Rješavanje problema n tijela pomoću GPU

Pavlović, Mario

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Mario Pavlović

RJEŠAVANJE PROBLEMA *N* TIJELA POMOĆU GPU

Diplomski rad

Voditelj rada:
doc. dr. sc. Zvonimir Bujanović

Zagreb, rujan, 2016.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Sadržaj

Uvod

Suvremeni grafički procesori (GPU) sastoje se od vrlo velikog broja procesorskih jezgri. Takva se arhitektura, osim za osnovnu namjenu izračunavanja i iscrtavanja trodimenzionalne scene, može iskoristiti u svrhu računanja opće namjene. Masovni paralelizam ostvariv pomoću GPU daje potencijalno velika ubrzanja klasičnih algoritama, od matričnih algoritama do algoritama na grafovima.

U ovom diplomskom radu ćemo analizirati potencijal iskorištavanja arhitekture GPU za implementaciju rješenja klasičnog problema međudjelovanja n tijela. Kod ovog problema potrebno je simulirati kretanje svakog od n tijela koja djeluju jedno na drugo gravitacijskom silom.

Prikazat ćemo rješenje sa *brute-force* pristupom koje je složenosti $O(n^2)$ i jako pogodno za paralelizaciju bez obzira koristimo li zajedničku memoriju ili ne. Centralni dio ovog diplomskog rada je algoritam Barnesa i Huta koji rješava problem n tijela u najgorem slučaju sa složenošću $O(n^2)$, a u prosjeku $O(n \log n)$. Prikazat ćemo implementaciju sva tri spomenuta algoritma pomoću tehnologije CUDA te detaljno objasniti korake algoritama.

Poglavlje 1

GPU i programski jezik CUDA

1.1 Grafički procesor (GPU)

Grafički procesor je procesor specijaliziran za procesuiranje računalne grafike te za izračune koji se mogu paralelizirati. Sastoji se od nekoliko multiprocesora koji mogu izvršavati naredbe posve neovisno. Sastavni dio je grafičke kartice i o njemu, uz brzinu memorije, najviše ovise njene performanse. Grafičkom procesoru je glavna zadaća obavljati obrađivanje scene ili izračuna, dok memorija služi kao spremnik za teksture i ostale neophodne podatke. Što je brža memorija na grafičkoj kartici, to brže se može pohraniti i dohvatiti podatke koje GPU obrađuje.

Prvi grafički čipovi su podržavali primitivne operacije kojima se izračunavanje potrebno za prikaz osnovnih elemenata (trokuta, krugova, kocki i slično) izvršavalo mnogo brže nego na glavnom procesoru (CPU). To je ujedno značilo da je glavni procesor oslobođen i ne mora izvršavati te operacije što rezultira ukupno boljim performansama sustava.

Stalnim zahtjevima tržišta za izračunavanjem u realnom vremenu, obrađivanjem 3D grafike visoke rezolucije, programabilni GPU je evoluirao u visoko paralelni, višedretveni i višejezgreni procesor s ogromnom računalnom snagom i veoma velikom propusnosti memorije.

CPU i GPU

Razlika između CPU i GPU je u tome što je GPU specijaliziran za intenzivno računanje tj. visoko paralelizirano računanje, a to je upravo potrebno za brzo iscrtavanje grafike na zaslon. GPU je dizajniran tako da se više tranzistora posvećuje računanju nego dohvatu podataka i kontroli toka kao što se vidi na slici 1.1[1]. Nema jedinstvenu kontrolu toka za

[1] Ova slika, kao i slika 1.2 su preuzete iz [1].

Slika 1.1: Raspored tranzistora na CPU i GPU

svaku dretvu nego više dretvi može imati zajedničku kontrolu toka. Kažemo da se te dretve izvršavaju istovremeno ili da su u istom *warpu*.

GPU je vrlo efikasan kada treba jedan kod izvršiti za više različitih podataka spremljenih u glavnu memoriju. Za takve kodove je odnos broja aritmetičkih i memorijskih operacija uvelike na strani aritmetičkih. Budući da se isti program izvršava za svaki podatak, nije potrebna kompleksna i složena zajednička kontrola toka. Svaka dretva uzima svoj podatak, tako da se dohvati podataka mogu odvijati paralelno. Umjesto da jedna dretva dohvaća blok podataka dohvaća se samo jedan podatak po dretvi. Mnoge se aplikacije mogu znatno ubrzati korištenjem ovog paralelnog prijenosa podataka. Prilikom iscrtavanja trodimenzionalnog tijela na zaslon, veliki skupovi piksela i točaka su pridruženi dretvama koje se paralelno izvode. Mnogi algoritmi koji nisu povezani s grafičkim prikazom tijela na zaslonu se također mogu ubrzati, primjerice algoritmi koji se bave fizičkim simulacijama, računalnom biologijom, financijskim izračunima i raznim drugim znanstvenim i komercijalnim djelatnostima.

1.2 Kratki opis tehnologije CUDA

CUDA je računalna platforma za paralelno računanje i programski model razvijen od strane tvrtke NVIDIA. Ona povećava performanse računanja iskorištavajući resurse grafičkog procesora. Napravljena je da bi se softverskim inženjerima omogućilo jednostavnije korištenje NVIDIA grafičkih procesora. Predstavljanje CUDA-e bilo je u studenom 2006. godine, a prvo izdanje 2007. godine.

Dizajnirana je za rad s jezicima visoke razine kao što su C, C++ i Fortran. Tri osnovna principa CUDA tehnologije su:

1. Hijerarhija grupa dretvi.
2. Zajednička (*shared*) memorija.
3. Sinkronizacija.

Ta tri principa zahtijevaju od programera podjelu problema na jednostavnije potprobleme koji se mogu rješavati nezavisno. Takvi potproblemi se mogu paralelizirati pomoću blokova dretvi jer se svaki blok izvršava na zasebnom multiprocesoru neovisno od ostalih blokova. Svaki potproblem bi se trebao razbiti na pogodnije dijelove koji se mogu rješavati paralelno sa punom kooperacijom između svih dretvi u bloku.

Ovakva dekompozicija problema čuva izražajnost jezika i dozvoljava dretvama suradnju prilikom rješavanja manjih problema. Budući da se GPU sastoji od nekoliko multiprocesora svaki blok dretvi može biti raspoređen na nekom od dostupnih. Ovisno o odnosu broja multiprocesora i broja blokova, neki će blokovi biti izvršeni paralelno, a neki sekvencijalno. Jednom preveden CUDA program se može izvršavati neovisno o broju multiprocesora u sklopu GPU-a kao što je prikazano na slici 1.2. Dakle, CUDA program je skalabilan u odnosu na broj multiprocesora. Očito je da će GPU s više multiprocesora brže izvršiti program nego oni sa manje jer mogu više blokova paralelno izvršavati. Ovakav skalabilni model CUDA tehnologije dozvoljava raširenost GPU arhitekture od modela za široku uporabu, do profesionalnih GPU Quadro i Tesla modela sa visokim performansama.

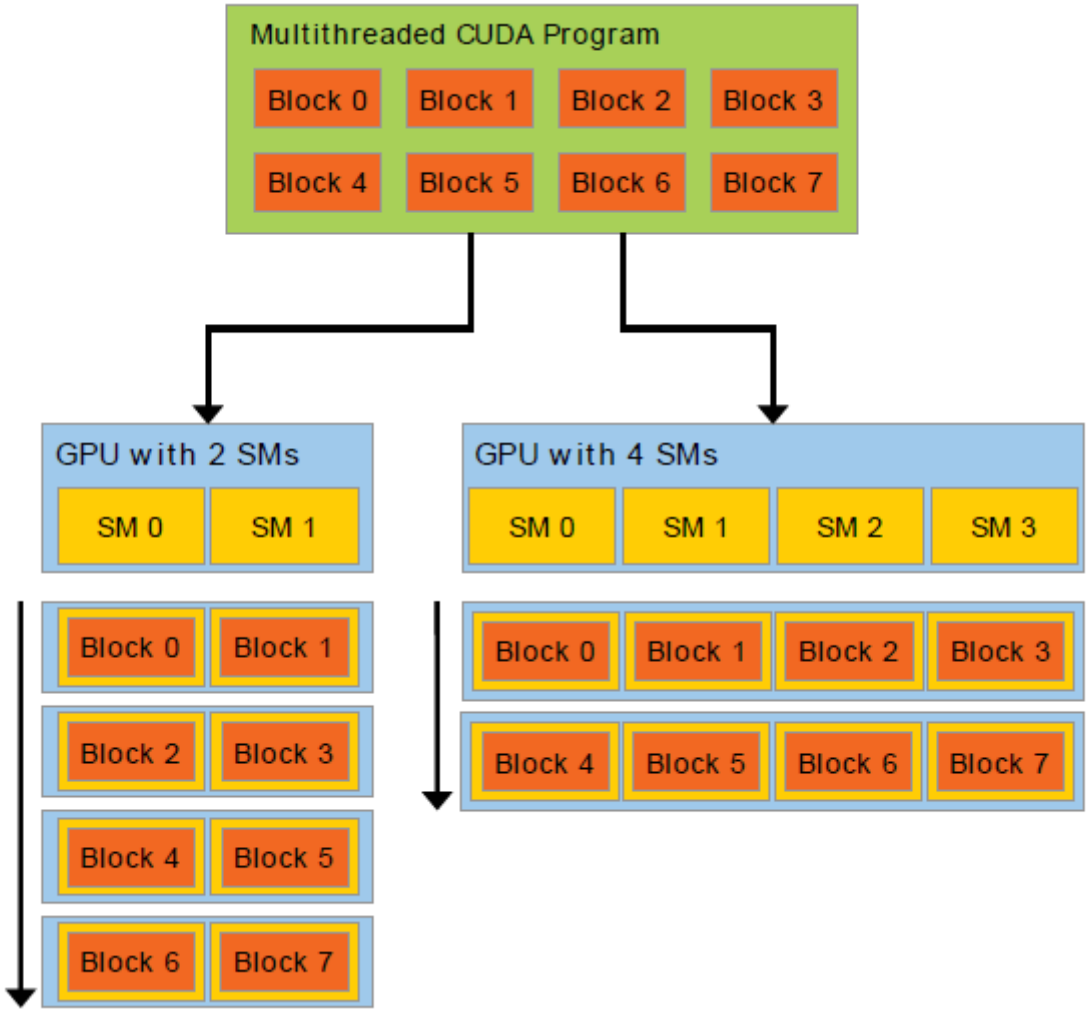
1.3 Arhitektura CUDA programa

Program napisan u CUDA-i se sastoji od dijela koji se izvršava sekvencijalno na CPU (tzv. host dio), te dijela koji se izvršava na GPU (tzv. device dio). Dio programa koji se izvodi na GPU radi paralelno na način kako je opisano ranije, a poziva se kao specijalna funkcija (tzv. kernel) iz host dijela. Te specijalne funkcije ćemo opisati nešto kasnije. Prije toga je potrebno definirati i alocirati polja koja će koristiti na GPU.

