

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

29 sierpnia 2018

Spis treści

1	Przedstawienie problemu symulacji N ciał	3
1.1	Szczególne przypadki	3
1.1.1	Problem dwóch ciał	3
1.1.2	Problem trzech ciał	3
1.2	Zastosowania	3
1.3	Implementacja i wykorzystane technologie	4
2	Teoretyczne podstawy symulacji N ciał	4
3	Naiwny algorytm symulacji N ciał	5
3.1	Jednowątkowa wersja naiwnego algorytmu	5
3.2	Zrównoleglenie naiwnego algorytmu	6
3.2.1	Architektura CUDA	6
3.2.2	Thrust	7
3.2.3	Implementacja wielowątkowa	7
3.3	Softening	9
3.4	Złożoność zrównoleglonej naiwnej wersji algorytmu	9
4	Równoległy algorytm Barnes-Huta	9
4.1	Drzewa Ósemkowe	9
4.2	Kody Mortona	10
4.3	Struktura węzła w drzewie ósemkowym	10
4.4	Równoległa implementacja drzew ósemkowych	11
4.5	Pseudokody poszczególnych kroków	12
4.5.1	Obliczanie kodów Mortona	12
4.5.2	Sortowanie kodów	13
4.5.3	Obsługa zduplikowanych kodów Mortona	13
4.5.4	Zliczanie węzłów wewnętrznych oraz łączenie węzłów	14
4.6	Złożoność budowania drzewa	16
5	Obliczanie sił oddziałujących na ciało	17
6	Wizualizacja	18
6.1	OpenGL	18
6.2	Renderowanie symulacji	19
7	Podsumowanie	20

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.

Podstawowe zagadnienia do symulacji N ciał zostały przedstawione przez **Isaaca Newtona** w 1687 roku, który wprowadził prawa (później zostaną przedstawione dokładniej), które obowiązują przy próbach analitycznego rozwiązania problemu [19]. Początki symulacji można datować na rok 1941, kiedy to **Erik Holmberg**, prowadził eksperymenty z galaktykami, które modelował jako okręgi, w których punkty koncentrowały się wokół wspólnego środka. Co ciekawe, w swoich eksperymentach reprezentował punkty w galaktyce poprzez żarówki [14]. Z kolei pierwsze obliczenia dla symulacji N ciał na komputerze zostały wykonane przez **Sebastiana von Hoerner**a w 1960 roku [13].

1.1 Szczególne przypadki

Problem analitycznego wyznaczenia dokładnego ruchu dowolnej liczby ciał jest trudny, więc można znaleźć wiele prac skupiających się jedynie na ustalonej, małej liczbie 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 analitycznie rozstrzygnięty przez J. Bernoulliego przy założeniu, że masa obiektu koncentruje się w jego środku [20, str. 1-49].

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 analitycznie w ogólności. Jednakże istnieją rozwiązania dla uproszczonych wersji tego problemu, jak na przykład gdy masy obiektów są równe [2] lub w przypadku gdy w układzie mamy trzy równoległe względem siebie linie, a każde z ciał porusza się po jednej z prostych [9], a także gdy masa jednego z ciał obecnych w systemie jest zaniedbywalnie mała (jest to tak zwany ograniczony problem trzech ciał - przedstawiony przez J. L. Lagrange'a w XVIII wieku) [21].

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 [16] lub dynamika systemów gwiazdnych [11]. W kosmologii symulacje są wykorzystywane do studiowania procesów tworzenia nieliniowych struktur jak galaktyczne halo z wpływem ciemnej materii [8]. Z kolei, bezpośrednie

symulacje N ciał są wykorzystywane na przykład do studiowania dynamicznej ewolucji klastrów gwiazd lub do symulacji dynamiki planetozymali. [6]. Zastosowania symulacji można zaobserwować również w innych dziedzinach, chociażby w algorytmach rysowania grafu skierowanego metodą siłową (ang. force-directed graph drawing) [7]. Powyższe przykłady pokazują jak różnorodny jest zakres zastosowań symulacji.

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 oddziaływające wzajemnie pomiędzy każdymi dwoma ciałami w systemie, przez co oblicza całkowitą siłę oddziałującą na każde ciało w jednym kroku symulacji. W drugiej części zaimplementuję wielowątkowo algorytm Barnes-Hut, który korzysta z drzew ósemkowych.

Repozytorium z całym kodem projektu jest dostępne pod następującym adresem <https://github.com/damian1996/N-Body-Simulation>. Całość została zaimplementowana w C++. Innymi technologiami wykorzystanymi w pracy są OpenGL, CUDA czy Thrust, których zastosowanie zostanie wspomniane później.

Pełna instrukcja instalacji niezbędnego oprogramowania do uruchomienia symulacji jest zawarta w repozytorium (dla linuxa Ubuntu).

2 Teoretyczne podstawy symulacji N ciał

W celu przedstawienia ogólnego sformułowania problemu potrzebujemy przytoczyć trzy prawa dynamiki sformułowane przez Isaaca Newtona [19, str. 83-84] :

Prawo 1. *W inercjalnym układzie odniesienia, jeśli na ciało nie działa żadna siła lub siły działające równoważą się, to ciało pozostaje w spoczynku lub porusza się ruchem jednostajnym prostoliniowym.*

Prawo 2. *W inercjalnym układzie odniesienia jeśli siły działające na ciało nie równoważą się (czyli wypadkowa sił \vec{F}_w jest różna od zera), to ciało porusza się z przyspieszeniem wprost proporcjonalnym do siły wypadkowej, a odwrotnie proporcjonalnym do masy ciała.*

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

Prawo 3. *Oddziaływania ciał są zawsze wzajemne. W inercjalnym układzie odniesienia siły wzajemnego oddziaływania dwóch ciał mają takie same wartości, taki sam kierunek, przeciwne zwroty i różne punkty przyłożenia (każda działa na inne ciało).*

Niezbędne jest również przytoczenie prawa powszechnego ciążenia Newtona [20, str.956]

Prawo 4. *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 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 m_i to masa ciała na które oddziałują inne ciała, m_j to 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, a G to stała grawitacji i wynosi

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

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.

Będziemy potrzebować wzorów na zmianę prędkości i pokonaną drogę przez obiekt w danym odcinku czasu w ruchu jednostajnie przyspieszonym prostoliniowym.

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

gdzie v_k jest prędkością po wykonaniu kroku symulacji, v_p jest prędkością początkową, a oznacza przyspieszenia, a t to czas kroku symulacji.

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.

