Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jani Bevk

Simulacije fluidov z metodo Monte Carlo na grafičnih procesnih enotah

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Uroš Lotrič
Somentor: prof. dr. Jurij Reščič

Ljubljana, 2018

Zahvala

Zahvaljujem se mentorju izr. prof. dr. Urošu Lotriču in somentorju prof. dr. Juriju Reščiču za vso strokovno pomoč, nasvete in usmerjanje pri izdelavi magistrske naloge. Iskreno se zahvaljujem tudi družini in prijateljem za podporo in pomoč skozi vsa leta študija.

Jani Bevk, 2018

"Don't try to be like Jackie. There is only one Jackie. Study computers instead."

— Jackie Chan

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
GPU	Graphics processing unit	Grafična procesna enota
CPU	Central processing unit	Centralna procesna enota
GPGPU	General-purpose computing on graphics processing units	Splošno-namensko računanje na grafičnih procesnih enotah
CUDA	Compute Unified Device Architecture	Platforma za vzporedno programiranje na grafičnih procesnih enotah
MCMC	Markov Chain Monte Carlo	Monte Carlo Markovske verige

Povzetek

Naslov: Simulacije fluidov z metodo Monte Carlo na grafičnih procesnih enotah

Molekulske simulacije so zbirka metod za izvajanje računalniških eksperimentov na modelih molekulskih sistemov. Služijo kot most med teoretičnimi napovedmi in eksperimentalnimi rezultati. S kompleksnostjo in velikostjo uporabljenega simulacijskega modela raste potreba po večji računski moči. Za splošno-namensko računanje se zaradi njihovega ugodnega razmerja med računsko zmogljivostjo in porabo energije ter ceno čedalje bolj uporabljajo grafične procesne enote. V tem delu smo se osredotočili na metodo Monte Carlo za simulacijo fluidov. Uspešno smo jo prilagodili za izvajanje na grafičnih procesnih enotah. Uporabili smo platformo CUDA in princip energijske dekompozicije. Tekom simulacije računamo energijo sistema in radialno porazdelitveno funkcijo, interakcije med atomi pa modeliramo s potencialom Lennard-Jones. Podprli smo tudi simulacijo molekul, sestavljenih iz več različnih atomov. Pohitritve vzporedne izvedbe smo analizirali v primerjavi z zaporedno izvedbo, ki se izvaja na centralni procesni enoti. Pri uporabi dvojne natančnosti za predstavitev števil v plavajoči vejici smo dosegli do 172-kratne pohitritve, pri uporabi enojne natančnosti pa skoraj 640-kratne pohitritve.

Ključne besede

simulacija Monte Carlo, molekulske simulacije, statistična termodinamika, grafične procesne enote, vzporedno računanje, CUDA

Abstract

Title: Simulations of fluids using the Monte Carlo method on graphic processing units

Molecular simulations are a set of methods for performing computer experiments on models of molecular systems. They act as a bridge between theoretical predictions and experimental results. The need for greater computational power grows with the complexity and size of the simulation model. Graphics processing units are increasingly being used for general-purpose computing due to their favourable ratio of computing capacity to power consumption and price. In our work, we focus on the Monte Carlo method for simulation of fluids. We have successfully adapted it for execution on graphics processing units using the CUDA platform and the energy decomposition principle. Throughout the simulation the system energy and radial distribution function are calculated. Inter-atom interactions are modelled using the Lennard-Jones potential. We have also implemented support for molecules composed of several different atoms. We have analysed the performance of our parallel implementation in comparison to a sequential implementation. We have achieved up to 172-fold speedups when using double precision for floating-point number representation and almost up to 640-fold speedups when using single precision.

Keywords

Monte Carlo simulation, molecular simulation, statistical thermodynamics, graphics processing units, parallel computing, CUDA

Poglavje 1

Uvod

Računalniške simulacije so postale nepogrešljivo orodje pri modeliranju naravnih sistemov. Statistična termodinamika omogoča, da na podlagi lastnosti atomov in/ali molekul, ki sestavljajo opazovani sistem, ter njihovih medsebojnih interakcij, izračunamo merljive fizikalne količine. Za modelni sistem lahko te količine izračunamo s simulacijami. Računalniške simulacije molekulskih sistemov igrajo čedalje bolj pomembno vlogo pri raziskavah s področja kemije, fizike in biologije, širijo pa se tudi na druga področja [34].

S kompleksnostjo uporabljenega simulacijskega modela in velikostjo sistema raste potreba po večji računski moči. Računsko moč običajno povezujemo s frekvenco centralnih procesnih enot, ki pa v zadnjih letih stagnira. Narašča le število jeder, kar omogoča sočasno izvajanje več ukazov. Programska oprema mora biti ustrezno prilagojena, da lahko izkoristi večjedrnost centralnih procesnih enot. Nekatera orodja za molekulske simulacije to seveda zmorejo, na primer ms2 [9] in MOLSIM [29] s pomočjo MPI ter Cassandra [30] s pomočjo OpenMP. Za računanje pa postajajo čedalje bolj priljubljene grafične procesne enote, ki na več sto procesnih jedrih izvajajo več deset tisoč vzporednih niti [25]. Temu botrujeta predvsem njihova cenovna dostopnost in razmerje med računsko zmogljivostjo in porabo energije.

Molekulske simulacije se običajno delijo na simulacije molekulske dinamike in simulacije Monte Carlo [10]. Veliko simulacij molekulske dinamike

je že prilagojenih za izvajanje na grafičnih procesnih enotah. Prilagajanje simulacij Monte Carlo pa se izkaže za precej težji proces [20]. Razlog je stohastična in zaporedna narava same metode, saj je med simulacijo molekule potrebno naključno premikati eno za drugo.

V tem delu smo se osredotočili na paralelizacijo in prilagoditev metode Monte Carlo za simulacijo fluidov, da se bo ta lahko izvajala na grafičnih procesnih enotah. Uporabili smo Metropolisov algoritem in z njim tekom simulacije računali energijo sistema in radialno porazdelitveno funkcijo. Za modeliranje interakcij med atomi smo uporabili potencial Lennard-Jones.

1.1 Pregled sorodnih del

Kljub temu da metoda Monte Carlo ni najbolj prijazna za paralelizacijo, obstaja nekaj izvedb, ki se izvajajo na grafičnih procesnih enotah.

Hailat *et al.* so s platformo CUDA uspešno paralelizirali metodo Monte Carlo za simulacijo delcev v Lennard-Jonesovem potencialu [12, 23]. S principom energijske dekompozicije so računanje interakcij med delci porazdelili na vse razpoložljive niti. Za izogibanje razhajanja niti (angl. *thread divergence*) in enakomerno obremenjenost niti so uporabili poseben preslikovalni algoritem. Ta določa, za kateri par delcev je odgovorna vsaka nit. Dosegli so do 30-kratne pohitritve.

Liang *et al.* so simulacijo Coulombovih interakcij med delci paralelizirali na podoben način [20]. Poleg računanja interakcij med delci so med niti porazdelili tudi premikanje delcev in sprejemanje/zavračanje teh premikov. S tem se v enem klicu ščepca izvedejo poskusi premikov vseh delcev v sistemu. Potrebna je bila kompleksna medsebojna sinhronizacija niti in blokov niti. Dosegli so kar 440-kratno pohitritev.

Anderson *et al.* pa so za paralelizacijo simulacije trdih delcev uporabili princip domenske dekompozicije [2, 3]. Prostor so razdelili na šahovnico, tako, da se lahko več poskusov premikov delcev izvede hkrati. Razdalje med delci v nesosednjih celicah so namreč dovolj velike, da delci nimajo

vpliva drug na drugega. Podprli so tudi možnost izvajanja na več grafičnih procesnih enotah hkrati s pomočjo ogrodja MPI.

Vse omenjene simulacije temeljijo na Markovskih procesih. Za izvajanje na grafičnih procesnih enotah so bile prilagojene tudi nekatere druge metode Monte Carlo, na primer inverzna metoda Monte Carlo [35].

1.2 Prispevki

Glavni prispevek magistrske naloge je vzporedna izvedba metode Monte Carlo za simulacije fluidov, ki se izvaja na grafičnih procesnih enotah. Obstoječe vzporedne izvedbe v pregledani literaturi simulirajo enoatomne delce, mi pa smo podprli tudi simulacijo molekul. Pri tem je potrebno premikati več atomov naenkrat in poleg translacije upoštevati tudi rotacijo molekul. Ker je poudarek paralelizacije predvsem na hitrosti izvajanja, smo analizirali pohitritve v primerjavi z zaporedno izvedbo metode. Grafične procesne enote s števili v formatu dvojne natančnosti računajo počasneje kot s števili v formatu enojne natančnosti [32]. Zato smo preučili tudi vpliv enojne in dvojne natančnosti na natančnost rezultatov in hitrost izvajanja.