Alociranje memorije i kopiranje podataka sa CPU na GPU i obrnuto

CUDA ima ugrađene funkcije za alociranje memorije na GPU te za kopiranje podataka sa CPU memorije na GPU memoriju i obrnuto. Alociranje memorije na GPU se vrši pozivom funkcije `cudaMalloc` koja prima referencu na prvi element polja i veličinu polja koje treba alocirati. Pozivom funkcije `cudaFree` koja prima pokazivač na prvi element polja oslobađa se prethodno zauzeta memorija.

Kopiranje podataka sa CPU na GPU i obrnuto se vrši pozivom funkcije `cudaMemcpy` koja prima pokazivač na prvi element polja kojeg treba kopirati, pokazivač na prvi element polja u koje se trebaju kopirati podatci, veličinu podataka koje treba kopirati i kao zadnji



Slika 1.2: Različit raspored blokova prilikom izvršavanja CUDA programa

argument prima varijablu koja označava tok podataka. Tok može biti od CPU prema GPU (`cudaMemcpyHostToDevice`) ili od GPU prema CPU (`cudaMemcpyDeviceToHost`).

Kerneli

Kao što smo naveli, kerneli su specijalne funkcije koje omogućavaju izvođenje naredbi na GPU. Definira se ključnom riječi `__global__`. Pogledajmo primjer poziva jednog

kernela:

```
f<<<N,M>>>(a);
```

Poziv `f<<<N, M>>>( a )` će izvršiti kod definiran u funkciji `f` na GPU. Pri tome će biti stvoreno `N` blokova, a svaki od blokova sastojat će se od `M` dretvi. Svaka dretva će u cijelosti izvršiti kod funkcije `f`. Pri tome, dretve koje pripadaju istom bloku imaju mogućnost sinkronizacije i međusobne komunikacije. Međutim, to nije moguće za dretve iz različitih blokova – programer nema nikakvu kontrolu nad time kojim će se redom izvoditi pojedini blokovi. Unutar funkcije `f` je omogućen pristup nekim dodatnim varijablama koje omogućavaju identifikaciju dretve. Svaka dretva koja izvršava kernel dobiva jedinstven broj dretve u bloku i svaki blok dobiva jedinstven broj bloka. Njima se može pristupiti u kernelu koristeći ugrađene varijable `threadIdx` i `blockIdx`. Dimenzija bloka se može dohvatiti ugrađenom varijablom `blockDim`.

```
__global__ square(int number)
{
  int id = threadIdx.x;
  printf("%d\n",id*id);
}
int main(void)
{
  int number = 10;
  square<<<1,10>>>(number);
  return 0;
}
```

Ovo je primjer jednostavnog CUDA programa koji računa kvadrate prvih deset brojeva. Kao što vidimo, sintaksa funkcije `main` je ista kao i u jeziku C ili C++ s izuzetkom poziva funkcije `square`. Prilikom poziva kernela upisuje se broj blokova i broj dretvi, kod nas su to 1 i 10. Uočimo da kernel dohvaća ugrađenu varijablu `threadIdx` te onda računa kvadrat broja dretve za svaku dretvu. Postoji ograničenje za broj dretvi po blokovima jer se od svih dretvi unutar bloka očekuje da će se izvoditi na jednom multiprocesoru. On ovisi o verziji GPU-a, trenutno najsnažnija verzija GPU-a podržava maksimalno 1024 dretve po bloku.

Kernel može biti izvođen paralelno na više blokova, tako da je ukupan broj dretvi koje izvršavaju kernel jednak broju dretvi po bloku pomnožen s brojem blokova. Osim kernela, postoje i funkcije koje se ne mogu pokrenuti sa CPU-a, nego se pokreću i izvršavaju na GPU. Te funkcije ispred svog imena imaju ključnu riječ `__device__` i za razliku od kernela čiji je povratni tip uvijek `void`, `__device__` funkcije mogu imati bilo koji povratni tip. Ove funkcije su pomoćne funkcije koje pozivaju kerneli.

Mehanizmi sinkronizacije i memorijske ograde

Budući da program koji se izvršava na GPU radi paralelno, odnosno, više dretvi izvršava isti kod, često se dogodi da dretve istovremeno pristupaju istom podatku ili pak da jedna dretva čita podatak iz memorije prije nego što je druga dretva mogla osvježiti njegovu vrijednost.

Uočimo da dretve unutar pojedinog bloka ne moraju izvršavati kod kernela istovremeno, odnosno, ne moraju pripadati istom *warpu*. Zbog toga, u GPU postoje ugrađene funkcije koje sinkroniziraju rad dretvi unutar bloka. Funkcija `__synchthreads` sinkronizira rad unutar jednog bloka na način da se dretve tog bloka zaustavljaju na toj naredbi. Tek kada su sve dretve iz bloka došle do `__synchthreads`, one nastavljaju sa izvođenjem koda.

Moguće je koristiti "laku" sinkronizaciju između blokova, tj. *memory fence* funkcije koje služe da bi se dala prednost dretvi koja upisuje nad dretvom koja čita podatak s nekog mjesta u glavnoj ili zajedničkoj memoriji ukoliko su postavile zahtjev istovremeno.

Zajednička (*shared*) memorija

Uz glavnu memoriju, na GPU postoji i zajednička (*shared*) memorija. Svaki blok ima svoju *shared* memoriju koju dijele sve dretve u bloku. Ukoliko želimo da neka dretva pročita iz zajedničke memorije što je druga dretva zapisala, potrebno je prije čitanja pozvati `__synchthreads`. Pristup zajedničkoj memoriji je znatno brži nego pristup glavnoj memoriji. Zajedničku memoriju možemo definirati statički;

```
__global__ someKernel(int a)
{
  __shared__ int value[10];
}
```

ili dinamički;

```
__global__ someKernel(int a)
{
  extern __shared__ int value[];
}
```

Kod dinamičkog definiranja zajedničke memorije, prilikom poziva kernela je potrebno navesti veličinu zajedničke memorije. To izgleda ovako:

```
someKernel<<<nBlock,nThread,sizeof(int)*10 >>>(a);
```

Nakon imena kernela koji pozivamo u oštrim zagradama upisujemo redom: broj blokova, broj dretvi po bloku i veličinu zajedničke memorije koju treba alocirati. Nakon toga idu argumenti funkcije.

Atomic funkcije

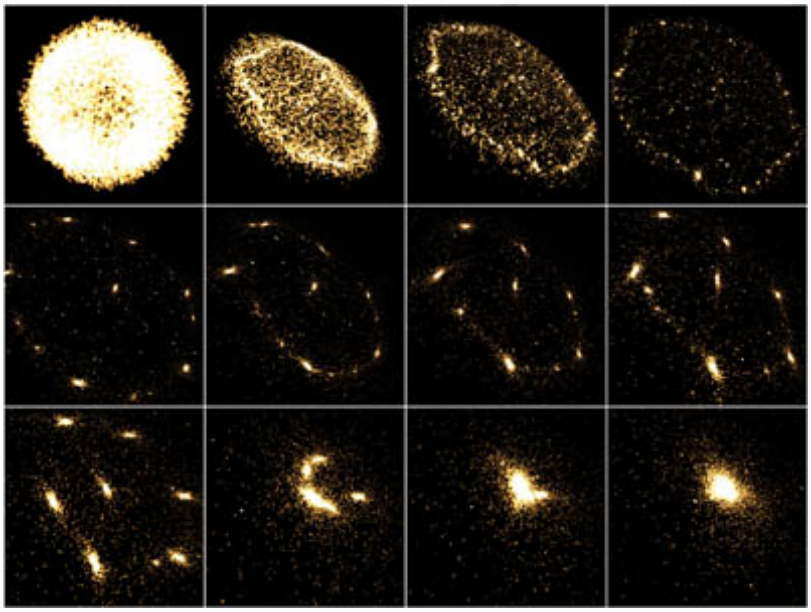
Ponekad je potrebno osigurati da samo jedna dretva može čitati ili upisati neki podatak. Za to nam služe *atomic* funkcije u CUDA-i. One osiguravaju da samo jedna dretva može obavljati kritičnu operaciju dok sve druge čekaju dok ne dobiju dozvolu.

Za primjer ćemo opisati funkciju `atomicCAS(mutex, val, newVal)`. Ona prima tri argumenta: pokazivač `mutex` tipa `(int*)` i dvije varijable tipa `int`. Funkcija uspoređuje vrijednosti na koju pokazuje pokazivač `mutex` s varijablom `val`. Ako su vrijednosti jednake tada se na mjestu na koje pokazuje `mutex` zapisuje vrijednost `newVal`.

Poglavlje 2

Problem *n* tijela

2.1 Opis problema



Slika 2.1: Okviri interaktivne 3D simulacije 16384 tijela

Simulacija *n* tijela numerički aproksimira evoluciju sustava u kojem svako tijelo neprekidno djeluje na sva druga tijela. Dobar primjer je astrofizička simulacija u kojoj tijela predstavljaju galaksije ili pojedine zvijezde koje se privlače međusobno djelovanjem gravitacijske sile kao na slici 2.1[1]. Ovakva simulacija se pojavljuje u mnogim drugim

[1] Ova slika je preuzeta iz [2].

računalnim problemima, primjerice: sklopovi proteina su proučavani korištenjem simulacije n tijela za računanje elektrostatičke i van der Waalsove sile, burni tok tekućina i računanje globalnog osvjetljenja u računalnoj grafici.

Jedan mogući pristup računanju interakcije n tijela je računanjem sile između svaka dva tijela. Ovakav pristup predstavlja računanje *grubom silom*. To je relativno jednostavna metoda koja je laka za implementirati. U pravilu se ne koristi jer je njezina složenost $O(n^2)$. Ipak, ovakav pristup se koristi u sofisticiranijim algoritmima kao jezgra za određivanje sila između tijela koja su relativno blizu.