Tak zdefiniowana symulacja N ciał spotyka się z następującymi problemami:

- Ciało w pojedynczym kroku symulacji porusza się ruchem jednostajnie przyspieszonym prostoliniowym, więc ruch w całej symulacji jest jedynie aproksymacją poprzez sklejanie ruchu obiektu w tych odcinkach czasu.
- Kiedy ciała są bardzo bliskie siebie, wtedy zadziała na nie ogromna siła, która w rzeczywistości będzie działała na ciało przez krótki czas. Może zdarzyć się, że krok symulacji jest znacznie większy niż rzeczywisty czas, w którym obiekty są blisko siebie i obliczona przez symulację siła jest o wiele większa niż powinna.

3 Naiwny algorytm symulacji N ciał

3.1 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 następująco:

Listing 1: Pseudokod naiwnego algorytmu

```
1 ustaw masę 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)
```

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 ciała uwzględniając całkowitą siłę, która na nie działa. Siła ta wynika z powyżej przytoczonego wzoru. Nietrudno spostrzec że pojedyncza iteracja jednowątkowego naiwnego algorytmu symulacji N ciał ma złożoność obliczeniową $\mathcal{O}(N^2)$.

3.2 Zrównoleglenie naiwnego algorytmu

Jednakże ten algorytm jest zbyt wolny dla dużej liczby ciał. W tym podrozdziale zoptymalizujemy go poprzez zrównoleglenie obliczeń. Wyliczenie siły oddziałującej na pewne ciało oraz analityczne wyznaczenie mu nowej pozycji są niezależne względem tych obliczeń dla pozostałych ciał. A to oznacza, że pojedynczy krok algorytmu możemy policzyć równolegle dla każdego obiektu.

3.2.1 Architektura CUDA

W tym celu wykorzystamy architekturę CUDA. Jest to uniwersalna architektura kart graficznych (które dzisiejszych czasach poza zastosowaniem w renderowaniu grafiki, są również często używane do masywnych obliczeń nawet na tysiącach wątków jednocześnie), umożliwiającą wykorzystanie ich mocy obliczeniowej w wielu problemach, które mogą się wykonywać zarówno sekwencyjne jak i wielowątkowo. Wykorzystałem CUDA w wersji v9.2.148 z compute capability 3.0 i wyżej.

Wątki na CUDA są pogrupowane w bloki. W compute capability 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 pozostałych 65535 [3, str. 238-242]. Dodatkowo wątki są grupowane w mniejsze grupy niż bloki, zwane warpami, które liczą po 32 wątki. Wątki w jednym warpie są uruchamiane jednocześnie i zarządzane przez warp scheduler [3, str. 70].

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 [3, str. 238-242]),
- globalna - najwolniejsza ze wszystkich rodzajów pamięci, ale wspólna dla wszystkich bloków.

3.2.2 Thrust

Thrust jest szablonową biblioteką dla CUDA bazującą 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 v9.2.148 [5].

3.2.3 Implementacja wielowątkowa

W implementacji wykorzystuję dwie szablonowe struktury z biblioteki Thrust. **host_vector**[5] jest odpowiednikiem **std::vector**. Rezyduje w pamięci hosta powiązanego z równoległym urządzeniem (ang. device). **device_vector**[5] różni się tym, że jest przechowywany w pamięci urządzenia. Implementację 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_vector** dla odpowiedzialnego za transport pozycji ciał do pamięci urządzenia.

Następnie w liniach 3-5 konwertuje **device_vector** 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 [4]). W linii 7 kopiuje nowe pozycje z urządzenia 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     NaiveSim<<<numberOfBlocks, threadsPerBlock>>>>(d_positions,
5         d_velocities, d_weights, numberOfBodies, dt);
6     pos = posD;
7 }
```

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

W sygnaturze kernela widzimy, że przyjmuje trzy raw pointery do odpowiednio pozycji, prędkości oraz mas. A poza tym dostaje również 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 jednowymiarowego schematu bloków, to wystarczy wymnożyć identyfikator bloku z rozmiarem pojedynczego bloku oraz dodać identyfikator wątku w bloku. (W każdym kolejnym kernelu *thid* będzie

liczony tak samo). W linii 7 mamy zabezpieczenie na wypadek gdyby obiekt o takim indeksie nie istniał, gdyż czasami liczba wywołanych wątków jest większa niż rzeczywista liczba obiektów.

Dalej mamy część, dla której równoleglimy nasz algorytm. Najpierw w liniach 12-23 mamy pętlę, w której liczymy siłę działającą na obiekt o identyfikatorze, który jest równy wcześniej wyliczonemu *thid* 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, tak jak we wzorze. Z kolei w ostatnich liniach tej pętli podstawiamy wszystkie wartości do wzoru, aby obliczyć siłę działającą na ciało.

Listing 3: Kernel NaiveSim