1.3 Metodologija

Da smo lahko ovrednotili pohitritve, smo metodo Monte Carlo za simulacije fluidov implementirali zaporedno, v programskem jeziku C++. Metodo smo nato paralelizirali, optimizirali in prilagodili za izvajanje na grafičnih procesnih enotah. Pri tem smo uporabili platformo CUDA. Pravilnost računanja energije sistema in radialnih porazdelitvenih funkcij smo preverili z referenčnim programom MOLSIM [29]. Učinkovitost paralelizacije smo ovrednotili z merjenjem in primerjanjem časov izvajanja zaporedne in vzporedne izvedbe pri različnih vrednostih vhodnih parametrov in velikostih problema.

1.4 Struktura naloge

Magistrsko delo je sestavljeno iz šestih poglavij. V poglavju 2 predstavimo molekulske simulacije in metodo Monte Carlo za simulacijo fluidov, ki je predmet te naloge. V poglavju 3 opišemo zaporedno izvedbo metode Monte Carlo, ki se izvaja na centralni procesni enoti. Osredotočili smo se na sam algoritem in njegove vhodne ter izhodne podatke. V poglavju 4 na kratko predstavimo splošno-namensko računanje na grafičnih procesnih enotah in platformo CUDA, ter opišemo našo prilagoditev te metode za izvajanje na grafičnih procesnih enotah. Najprej opišemo osnovno vzporedno izvedbo, nato pa še optimizacijo le-te. V poglavju 5 preverimo pravilnost delovanja naših izvedb in ovrednotimo dosežene pohitritve. Sklepne ugotovitve predstavimo v poglavju 6.

Poglavje 2

Molekulske simulacije

Molekulske simulacije so zbirka metod za izvajanje računalniških eksperimentov na modelih molekulskih sistemov. Modeli so lahko poljubno kompleksni, vendar se moramo zavedati kompromisa med natančnostjo in preprostostjo modela. Za najmanjšo enoto modela pogosto vzamemo atom, možno pa je tudi modeliranje elektronov v atomu.

Vsi problemi niso natančno rešljivi, z računalniškimi simulacijami pa lahko pridemo do numeričnih rešitev. Računalniške simulacije so pomembno orodje pri izvajanju teoretičnih raziskav, saj jih lahko uporabimo za testiranje teorij. Priročne so predvsem, ko izvedba pravih eksperimentov ni mogoča zaradi tehničnih omejitev, časa ali financ. Rezultate računalniških simulacij lahko seveda primerjamo tudi z rezultati pravih eksperimentov. S tem preverimo pravilnost uporabljenega modela. Simulacije so torej nekakšen most med teoretičnimi napovedmi, modelom sistema in eksperimentalnimi rezultati.

Molekulske simulacije se običajno delijo na simulacije molekulske dinamike in simulacije Monte Carlo [10]. Simulacije molekulske dinamike simulirajo gibanje in interakcije molekul skozi čas. Njihovo premikanje je določeno s silami, ki nanje delujejo, in sistemom Newtonovih enačb gibanja. Simulacije Monte Carlo pa so stohastični procesi. Temeljijo na naključnih poskusih premikanja molekul in računanju sprememb intermolekularne energije. V

tem delu smo se osredotočili na simulacije Monte Carlo.

2.1 Metoda Monte Carlo

Metode Monte Carlo so družina računskih algoritmov, ki za reševanje problemov uporabljajo naključnost. Kljub njihovi stohastični naravi so te metode primerne tudi za reševanje povsem deterministčnih problemov. Zanašajo se na ponavljajoče naključno vzorčenje iz določene verjetnostne porazdelitve in veliko število izračunov. S pomočjo zakona o velikih številih in drugih statističnih metod sklepanja to omogoča napovedovanje obnašanja kompleksnih matematičnih sistemov.

Problemi, ki jih rešujemo z metodami Monte Carlo, večinoma spadajo v tri kategorije [19]:

- *Vzorčenje*: Z opazovanjem številnih realizacij naključnega objekta zbiramo informacije o njem. Primer tega je simulacijsko modeliranje, kjer naključni proces posnema obnašanje nekega resničnega sistema.

- *Ocenjevanje*: Ocenjujemo določene numerične spremenljivke, povezane s simulacijskim modelom. Na primer ocenjevanje vrednosti integrala s pričakovano vrednostjo slučajne spremenljivke.

- *Optimizacija*: Metode Monte Carlo so zelo močno orodje za iskanje ekstremov realnih funkcij.

Metode Monte Carlo se dandanes uporabljajo na najrazličnejših področjih. Med drugim se uporabljajo na področjih inženirstva (občutljivostne in druge analize), statistike, ekonomije in financ (ocenjevanje tveganj in investicij), umetne inteligence (drevesno preiskovanje Monte Carlo), računalniške grafike (angl. *path tracing*), računske biologije, računske fizike, ter seveda na področju računske kemije. Uporabne so zlasti v primerih, kjer analitične ali numerične rešitve ne obstajajo ali pa so pretežke za izvedbo.

Metoda Monte Carlo ni ena sama, ampak obstaja več različic, prilagojenih specifičnim problemom. Številne sledijo naslednjemu vzorcu [14]:

1. definicija modela in domene možnih vhodov,
2. večkratno vzorčenje iz verjetnostne porazdelitve domene,
3. izračun želene statistike na podlagi vzorcev.

Ena od različic metode Monte Carlo, ki se uporablja v molekulskih simulacijah, je Metropolisov algoritem.

2.1.1 Metropolisov algoritem

Metropolisov algoritem, razvit leta 1953, je metoda Monte Carlo, ki temelji na vzorčenju [22]. Uporablja se za generiranje stanj sistema glede na Boltzmannovo porazdelitev. Zaporedje stanj generira s pomočjo Markovskih verig, zato je novo stanje sistema vedno odvisno samo od trenutnega stanja. Algoritem namreč spada v razred metod, imenovanih Monte Carlo Markovske verige (angl. *Markov Chain Monte Carlo methods, MCMC*) [11].

Algoritem deluje s ponavljanjem iteracij, kjer v vsaki iteraciji naključno generiramo novo stanje sistema. To stanje z neko verjetnostjo bodisi sprejmemo bodisi zavrnemo. Več kot je iteracij, bolj se porazdelitev stanj približuje želeni porazdelitvi.

Algoritem Metropolis-Hastings je izpeljan iz Metropolisovega algoritma in je bolj splošen [15]. Omogoča generiranje stanj glede na katerokoli verjetnostno porazdelitev. Edini pogoj je, da se lahko izračuna vrednost neke funkcije, ki je sorazmerna gostoti porazdelitve. Točne gostote porazdelitve ni potrebno izračunati.

Z Metropolisovim algoritmom lahko simuliramo sistem molekul tako, da generiramo naključne premike molekul, ki jih bodisi sprejmemo bodisi zavrnemo. Pri tej simulaciji najmanjšo enoto Markovske verige, kjer poskušamo premakniti eno molekulo, imenujemo *korak* Monte Carlo. *Cikel* Monte Carlo

sestoji iz N korakov, kjer je N število vseh molekul v simuliranem sistemu. Vsak korak je sestavljen iz treh osnovnih *faz* [10, 20]:

1. **Izbira** (angl. *selection*):
   Izberemo molekulo i s konfiguracijo κ_i, ki jo bomo poskusili premakniti v tem koraku. Oznaka κ_i predstavlja lokacijo in orientacijo molekule.

2. **Poskus premika** (angl. *trial*):
   Generiramo naključen premik molekule i, da dobimo njeno novo konfiguracijo, $\kappa_i' = \kappa_i + \Delta$. Nato po enačbi

   $$\Delta E = E(\kappa_i') - E(\kappa_i) \tag{2.1}$$

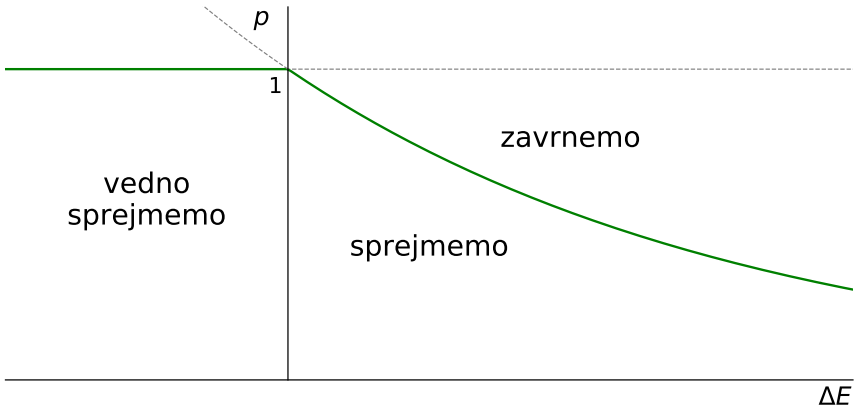
   izračunamo spremembo energije celotnega sistema, ki bi jo povzročil ta premik. Funkcija $E(\kappa)$ izračuna energijo, ki jo prispeva molekula s konfiguracijo κ.