2.2 Računanje sile između svaka dva tijela

Danih n tijela imaju koordinate $\vec{x_i} \in \mathbb{R}^3$ i početne brzine $\vec{v_i} \in \mathbb{R}^3$ za $i \in 1, 2, ..., n$. Vektor sile $\vec{f_{ij}}$ na tijelo i uzrokovan djelovanjem gravitacijske sile tijela j dan je jednadžbom:

$$\vec{f_{ij}} = G\frac{m_i m_j}{||\vec{r}_{ij}||^2} \cdot \frac{\vec{r}_{ij}}{||\vec{r}_{ij}||}, \tag{2.1}$$

gdje su m_i i m_j mase tijela i i j, a $\vec{r}_{ij}$ je vektor smjera sile od tijela i do tijela j dan sa $\vec{r}_{ij} = \vec{x}_j - \vec{x}_i$. G je gravitacijska konstanta. Uočimo da je lijevi faktor (magnituda sile) proporcionalna produktu masa i obrnuto proporcionalna kvadratu udaljenosti između tijela i i j. Desni faktor označava smjer djelovanja sile odnosno jedinični vektor od tijela i u smjeru tijela j.

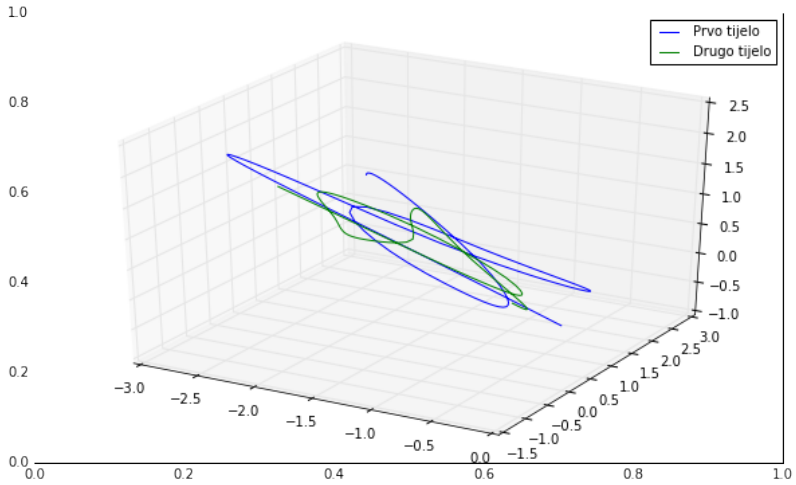
Cjelokupna sila na tijelo i, u oznaci $\vec{F_i}$, uzrokovana reakcijom s preostalih $n - 1$ tijela dobiva se zbrajanjem svih interakcija:

$$\vec{F_i} = \sum_{\substack{1 \le j \le N \\ j \neq i}} \vec{f_{ij}} = Gm_i \sum_{\substack{1 \le j \le N \\ j \neq i}} \frac{m_j \vec{r}_{ij}}{||\vec{r}_{ij}||^3}. \tag{2.2}$$

Primjetimo da će sila između tijela prilikom približavanja rasti bez ograničenja. To predstavlja problem jer će se dogoditi da u jednom trenutku dijelimo s 0. Da bi izbjegli tu situaciju potrebno je dodati *ublažujući faktor* u nazivnik. Ovo ima smisla jer u astrofizičkim izračunima sudari su uglavnom isključeni. Tijela predstavljaju galaksije koje mogu prolaziti jedna kroz drugu. Stoga uvodimo *ublažujući faktor* $\epsilon > 0$ u nazivnik tako da jednadžba sada izgleda:

$$\vec{F_i} \approx Gm_i \sum_{1 \le j \le N} \frac{m_j \vec{r}_{ij}}{\left(||\vec{r}_{ij}||^2 + \epsilon^2\right)^{\frac{3}{2}}}. \tag{2.3}$$

Uočimo da uvjet $i \neq j$ više nije potreban jer $\vec{f_{ii}} = 0$ zbog toga što je tada u nazivniku $\epsilon^2 > 0$, a u brojniku je 0. Tijela se ponašaju kao sferične galaksije. Ublažujući faktor ograničava magnitudu sile između dva tijela, što je poželjno kod numeričkog računanja stanja sustava.



Slika 2.2: Prikaz putanja dva tijela u međudjelovanju tri tijela

Da bismo izračunali sljedeću poziciju, odnosno, pomak svakog od n tijela, potrebno je izračunati njegovo ubrzanje, $\vec{a_i} = \frac{\vec{F_i}}{m_i}$. Uvrštavanjem izraza za F_i, dobivamo sljedeću formulu:

$$\vec{a_i} \approx G \sum_{1 \leq j \leq N} \frac{m_j \vec{r}_{ij}}{\left(\|\vec{r}_{ij}\|^2 + \epsilon^2\right)^{\frac{3}{2}}}. \tag{2.4}$$

Za primjer, pogledajmo implementaciju algoritma koja simulira međudjelovanje n tijela pomoću formule 2.4. Podatke o tijelu ćemo spremati u strukturi definiranoj ovako:

```
typedef struct
{
  double x, y, z, m;
}double4;
```

pri čemu su x, y, z koordinate iz $\mathbb{R}^3$, a masa tijela m iz $\mathbb{R}$. Podatke o ubrzanju i brzini spremamo u strukturu definiranoj ovako:

```
typedef struct
{
  double x, y, z;
}double3;
```

pri čemu su x, y, z koordinate iz $\mathbb{R}^3$.

```
double3 calculatePositions(double4 * bodies, double4 body,
double3 * initVel, int N)
```

```
{
  double3 acc,dv,position;
  acc.x = 0; acc.y = 0; acc.z = 0;
  for(int i = 0; i < N; ++i)
  {
    dv.x = bodies[i].x - body.x;
    dv.y = bodies[i].y - body.y;
    dv.z = bodies[i].z - body.z;

    double norm = sqrt(dv.x*dv.x + dv.y*dv.y+dv.z*dv.z);
    double result = norm*norm+E*E;

    result = sqrtf(result*result*result);
    result = bodies[i].m/result;

    acc = vecAdd(acc,vecMulScal(dv,result));
  }
 position.x = body.x + (initVel->x + acc.x*T/2)*T;
 position.y = body.y + (initVel->y + acc.y*T/2)*T;
 position.z = body.z + (initVel->z + acc.z*T/2)*T;

 initVel->x = initVel->x + acc.x*T;
 initVel->y = initVel->y + acc.y*T;
 initVel->z = initVel->z + acc.z*T;
 return position;
}
```

Algoritam radi tako da se za svako tijelo poziva funkcija `calculatePositions`. Funkcija prima sljedeće parametre: polje `bodies` u kojem su pohranjena sva tijela, tijelo `body` za kojeg se računa nova pozicija, pokazivač na varijablu u kojoj je spremljena početna brzina i ukupan broj tijela. Varijabla `T` je unaprijed definirana i označava vremenski interval. `E` je *ublažujući faktor* koji sami postavljamo. Funkcija prolazi po svim tijelima u petlji te za svako od njih računa doprinos ubrzanju. Kad se izračuna ubrzanje, pomoću početne brzine, starih pozicija i ubrzanja računaju se nove pozicije po formuli:

$$\vec{s} = \vec{s_0} + \left(\vec{v_0} + \frac{\vec{a}t}{2}\right)t. \tag{2.5}$$

Prilikom računanja evolucije sustava bit će potrebno računati i novu brzinu pa dobivamo sljedeću formulu:

$$\vec{v} = \vec{v_0} + \vec{a}t. \tag{2.6}$$

Nakon što smo definirali načine na koji ćemo računati evoluciju sustava n tijela, možemo pristupiti implementaciji algoritama na GPU.

Poglavlje 3

Algoritmi za rješavanje problema *n* tijela

U ovom dijelu ćemo opisati i implementirati tri algoritma u CUDA-i za rješavanje problema *n* tijela pomoću GPU. Algoritmi su sljedeći:

1. Jednostavni algoritam za rješavanje problema *n* tijela, koji predstavlja direktnu paralelizaciju funkcije `calculatePositions` iz prethodnog poglavlja.

2. Varijanta jednostavnog algoritma iz točke 1, koja koristi zajedničku (*shared*) memoriju na GPU.

3. Algoritam Barnesa i Huta za rješavanje problema *n* tijela.

Prva dva algoritma su vremenske složenosti $O(n^2)$, a treći algoritam je u najgorem slučaju vremenske složenosti $O(n^2)$, dok je u prosjeku približno $O(n \log n)$ složenosti.

3.1 Jednostavni algoritam za rješavanje problema *n* tijela

Ovaj algoritam se sastoji od samo jednog kernela. On za svako tijelo dohvaća podatke te u petlji računa njegovo ubrzanje nastalo interakcijom s ostalim tijelima. Pomoću ubrzanja i početne brzine računa se nova pozicija tijela po formuli (2.5).

Kod algoritma:

```
__device__ void setNewPositions(double4 * bodies, double3 * acc,
 double3 * initVel,int id)
{
   bodies[id].x += (initVel[id].x + acc[id].x*T/2)*T;
   bodies[id].y += (initVel[id].y + acc[id].y*T/2)*T;
   bodies[id].z += (initVel[id].z + acc[id].z*T/2)*T;
```

```
   initVel[id] = vecAdd(initVel[id],vecMulScal(acc[Id],T));
}
__device__ double3 getDirectionVector(double4 body1, double4 body2)
{
   double3 result;
   result.x = body2.x - body1.x;
   result.y = body2.y - body1.y;
   result.z = body2.z - body1.z;
   return result;
}
__global__ calculateForces(double4 * bodies, double3 * acc,
double3 * initVel,int N)
{
   int id = blockIdx*blockDim.x + threadIdx.x;
   if(id >= N) return;
   double3 tempAcc = getInitAcc();
   double4 bodyCoords = bodies[id];
   for(int i = 0; i < N; ++i)
   {
        double3 dv = getDirectionVector(bodyCoords,bodies[i]);
        double result = norm3df(dv);
        result = result*result + E*E;
        result = sqrtf(result*result*result);
        result = bodies[i].m/result;

        tempAcc = vecAdd(tempAcc,vecMulScal(dv,result));
   }
   tempAcc = vecMulScal(tempAcc,G);
   acc[id] = tempAcc;
   setNewPositions(bodies,acc,initVel,id);
}
```

Objašnjenje jednostavnog algoritma za rješavanje problema *n* tijela

Kernel `calculateForces` prima sljedeće parametre: polje `bodies` u kojem su pohranjena sva tijela u sustavu, polje `acc` u koje ćemo spremati izračunato ubrzanje, polje `initVel` u kojem je pohranjena početna brzina i broj `N` koji označava broj tijela u sustavu. Prvo se određuje `id` dretve te se svakoj dretvi dodjeljuje po jedno tijelo. Za svaku dretvu se definira i inicijalizira početno ubrzanje koje iznosi 0 po svim koordinatama s funkcijom `getInitAcc`. Redak `if(id>=N) return;` služi da bi se blokirale sve dretve koje imaju indeks veći od zadnjeg indeksa tijela.