```

1  const double G = 6.674 * (1e-11);
2  template <typename T>
3  __global__ void
4  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 j in {0..numberOfBodies-1}
13     {
14         if (j != thid) {
15             double d[3];
16             for(int k=0; k<3; k++)
17                 d[k] = pos[j*3 + k] - pos[k];
18             float dist = (d[0]*d[0] + d[1]*d[1] + d[2]*d[2]);
19             dist = dist*sqrt(dist);
20             float F = G * (weighI * weigh[j]);
21             for(int k=0; k<3; k++)
22                 force[k] += F * d[k] / dist;
23         }
24     }
25     for k in {0..2}
26     {
27         float acc = force[k] / weighI;
28         pos[thid*3+k] += velo[thid*3+k]*dt + acc*dt*dt/2;
29         velo[thid*3+k] += acc*dt;
30     }
31 }
```

Mając policzoną siłę oddziałującą na obiekt, przechodzimy do wyliczenia nowej pozycji dla obiektu. Jako, że ruch obiektu aproksymujemy poprzez przyjęcie, że obiekt w pojedynczym kroku porusza się ruchem jednostajnym przyspieszonym, a następnie łączymy ze sobą kolejne kroki symulacji, to możemy w tym celu wykorzystać wcześniej wprowadzone wzory. Najpierw w linii 25, korzystając z prawa 2, wyliczamy wektor przyspieszenia 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.

Wykorzystując architekturę CUDA, mogliśmy przyspieszyć nasz algorytm. Jednakże, symulacja implementowana tym algorytmem ma jeszcze jeden duży problem. Tak jak wspomnie-

liśmy wcześniej, gdy obiekty są bardzo blisko siebie, to znaczy odległość między nimi jest bliska zeru, to wtedy siła, którą na siebie działają jest ogromna. Wtedy w naszej symulacji pojawiają się błędy, gdyż czas pojedynczej iteracji symulacji może być znacznie większy niż czas w rzeczywistości, przez który ciała są blisko siebie.

3.3 Softening

Optymalizacją dla tego problemu jest, tak zwany **softening**[1, str. 21]. 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 odległości w mianowniku wzoru. W pracy przyjąłem, że $\epsilon = 0.01$.

3.4 Złożoność zrównoleglonej naiwnej wersji algorytmu

Równoległa wersja naiwnego algorytmu wciąż wykonuje pracę rzędu $\mathcal{O}(N^2)$, jednakże dzięki temu, że dla każdego obiektu siły na niego działające oraz nową pozycję liczymy na osobnym wątku, to czas działania algorytmu to $\mathcal{O}(N)$, przy założeniu, że mamy do dyspozycji N procesorów.

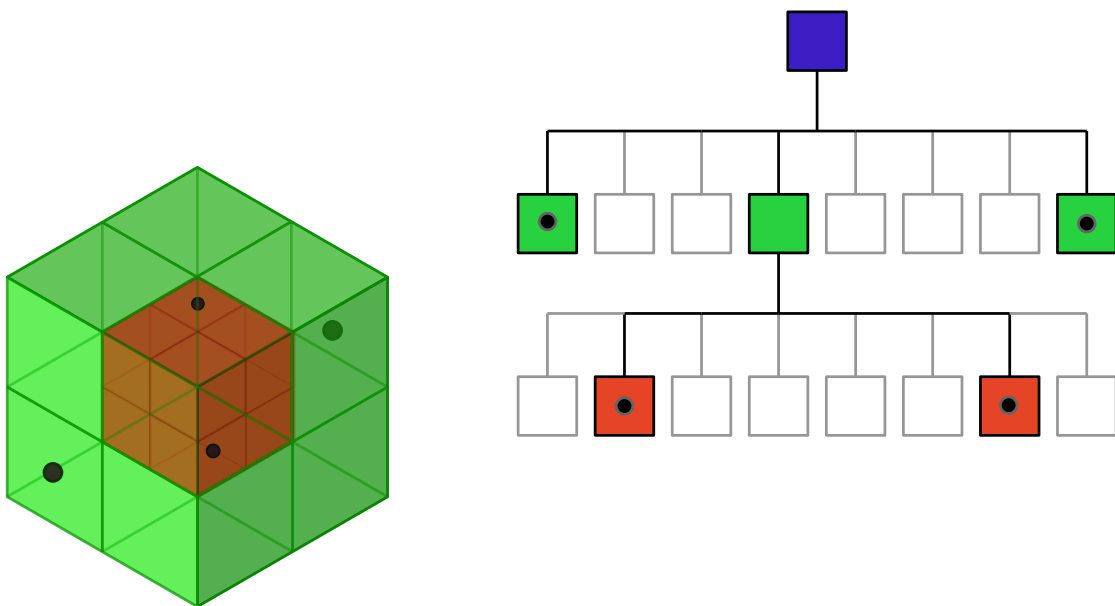
4 Równoległy algorytm Barnes-Huta

Zrównoleglony naiwny algorytm jest szybszy niż wersja jednowątkowa, jednakże wciąż jest zbyt wolny do symulacji układów z bardzo dużą liczbą ciał. W tym celu zaimplementujemy drugi algorytm, zwany algorytmem Barnes-Huta [15, str. 446-449]. Składa się on z dwóch części: tworzenia drzewa ósemkowego (ang. octree) oraz obliczania sił działających na ciało z wykorzystaniem stworzonych drzew.

4.1 Drzewa Ósemkowe

Algorytm jest oparty na drzewach, a dokładniej drzewach ósemkowych [18]. Drzewo ósemkowego przedstawiamy następująco (rys. Rysunek 1):

- cała przestrzeń trójwymiarowa jest przedstawiana przez sześcian, który jest reprezentowany przez korzeń drzewa
- drzewo składa się z węzłów wewnętrznych oraz liści
- każdy węzeł wewnętrzny ma osiem następników, z których każdy reprezentuje jeden z ośmiu podsześcianów, na które dzielimy duży sześcian reprezentowany przez węzeł
- jedynie liście drzewa zawierają punkty układu
- drzewo ma ustaloną głębokość K



Rysunek 1: Przykład drzewa ósemkowego (dla uproszczenia rysunku bez ustalonej głębokości) z czterema węzłami o następujących współrzędnych: $(-0.5, -0.5, 0.5)$, $(0.5, 0.5, 0.5)$, $(-0.4, 0.4, -0.7)$, $(-0.7, 0.6, -0.4)$ dla przestrzeni ograniczonej sześcianem o środku w punkcie $(0, 0, 0)$ oraz boku równym 2.

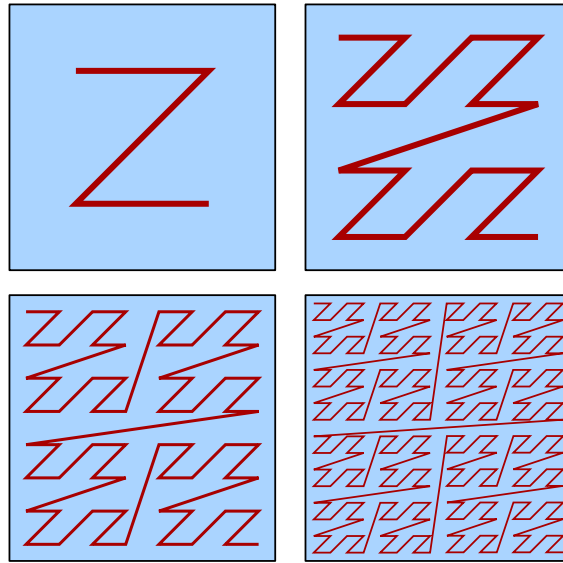
4.2 Kody Mortona

Zanim pokażemy strukturę węzła w naszej implementacji drzewa ósemkowego, przytoczymy czym są kody Mortona, które będą niezbędnym narzędziem do ustalenia położenia węzłów w drzewie o ustalonej głębokości.

Kodami Mortona, nazywamy mapowanie punktu w przestrzeni do listy liczb (w naszym przypadku mapujemy punkty w 3-wymiarowej przestrzeni do liczb całkowitych, a konkretniej interesować nas będzie ich bitowa reprezentacja). Porządek Mortona jest definiowany przez **krzywą wypełniającą powierzchnię (ang. space-filling curve)**, która jest w kształcie litery Z , więc czasami jest nazywana **Z -krzywą**. Krzywa ta ma ważną własność - jeśli elementy są blisko siebie w drzewie, to w tym porządku ta bliskość jest zachowana w ułożeniu w pamięci. W pracy traktujemy kod Mortona jako 64-bitową liczbę całkowitą, w której 15 bitów (łącznie 45 dla wszystkich trzech wymiarów) oznacza kolejne rozgałęzienia dla naszego węzła, czyli jedną z dwóch części, w których punkt kolejno łąduje, gdy umieszczamy go w coraz mniejszych sześcianach, aż do przyjętej głębokości równej 15 (czyli $K = 15$).

4.3 Struktura węzła w drzewie ósemkowym

Przedstawia się ona następująco:



Rysunek 2: Przykład Z-curve

Listing 4: Struktura OctreeNode