3. **Odločitev** (angl. *acceptance/rejection*):
   Če bi se energija celotnega sistema zmanjšala, potem poskus premika vedno sprejmemo. V nasprotnem primeru pa poskus premika sprejmemo z verjetnostjo, ki jo določa Boltzmannova porazdelitev. Verjetnost sprejetja poskusa premika je torej definirana z enačbo

   $$p = \begin{cases} 1, & \Delta E \leq 0 \\ e^{-\beta \Delta E}, & \Delta E > 0 \end{cases} . \tag{2.2}$$

   Izraz $e^{-\beta \Delta E}$ je *Boltzmannov faktor*, kjer je $\beta = (k_\mathrm{B}\, T)^{-1}$, k_B Boltzmannova konstanta, T pa temperatura sistema. Če poskus premika sprejmemo, potem dejansko premaknemo molekulo i in posodobimo njeno konfiguracijo, $\kappa_i \leftarrow \kappa_i'$. Če poskus premika zavrnemo, potem njena konfiguracija κ_i ostane nespremenjena in jo znova upoštevamo, konfiguracijo κ_i' pa zavržemo. Shemo odločanja lahko vidimo na sliki 2.1.

Fazo izbire lahko izvedemo na enega od dveh načinov, bodisi naključno bodisi zaporedno. Ker pri naključnem izbiranju molekule izbiramo povsem



Slika 2.1: Graf verjetnosti sprejetja poskusa premika in shematski prikaz odločitve. Povzeto po [37].

naključno, je v enem ciklu vsaka molekula izbrana *v povprečju* enkrat. Pri zaporednem izbiranju pa molekule označimo in jih izbiramo eno za drugo, zato je znotraj enega cikla vsaka molekula izbrana *točno* enkrat.

V fazi poskusa premika novo konfiguracijo molekule sicer generiramo naključno, vendar na podlagi njene trenutne konfiguracije. Nova konfiguracija molekule je zato vedno v bližini trenutne. Gre za naključni sprehod (angl. *random walk*), s katerim gradimo Markovsko verigo. Ker algoritem vedno premika samo eno molekulo naenkrat, je namreč novo stanje celotnega sistema odvisno le od trenutnega stanja sistema.

Sistem vedno teži k stanju z minimalno energijo. Zato je v fazi odločitve sprejemanje in zavračanje poskusov premikov osredotočeno na zniževanje energije celotnega sistema. Včasih pa je naključno sprejet tudi premik, ki energijo sistema zviša. To je ključna lastnost Metropolisovega algoritma. Pri preiskovanju stanj sistema algoritem hkrati išče premike, ki stanje najbolj izboljšajo, in se izogiba lokalnim ekstremom. Verjetnost sprejetja neugodnega stanja je izpeljana iz principa podrobnega ravnotežja (angl. *detailed balance*) [10]. Ta določa pogoj, da sistem ostane v ravnovesju, ko ga doseže. Za sistem v ravnovesju mora biti premik iz trenutnega v novo stanje enako verjeten kot premik v obratno smer. Pri naključnem izbiranju sta verjetnosti

premikov enaki, saj je verjetnost izbire neke molekule vedno enaka ne glede na prejšnjo izbiro. To pa ne drži pri zaporednem izbiranju, saj je verjetnost izbire iste molekule kot v prejšnjem koraku vedno enaka nič. Izkaže se, da je pogoj podrobnega ravnotežja zadosten, ni pa potreben [21]. Tudi zaporedno izbiranje privede do veljavne simulacije Monte Carlo.

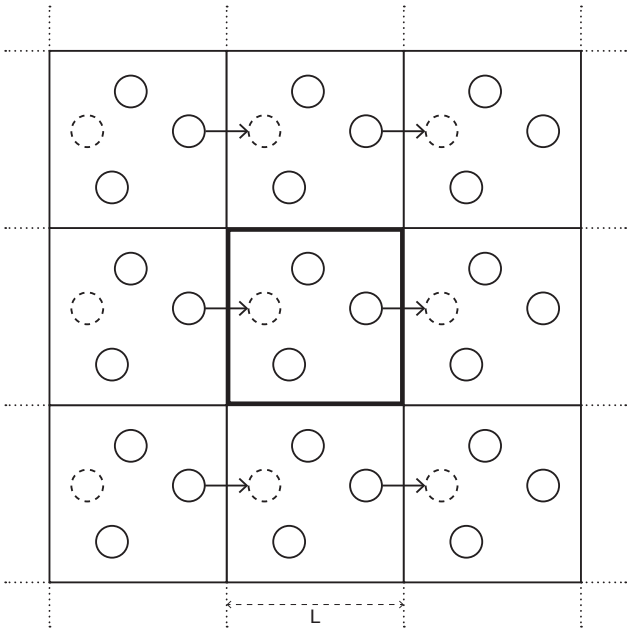
Namen molekulskih simulacij je izračunati ravnovesne lastnosti sistema molekul. Cikle simulacije ponavljamo, dokler ne dosežemo ravnovesnega stanja in želene natančnosti. Ko je sistem v ravnovesju, je verjetnostna porazdelitev enaka Boltzmannovi porazdelitvi.

2.2 Periodični robni pogoji

Z molekulskimi simulacijami običajno želimo pridobiti informacije o lastnostih makroskopskih vzorcev. Žal pa smo v računalniških simulacijah omejeni s hitrostjo izvajanja programa in velikostjo pomnilnika. Zato se omejimo na manjše število atomov N_a, ki jih zapremo v nek omejen prostor. Najenostavnejši primer takšnega prostora v treh dimenzijah je kockasta škatla z dolžino robu L, iz katere atomi ne morejo pobegniti.

Pomanjkljivost takšnih fiksnih robnih pogojev je pojav površinskih učinkov. Na atome, ki so bližje površini škatle, delujejo drugačne sile kot na večino ostalih atomov v notranjosti. Problem je najbolj izrazit pri manjših simulacijah, saj je delež atomov na površini sorazmeren z $N^{-1/3}$ [10].

Tej težavi se lahko izognemo tako, da sistem navidezno povečamo z uporabo periodičnih robnih pogojev (angl. *periodic boundary conditions*). Simulacijsko škatlo neskončnokrat kloniramo in razporedimo po prostoru v neskončno mrežo, kot je prikazano na sliki 2.2. Ko se atom v prvotni škatli premakne, se istočasno na enak način premaknejo tudi vse slike tega atoma v ostalih škatlah. Če atom izstopi iz škatle, potem vanjo na nasprotni strani vstopi ena od slik atoma. Škatle sedaj nimajo več fizičnih površin, zato problema površinskih učinkov ni več.



Slika 2.2: Shema periodičnih robnih pogojev v dveh dimenzijah. V treh dimenzijah so škatle po prostoru razporejene tudi po tretji dimenziji.

Pri simulacijah moramo običajno izračunati energijo celotnega sistema. Da izračunamo prispevek energije atoma i, moramo upoštevati tako njegove interakcije z vsemi preostalimi $N_a - 1$ atomi kot tudi njegove interakcije z vsemi slikami atomov. To v praksi ni možno zaradi neskončnega števila slik atomov. Z uporabo kriterija minimalne slike (angl. *minimum image criterion*) lahko izračunamo približek. Upoštevamo le interakcije med atomom i in najbližjimi slikami preostalih $N_a - 1$ atomov, interakcije s preostalimi slikami pa zanemarimo.

V simulaciji spremljamo lokacije N_a atomov v prvotni škatli. Predpostavimo, da je središče koordinatnega sistema v centru te škatle. Koordinate atomov so torej vedno med $-\frac{L}{2}$ in $\frac{L}{2}$. Za računanje s koordinatami atomov, upoštevajoč periodične robne pogoje, je priročno definirati funkcijo wrap($\boldsymbol{x}$):

$$\mathrm{wrap}(\boldsymbol{x}) = \boldsymbol{x} - L \cdot \mathrm{nint}\left(\frac{\boldsymbol{x}}{L}\right) \quad . \tag{2.3}$$

Funkcija nint($\boldsymbol{x}$) zaokroži komponente vektorja $\boldsymbol{x}$ na najbližje celo število. Če pri premikanju atom zapusti škatlo, običajno namesto njega začnemo spremljati njegovo vstopajočo sliko. Z uporabo enačbe

$$\boldsymbol{x}' = \mathrm{wrap}(\boldsymbol{x}) \tag{2.4}$$

poskrbimo, da so nove koordinate premaknjenega atoma oziroma njegove slike vedno znotraj škatle. Alternativa temu pristopu je, da atom ob izstopu iz škatle še vedno spremljamo, brez spreminjanja njegovih koordinat [8]. V kolikor pravilno upoštevamo kriterij minimalne slike, ta odločitev ne vpliva na rezultat simulacije.