Petlja prolazi po svim tijelima te računa doprinos svakog od njih na način kao što je opisano jednadžbom (2.4). Funkcija `getDirectionVector` prima za argumente dva tijela te vraća njihovu razliku po prve tri koordinate (osim mase). Ona se za sve dre-

tve izračunava n puta. Nakon što se izračuna doprinos jednog tijela ubrzanju, dodaje se funkcijom `vecAdd` koja prima dva argumenta tipa `double3` i vraća njihov zbroj na već postojeće ubrzanje. Funkcija `vecMulScal` prima vektor `double3` i skalar `double` te vraća vektor pomnožen sa skalarom.

Nakon što je dretva izašla iz petlje, dobivena vrijednost se sprema u polje `acc` te zatim pomoću izračunatog ubrzanja, položaja tijela i početne brzine postavljaju se nove koordinate tijela i to kao što je definirano u (2.5). Zatim se nove koordinate spremaju u polje `bodies` tako da se pregaze stare. Uz poziciju svaki put se računa i brzina po formuli (2.6).

Ovaj način rješavanja je mnogo efikasniji od implementacije istog algoritma na CPU. Ostvareno je potpuno paralelno i nezavisno računanje nove pozicije za svako od n tijela. Koliko će ovaj algoritam biti brži od sekvencijalnog, ovisi o tome koliko GPU ima multiprocesora i o tome koliko dretvi svaki multiprocesor može istovremeno izvršavati. No, ovaj algoritam ima i dva velika nedostatka:

1. Velika složenost koja iznosi $O(n^2)$ postaje problem kada se broj tijela poveća.
2. Pristupanje podatcima iz glavne memorije. Treba izbjeći pristupanje podatcima u glavnoj memoriji ako je to moguće tj. koristiti blokovsku zajedničku (*shared*) memoriju.

3.2 Jednostavni algoritam za rješavanje problema *n* tijela koji koristi zajedničku memoriju

Ovaj algoritam se kao i prethodni sastoji samo od jednog kernela. Logika izračunavanja je ista kao i u prethodnom algoritmu, no uz potpuno drugačiji pristup memoriji.

```
__device__ double3 bodyToBodyInteraction(double4 b1, double4 b2,
double3 a )
{
  double3 dv = getDirectionVector(b1,b2);

  double result = norm3df(dv);
  result = result*result + E*E;
  result = sqrtf(result*result*result);
  result = b2.m/result;
  a = vecAdd(a,vecMulScal(dv,result));
  return a;
}
__device__ double3 tileCalculation(double4 body, double3 a)
{
  int i;
  extern __shared__ double4 shBodies[];
```

```
  for (i = 0; i < blockDim.x; i++)
  {
    a = bodyToBodyInteraction(body, shBodies[i], a);
  }
  return a;
}
__global__ void calculateForces(double4 * bodies, double3 * acc,
double3 * initVel,unsigned N)
{
  int id = blockIdx.x*blockDim.x + threadIdx.x;
  if(id >= N) return;
  extern __shared__ double4 shBodies[];
  int p = blockDim.x;
  int i, tile;
  double4 b1;
  b1.x = 0.0; b1.y = 0.0; b1.z = 0.0; b1.m = 0.0;
  double3 a = getInitAcc();
  double4 t  = bodies[id];
  for (i = 0, tile = 0; i < N; i += p, tile++)
  {
    int idx = tile * blockDim.x + threadIdx.x;
    if(idx >= N ) shBodies[threadIdx.x] = b1;
    else shBodies[threadIdx.x] = bodies[idx];
    __syncthreads();
    a = tileCalculation(t,a);
    __syncthreads();
  }
  a = vecMulScal(a,G);
  acc[id] = a;
  setNewPositions(bodies,acc,initVel,id);
}
```

Objašnjenje jednostavnog algoritma za rješavanje problema *n* tijela koji koristi zajedničku memoriju

Device funkcija `bodyToBodyInteraction` prima dva tijela (dva `double4`) i ubrzanje (`double3`) te računa iznos ubrzanja za prvo tijelo nastalo djelovanjem drugog tijela. Rezultat se dodaje na već postojeće ubrzanje. U ovom se kernelu koriste funkcije `getDirectionVector`, `vecAdd` i `vecMulScal` definirane na isti način kao i u prethodnom algoritmu.

Funkcija `tileCalculation` prima jedno tijelo (`double4`) i njegovo parcijalno izračunato ubrzanje (`double3`). Ona za svako tijelo prethodno spremljeno u `shared` memoriju izvršava funkciju `bodyToBodyInteraction` koja prima dva tijela i ubrzanje.

Kernel funkcija `calculateForces` prima iste argumente kao i prethodni algoritam. U njoj se prvo dohvaća `id` dretve te se nakon toga inicijalizira `extern __shared__ double4` memorija u koju će svaka dretva spremati tijela za računanje ubrzanja.

Prefiks `extern` znači da se zajednička memorija prvo mora dinamički alocirati prilikom poziva kernela, na način kako je opisano u prvom poglavlju.

U sljedećem koraku algoritma, definiramo i inicijaliziramo na nulu ubrzanje `a` pomoću `getInitAcc`. Nakon toga i-ta dretva dohvaća i-to tijelo i sprema ga u lokalnu varijablu. Na taj način ga ne mora svaki put dohvaćati iz glavne memorije.

Nakon toga slijedi petlja u kojoj se tijela ubacuju u zajedničku memoriju. Petlja će prvo ubacivati tijela koja su u bloku 0, njih `blockDim.x`, te će nakon toga svaka dretva iz bloka izračunati doprinos svih tijela iz bloka 0 ubrzanju tijela koje pripada toj dretvi. Zatim će se ovaj postupak ponoviti za blok 1, pa za blok 2 i tako dalje. Primjetimo da će i-ta dretva ubacivati redom $(i + D \cdot p)$-to tijelo u zajedničku memoriju gdje je D dimenzija bloka, a p je broj bloka. To osigurava da nikoje dvije dretve neće ubaciti isto tijelo.

`__syncthreads` služi da bi se dretve zaustavile na toj naredbi. Dretve ne smiju krenuti u računanje prije nego sva tijela ubace u zajedničku memoriju. Ukoliko to nije slučaj, moglo bi se dogoditi da neke dretve koje su ranije završile s ubacivanjem tijela počnu računati ubrzanje za tijelo koje nije ubačeno.

Nakon petlje ubrzanje nije još izračunato, odnosno treba ga pomnožiti s gravitacijskom konstantom G, nakon čega se nova pozicija računa kao i u prethodnom algoritmu.

Poziv kernela izgleda ovako:

```
calculateForces << <blocksPerGrid, threadPerBlock,
threadPerBlock*sizeof(double4) >> >
(d_bodies, d_acc, d_initialVelocity, N);
```

3.3 Algoritam Barnesa i Huta

Uvod u algoritam

Budući da prethodni algoritmi zbog svoje velike kompleksnosti nisu primjenjivi za računanje u stvarnom vremenu, potreban je brži algoritam. J. Barnes i P. Hut su 1986. godine objavili hijerarhijski $O(n \log n)$ algoritam za rješavanje problema n tijela. Taj algoritam ima potpuno drugačiji pristup računanju no opisani algoritmi. Pristup je baziran na organizaciji tijela u specijalno stablo, tzv. octtree, ovisno o poziciji svakog tijela u prostoru. Svaki list u stablu će predstavljati jedno tijelo, a preostali čvorovi su tzv. ćelije. Svaka ćelija čuva informaciju o ukupnoj masi svih tijela koja se nalaze u podstablu kojem je ta ćelija korijen. Također, u svakoj ćeliji čuvamo i koordinate centra mase svih tijela iz podstabla. Ušteda vremena će se postići zbog zamjene svih tijela u podstablu sa fiktivnim tijelom čiji

su podatci u ćeliji. Fiktivna tijela ćemo koristiti umjesto svih tijela u podstablu samo ako je ćelija "dovoljno daleko".

Efikasna implementacija algoritma za problem n tijela u CUDA-i baziranog na octtree-u se sastoji od šest odvojenih kernela. Prvo se prostor hijerarhijski raspolavlja te se zapisuje pomoću octtree-a. Octtree je trodimenzionalni ekvivalent binarnog stabla u kojem svaki čvor ima maksimalno 8 djece. Nakon što se stablo izgradi, za svako podstablo se računaju informacije o ukupnoj masi svih tijela koja se nalaze u njemu. Kad smo sve to napravili, za svako tijelo obilazimo stablo te računamo silu u odnosu na ostale čvorove u stablu uzimajući u obzir njihovu udaljenost. Algoritam Barnesa i Huta korištenjem aproksimacija prilikom obilaska smanjuje kompleksnost u prosjeku na $O(n \log n)$ te nam je zbog toga zanimljiv.

Veliki problem s kojim se algoritam suočava je taj što se stablo za svaki vremenski korak mora iznova graditi i svaki put se treba prolaziti kroz njega. To je neophodno da bi se algoritam efikasno implementirao u CUDA-i. Algoritam zahtjeva rekurziju, a rekurzija nije podržana na trenutnim GPU-ovima, tako da se moraju koristiti iteracije. U algoritmu ćemo prikazati korištenje nekih od svojstava GPU-a na razne načine. Primjerice, isključivo korištenje polja (pisanje i čitanje) u glavnoj GPU memoriji ne dolazi u obzir jer je puno zahtjeva na *sporu* memoriju. Stoga će se koristiti kombinacija učitavanja podataka u zajedničku memoriju, registre, cache i zaustavljanje dretvi da bi se smanjio broj zahtjeva na glavnoj memoriji.