```

1 struct OctreeNode {
2     int children[8] = {-1, -1, -1, -1, -1, -1, -1, -1};
3     int position = -1;
4     std::vector<int> duplicates;
5     float totalMass = 0.0;
6     float centersOfMass[3] = {0.0, 0.0, 0.0};
7 };

```

W powyższej strukturze przechowujemy indeksy następników w globalnej tablicy węzłów. Mamy również pole `position`, które przyjmuje `-1`, gdy węzeł jest wewnętrzny oraz indeks węzła (na tablicę globalną), gdy węzeł jest liściem. Wektor `duplicates` przechowuje w węźle indeksy wszystkich punktów o takim samym kodzie Mortona (czyli liści będzie tyle samo co unikatowych kodów Mortona stworzonych na bazie punktów wejściowych). Dodatkowo struktura węzła zawiera całkowitą masę poddrzewa oraz środek masy.

Środek masy (barycentrum) ciała jest punktem w przestrzeni, który zachowuje się tak, jak gdyby w nim skupiona była cała masa układu ciał. Jest zadany następującym wzorem:

$$x_{srm} = \frac{\sum_{j=1}^N m_j \cdot x_j}{\sum_{j=1}^N m_j},$$

gdzie m_j oznacza masę j -tego ciała, x_j oznacza jego x -ową współrzędną (dla dwóch pozostałych wzory są analogiczne).

4.4 Równoległa implementacja drzew ósemkowych

Z wykorzystaniem powyżej zdefiniowanych kodów zaimplementujemy algorytm inspirowany algorytmem równoległego tworzenia drzewa ósemkowego wspomnianym w pracy

Tero Karras'a [17].

Zmodyfikowany algorytm (skrótowo) składa się z pięciu kroków :

- Obliczenie kodów Mortona dla każdego punktu układu. Realizujemy to poprzez kernel, który na podstawie przedziałów, w których zawierają się punkty w całym układzie dla każdego z wymiarów (znajdujemy dolną i górną granicę w każdym z wymiarów), wyznacza przez K poziomów coraz to dokładniejsze położenie poprzez ustawianie kolejnych bitów w liczbie binarnej.
- Sortujemy kody Mortona. W tym celu wykorzystujemy bibliotekę Thrust, a dokładniej funkcję **sort_by_key**.
- Kody liczymy do głębokości K, więc istnieje możliwość pojawienia się duplikatów, które musimy zachować.
- Liczymy ile wierzchołków będzie miało nasze drzewo (znamy jedynie liczbę liści, potrzebujemy policzyć również liczbę węzłów wewnętrznych). Dodatkowo tworzymy wszystkie wierzchołki w drzewie.
- Łączymy węzły według relacji (rodzic, dziecko). W tym celu musimy odpowiednio przyporządkować każdemu węzłowi jego węzeł nadrzędny.

4.5 Pseudokody poszczególnych kroków

4.5.1 Obliczanie kodów Mortona

Listing 5: Krok 1

```
1 __global__
2 void calculateMortonCodes(T* pos, unsigned long long* codes, int numberOfBodies, T* mins, T* maxs) {
3     float start[3] = {mins[0], mins[1], mins[2]};
4     float middle[3] = {(maxs[0] - mins[0])/2, (maxs[1] - mins[1])/2, (maxs[2] - mins[2])/2};
5     unsigned long long code = 0;
6     for i in {0..K-1}
7     {
8         for j in {0..2}
9         {
10             code <<= 1;
11             if(start[j] + middle[j] < pos[3*thid + j])
12             {
13                 code |= 0x1;
14                 start[j] += middle[j];
15             }
16             middle[j] /= 2;
17         }
18     }
19     codes[thid] = code;
20 }
```

W przypadku obliczania kodów Mortona, potrzebujemy minimalnego oraz maksymalnego punktu w każdym z wymiarów (szukamy ich z pomocą funkcji `minmax_element()` z biblioteki Thrust). Dzięki temu jesteśmy w stanie wyznaczyć początek oraz środek przedziału dla obiektu na danej głębokości. Poprzez to możemy stwierdzić, w której połowie znajduje się nasz obiekt(czyli na danej głębokości przydzielić mu 0 lub 1).

4.5.2 Sortowanie kodów

Listing 6: Krok 2

```
1 thrust::device_vector<int> sortedNodes(numberOfBodies);
2 int* d_sortedNodes = thrust::raw_pointer_cast(sortedNodes.data());
3 fillNodes<<<blocks, THREADS_PER_BLOCK>>>(d_sortedNodes, numberOfBodies);
4
5 thrust::sort_by_key(mortonCodes.begin(), mortonCodes.end(), sortedNodes.begin());
```

Rolą wektora `sortedNodes` jest zachowanie początkowej numeracji punktów (początkowe indeksy wpisujemy do wektora `sortedNodes` poprzez kernel `fillNodes` w czasie stałym), to znaczy sortujemy pary (**kodMortona, początkowyIndeks**).

4.5.3 Obsługa zduplikowanych kodów Mortona

Listing 7: Krok 3

```
1 thrust::device_vector<int> groups(numberOfBodies);
2 int* d_groups = thrust::raw_pointer_cast(groups.data());
3 computeDifferences<<<blocks, THREADS_PER_BLOCK>>>(d_codes, d_groups);
4 thrust::inclusive_scan(groups.begin(), groups.end(), groups.begin());
5 thrust::device_vector<int> first(numberOfBodies);
6 thrust::device_vector<int> last(numberOfBodies);
7 int* d_first = thrust::raw_pointer_cast(first.data());
8 int* d_last = thrust::raw_pointer_cast(last.data());
```

Wektor `groups` będzie przechowywał informację o indeksie grupy węzłów o tym samym kodzie Mortona. Obliczamy to poprzez połączenie kernela *computeDifferences*, który dla danej komórki i stawia 1 gdy $mortonCodes[i - 1] \neq mortonCodes[i]$, 0 w przeciwnym przypadku (pierwsza komórka jest inicjowana na 0). A następnie liczymy sumy prefiksowe na wektorze `groups` w celu wyznaczenia indeksu każdej z grup. Następnie uruchmiamy, kernel *CalculateFirstAndLast*, który służy do wyznaczenia początku oraz końca przedziału w tablicy globalnej punktów dla każdego kodu Mortona (czyli dla każdej grupy). Kernel przedstawia się następująco:

Listing 8: Kernel `calculateFirstAndLast`