Pri določanju energije sistema moramo običajno izračunati razdalje med atomi. Pri tem moramo upoštevati kriterij minimalne slike. Razdaljo r_{ij} med atomoma i in j s koordinatami $\boldsymbol{x}_i$ in $\boldsymbol{x}_j$ izračunamo po enačbah

$$\boldsymbol{\Delta x} = \mathrm{wrap}(\boldsymbol{x}_j - \boldsymbol{x}_i) \quad \text{in} \tag{2.5}$$

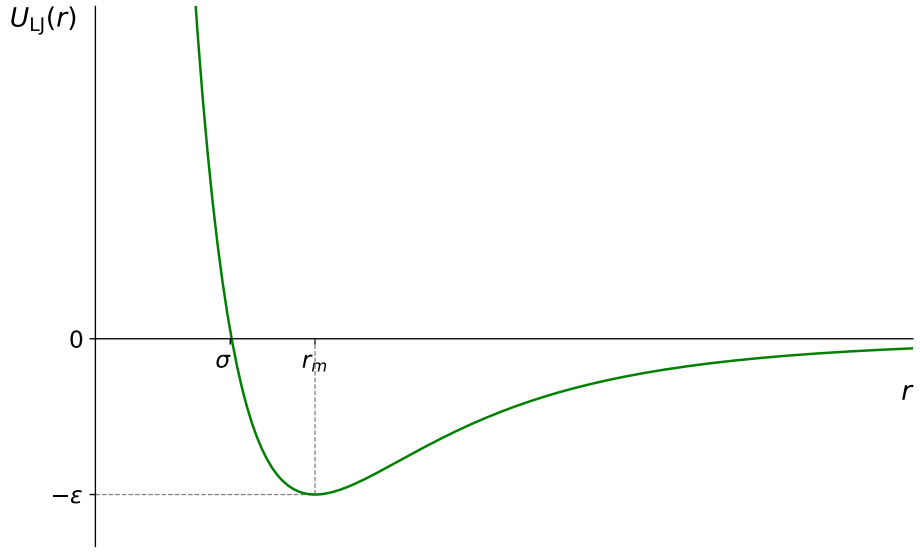
$$r_{ij} = \sqrt{(\boldsymbol{\Delta x}_x)^2 + (\boldsymbol{\Delta x}_y)^2 + (\boldsymbol{\Delta x}_z)^2} \quad . \tag{2.6}$$

2.3 Potencial Lennard-Jones

Potencial Lennard-Jones je relativno preprost matematični model, ki opisuje potencialno energijo interakcij med dvema nevtralnima atomoma ali molekulama glede na njuno medsebojno razdaljo. Zaradi svoje računske preprostosti se pogosto uporablja pri računalniških simulacijah, čeprav obstajajo natančnejši modeli. Podan je z enačbo

$$U_{\mathrm{LJ}}(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \quad , \tag{2.7}$$

kjer je r razdalja med delcema, σ razdalja, pri kateri je potencialna energija enaka nič, in ϵ globina potencialne jame [1]. Graf potencialne energije lahko vidimo na sliki 2.3. Potencialna energija med delcema je najnižja pri medsebojni razdalji $r_m = \sqrt[6]{2}\,\sigma$. Takrat rečemo, da sta delca v ravnovesni legi. Velja $U_{\mathrm{LJ}}(r_m) = -\epsilon$. Globlja kot je potencialna jama, močnejše so interakcije med delci.



Slika 2.3: Graf potenciala Lennard-Jones v odvisnosti od razdalje med delcema r.

Med dvema nevtralnima delcema delujejo tako privlačne kot odbojne sile, njihova velikost pa je odvisna od njune medsebojne razdalje. Zato je tudi enačba (2.7) sestavljena iz dveh členov [38]. Prvi člen (r^{-12}) opisuje odbojne sile in prevladuje, ko je r manjši od ravnovesne razdalje. Modelira Paulijeve odbojne sile, ki nastanejo zaradi prekrivanja elektronskih ovojnic. Drugi člen (r^{-6}) pa opisuje van der Waalsove privlačne sile in prevladuje, ko je r večji od ravnovesne razdalje.

Enačba (2.7) opisuje potencialno energijo med dvema enakima delcema, ki imata enake vrednosti σ in ϵ. Če pa želimo računati interakcije med dvema različnima delcema i in j z vrednostmi σ_i, ϵ_i, σ_j in ϵ_j, moramo uporabiti eno od pravil kombiniranja (angl. *combining rules*). V računalniških simulacijah se zelo pogosto uporabljata pravili Lorentz-Berthelot [6]. Opisujeta ju enačbi

$$\sigma_{ij} = \frac{\sigma_i + \sigma_j}{2} \quad \text{, in} \tag{2.8}$$

$$\epsilon_{ij} = \sqrt{\epsilon_i \, \epsilon_j} \quad . \tag{2.9}$$

Vrednosti σ_{ij} in ϵ_{ij} uporabimo v enačbi (2.7) namesto vrednosti σ in ϵ.

Če je sistem, ki ga modeliramo, sestavljen zgolj iz dveh delcev, je energija sistema kar enaka potencialni energiji med njima. Običajno pa je sistem sestavljen iz več delcev. Takrat energijo sistema izračunamo kot vsoto potencialne energije med vsemi pari N delcev po enačbi

$$U = \sum_{i=1}^{N} \sum_{j=i+1}^{N} U_{\mathrm{LJ}}(r_{ij}) \quad . \tag{2.10}$$

Potencialno energijo $U_{\mathrm{LJ}}(r)$ moramo torej izračunati $\frac{N(N-1)}{2}$-krat.

2.4 Radialna porazdelitvena funkcija

Ena od funkcij, ki opisuje lokalno strukturo sistema delcev, je radialna porazdelitvena funkcija (angl. *radial distribution function*). Označujemo jo z $g(r)$. Definira, kako se z razdaljo r od nekega referenčnega delca spreminja lokalna gostota delcev. Določimo jo lahko tako eksperimentalno, na primer s tehnikami razprševanja sevanja, kot tudi z računalniškimi simulacijami.

V simulacijah $g(r)$ izračunamo kot razmerje med povprečno številsko gostoto $\rho(r)$ na razdalji r od vsakega danega delca in številsko gostoto ρ idealnega plina [1]. To lahko zapišemo z enačbo

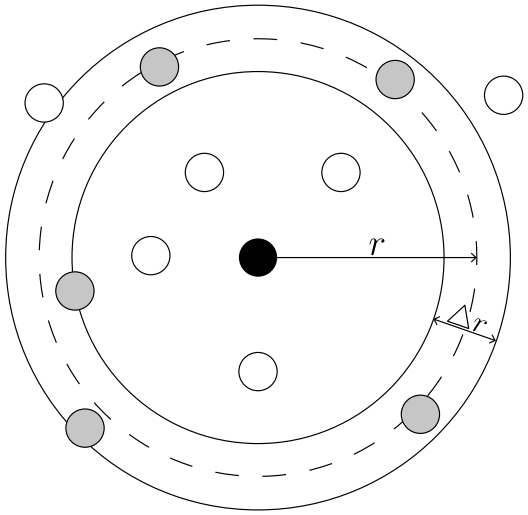
$$g(r) = \frac{\rho(r)}{\rho} \quad . \tag{2.11}$$

Številska gostota idealnega plina je krajevno neodvisna in jo izračunamo po enačbi

$$\rho = \frac{N}{V} \quad , \tag{2.12}$$

kjer je N število delcev v sistemu, V pa volumen sistema. Za izračun povprečne številke gostote $\rho(r)$ najprej izračunamo razdalje med vsemi pari delcev in jih razvrstimo v histogram $H(r)$ z intervalom Δr. Indeks intervala, kamor spada razdalja r, izračunamo po enačbi

$$i_{bin} = \left\lfloor \frac{r}{\Delta r} \right\rfloor \quad . \tag{2.13}$$



Slika 2.4: Shema določanja $g(r)$ v dveh dimenzijah. Prostor okrog vsakega delca je razdeljen na kolobarje debeline Δr. V treh dimenzijah prostor razdelimo na sferične lupine.