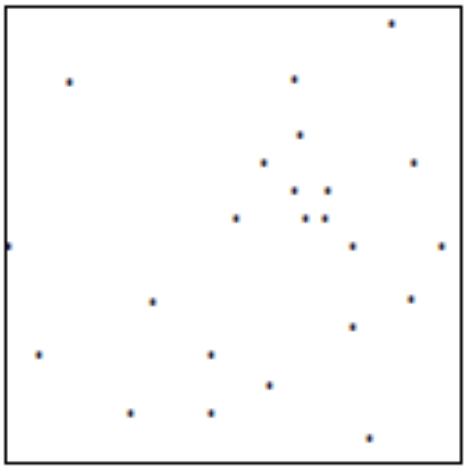
Često se javlja problem kod jednog *warpa* dretvi. Budući da dretve u jednom *warpu* istovremeno pokušaju pristupiti istom elementu, koristimo jednu dretvu u *warpu* koja će dohvaćati podatak iz glavne memorije i spremati je u zajedničku memoriju. Tako će dijeliti podatke s ostalim dretvama bez potrebe sinkronizicije.

Također će se koristiti operacije specifične za GPU kao što su glasovanje dretvi (*thread-voting*) da bi se značajno poboljšale performanse. *Thread fence* instrukcijama će se implementirati "perolaka" sinkronizacija bez upotrebe atomskih operacija. Na slikama ćemo ilustrirati dvodimenzionalnu varijantu problema (izgradnju tzv. quadtree stabla), dok će konačna implementacija rješavati punu trodimenzionalnu varijantu (i izgraditi octtree stablo).

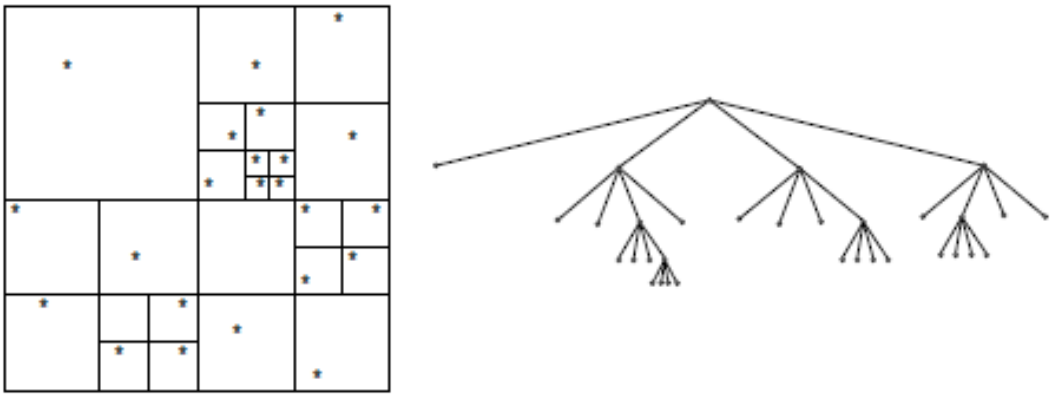
Koraci algoritma

Prije nego što krenemo u implementaciju opisat ćemo ukratko šest koraka algoritma koji se sekvencijalno (jedan iza drugog) izvode na GPU pozivom kernela za svakog od njih. Ovaj način razbijanja algoritma na više kernela je dobar iz razloga što možemo podešavati na koliko blokova i s koliko dretvi po bloku želimo da se pojedini kernel izvršava.

1. Prvi kernel izračunava vršnu ćeliju (korijen stabla) koja će obuhvaćati sva tijela, tako



Slika 3.1: Inicijalna ćelija - rezultat izračuna prvog kernela. Svaka zvjezdica predstavlja jedno tijelo.



Slika 3.2: Hijerarhijska dekompozicija tijela u 2D (lijevo); dobiveno quadtree stablo (desno)

da traži maksimume i minimume po svim koordinatima za sva tijela. Vidi sliku 3.1[1].

2. Drugi kernel hijerarhijski raspolavlja pojedine ćelije po svakoj od koordinata sve dok u svakoj ćeliji ne preostane najviše po jedno tijelo. Ovo ćemo implementirati posebnim algoritmom koji će svako tijelo ubaciti u octtree. Vidi sliku 3.2, na kojoj prvo dijete svakog čvora odgovara gornjem lijevom pod-kvadrantu pripadne ćelije, drugo gornjem desnom, treće donjem lijevom, a četvrto donjem desnom. Slično ograniziramo i pod-oktante u octtree-u.

3. Treći kernel za svaku ćeliju računa centar mase i ukupnu masu svih tijela ili ćelija u njoj.

4. Četvrti kernel sortira tijela tako da tijela koja su po koordinatama blizu budu u istom bloku kad se izračunavanje bude izvršavalo u sljedećem kernelu.

5. Peti kernel računa silu koja djeluje na svako tijelo. Počevši od korijena stabla, provjerava se je li centar mase ćelije udaljen dovoljno daleko od trenutnog tijela za svaku ćeliju. Ako jest, suma sila kojom djeluju tijela iz podstabla aproksimira se jednom silom koja djeluje iz centra mase. Ako je centar mase blizu trenutnog tijela, ovaj postupak se ponavlja za svako dijete ćelije.

6. Šesti kernel postavlja tijela na nove pozicije i postavlja novu brzinu.

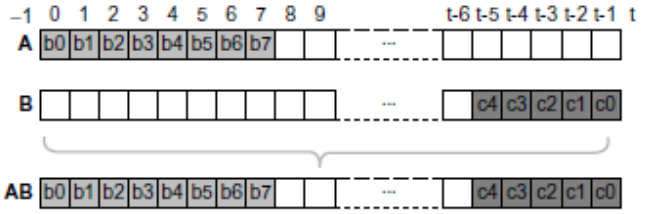
Algoritam i implementacija

Ovdje opisujemo implementaciju algoritma Barnesa i Huta u CUDA-i, s fokusom na optimizaciju koju ćemo koristiti da bi se algoritam efikasno izveo na GPU. Prvo ćemo opisati neke od globalnih optimizacija koje se provlače kroz sve kernele pa ćemo onda promatrati svaki kernel posebno. Ova generalna optimizacija se može primjeniti i na druge algoritme.

Globalna optimizacija

Dinamičke strukture podataka kao što su stabla obično su građene od objekata koji se čuvaju na heap-u. Svaki takav objekt sadrži više polja tj. svaki pokazivač na dijete se mora dinamički alocirati. Budući da je dinamička alokacija i pristup objektima na heap-u spor proces, koristimo strukturu baziranu na poljima. Dakle, u kodu ćemo koristiti polje indeksa umjesto pokazivače na čvorove stabla. Sam octtree će biti implementiran kao $2n \times 8$ matrica jer svaka ćelija može imati maksimalno 8 djece, a maksimalan broj ćelija moramo pretpostaviti. U ovoj implementaciji se pretpostavlja da je maksimalan broj ćelija $2n$. Postoji mogućnost da će nam broj ćelija biti nedovoljan, ali na to u ovom trenutku

[1] Ova slika, kao i slike 3.2,3.3,3.4 su preuzete iz [3].



Slika 3.3: Polje indeksa za konstrukciju stabla

ne možemo utjecati jer bi zahtjevalo traženje najmanje udaljenosti između dva tijela među svim tijelima što je složenosti $O(n^2)$.

Da bismo ubrzali kod koristit ćemo isto polje indeksa za tijela i za ćelije. Za indekse tijela ćemo koristiti početne, a za indekse ćelija zadnje indekse polja indeksa. Vidi sliku 3.3. Ukoliko je neki element polja jednak -1, smatramo da je taj element slobodan, tj. da ga ne zauzima tijelo ili ćelija. Prednosti ovog pristupa su:

1. Jednostavna usporedba indeksa polja s brojem tijela daje odgovor radi li se o ćeliji ili o tijelu.

2. Lako možemo odgovoriti na pitanje radi li se o tijelu ili se radi o neiskorištenom indeksu, budući da je -1 manji od broja tijela.

Adrese prvih elemenata različitih polja ostaju nepromijenjene tijekom cijelog algoritma, pa ih možemo samo jednom, na početku, kopirati u konstantnu memoriju na GPU. Ovaj pristup je znatno brži nego prosljeđivanje podataka svakom kernelu posebno.

Algoritam kopira podatke s CPU-a na GPU samo na početku. Tada CPU šalje GPU-u početne položaje tijela i početnu brzinu. Kad se izračunaju svi podatci, oni se kopiraju s GPU-a na CPU. Ovaj pristup izbjegava učestalo korištenje tzv. *uskog grla* za vrijeme simulacije. To je moguće zato što cijeli algoritam izvršavamo na GPU.

Prvi korak algoritma

Prvi kernel računa vršnu ćeliju, ili korijen octtree-a na način da traži maksimume odnosno minimume po svim koordinatama za sva tijela. Time se dobije kocka u kojoj su smještena sva tijela. Vidi sliku 3.1. Svakom bloku dodijeljeno je onoliko tijela koliko ima dretvi u tom bloku. Svaki blok će izračunati minumum i maksimum koordinate svih tijela koja su mu dodijeljena. Na kraju, po jedna dretva iz svakog bloka će osvježiti varijablu u globalnoj memoriji u kojoj čuvamo min/max koordinate svih tijela.

Prilikom čitanja podataka svaka dretva u bloku prvo prenosi tijelo iz glavne u zajedničku memoriju. Na taj način se smanjuju zahtjevi na glavnu memoriju, a povećavaju se na zajedničku koja je brža.

Kod kernela koji pronalazi minimume po svim koordinatama:

```
__global__ void minimumPoint( double4 * bodies, int N, volatile
 double4 * min, int * mutex )
{
  extern __shared__ double4 sBodies[];
  unsigned int id = threadIdx.x + blockIdx.x*blockDim.x;
  if(id >= N) return;
  sBodies[threadIdx.x] = bodies[id];
  __syncthreads();
  for (unsigned int s=1; s < blockDim.x; s *= 2)
  {
    int index = 2 * s * threadIdx.x;
    if (index+s < blockDim.x && (blockIdx.x*blockDim.x+index+s) <
     N)
    {
      if(id+s < N)
      {
        sBodies[index].x = min(sBodies[index].x,sBodies[index+s].x);
        sBodies[index].y = min(sBodies[index].y,sBodies[index+s].y);
        sBodies[index].z = min(sBodies[index].z,sBodies[index+s].z);
      }
    }
    __syncthreads();
  }
  if(threadIdx.x == 0)
  {
    while(atomicCAS(mutex,0,1)!=0);
    min->y = min(min->y,sBodies[0].y);
    min->x = min(min->x,sBodies[0].x);
    min->z = min(min->z,sBodies[0].z);
    atomicExch(mutex,0);
  }
}
```

U ovom kernelu se prvo redukcijom računaju minimumi za svaki blok posebno. To se događa u petlji. Nakon toga, prva dretva osvježava vrijednost globalnog minimuma svih blokova. Ona će minimum izračunat unutar ovog bloka usporediti s trenutnom vrijednosti globalnog minimuma.