```
1 __global__
2 void calculateFirstAndLast(int* first, int* last, unsigned long long* codes, int numberOfBodies) {
3     if (mortonCodes[thid-1] != mortonCodes[thid]) {
4         last[group[thid-1]] = thid-1, first[group[thid]] = thid;
5     }
6 }
```

Gdy już to policzymy to możemy stworzyć początkowe drzewo ósemkowe składające się tylko z liści.

Listing 9: Stworzenie wektora octree

```
1 thrust::device_vector<OctreeNode> octree(numberOfBodies);
2 OctreeNode* d_octree = thrust::raw_pointer_cast(uniquePointsCount);
3
4 AssignDuplicatesToNode<<<blocks, THREADS_PER_BLOCK>>>(octree, groups, first, last);
```

Po stworzeniu wektora reprezentującego drzewo przypisujemy każdemu węzłowi indeksy wszystkich punktów, które mają i -ty kod Mortona, czyli należą do i -tej grupy (kernel *AssignDuplicatesToNode* wstawia je do wektora *duplicates* odpowiedniego węzła z *octree*).

4.5.4 Zliczanie węzłów wewnętrznych oraz łączenie węzłów

W następnym kroku tworzymy wektor **parentsNumbers**, którego zadaniem jest przechowywanie informacji o rodzicu każdego z węzłów (informacja ta będzie niezbędna przy łączeniu wierzchołków) na każdym z poziomów (jak wspomnieliśmy nasze drzewo ma głębokość K). Poniższy fragment kodu zawiera również trzy istotne zmienne:

- **childrenCount** - zlicza liczbę nowych węzłów na każdym z poziomów drzewa,
- **previousChildrenCount** - przechowuje liczbę węzłów z poprzedniego poziomu,
- **allChildrenCount** - zlicza liczbę wszystkich węzłów w drzewie ósemkowym.

Listing 10: Przechowywanie węzłów drzewa

```
1 thrust::device_vector<int> parentsNumbers(uniquePointsCount);
2 int* d_parentsNumbers = thrust::raw_pointer_cast(parentsNumbers.data());
3
4 int childrenCount = uniquePointsCount;
5 int allChildrenCount = uniquePointsCount;
6 int previousChildrenCount = 0;
```

Kluczowym punktem tych dwóch części jest poniższa pętla. Iteruje się ona $K = 15$ razy, zaczynając od dzieci drzewa (podejście **bottom-up**), oznacza to także, że zaczyna od węzłów ograniczających najmniejszy obszar, czyli obszaru ograniczającego nasze punkty najbardziej.

W każdej z iteracji chcemy wyznaczyć rodziców dla węzłów. W tym celu pomocny jest kernel **calculateDuplicates** w połączeniu z **thrust::inclusive_scan**, które pozwalają wyznaczyć duplikaty wśród rodziców (czyli węzły na wyższym poziomie, które mają identyczne kody Mortona, więc mają takie same następniki). Kernel ustawia wartość 1 w danej komórce wektora, jeśli rodzice dwóch kolejnych węzłów są różni (wystarczy sprawdzić dwa kolejne, gdyż dzięki sortowaniu dzieci każdego z węzłów są obok siebie), a następnie **inclusive_scan** robi sumę prefiksową tej tablicy, aby wyznaczyć te, które są takie same (mają przypisane takie same liczby po obliczeniu sum prefiksowych).

Listing 11: Kernel calculateDuplicates

```

1 __global__
2 void calculateDuplicates(unsigned long long* mortonCodes, int* result, int N) {
3     int thid = blockIdx.x*blockDim.x + threadIdx.x;
4     if(thid >= N || thid == 0) return;
5     unsigned long long code = mortonCodes[thid];
6     unsigned long long previous_code = mortonCodes[thid-1];
7     code >>= 3;
8     previous_code >>= 3;
9     result[thid] = (code != previous_code);
10 }

```

Po wyznaczeniu nowych wierzchołków na kolejnym poziomie, wstawiamy je do wektora **octree**, a następnie przesuwamy im indeks, aby miały indeksy zaczynające się od pierwszego wolnego (po ostatnim węźle z niższego poziomu). Widzimy, że tym sposobem **root** drzewa będzie ostatnim węzłem w naszej tablicy.

Kiedy stworzymy węzły na danym poziomie, to następnym zadaniem jest połączenie ich z węzłami będącymi na niższym poziomie. Relizujemy to poprzez kernel **connectChildren**.

Dodatkowo w tym kernelu nie liczymy całkowitego środka masy dla punktów, a jedynie liczniki ze wzoru, czyli sumę iloczynów mas i pozycji ciał w układzie. W celu dokończenia wyznaczania barycentrum po utworzeniu całego drzewa, odpalamy kernel, który dla każdego z węzłów podzielimy obecne środki mas przez *totalMass* (które również wyliczyliśmy).

Listing 12: Kernel connectChildren

```

1 __global__
2 void connectChildren(unsigned long long int* mortonCodes, int* parentsNumbers, OctreeNode* octree,
3     int N, int previousChildrenCount, int* sortedNodes, double* pos, double* wei, int level)
4 {
5     unsigned long long int childNumber = mortonCodes[thid]&0x7;
6     octree[parentsNumbers[thid]].children[childNumber] = thid+previousChildrenCount;
7     octree[parentsNumbers[thid]].position = -1;
8     octree[thid+previousChildrenCount].position = level == 0 ? sortedNodes[thid] : -1;
9     int childId = sortedNodes[thid];
10    if(level == 0) {
11        octree[thid].totalMass = wei[childId];
12        for k in {0..2}
13        {
14            octree[thid].centersOfMass[k] = (wei[childId] * pos[3*childId + k]) / wei[childId];
15        }
16    }
17    int pthid = parentsNumbers[thid];
18    atomicAdd(&octree[pthid].totalMass, octree[thid+previousChildrenCount].totalMass);
19    for k in {0..2}:
20    {
21        atomicAdd(&octree[pthid].centersOfMass[k],
22            octree[thid+previousChildrenCount].centersOfMass[k]);
23    }
24 }

```

Cele, które ten kernel realizuje, są następujące:

- ustawienie odpowiedniego dziecka dla rodzica(informacja z kodu Mortona),
- ustawienie pola position, to znaczy -1 gdy węzeł wewnętrzny, w przeciwnym przypadku identyfikator węzła,
- aktualizacja totalMass oraz centersOfMass dla węzła,
- dodatkowo, gdy funkcja przetwarza liście drzewa, to ustawia im początkowe wartości dla totalMass oraz centersOfMass.

Całą pętlę realizującą kroki piąty oraz szósty, przedstawia pseudokod w listingu 13.

Listing 13: Kroki 4-5

