douglas C. Heggie. *CHAOS IN THE N-BODY PROBLEM OF STELLAR DYNAMICS.* 1991.

[JEi] J.Eiland. *N-Body Simulation of the Formation of the Earth-Moon System from a Single Giant Impact.*

[Lar07] Jan Prins Lars Nyland Mark Harris. "GPU Gems 3". In: 2007. Chap. Fast N-Body Simulation with CUDA. Chapter 31, pp. 677–694.

[Lin99] Tancred Lindholm. "Seminar presentation. N-body algorithms". In: *University of Helsinki* (1999).

[Mar11] Keshav Pingali Martin Burtscher. "GPU Computing Gems Emerald Edition". In: NVIDIA Corporation, Wen-mei W. Hwu, 2011. Chap. An Efficient CUDA Implementation of the Tree-Based Barnes HUT N-Body Algorithm. Chapter 6, pp. 75–92.

[Rog71] Jerry E. White Roger R. Bate Donald D. Mueller. "Fundamentals of astrodynamics". In: DOVER PUBLICATIONS, 1971. Chap. 1 TWO-BODY ORBITAL MECHANICS, pp. 1–49.