Lahko si predstavljamo, da prostor okrog vsakega delca razdelimo na sferične lupine debeline Δr, kot je prikazano na sliki 2.4. V vsaki lupini preštejemo število delcev, ki jih ta vsebuje. Nato lahko izračunamo $\rho(r)$ po enačbi

$$\rho(r) = \frac{H(r)}{\Delta V(r)} \cdot \frac{1}{(N-1)} \quad , \tag{2.14}$$

kjer je $\Delta V(r)$ volumen sferične lupine na razdalji r. Izračunamo ga po enačbi

$$\Delta V(r) = \frac{4}{3}\pi \left[\left(r + \frac{\Delta r}{2} \right)^3 - \left(r - \frac{\Delta r}{2} \right)^3 \right] \quad . \tag{2.15}$$

S faktorjem $(N-1)^{-1}$ v enačbi (2.14) normaliziramo vrednosti iz histograma $H(r)$, saj ta vsebuje število razdalj med vsemi *pari* delcev.

Enačbe (2.11), (2.12) in (2.14) predpostavljajo, da so vsi delci v sistemu enaki. Ko imamo v sistemu M različnih tipov delcev, običajno izračunamo porazdelitvene funkcije $g(\tau_1, \tau_2, r)$ za vse kombinacije parov tipov delcev. Z τ označimo tip delcev. Število kombinacij parov tipov delcev je

$$M_c = \binom{M+1}{2} = \frac{M\,(M+1)}{2} \quad . \tag{2.16}$$

Na primer, za sistem z dvema tipoma delcev, A in B, izračunamo $g(A, A, r)$, $g(A, B, r)$ in $g(B, B, r)$. Za vsak par tipov delcev kreiramo svoj histogram razdalj $H(\tau_1, \tau_2, r)$. Ker imamo več različnih tipov delcev, moramo še posebej paziti na normalizacijo. Enačbo (2.11) preoblikujemo:

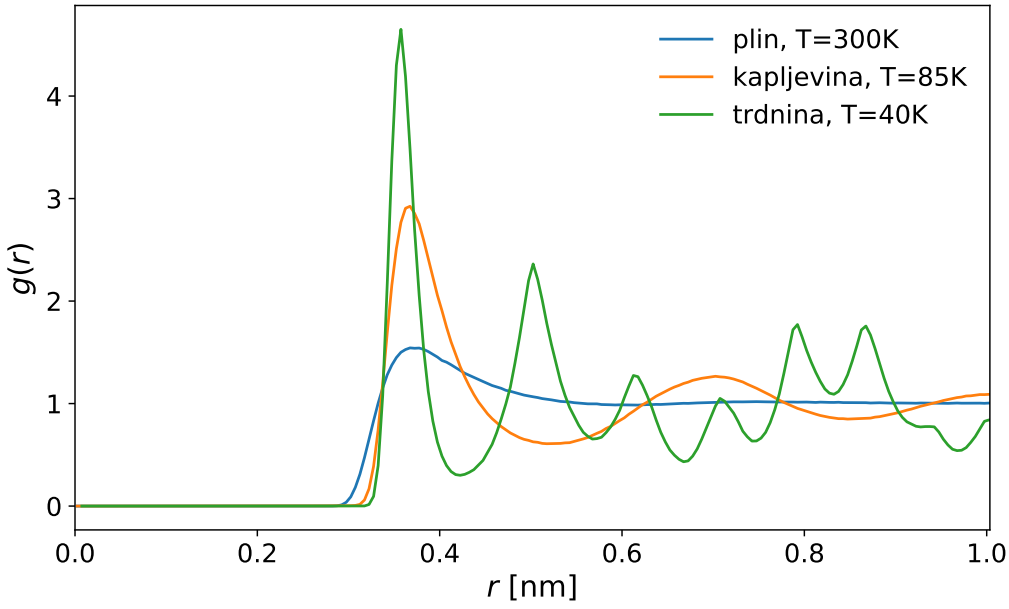
$$\begin{aligned} g(r) &= \frac{\rho(r)}{\rho} \\ &= \left(\frac{H(r)}{\Delta V(r)} \cdot \frac{1}{(N-1)}\right)\left(\frac{N}{V}\right)^{-1} \\ &= \frac{H(r) \cdot V}{\Delta V(r)} \cdot \frac{1}{N(N-1)} \quad . \end{aligned} \tag{2.17}$$

Faktor $(N(N-1))^{-1}$ interpretiramo kot normalizacijski faktor. Če obravnavamo samo delce enakega tipa, recimo tipa A, potem pri kreiranju histograma izračunamo razdaljo od vsakega izmed N_A delcev do preostalih $N_A - 1$ delcev. Če pa obravnavamo delce različnih tipov, izračunamo razdaljo od vsakega izmed N_A delcev tipa A do vseh N_B delcev tipa B in obratno. V prvem primeru smo za pripravo histograma izračunali $N(N-1)$ razdalj, v drugem pa $2N_AN_B$ razdalj. Na podlagi tega in enačbe (2.17) lahko izračunamo $g(\tau_1, \tau_2, r)$ z enačbama

$$g(\tau_1, \tau_2, r) = \frac{H(\tau_1, \tau_2, r) \cdot V}{\Delta V(r)} \cdot \alpha_{\tau_1\tau_2} \quad \text{in} \tag{2.18}$$

$$\alpha_{\tau_1\tau_2} = \begin{cases} (N_{\tau_1}\,(N_{\tau_1} - 1))^{-1}, & \text{če } \tau_1 = \tau_2 \\ (2\,N_{\tau_1}\,N_{\tau_2})^{-1}, & \text{sicer} \end{cases} \quad . \tag{2.19}$$

Radialne porazdelitve funkcije so pomembne, ker lahko z njihovo pomočjo povežemo mikroskopske podrobnosti z makroskopskimi lastnostmi. So namreč dober pokazatelj agregatnega stanja. Kot lahko vidimo na sliki 2.5 se vrednosti $g(r)$ za pline, kapljevine in trdne snovi zelo razlikujejo. Glede na enačbo (2.11) za idealni plin velja $g(r) = 1$. Vsakršno odstopanje odraža korelacije med delci zaradi medmolekulskih interakcij.



Slika 2.5: Graf radialne porazdelitvene funkcije za argon v različnih agregatnih stanjih. $\sigma = 0{,}3345\,\mathrm{nm}$ in $\epsilon = 1{,}045\,\mathrm{kJ/mol}$.

Poglavje 3

Izvedba na centralni procesni enoti

Glavni cilj te magistrske naloge je prilagoditi metodo Monte Carlo za izvajanje na grafičnih procesnih enotah in jo implementirati. S tem želimo doseči čim večje pohitritve pri simuliranju fluidov.

Uporabili smo Metropolisov algoritem, opisan v poglavju 2.1.1. Podprli smo tako simulacijo enoatomnih delcev kot tudi molekul. V simulaciji računamo energijo sistema in radialno porazdelitveno funkcijo. Interakcije med atomi modeliramo z Lennard-Jonesovim potencialom, opisanim v poglavju 2.3. Algoritem smo najprej implementirali na klasičen, zaporedni način, za izvajanje na centralni procesni enoti. To izvedbo smo nato uporabili kot referenco za ovrednotenje pohitritev izvedbe na grafičnih procesnih enotah.

Zaporedno izvedbo smo napisali v programskem jeziku C++. Razvijali smo v integriranem razvojnem okolju Microsoft Visual Studio 2017 [24]. Da je kodo možno prevesti in izvajati na različnih operacijskih sistemih, smo uporabili orodje CMake [7]. To je sistem za gradnjo programov, ki olajša njihovo prevajanje in povezovanje.

3.1 Vhodni podatki in simulacijski model

Za simulacijo sistema molekul najprej potrebujemo opis simulacijskega modela, ki ga program ob zagonu prebere iz treh datotek. Lokacijo datotek določimo s parametri ukazne vrstice.

Prva datoteka, imenovana `model.txt`, vsebuje osnovne parametre simulacijskega modela. Vsaka vrstica datoteke vsebuje ime parametra in njegovo vrednost, ločena z enačajem. Primer vsebine datoteke lahko vidimo na izseku kode 3.1, opis parametrov pa v tabeli 3.1.