```

1  for(int i = 0; i < K; ++i) {
2      blocks = (childrenCount+THREADS_PER_BLOCK-1)/THREADS_PER_BLOCK;
3      thrust::fill(parentsNumbers.begin(), parentsNumbers.end(), 0);
4      calculateDuplicates<<<blocks, THREADS_PER_BLOCK>>>(<
5          d_codes,
6          d_parentsNumbers,
7          childrenCount
8      );
9
10     thrust::inclusive_scan(parentsNumbers.begin(), parentsNumbers.end(), parentsNumbers.begin());
11     octree.insert(octree.end(), parentsNumbers[childrenCount-1]+1, OctreeNode());
12     d_octree = thrust::raw_pointer_cast(octree.data());
13
14     thrust::for_each(parentsNumbers.begin(), parentsNumbers.end(),
15         thrust::placeholders::_1 += allChildrenCount);
16
17     connectChildren<<<blocks, THREADS_PER_BLOCK>>>(<
18         d_codes,
19         d_parentsNumbers,
20         d_octree,
21         childrenCount,
22         previousChildrenCount,
23         d_sortedNodes,
24         d_positions,
25         d_weights,
26         i
27     );
28
29     thrust::for_each(mortonCodes.begin(), mortonCodes.end(), thrust::placeholders::_1 >= 3);
30     auto it = thrust::unique(mortonCodes.begin(), mortonCodes.end());
31     mortonCodes.erase(it, mortonCodes.end());
32     d_codes = thrust::raw_pointer_cast(mortonCodes.data());
33     childrenCount = thrust::distance(mortonCodes.begin(), it);
34     previousChildrenCount = allChildrenCount;
35     allChildrenCount += childrenCount;
36 }

```

4.6 Złożoność budowania drzewa

Przyjrzyjmy się kolejnym operacjom w naszym algorytmie. Krok pierwszy wykonujemy w czasie stałym. Następnie sortowanie z biblioteki thrust, które wykonuje się poprzez równoległy radixsort, czyli ma złożoność obliczeniową $\mathcal{O}(\log N)$. Krok trzeci również wykonuje się

logarytmicznie, a jego operacją dominującą jest `thrust::inclusive_scan`. Ostatnia część jest bardziej wymagająca czasowo, gdyż ma złożoność obliczeniową $\mathcal{O}(K \cdot \log N)$. Czyli widzimy, że jest operacją dominującą całego algorytmu. Z tego wynika, że złożoność algorytmu budowania drzewa, to $\mathcal{O}(K \cdot \log N)$, gdzie w naszej symulacji $K = 15$.

5 Obliczanie sił oddziałujących na ciało

Drzewo zostało stworzone w celu szybszego obliczania siły jaka działa na ciało.

Wersje jednowątkowa oraz wielowątkowa algorytmu obliczania siły różnią się jedynie tym, że ta druga robi to na osobnym wątku dla każdego ciała. Dodatkowo zaimplementowałem ją iteracyjnie.

Algorytm polega na przejściu drzewa od korzenia i na sprawdzaniu czy węzeł jest wewnętrzny czy zewnętrzny. Gdy węzeł jest liściem, to liczymy siłę między naszym punktem a wszystkim punktami z wektora *duplicates* (może być ich więcej niż jeden przez duplikaty w kodach Mortona). Każdą z sił oddziałujących na punkt liczymy identycznie jak w naiwnym algorytmie. Gdy węzeł jest wewnętrzny, to postępujemy następująco:

Niech s będzie szerokością obszaru obejmowanego przez sześcian oraz niech d będzie odległością między środkiem masy obszaru a punktem. Jeśli $s/d < \theta$, gdzie przyjęliśmy, że $\theta = 2.0$, to liczymy siłę między środkiem masy obszaru a punktem. W przeciwnym przypadku wywołujemy się rekurencyjnie na ośmiu podsześcianach. Ogólnie, im θ jest większa, to obliczenia są mniej dokładne, czyli więcej poddrzew traktujemy jako jeden węzeł [1].

Dodatkowo musimy rozstrzygać czy nasz punkt i przetwarzany węzeł nie są tym samym węzłem, aby nie liczyć siły między nimi.

W poniższej iteracyjnej implementacji algorytmu obliczania siły korzystamy z dwóch tablic, które będą nam symulować stos. Podczas przetwarzania wierzchołka wrzucamy na stos dwie pary. Pierwszą będzie **(idObecnegoWęzła, nrDziecka+1)**, czyli po przetworzeniu całego poddrzewa dziecka o `nrDziecka`, przejdziemy do przetwarzania kolejnego dziecka. Druga para to **(idDziecka, 0)**, czyli wrzucamy na stos pierwszego dziecka obecnego węzła (to znaczy węzeł na którym będziemy wywoływać się rekurencyjnie).

Z kolei w celu przechowywania granic wyznaczających obszar, który ogranicza dany węzeł trzymamy jedynie tablice z minimalnymi i maksymalnymi punktami w każdym z wymiarów dla korzenia. Następnie, gdy jesteśmy na poziomie i , to dzielimy długość boku sześcianu korzenia przez 2^i , co pozwala nam uzyskać bok węzła na poziomie i .

Listing 14: Obliczanie siły oddziałującej na ciało w układzie

```

1  __global__
2  void computeForces(OctreeNode* octree, double* velocities, double* weights,
3    double* pos, double* mins, double* maxs, int AllNodes, int N, double dt)
4  {
5    float forces[3] = {0.0, 0.0, 0.0};
6    float p = {pos[3*thid], pos[3*thid + 1], pos[3*thid + 2]};
7    unsigned int stack[16], child[16];
8    float widthCube = maxs[0]-mins[0];
9    int top = -1;
10   stack[++top] = AllNodes - 1, child[top] = 0;
11
12   while(top>=0)
13   {
14     int prevTop = top, nextChild = child[top], idx = stack[top--];
15     if(idx == -1) continue;
16
17     if(octree[idx].position == -1) {
18       double d = distanceBetween(octree[idx].centerOfMass(), p);
19       double s = widthCube/(1<<prevTop);
20       bool isFarAway = (s/d < theta);
21       if(isFarAway) {
22         forces += computeForcesBetween(octree[idx].centerOfMass, p);
23       }
24       else {
25         if(nextChild==numberOfChilds) continue;
26         stack[++top] = idx, child[top] = nextChild + 1;
27         stack[++top] = octree[idx].children[nextChild], child[top] = 0;
28       }
29     }
30     else {
31       if(thid == octree[idx].position) continue;
32       for point in octree[idx].duplicates
33         forces += computeForcesBetween(octree[idx].getPoint(point), p);
34     }
35   }
36   UpdatePositionAndVelocity(forces, positions, velocities, weights, N, dt);
37 }

```

Złożoność obliczeniowa algorytmu obliczania siły przy równomiernym rozkładzie punktów wynosi $\mathcal{O}(\log N)$ i zostało to pokazane w pracy Barnes-Hut'a [15].

6 Wizualizacja

W celu wizualizacji symulacji wykorzystałem OpenGL3.

6.1 OpenGL

OpenGL [10] jest API, które jest przeznaczone głównie do tworzenia grafiki. OpenGL zazwyczaj wykorzystuje kartę graficzną (GPU), więc tworzenie grafiki następuje szybciej niż innymi sposobami. Ten efekt nazywamy przyspieszeniem sprzętowym. OpenGL wykorzystywany jest często przez gry komputerowe i wygaszacze ekranu.

6.2 Renderowanie symulacji

Najistotniejszym punktem kodu OpenGL-owego jest rysowanie symulacji w każdym kroku, za które odpowiada funkcja **Render**:

Listing 15: Pseudokod renderowania symulacji