Uočimo da je dohvaćanje i osvježavanje vrijednosti globalnog minimuma potrebno napraviti pomoću atomic operacija kako to različiti blokovi ne bi pokušali napraviti istovremeno – dakle, riječ je o tzv. kritičnoj sekciji koda. Kritična sekcija se oslobađa pomoću

naredbe `atomicExch(mutex, val)` tako da se na mjestu na koje pokazuje `mutex` postavi vrijednost `val`.

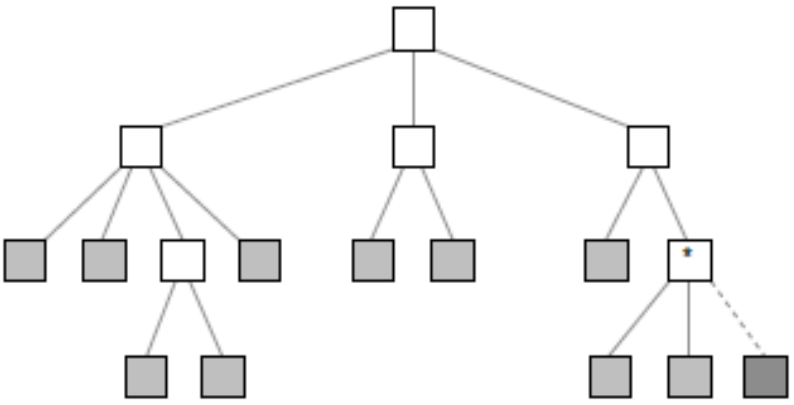
U ovom kernelu se najbolji rezultati postižu ako se broj dretvi po bloku poveća na maksimum (trenutno 1024) tako da pristup kritičnoj sekciji traži što manji broj dretvi.

U argumentima funkcije se može vidjeti da je pokazivač na prvi element polja `min` označen sa `volatile`. To znači da su sve optimizacije koje radi prevodilac isključene te da varijablu `min` može mijenjati bilo koja dretva u svakom trenutku.

Drugi korak algoritma

Drugi kernel implementira iterativni algoritam koji gradi stablo koristeći tzv. perolake (engl. *leightweight*) lokote. Oni zaključavaju samo pokazivače na djecu u polju indeksa. Drugim riječima, lokoti se koriste samo na listovima.

Stablo se na početku sastoji samo od korijena, odnosno vršne ćelije dobivene u prethodnom koraku. Svaka dretva ubacuje po jedno tijelo tako da prolazi od korijena stabla sve do ćelije u koju će moći ubaciti tijelo. Tu ćeliju pronalazi funkcija `findInsertionPoint`. Nakon toga, funkcija `findOctant` pronalazi oktant ćelije kojem tijelo pripada. Dretva će pokušati zaključati nađeni pokazivač (indeks u polju `d_indexes`) tako da će u njega pomoću funkcije `atomicCAS` zapisati vrijednost -2. To znači da je indeks zaključan i druge dretve moraju čekati da ga dretva koja ga je zaključala oslobodi. Ako dretva uspije zaključati indeks, tada ona ubacuje novo tijelo u stablo tako što zapiše broj tijela preko -2 što ujedno i otpušta lokot, vidi sliku 3.4. Ako je na traženom indeksu u polju `d_indexes`



Slika 3.4: Zaključavanje lista ćelije sa zvijezdicom

već umetnuto tijelo, tada dretva prvo kreira novu ćeliju pomoću atomic funkcije ugrađene u funkciju `findNewCell`. Ta funkcija pronalazi sljedeće neiskorišteno mjesto u polju indeksa tako da pretražuje polje od najvišeg indeksa. U tu novu ćeliju se nakon toga ubacuje

originalno i novo tijelo. Zatim se izvršava memorijska ograda `__threadfence` koja osigurava da to novo podstablo bude vidljivo ostalim dretvama i blokovima. Tek tada se nova ćelija ubacuje u stablo oslobađajući lokot.

Svaka dretva u petlji pokušava dobiti lokot za željeni indeks u polju `d_indexes`. Ovo može preopteretiti glavnu memoriju konstantnim zahtjevima dretvi za korištenje traženog indeksa. To će usporiti dretve koje su uspjele dobiti lokot, osobito u početku kada je stablo malo.

Sve dok je bar jedna dretva u *warpu* (podsjetimo, *warp* je skupina dretvi koje se izvršavaju zajedno) uspjela dobiti lokot, divergencija dretvi će privremeno isključiti sve ostale dretve u *warpu* koje nisu uspjele dobiti lokot, sve dok uspješna dretva ne izvrši ubacivanje tijela i oslobodi svoj lokot. Ovaj proces smanjuje broj pristupa memoriji jer bi većina pokušaja dobivanja lokota bila beskorisna zbog toga što bi pronašli zaključan lokot.

Kod kernela koja gradi stablo izgleda ovako:

```
__global__ void buildOctree(double4 * bodies, int N,
 volatile int * d_indexes, int * d_pomIndexes,double4 * maxB,
  double4 * minB)
{
  int i = threadIdx.x + blockIdx.x*blockDim.x;
  if(i >= N) return;
  bool finish = false;
  int insertionPoint,oldBody;
  Cell m,M;

  m = *minB;
  M = *maxB;

  while(finish == false)
  {
    insertionPoint = findInsertionPoint(bodies[i],N,&m,
    &M,d_indexes);
    int octant = findOctant(m,M,bodies[i]);
    int child = d_indexes[insertionPoint*8+octant];
    if(child != locked)
    {
      oldBody = child;
      if(child < N &&  child ==
       atomicCAS((int*)&d_indexes[insertionPoint*8+octant],
       child,locked))
      {
        if(child == -1)
        {
          d_indexes[insertionPoint*8+octant]=i;
          finish = true;
        }
```

```
        else
        {
          int storeVar,p=0,oldOctant;
          int pomInsertionPoint = insertionPoint;
          int newCell;
          oldOctant = octant;
          while(finish == false)
          {
            newCell =
            findNewCell(pomInsertionPoint,N,d_pomIndexes);
            if(p == 0) storeVar = newCell;
            getOctantMesures(octant,&m,&M);
            int exstOctant =
            findOctant(m,M,bodies[oldBody]);
            d_indexes(newCell,exstOctant) = oldBody;
            octant = findOctant(m,M,bodies[i]);
            if(octant != exstOctant)
            {
              d_indexes(newCell,octant) = i;
              finish = true;
            }
            if(p != 0)
              d_indexes(pomInsertionPoint,child) = newCell;
            pomInsertionPoint = newCell;
            child = octant;
            p++;
          }
          d_indexes(insertionPoint,oldOctant) = storeVar;
        }
      }
    }
    __syncthreads();
  }
}
```

Da bi se spriječio slučaj u kojemu nijedna dretva u *warpu* ne bi dobila lokot, umećemo barijeru `__synchthreads` koja nije nužna za korektnost, ali se sinkroniziraju dretve u jednom bloku. Ova barijera zaustavlja *warpove* za koje postoji velika vjerojatnost da ne bi mogli dobiti lokot, a zadržavali bi dretve koje su ga dobile. Funkcija `getOctantMesures` postavlja nove konture ćelije, novi maksimum i minimum, ovisno u kojoj ćeliji se nalazi tijelo.

Kernel završava s radom nakon što sve dretve uspješno ubace dohvaćena tijela. Nakon toga prelazimo na sljedeći korak algoritma.

Treći korak algoritma

Treći kernel prolazi kroz cijeli octtree od zadnje dodane ćelije pa sve do korijena te za svaku od njih računa centar mase i sumu masa svih tijela u ćeliji.

Na početku sve ćelije imaju inicijalizirane mase i pozicije na 0, te imaju svojstvo `success` koje služi da bi se znalo je li dretva izračunala centar mase i kumulativnu masu. U ovom kernelu, svakoj dretvi pridružena je jedna ćelija. Budući da većina ćelija u octtree-u ima kao djecu isključivo tijela, one mogu odmah računati podatke za ćeliju. Ukoliko imaju ćelije za djecu, onda moraju čekati dok te ćelije ne postave zastavicu `success` na `true`.

Računanje kreće od ćelija s nižim indeksom. Tako će ćelije s većim indeksom, kojima su djeca uglavnom druge ćelije, većinom imati djecu čija je zastavica `success` postavljena na `true`. Nakon što se centar mase spremi, izvršava se *memory fence* funkcija. Tada se sprema kumulativna masa te se ažurira varijabla `success`. Memorijska ograda osigurava da promjene budu vidljive svim ostalim dretvama. Na ovaj način nisu potrebne atomic funkcije jer svojstvo ćelije `success` služi kao zastavica spremnosti.