Izsek kode 3.1: Primer vsebine datoteke `model.txt` z osnovnimi parametri modela. Znak `#` označuje komentar.

```
n_atoms=100
n_particles=50
n_atom_types=2
n_mc_passes=1000
temp=240 # [K]
box=5.84 # [Å]
mcmove_tran=0.5 # [Å]
mcmove_rot=0.5236 # rad

# Atom of type 0
atom[0].sigma=1.0 # [Å]
atom[0].epsilon=1.0 # [kJ/mol]
atom[0].min_r=0.345 # [Å]

# Atom of type 1
atom[1].sigma=0.5 # [Å]
atom[1].epsilon=0.5 # [kJ/mol]
atom[1].min_r=0.15 # [Å]
```

Model, ki ga opisuje ta datoteka, ima koordinatno izhodišče na sredini škatle z dolžino robu L. Škatla vsebuje N molekul, ki so sestavljene iz skupaj N_a atomov. Če velja $N = N_a$, potem je vsaka molekula sestavljena iz točno enega atoma. Sistem seveda lahko vsebuje atome različnih kemijskih elementov. V vhodnih datotekah in programu so tipi atomov označeni z indeksi od 0 do $M - 1$, kjer je M število različnih tipov atomov v sistemu. Za vsak tip atoma τ datoteka vsebuje parametre σ_τ, ϵ_τ in $r_{\min,\tau}$. Parametra σ_τ in ϵ_τ potrebujemo za izračun Lennard-Jonesovega potenciala. Parameter

parameter	oznaka	opis
`n_particles`	N	Število molekul v sistemu.
`n_atoms`	N_a	Število vseh atomov v sistemu.
`n_atom_types`	M	Število različnih tipov atomov.
`n_mc_passes`	N_{MC}	Število ciklov Metropolisovega algoritma.
`temp`	T	Temperatura sistema v kelvinih.
`box`	L	Dolžina robu škatle, ki omejuje sistem.
`mcmove_tran`	Δx	Maksimalna translacija molekule pri naključnem premikanju.
`mcmove_rot`	$\Delta\varphi$	Maksimalna rotacija molekule pri naključnem premikanju.
`atom[`τ`].sigma`	σ_τ	Parameter σ za potencial Lennard-Jones za tip atoma τ.
`atom[`τ`].epsilon`	ϵ_τ	Parameter ϵ za potencial Lennard-Jones za tip atoma τ.
`atom[`τ`].min_r`	$r_{\min,\tau}$	Polmer atoma tipa τ.

Tabela 3.1: Opis parametrov modela iz datoteke `model.txt`. V prvem stolpcu je ime parametra iz datoteke same, v drugem stolpcu pa oznaka parametra, ki je uporabljena v tem delu.

$r_{\min,\tau}$ lahko uporabimo za modeliranje atomov kot trdih krogel s polmerom $r_{\min,\tau}$, ki se ne morejo prekrivati. Skozi celotno simulacijo katerakoli dva atoma tipov A in B ne moreta biti bližje kot $r_{\min,A} + r_{\min,B}$.

Preostali dve datoteki, `particles.csv` in `atoms.csv`, opisujeta začetno konfiguracijo vseh molekul in atomov v sistemu. Konfiguracija molekule je določena z njeno pozicijo in orientacijo v prostoru. Pozicijo molekule k v treh dimenzijah predstavimo z vektorjem koordinat njenega središča, $\boldsymbol{x}_{\mathrm{m},k} = [x\, y\, z]^T$, njeno orientacijo pa z rotacijsko matriko $\boldsymbol{R}_k$ dimenzij 3×3. V simulacijskem modelu je vsaka molekula sestavljena iz enega ali več atomov. Pozicija atoma i, ki je del molekule k, je določena s stolpičnim vektorjem lokalnih koordinat, $\boldsymbol{x}_{\mathrm{loc},i}$, ki so relativne glede na središče molekule. Globalne koordinate atoma i izračunamo po enačbi

$$\boldsymbol{x}_{\mathrm{a},i} = \boldsymbol{x}_{\mathrm{m},k} + \boldsymbol{R}_k\, \boldsymbol{x}_{\mathrm{loc},i} \quad . \tag{3.1}$$

Datoteka `particles.csv` je zapisana v formatu `csv` in vsebuje N vrstic. Vrstica z indeksom k vsebuje podatke o molekuli k. Podatki v vrsticah so ločeni z vejico in zaporedoma predstavljajo:

- $i_{\mathrm{start},k}$, indeks prvega atoma v molekuli k,
- n_k, število atomov v molekuli k,
- komponente vektorja $\boldsymbol{x}_{\mathrm{m},k}$, in
- komponente matrike $\boldsymbol{R}_k$.

Datoteka `atoms.csv` je prav tako zapisana v formatu `csv`, vsebuje pa N_a vrstic. Vrstica z indeksom i vsebuje podatke o atomu i. Zaporedoma so to:

- τ_i, tip atoma i,
- komponente vektorja $\boldsymbol{x}_{\mathrm{a},i}$, in
- komponente vektorja $\boldsymbol{x}_{\mathrm{loc},i}$.

Poleg opisanih vhodnih datotek program potrebuje še dva parametra, ki opisujeta računanje radialne porazdelitvene funkcije. To sta interval Δr za pripravo histograma in frekvenca računanja histograma ν_H. Razdalje med atomi razvrstimo v histogram vsakih ν_H ciklov algoritma. Oba parametra specificiramo s parametri ukazne vrstice.

3.2 Priprava podatkov

Ob zagonu program najprej prebere vsebino vhodnih datotek in jo shrani v pomnilnik. Za lažje delo s podatki smo definirali strukture na izseku kode 3.2. Podatke smo shranili v polja tipa `std::vector<T>`. Z uporabo ključne besede `typedef` in direktiv predprocesorja smo definirali nov podatkovni tip `real_t` za podatke v plavajoči vejici. Tako se lahko odločimo med enojno in dvojno natančnostjo kar z argumenti, ki jih podamo prevajalniku.

Ko so vsi potrebni podatki naloženi v pomnilnik, nastavimo vrednosti, ki se tekom simulacije ne bodo spreminjale. Tako jih izračunamo samo enkrat in ne vsakič, ko jih potrebujemo. Izračunamo Boltzmannov faktor β, ki ga potrebujemo v fazi odločitve Metropolisovega algoritma. Boltzmannov faktor je odvisen le od temperature sistema. Shranimo ga kar v strukturo tipa `model_t`. Izračunamo tudi parametre σ, ϵ in $r^2_{\min}$ za vse kombinacije tipov atomov. Uporabimo enačbe (2.8), (2.9) in

$$r^2_{\min}(\tau_1, \tau_2) = (r_{\min,\tau_1} + r_{\min,\tau_1})^2 \quad . \tag{3.2}$$

Te podatke shranimo v polje tipa `std::vector<energy_params_t>`. Indeks polja predstavlja indeks kombinacije dveh tipov atomov. Ker v izvedbi za oznake tipov atomov uporabljamo števila od 0 do $M-1$, lahko indeks kombinacije $c(\tau_1, \tau_2)$ enostavno izračunamo po enačbi

$$c(\tau_1, \tau_2) = b + a \cdot M - \binom{a+1}{2} \quad , \tag{3.3}$$

kjer je $a = \min(\tau_1, \tau_2)$ in $b = \max(\tau_1, \tau_2)$.

Za izračun radialne porazdelitvene funkcije potrebujemo M_c histogramov, za vsako kombinacijo parov tipov atomov svojega. Vsak histogram mora imeti

$$N_{\text{bins}} = \left\lceil \frac{L}{2\,\Delta r} \right\rceil \tag{3.4}$$

stolpcev. Uporabimo dvodimenzionalno polje tipa `std::vector<std::vector<uint32_t>>`, kjer indeks prve dimenzije predstavlja $c(\tau_1, \tau_2)$. Vse stolpce histogramov nastavimo na nič.

Izsek kode 3.2: Definicija C++ struktur za opis modela.

```
#ifdef PREC_REAL_DOUBLE
typedef double real_t;
#else
typedef float real_t;
#endif

typedef struct { real_t x, y, z; } vec3_t;
typedef struct { real_t m[3][3]; } mat3_t;

typedef struct {
  uint32_t ix_start, n_atoms;
  vec3_t position;
  mat3_t orientation;
} particle_t;

typedef struct {
  vec3_t position;
  uint32_t type;
} atom_t;

typedef struct {
  uint32_t n_atoms, n_particles, n_mc_passes, n_atom_types;
  float temp;
  real_t box, mcmove_tran, mcmove_rot;
  real_t factor_beta;
} model_t;

typedef struct {
  real_t min_r, sigma, epsilon;
} atom_info_t;

std::vector<particle_t> particles; // podatki o molekulah (i_start, n, x_m, R)
std::vector<atom_t> atoms; // pozicije in tip atomov (x_a, τ)
std::vector<vec3_t> atoms_local; // lokalne koordinate atomov (x_loc)
std::vector<atom_info_t> atom_infos; // podatki o tipu atoma (σ, ϵ, r_min)
model_t model;
std::vector<energy_params_t> energy_params; // σ(τ1,τ2), ϵ(τ1,τ2), r²_min(τ1,τ2)
std::vector<std::vector<uint32_t>> histogram; // histogram (H(τ1,τ2,r))
```

3.3 Izračun začetne energije sistema

Tekom simulacije moramo v vsakem koraku vedeti kolikšna je trenutna energija sistema. Za izračun celotne energije sistema je po enačbi (2.10) potrebno sešteti prispevek energije vseh parov atomov v sistemu. Namesto računanja v vsakem koraku algoritma, lahko celotno energijo sistema izračunamo samo enkrat, na začetku simulacije. V vsakem nadaljnjem koraku algoritma pa izračunamo le spremembo energije, ki jo povzroči morebitni premik atomov, in jo prištejemo energiji sistema iz prejšnjega koraka. To je bolj učinkovito, saj moramo za izračun spremembe energije upoštevati le interakcije premaknjenih atomov z vsemi ostalimi atomi. Energijo sistema v začetnem stanju izračunamo po algoritmu 3.1.