```
1 void Render()
2 {
3     glUseProgram(program);
4     glm::mat4 view = glm::lookAt(
5         glm::vec3(
6             camera_radius*sin(camera_theta),
7             camera_radius*cos(camera_theta)*cos(camera_phi),
8             camera_radius*cos(camera_theta)*sin(camera_phi)
9         ),
10        glm::vec3(0, 0, 0),
11        glm::vec3(0, 1, 0)
12    );
13    glm::mat4 projection = glm::perspective(
14        glm::radians(45.0f),
15        1.0f*width/height,
16        0.1f,
17        100.0f
18    );
19    glm::mat4 mvp = projection * view;
20    GLuint MatrixID = glGetUniformLocation(program, "MVP");
21    glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &mvp[0][0]);
22
23    UpdateBuffers();
24    DrawPoints();
25 }
```

Render polega głównie na wykorzystaniu macierzy Model-View-Projection (MVP) [12]. W linii 19 mnożymy wszystkie macierze ze sobą (pojedyncza macierz odpowiada za jeden krok MVP), uzyskując wyrenderowaną pozycję naszych obiektów.

Bardziej szczegółowo, najpierw ustawiamy kamerę, do czego służy funkcja **lookAt** [10], która jako pierwszy argument przyjmuje współrzędne kamery, następnie punkt na który patrzymy, a w ostatnim parametrze ustawiamy, że patrzymy z góry do dołu.

Z kolei funkcja **perspective** [10] odpowiada za rzutowanie, a w argumentach przyjmuje:

- kąt z jakiego patrzymy na punkt podany w radianach
- **aspect ratio**, czyli proporcje pomiędzy szerokością i wysokością obrazu
- **far clipping plane** oraz **near clipping plane**, które wyznaczają płaszczyzny, które ograniczają obszar renderowania

7 Podsumowanie

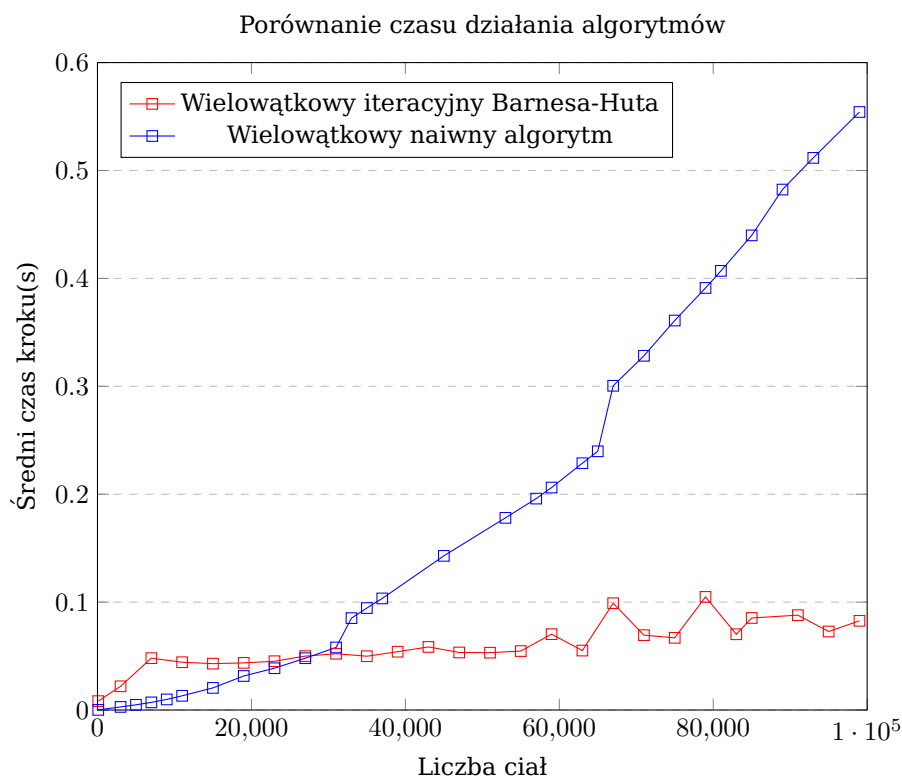
W pracy zaimplementowałem cztery algorytmy:

- Jednowątkowy naiwny algorytm
- Wielowątkowy naiwny algorytm
- Jednowątkowy algorytm Barnes-Huta
- Wielowątkowy algorytm Barnes-Huta

Symulacje, których wyniki zostały zebrane w poniższych wykresach były uruchamiane na karcie NVIDIA GeForce GTX 980. Poniżej przedstawiam wykresy przedstawiające wyniki moich implementacji algorytmów wielowątkowych z powyższej listy. Algorytmy 1 i 3 zostały wykorzystane jedynie do testowania poprawności symulacji.

Dane testowe były losowe o równomiernym rozkładzie. Losowane były początkowe:

- pozycje ciał w przedziale $\{-1.0, 1.0\}$,
- prędkości ciał w przedziale $\{-0.01, 0.01\}$,
- masy ciał w przedziale $\{1000.0, 100000.0\}$.



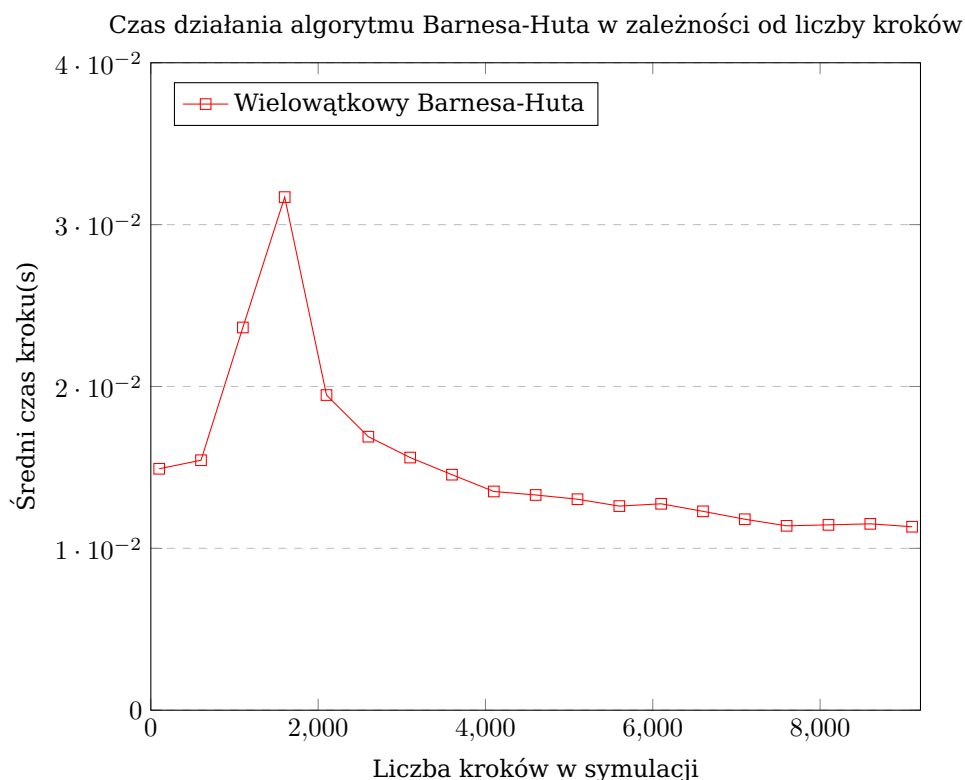
Rysunek 3: Porównanie czasu działania algorytmów

Powyżej widać wykres przedstawiający czas wykonywania się poszczególnych algorytmów przy jednakowych danych wejściowych oraz przez identyczną liczbę kroków. W testach algorytmy były uruchamiane przez 500 kroków, a następnie całkowity czas działania algorytmu był dzielony przez liczbę kroków. Wykres przedstawia czasy działania obu algorytmów równoległych dla wybranych liczby obiektów z zakresu $\{0, 100000\}$.

Zauważmy że dla coraz większej liczby ciał, równoległa implementacja algorytmu Barnes-Huta jest o wiele szybsza niż wielowątkowy algorytm liczący dla każdego ciała siłę działającą na niego przez każde inne ciało.

Zachodzi również zależność, że im więcej tur, tym algorytm Barnes-Huta spisuje się lepiej, gdyż część obiektów oddala się względem innych obiektów i przez to często grupka obiektów jest traktowana jako jeden obiekt i nie musimy liczyć siły z każdym ciałem z osobna, a za to liczymy siłę tylko raz z środkiem masy tej grupki ciał.