Osim računanja centra mase i kumulativne mase računa se i broj djece za svaku ćeliju (u polju `numberOfChildren`), što će znatno ubrzati sljedeći kernel.

```
__global__ void computeCentersOfGravity( double4 * bodies, int N,
volatile double4 * cells, int * d_indexes, volatile int * success,
 volatile int * numberOfChildren)
{
  i = threadIdx.x + blockIdx.x*blockDim.x;
  int missing;
  if(i+N < N*2)
  {
    int missingChildren[8];
    for(j = 0; j < 8; ++j)
      missingChildren[j] = -2;

    missing = 0;
    cells[i] = initializeCell();
    numberOfChildren[i] = 0;
    for(j = 0; j < 8; ++j)
    {
      int child = d_indexes[(i+N)*8+j];
      if(child != -1)
      {
        if(childIsReady(child,success,N))
        {
          if(child < N)
          {
            cells[i].x += bodies[child].x * bodies[child].m;
            cells[i].y += bodies[child].y * bodies[child].m;
            cells[i].z += bodies[child].z * bodies[child].m;
```

```
            numberOfChildren[i]++;
            cells[i].m += bodies[child].m;
          }
          else
          {
            int position1 = child-N;
            cells[i].x += cells[position1].x * cells[position1].m;
            cells[i].y += cells[position1].y * cells[position1].m;
            cells[i].z += cells[position1].z * cells[position1].m;
            numberOfChildren[i] += numberOfChildren[position1];
            cells[i].m += cells[position1].m;
          }
        }
        else
        {
          int insertionPoint =
            insertPointForMissingChild(missingChildren);
          missingChildren[insertionPoint] = child;
          missing++;
        }
      }
    }
    if(missing == 0)
    {
      cells[i].x = cells[i].x/cells[i].m;
      cells[i].y = cells[i].y/cells[i].m;
      cells[i].z = cells[i].z/cells[i].m;
      __threadfence();
      success[i] = 1;
    }
    if(missing != 0)
    {
      int childIndex = 0;
      do
      {
        childIndex = nextChild(missingChildren,childIndex);
        int child = missingChildren[childIndex];
        if(childIsReady(child,success,N)
        && childIndex != -1)
        {
          missing--;
          int position1 = child-N;
          cells[i].x += cells[position1].x * cells[position1].m;
          cells[i].y += cells[position1].y * cells[position1].m;\\
          cells[i].z += cells[position1].z * cells[position1].m;
          cells[i].m += cells[position1].m;
          numberOfChildren[i] += numberOfChildren[position1];
```

```
          missingChildren[childIndex] = -2;
        }
        if(missing == 0)
        {
          cells[i].x = cells[i].x/cells[i].m;
          cells[i].y = cells[i].y/cells[i].m;
          cells[i].z = cells[i].z/cells[i].m;
          __threadfence();
          success[i] = 1;
        }
        childindex++;
        __syncthreads();
      }while(missing != 0);
    }
   }
  return;
}
```

Polje `missingChildren` sadrži indekse sve djece za koje još nije izračunat centar mase. Funkcija `initializeCell` postavlja vrijednosti jedne ćelije (masu i koordinate) na 0. Unutar prve petlje se računaju kumulativne mase, broj djece itd. za sve ćelije čija djeca su spremna (`success = true`). Ukoliko dijete nije spremno, povećava se brojač `missing` te se dijete ubacuje u polje `missingChildren` uz pomoć funkcije `insertPointForMissingChild`.

Sve ćelije koje imaju `missing != 0` ulaze u `do while` petlju gdje prolaze po polju `missingChildren` sve dok ne uspiju izračunati doprinos sve djece traženim podatcima.

Funkcija `nextChild` vraća indeks sljedećeg djeteta promatrane ćelije, a funkcija `childIsReady` provjerava je li `success` od djeteta `true`.

Nakon ovoga imamo sve potrebno da bi krenuli u računanje sila između tijela. No, da bi to učinili znatno brže prije toga ćemo izvršiti još jedan korak .

Četvrti korak algoritma

Ovaj kernel paralelno sortira tijela. Kriterij sortiranja je taj da tijela koja su prostorno blizu po mogućnosti budu u istom bloku. To se radi tako da se prebrojava broj tijela *"lijevo"* od odabranog tijela u stablu.

Definirajmo relaciju "biti lijevo". Neka je T stablo tako da je $T = (G, E)$ gdje su G čvorovi, a E bridovi stabla. Kažemo da je čvor a *"lijevo"* od čvora b ako su a i b listovi te ako u preorder obilasku stabla čvor a obiđemo prije čvora b.

Označimo sa L broj tijela koja su lijevo od tijela i. Nakon sortiranja, tijelo i će u polju `bodyIndexes` biti spremljeno na index L.

Sortiranje se odvija u potpunosti paralelno jer nema nikakvih kritičnih operacija niti su potrebni ikakvi lokoti: svaka dretva može odrediti novu poziciju svojeg tijela u potpunosti

neovisno o drugim dretvama. Ovaj kernel je od presudne važnosti za brzinu izvođenja petog koraka algoritma.

```
__global__ void sortBodyIndexes(int N,double4 * bodies,
double4 * cells, int * bodyIndexes,
int * d_indexes,volatile int * numberOfChildren )
{
  int id = blockIdx.x*blockDim.x + threadIdx.x;
  if(id >= N) return;

  int cellIndex = N*2-1;
  int index = 0,j;
  Cell m,M;
  m = minBody;
  M = maxBody;
  while(1)
  {
    if(cellIndex < N) break;
    int octant = findOctant(m,M,bodies[id]);
    for(j = 0; j < octant;++j)
    {
      int tempInd = d_indexes(cellIndex,j);
      if(tempInd < N)
      {
        if(tempInd != -1)
        {
          index++;
        }
      }
      else
      {
        index += numberOfChildren[tempInd-N];
      }
    }
    cellIndex = d_indexes(cellIndex,octant);
    getOctantMesures(octant,&m,&M);
  }
  bodyIndexes[index] = id;
}
```

Svaka dretva počinje računati od korijena stabla. Nakon toga ulazi u `while` petlju iz koje će izaći tek izračuna broj tijela koja su "lijevo" u stablu od tijela i . U svakom koraku petlje dretva pronalazi oktant za promatranu ćeliju kojem pripada tijelo. Ona za pronađeni oktant prebrojava sve brojeve tijela za ćelije koje imaju indeks oktanta manji od pronađenog za traženo tijelo (to znači da su lijevo od traženog tijela). Ovo se događa na svakoj razini stabla. Nakon što sva tijela pronađu svoj indeks funkcija je gotova. Složenost ovog kernela

je $O(\log n)$ što je odlično.

Peti korak algoritma

Peti korak algoritama vrši središnju zadaću računanja ubrzanja za svako tijelo. On troši najveći dio vremena i najvažniji je za optimizaciju. Za svaki blok prolazi kroz stablo kvazi preorderom obilaskom: svi listovi neće nužno biti obiđeni. Za svaki čvor u obilasku se računa udaljenost od svih tijela u bloku. Nakon toga se *thread-voting* funkcijom `__all`, koja se vrlo brzo izračuna, odlučuje hoće li se računati sila u odnosu na ćeliju ili će se računati u odnosu na svako dijete od ćelije.

Kod petog kernela izgleda ovako:

```
__global__ void computeForcesBetweenBodies(double4 * bodies, int
 N, double4 * cells, int * bodyIndexes,int * d_indexes,
 Stack * stacks, double3 * result)
{
  int id = threadIdx.x + blockIdx.x*blockDim.x;
  if(id >= N) return;
  __shared__ double4 s;
  __shared__ int nextNode;

  double3 a = getInitAcc();
  double4 body = bodies[bodyIndexes[id]];
  if(threadIdx.x == 0)
  {
    Stack_Init(stacks+blockIdx.x);
    initialiseStack(stacks+blockIdx.x,d_indexes,N*2-1);
    __threadfence_block();
  }
  int depth = 0;
  while(depth >=0)
  {
    while(!Empty(&stacks[blockIdx.x]))
    {
      if(threadIdx.x == 0)
      {
        nextNode = Stack_Top(stacks+blockIdx.x);
        Stack_Pop(stacks+blockIdx.x);
        if(nextNode >= N)
        s = cells[nextNode-N];
        else s = bodies[nextNode];
      }
      __threadfence_block();
      if(nextNode)
      {
        if(nextNode < N || __all(distance >= cutoff) )
```

```
          computeForces(body,s,&a);
        else
        {
          depth++;
          if(threadIdx.x == 0)
          {
            initialiseStack(stacks+blockIdx.x,d_indexes,nextNode);
          }
          __threadfence_block();
        }
      }
      else
      {
        depth = max(0,depth-1);
      }
    }
    depth--;
  }
  a = vecMulScal(a,G);

  result[bodyIndexes[id]] = a;
}
```

Na početku kernela u zajedničkoj memoriji se definira varijabla `s` u koju ćemo spremati podatke o ćeliji ili tijelu koje će se skidati s vrha stoga i varijabla `nextNode` koja označava indeks ćelije u polju indeksa koja je skinuta s vrha stoga.

Zatim se inicijalizira ubrzanje funkcijom `getInitAcc` te svaka dretva dohvati jedno tijelo pomoću sortiranih indeksa iz glavne memorije u polju `bodyIndexes`. Na taj način dretve u bloku računaju udaljenost za prostorno bliska tijela. Stoga je mnogo veća vjerojatnost da će *thread-voting* funkcija `__all` biti usuglašena za većinu čvorova. Poslije toga prva dretva u bloku inicijalizira stog puneći ga djecom korijena stabla počevši od zadnjeg djeteta.

Sljedeći korak je kvazi preorder obilazak stabla. On se odvija tako da najprije prva dretva u bloku skida element s vrha stoga te indeks pohranjuje u `nextNode`, a podatke u `s`. Potom svima u bloku objavljuje novo stanje u dvjema varijablama.

Ukoliko je tijelo s vrha stoga udaljeno više od zadane granice (tzv. varijable `cutoff`) od svih tijela u bloku, tada pokrećemo `computeForces` za tu ćeliju i sva tijela u bloku. Ona ažurira ubrzanje. No, ukoliko postoji jedno tijelo koje je za manje od zadane granice udaljeno od ćelije ubacuju se sva djeca od te ćelije u stog. To radi funkcija `initializeStack`. Ovo vrijedi samo kad se radi o ćeliji.

Kernel završava kad se za sve blokove izvrši obilazak po stablu. Nakon toga preostaje još samo ažurirati podatke.

Šesti korak algoritma

Posljednji kernel bilježi i sprema rezultate prethodnih kernela. On se odvija u potpunosti paralelno. U njemu svaka od n dretvi uzima jedno tijelo, njegovo ubrzanje i početnu brzinu te na osnovu formule (2.5) izračunava novu poziciju tijela i brzinu po formuli (2.6).