Algoritem 3.1 Psevdokoda algoritma za izračun energije sistema v začetnem stanju

```
1:  function CalculateInitialEnergy()
2:      E ← 0
3:      for k = 0 to N − 2 do                          ▷ za vsako molekulo k
4:          for i = i_start,k to i_start,k + n_k − 1 do    ▷ za vsak atom i v molekuli k
5:              for l = k + 1 to N − 1 do              ▷ za vsako nadaljnjo molekulo l
6:                  for j = i_start,l to i_start,l + n_l − 1 do   ▷ za vsak atom j v molekuli l
7:                      r^2 ← dist^2(x_a,i, x_a,j)
8:                      E ← E + LJ(r^2, σ(τ_i, τ_j), ε(τ_i, τ_j))
9:                  end for
10:             end for
11:         end for
12:     end for
13:     return E
14: end function
```

Funkcija $dist^2(\boldsymbol{x}_1, \boldsymbol{x}_2)$ izračuna kvadrat razdalje med koordinatama $\boldsymbol{x}_1$ in $\boldsymbol{x}_2$ in pri tem upošteva periodične robne pogoje. To stori na podoben način kot enačba (2.6), le da izpusti korenjenje in vrne kvadrat razdalje namesto razdalje. Kvadrat razdalje lahko izkoristimo pri računanju potenciala Lennard-Jones. V enačbi (2.7) namreč nastopata r^6 in r^{12}, ki ju lahko na-

domestimo z $(r^2)^3$ in $(r^2)^6$. S tem se izognemo nepotrebnemu korenjenju in potenciranju razdalje. Z algoritmom 3.2 definiramo funkcijo $LJ(r^2, \sigma, \epsilon)$, ki to izkoristi.

Algoritem 3.2 Psevdokoda optimiziranega algoritma za izračun potenciala Lennard-Jones

```
1: function LJ(r^2, σ, ε)
2:     a ← (σ · σ)/r^2
3:     b ← a · a · a
4:     return 4 ε (b · b − b)
5: end function
```

3.4 Generiranje naključnih premikov molekul

V drugi fazi vsakega koraka Metropolisovega algoritma moramo generirati naključni premik molekule. Ker so molekule lahko sestavljene iz več atomov, moramo poleg translacijskega premika upoštevati tudi rotacijski premik. Predpostavimo, da so molekule toge. Zato se lokalne koordinate atomov $\boldsymbol{x}_{\mathrm{loc}}$ znotraj molekule tekom simulacije ne spreminjajo.

Pri translacijskem premiku središče molekule k naključno premaknemo za največ Δx v vsaki dimenziji. Potrebujemo generator (psevdo)naključnih števil, ki so enakomerno porazdeljena na intervalu $[0, 1]$. V naši izvedbi smo uporabili funkcijo `std::rand()` iz knjižnice `cstdlib`. Funkcijo prilagodimo, da vrne števila iz želenega intervala: `float rand() { return std::rand()/(float) RAND_MAX; }`. Nove koordinate središča molekule izračunamo po enačbi

$$\boldsymbol{x}'_{\mathrm{m},k} = \mathrm{wrap}(\boldsymbol{x}_{\mathrm{m},k} + (2 \cdot \boldsymbol{v}_{rand} - 1)\, \Delta x) \quad , \tag{3.5}$$

kjer je $\boldsymbol{v}_{rand}$ vektor treh naključnih števil, $\mathrm{wrap}(x)$ pa funkcija, definirana z enačbo (2.3).

Pri rotacijskem premiku pa molekulo naključno zasučemo za največ $\Delta\varphi$ v naključni smeri. Novo orientacijo molekule izračunamo po enačbi

$$\boldsymbol{R}'_k = \boldsymbol{R}_{rand}\,\boldsymbol{R}_k \quad , \tag{3.6}$$

kjer je $\boldsymbol{R}_{rand}$ naključna rotacijska matrika. Uporabili smo Arvov algoritem za hitro generiranje naključnih rotacijskih matrik [4]. Algoritem najprej naredi rotacijo za naključen kot okrog osi z, nato pa to os zavrti na naključen položaj. Pri tem uporabi Householderjeve transformacije. Becker je algoritem dopolnil tako, da lahko omejimo maksimalen kot naključne rotacije [5]. Algoritem lahko vidimo na izseku kode 3.3. Za delovanje potrebuje tri naključna števila. S parametrom `deflection` določimo maksimalen kot rotacije. Izračunamo ga po enačbi

$$\texttt{deflection} = \frac{1-\cos(\Delta\varphi)}{2} \quad . \tag{3.7}$$

Ko imamo novo lokacijo in orientacijo molekule, izračunamo še nove globalne koordinate atomov, ki sestavljajo to molekulo. Uporabimo enačbo

$$\boldsymbol{x}'_{\mathrm{a},i} = \boldsymbol{x}'_{\mathrm{m},k} + \boldsymbol{R}'_k\,\boldsymbol{x}_{\mathrm{loc},i} \quad . \tag{3.8}$$

Nove globalne koordinate atomov smo shranili v podatkovno strukturo slovar (`std::map<uint32_t, vec3_t>`).

Izsek kode 3.3: Izračun naključne rotacijske matrike. Povzeto po [4] in [5].

```
mat3_t rotation_random(const float deflection = 1.f) {
  const real_t
    theta = (rand() - 0.5f) * 2 * PI * deflection, // rotation around pole
    phi = rand() * 2 * PI, // direction of pole deflection.
    z = rand() * 2.f * deflection; // magnitude of pole deflection.

  const real_t r = std::sqrt(z),
               Vx = std::sin(phi) * r,
               Vy = std::cos(phi) * r,
               Vz = std::sqrt(2.f - z);

  const real_t st = std::sin(theta),
               ct = std::cos(theta),
               Sx = Vx * ct - Vy * st,
               Sy = Vx * st + Vy * ct,

  mat3_t M;
  M.m[0][0] = (Vx * Sx - ct) * -1.f;
  M.m[0][1] = (Vx * Sy - st) * -1.f;
  M.m[0][2] = Vx * Vz;
  M.m[1][0] = (Vy * Sx + st) * -1.f;
  M.m[1][1] = (Vy * Sy - ct) * -1.f;
  M.m[1][2] = Vy * Vz;
  M.m[2][0] = (Vz * Sx) * -1.f;
  M.m[2][1] = (Vz * Sy) * -1.f;
  M.m[2][2] = 1.f - z;
  return M;
}
```

3.5 Glavni del algoritma

Naša zaporedna izvedba dosledno sledi opisu Metropolisovega algoritma v poglavju 2.1.1. Algoritem 3.3 je glavni del Metropolisovega algoritma.

Algoritem 3.3 Psevdokoda zaporednega Metropolisovega algoritma.

priprava podatkov ▷ poglavje 3.2
$n_{acc} \leftarrow 0, n_{rej} \leftarrow 0$ ▷ števca za sprejete/zavrnjene premike
$E \leftarrow$ CalculateInitialEnergy() ▷ trenutna energija sistema
$E_{sum} \leftarrow E$ ▷ vsota energije sistema tekom simulacije

for $i_{MC} = 1$ **to** N_{MC} **do** ▷ za vsak cikel simulacije i_{MC}
    for $k = 0$ **to** $N - 1$ **do** ▷ za vsako molekulo k
        $\boldsymbol{x}'_{\mathrm{m},k}$, $\boldsymbol{R}'_k$, $\boldsymbol{x}'_{\mathrm{a}} \leftarrow$ generiraj naključni premik molekule ▷ poglavje 3.4
        $E_{old} \leftarrow$ CalculateEnergy$(k, \boldsymbol{x}_{\mathrm{a}}, \textbf{false}, i_{MC})$ ▷ energija pred premikom
        $E_{new} \leftarrow$ CalculateEnergy$(k, \boldsymbol{x}'_{\mathrm{a}}, \textbf{true}, i_{MC})$ ▷ energija po premiku
        $\Delta E \leftarrow E_{new} - E_{old}$

        $accept \leftarrow$ Decide(ΔE) ▷ sprejmemo odločitev o premiku
        if *accept* **then**
            $E \leftarrow E + \Delta E$
            $n_{acc} \leftarrow n_{acc} + 1$
            $\boldsymbol{x}_{\mathrm{m},k}$, $\boldsymbol{R}_k$, $\boldsymbol{x}_{\mathrm{a}} \leftarrow \boldsymbol{x}'_{\mathrm{m},k}$, $\boldsymbol{R}'_k$, $\boldsymbol{x}'_{\mathrm{a}}$
        else
            $n_{rej} \leftarrow n_{rej} + 1$
        end if

        $E_{sum} \leftarrow E_{sum} + E$
    end for
end for

pripravi izhodne podatke za izpis

Algoritem najprej pripravi potrebne podatke in izračuna začetno energijo sistema. Na vrednost nič nastavimo še dva števca, n_{acc} in n_{rej}, ki štejeta število sprejetih in zavrnjenih premikov. Definiramo tudi dve spremenljivki, E in E_{sum}, ki hranita trenutno in skupno energijo sistema.