Widzimy to na drugim wykresie, gdzie mamy ustalone, że ciał w symulacji jest 1000, a zmienna jest liczba symulowanych kroków.



Rysunek 4: Czas działania algorytmu Barnes-Huta w zależności od liczby kroków

Przeprowadzone eksperymenty dowodzą wysokiej przydatności architektury CUDA w implementacji algorytmu naiwnego oraz algorytmu Barnes-Huta dla symulacji N ciał.

Literatura

- [1] Sverre J. Aarseth. *Gravitational N-Body Simulations. Tools and Algorithms 1 edition*. Cambridge University Press, 2003.
- [2] Richard Montgomery Alain Chenciner. "A remarkable periodic solution of the three-body problem in the case of equal masses". W: *Annals of Mathematics* 152 (2000), s. 881–901.
- [3] NVIDIA Corporation. *CUDA C Programming Guide*. v9.2.148, 2018. url: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html> (term. wiz. 01.08.2018).
- [4] NVIDIA Corporation. *NVIDIA CUDA Runtime API*. v9.2.148, 2018. url: <http://docs.nvidia.com/cuda/cuda-runtime-api/index.html> (term. wiz. 01.08.2018).
- [5] NVIDIA Corporation. *THRUST*. v9.2.148, 2018. url: <https://docs.nvidia.com/cuda/thrust/index.html> (term. wiz. 01.08.2018).
- [6] J. Stadel D. Richardson T. Quinn i G. Lake. "Direct Large-Scale N-Body Simulations of Planetesimal Dynamics". W: *Icarus* 143 (2000), s. 45–59.
- [7] A.Bezgodov D.Egorov. "Improved Force-Directed Method of Graph Layout Generation with Adaptive Step Length". W: *Procedia Computer Science* 66 (2015), s. 689–696.
- [8] G.Chincarini E.D'Onghia C.Firmani. "The Halo Density Profiles with Non-Standard N-body Simulations". W: *Monthly Notices of the Royal Astronomical Society* 338 (2003), s. 156–164.
- [9] M.Kramer G.Lodge J. A. Walsh. "A Trilinear Three-Body Problem". W: *International Journal of Bifurcation and Chaos* 13 (2003), s. 2141–2155.
- [10] Khronos Group. *OpenGL API, OpenGL Shading Language and GLX Specifications*. OpenGL 4.6. 2017. url: https://www.khronos.org/registry/OpenGL/index_gl.php (term. wiz. 14.05.2018).
- [11] Douglas C. Heggie. "CHAOS IN THE N-BODY PROBLEM OF STELLAR DYNAMICS". W: *Predictability, Stability, and Chaos in N-Body Dynamical Systems* (1991), s. 47–62.
- [12] Sam Hokevar. *Tutorial 3 : Matrices*. version 2. url: <http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/> (term. wiz. 01.12.2004).
- [13] S. von Hoerner. "Die numerische Integration des n-Körper-Problemes für Sternhaufen. I". W: *Zeitschrift für Astrophysik* 50 (1960), s. 184–214.
- [14] Erik Holmberg. "On the Clustering Tendencies among the Nebulae." W: *Astrophysical Journal* 94 (1941), s. 385.
- [15] P. Hut J. Barnes. "A hierarchical $O(N \log N)$ force-calculation algorithm". W: *Nature* 324 (1986), s. 446–449.
- [16] J.Eiland. *N-Body Simulation of the Formation of the Earth-Moon System from a Single Giant Impact*. 2013.
- [17] Tero Karras. "Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees". W: *High Performance Graphics* (2012).

- [18] Donald Meagher. "Geometric Modeling Using Octree Encoding". W: *Computer Graphics and Image Processing* 19 (1981), s. 129–147.
- [19] Isaac Newton. *Philosophiae Naturalis Principia Mathematica*, 1st American edition. Daniel Adee, New York, 1846.
- [20] Jerry E. White Roger R. Bate Donald D. Mueller. "Fundamentals of astrodynamics". W: DOVER PUBLICATIONS, 1971. Rozd. 1 TWO-BODY ORBITAL MECHANICS, s. 1–49.
- [21] C. F. Peters V.Szebehely. "A New Periodic Solution of the Problem of Three Bodies". W: *The Astronomical Journal* 72 (1967), s. 1187–1190.