3.4 Usporedba performansi algoritama

Nakon implementacije, proveli smo testiranje performansi svih opisanih algoritama. Računalo na kojem su testirane performanse je server fermi.math.hr sa sljedećim komponentama:

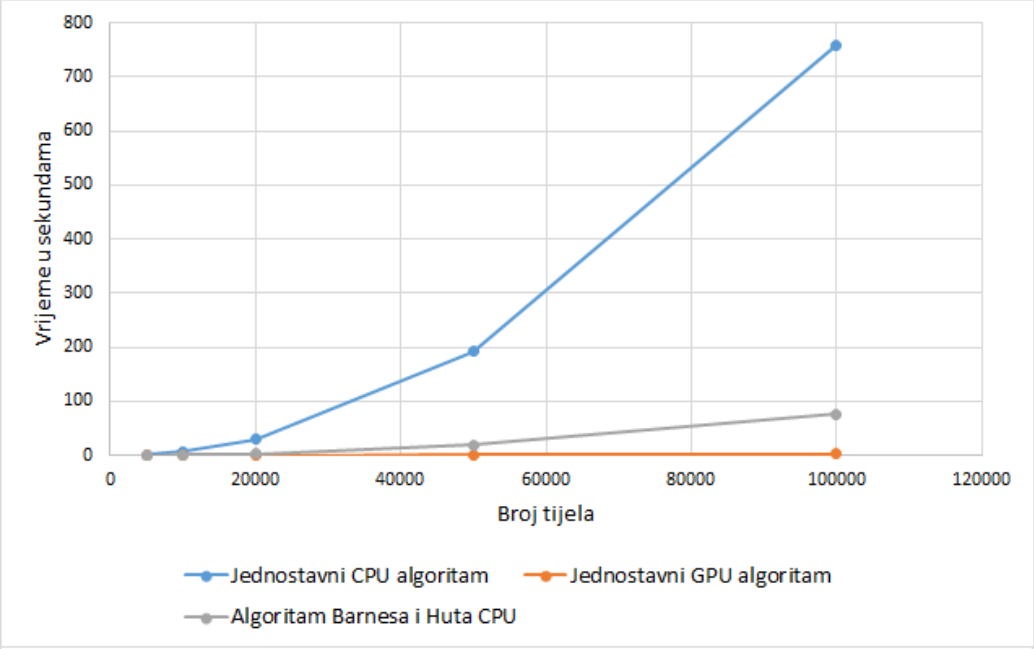
- 2x Intel(R) Xeon(R) CPU E5620 @ 2.40GHz (ukupno 8 CPU jezgri)
- 24 GB RAM
- Nvidia Tesla S2050 Computing System (u testovima koristimo samo jedan Fermi GPU sa 3GB memorije)
- Nvidia CUDA 7.5

Svi algoritmi implementirani su u C-u, te im je korektnost potvrđena usporedbom rezultata s najjednostavnijim CPU algoritmom opisanim na kraju drugog poglavlja. Najprije ćemo usporediti performanse algoritama implementiranih na CPU s najjednostavnijim algoritmom implementiranim na GPU, pogledati sliku 3.5. Iz slike zaključujemo da najjednostavniji algoritam implementiran na GPU ima mnogo bolje performanse nego algoritmi implementirani na CPU. Uočimo da algoritam Barnesa i Huta implementiran na CPU ima uvjerljivo bolje performanse nego obični algoritam implementiran na CPU.

Nakon toga ćemo usporediti performanse algoritama implementiranih na GPU, pogledati sliku 3.6. Primjetimo da algoritam Barnesa i Huta ima najbolje performanse već za 100000 tijela. Jednostavni $O(n^2)$ algoritmi su bolji za probleme s manjim brojem tijela. Također, uočimo da jednostavni algoritam koji koristi zajedničku memoriju daje bolje performanse od algoritma koji ne koristi zajedničku memoriju. U tablici 3.1 možemo vidjeti vremena izvršavanja svih dosad implementiranih algoritama za 100000 i 500000 tijela.

U tablici 3.2 možemo vidjeti vremena izvršavanja po koracima za algoritam Barnesa i Huta. Uočimo kako kerneli 2 i 5 zahtijevaju najviše vremena za izvršavanje.

Drugi kernel se izvršava dugo jer implementacija koristi lokote što bitno usporava kernel. Za peti kernel smo napomenuli da troši dosta procesorskog vremena. Ovdje on traje nešto kraće od drugog kernela jer njegova složenost ovisi o veličini varijable `cutoff`. Ukoliko je varijabla `cutoff` jako malena, tada će kernel prilikom računanja obilaziti dijelove stabla koji su blizu korijena što će uvećati grešku, ali ubrzati računanje. Ukoliko



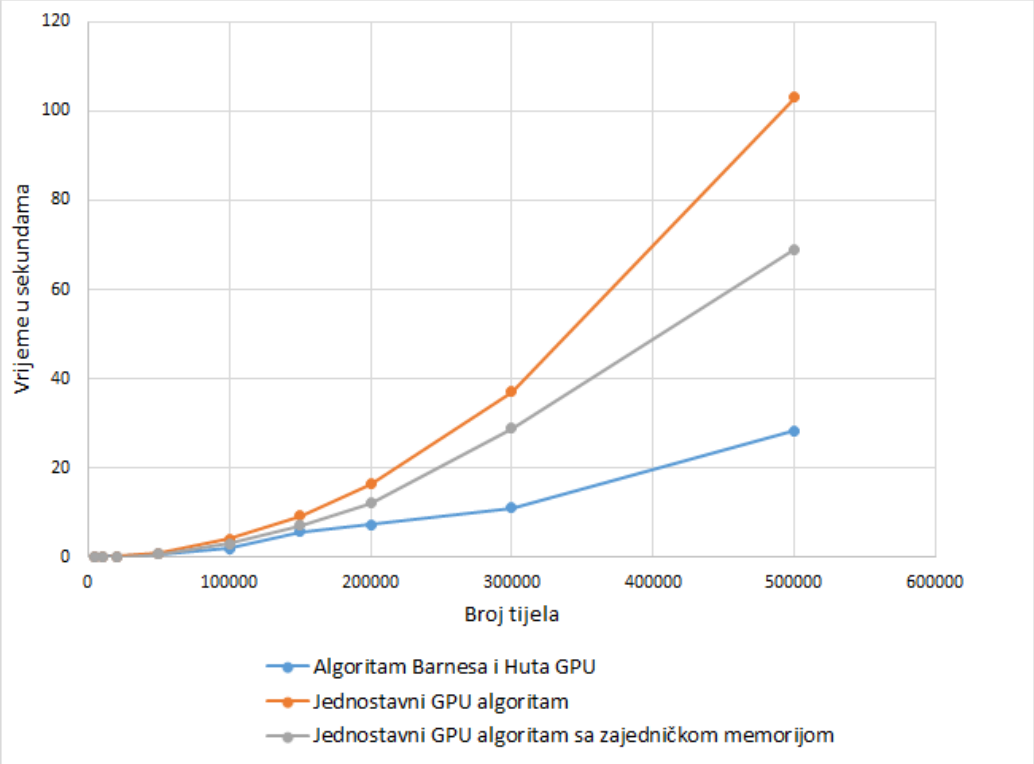
Slika 3.5: Usporedba CPU i GPU algoritama za rješavanje problema n tijela

Tablica 3.1: Vrijeme izvođenja algoritama za rješavanje problema n tijela mjereno u sekundama

Ime algoritma/broj tijela	100000	500000
$O(n^2)$ algoritam na CPU	758,57	> 1000
Algoritam Barnesa i Huta na CPU	77,15	> 1000
$O(n^2)$ algoritam na GPU	4,13	103,09
$O(n^2)$ algoritam na GPU korištenjem zaj. mem.	3,1	69,09
Algoritam Barnesa i Huta na GPU	2	28,38

je obrnut slučaj, može se dogoditi da algoritam bude spor kao i jednostavni algoritmi za računanje n tijela. Moramo paziti da varijabla `cutoff` bude takva da greška bude dovoljno mala i da vrijeme izvršavanja ne bude preveliko.

Rezultati su pokazali da algoritam Barnesa i Huta ima najbolje performanse od svih gore navedenih algoritama. Zbog toga možemo zaključiti da je implementacija algoritma Barnesa i Huta svrsishodna.



Slika 3.6: Usporedba performansi algoritama za rješavanje problema *n* tijela implementiranih na GPU

Tablica 3.2: Vremena izvođenja algoritma Barnesa i Huta po koracima za 150000 tijela

	Kernel 1	Kernel 2	Kernel 3	Kernel 4	Kernel 5	Kernel 6
Vrijeme u ms	20	3400	40	10	2355	5

Bibliografija

[1] *CUDA C Programming Guide*, http://docs.nvidia.com/cuda/.

[2] Hubert Nguyen: *GPU Gems 3*, Addison-Wesley Professional, 2007.

[3] Martin Burtscher, Keshav Pingali: *An Efficient CUDA Implementation of the Tree-Based Barnes Hut n-Body Algorithm*, http://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/nbody-problem.pdf.

[4] Wen-mei W. Hwu: *GPU Computing Gems Emerald Edition*, Morgan Kaufmann, 2011.

Sažetak

Ovaj rad se bavi implementacijom algoritama za rješavanje problema *n* tijela i stavlja naglasak na njihovu implementaciju u NVIDIA-inom programskom okviru CUDA. CUDA je tehnologija za implementaciju programa opće namjene na grafičkim karticama, te omogućava efikasno iskorištavanje njihovih masovno paralelnih procesora.

U radu se opisuju tehnike za efikasniju implementaciju kodova u tehnologiji CUDA općenito, korištenjem specifičnosti programskog jezika i hardverske arhitekture. Kroz implementaciju se naglašava korištenje optimalnih metoda koje će ubrzati izvršavanje koda.

Također se pokazuju rezultati izvršavanja algoritama, te opisuju njihove prednosti i nedostatci.

Summary

This work deals with implementation of algorithms for solving the n body problem. It puts accent on implementation in NVIDIA's programming framework CUDA. This technology is used for implementing general purpose programs on graphics processing units, and enables efficient usage of their massively parallel processors.

This works describes tehniques for more efficient implementations of CUDA programs in general, using the advantageous properties of the programming model and underlying hardware architecture.

Special emphasis is put on optimal methods that speed up the computation. This work also shows performance results for the implemented algoritms, comparing their strenghts and weaknesses.

Životopis

Mario Pavlović je rođen 16. siječnja 1993. godine u Kaknju u BiH. Završio je srednju školu KŠC "Don Bosco" u Žepču, BiH. Za vrijeme srednje škole sudjeluje na natjecanjima iz matematike, povijesti i geografije. Godine 2011. upisao je Preddiplomski sveučilišni studij matematike na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu.

Godine 2014. je na istom odsjeku i fakultetu upisao Diplomski studij Računarstvo i matematika. Od svibnja 2016. radi u Combis d.o.o. kao dizajner baze podataka. Osim matematike i računarstva, izrazito ga zanima književnost, povijest i geografija.