Algoritem vsebuje dve glavni gnezdeni zanki. Zunanja zanka šteje cikle algoritma, notranja zanka pa se v vsakem ciklu zaporedno sprehodi čez vse molekule k v sistemu. Odločili smo se namreč za zaporedno izbiranje molekul, da bo v nadaljevanju prilagoditev algoritma za izvajanje na grafičnih procesnih enotah enostavnejša.

V vsakem koraku algoritma generiramo poskus premika molekule k. Uporabimo postopek, opisan v poglavju 3.4. Ko imamo premaknjene koordinate atomov v molekuli, lahko izračunamo spremembo energije, ki bi jo ta premik povzročil. Spremembo energije izračunamo po enačbi (2.1) kot razliko med prispevkoma energije molekule k pred in po premiku. Uporabimo funkcijo CalculateEnergy, definirano z algoritmom 3.4.

Algoritem 3.4 Psevdokoda za izračun prispevka energije molekule. S parametrom *moved* povemo ali obdelujemo premaknjeno ali originalno molekulo.

```
1: function CalculateEnergy(k, x'_a, moved, i_MC)
2:     E_k ← 0
3:     for i = i_start,k to i_start,k + n_k − 1 do          ▷ za vsak atom i v molekuli k
4:         for l = 0 to N − 1, l ≠ k do                      ▷ za vsako molekulo l, l ≠ k
5:             for j = i_start,l to i_start,l + n_l − 1 do   ▷ za vsak atom j v molekuli l
6:                 r² ← dist²(x'_a,i, x_a,j)

7:                 if moved and r² ≤ r²_min(τ_i, τ_j) then   ▷ se atoma prekrivata?
8:                     return NaN
9:                 end if
10:                if not moved and (i_MC mod ν_H = 0) then
11:                    r = √r²
12:                    H(τ_i, τ_j, r) ← H(τ_i, τ_j, r) + 1   ▷ posodobimo histogram
13:                end if

14:                E_k ← E_k + LJ(r², σ(τ_i, τ_j), ϵ(τ_i, τ_j))   ▷ prispevek energije
15:            end for
16:        end for
17:    end for
18:    return E_k
19: end function
```

S to funkcijo izračunamo vsoto potencialnih energij, ki so posledica interakcij med atomi i v molekuli k in atomi j v vseh preostalih molekulah. Za vsako interakcijo najprej izračunamo kvadrat razdalje r^2 med atomoma i in j. Če računamo prispevek energije premaknjene molekule, preverimo, ali je razdalja med atomoma dovolj velika. Če ni, funkcija takoj vrne vrednost NaN (angl. *not a number*). Z NaN označimo, da je prišlo do prekrivanja atomov in premika ni mogoče sprejeti. Zato tudi ni smiselno nadaljevati z računanjem prispevka energije tega delca. S pravilno konfiguracijo vhodnih parametrov lahko preverjanje prekrivanja preskočimo. Prispevek energije izračunamo na podlagi r^2 z algoritmom 3.2.

Poleg izračuna energije molekule ta funkcija tudi posodobi histogram za računanje radialne porazdelitvene funkcije. To storimo tukaj, ker že imamo izračunan r^2. Histogram posodobimo vsakih i_{MC} ciklov algoritma pri računanju prispevka energije originalne molekule. Pri računanju prispevka energije premaknjene molekule tega ne moremo početi, saj se lahko funkcija zaradi prekrivanja predčasno zaključi.

Na podlagi spremembe energije se z algoritmom 3.5 lahko odločimo ali bomo premik molekule sprejeli ali zavrnili. Če je pri premiku prišlo do prekrivanja, premik zavrnemo. V nasprotnem primeru premik sprejmemo z verjetnostjo, ki jo določa enačba (2.2). Če je sprememba energije negativna, premik takoj sprejmemo. Če ni, pa primerjamo naključno število z intervala $[0, 1]$ z vrednostjo $e^{-\beta\Delta E}$. Premik sprejmemo, če je naključno število manjše.

Algoritem 3.5 Psevdokoda odločitve o sprejetju poskusa premika molekule.

```
1: function DECIDE(ΔE)
2:     return ΔE ≠ NaN and (ΔE ≤ 0 or rand() < exp(−β ΔE))
3: end function
```

V kolikor premik zavrnemo, potem zgolj povečamo števec zavrnjenih premikov za ena. Če pa premik sprejmemo, potem poleg povečanja števca sprejetih premikov posodobimo konfiguracijo molekule in atomov v njej s premaknjenimi vrednostnimi. Spremembo energije prištejemo trenutnemu stanju energije E. To nato vedno, ne glede na odločitev o sprejemu premika,

prištejemo k skupni vsoti energije E_{sum}.

Simulacije je konec, ko dosežemo število ciklov, ki smo ga specificirali z vhodnimi podatki.

3.6 Izhodni podatki

Po koncu simulacije moramo normalizirati vrednosti v histogramih $H(\tau_1, \tau_1, r)$, da dobimo radialne porazdelitvene funkcije $g(\tau_1, \tau_1, r)$. Sprehodimo se čez histograme in uporabimo enačbi (2.18) in (2.19). Radialne porazdelitvene funkcije shranimo v datoteko `distribution.csv` v formatu `csv`. Vsak stolpec vsebuje porazdelitveno funkcijo za eno kombinacijo parov tipov atomov. Kasneje lahko na podlagi te datoteke kreiramo vizualizacijo porazdelitvenih funkcij z grafom. Primer lahko vidimo na sliki 5.3.

Poleg datoteke s porazdelitvenimi funkcijami shranimo še končno stanje sistema molekul: $\boldsymbol{x}_{\mathrm{m}}$, $\boldsymbol{R}$ in $\boldsymbol{x}_{\mathrm{a}}$. Uporabimo enak format zapisa kot pri vhodnih datotekah `particles.csv` in `atoms.csv`.

Izpišemo še energijo sistema ob koncu simulacije E, odstotek sprejetih poskusov premika $A_{\%}$ in povprečno energijo atoma tekom simulacije E_{avg}. Uporabimo enačbi

$$A_{\%} = \frac{n_{acc}}{n_{acc} + n_{rej}} \quad \text{in} \tag{3.9}$$

$$E_{avg} = \frac{E_{sum}}{N_{MC} \cdot N \cdot N_a} \quad . \tag{3.10}$$

Izpišemo tudi skupni čas simulacije in povprečni čas za en cikel simulacije. Primer izpisa lahko vidimo na izseku kode 3.4.

Izsek kode 3.4: Primer izpisa rezultata simulacije.

```
A%       = 47.8112  # %
E_avg    = -2.98781 # kJ/mol (per atom)
E        = -3164.11 # kJ/mol

t        = 103.344s # s
t_mc_avg = 103.291  # ms
```

Poglavje 4

Izvedba na grafičnih procesnih enotah

Grafične procesne enote nam ponujajo zelo veliko število vzporednih niti. Simulacijo moramo najprej paralelizirati in razdeliti delo med niti. Na žalost je paralelizacija simulacij Monte Carlo precej bolj zapletena kot paralelizacija simulacij molekulske dinamike [20]. Zaradi Markovske narave same metode je vsak korak simulacije odvisen od rezultatov vseh prejšnjih korakov. To omejuje količino računanja, ki ga lahko izvedemo vzporedno.

Že pri zaporedni izvedbi smo se odločili za zaporedno izbiranje molekul v fazi izbire, saj ima ta večji potencial za paralelizacijo. Pri tem načinu namreč vnaprej vemo vrstni red poskusov premikanja molekul v ciklu, kar lahko s pridom izkoristimo.

Obstajajo trije glavni načini, kako se lahko lotimo paralelizacije metode Monte Carlo [2, 23]:

1. **Nerodni paralelizem** (angl. *embarrassingly parallel*): Je najpreprostejši način, saj simulacijo zgolj zaženemo večkrat, da se jih izvaja več sočasno. Simulacije so neodvisne in ne zahtevajo medsebojne komunikacije.

2. **Energijska dekompozicija** (angl. *energetic decomposition*, tudi *farm decomposition*): Računanje energije razdelimo med vzporedne niti. Niti

so med seboj sicer neodvisne, a poskus premika molekule in odločanje se še vedno izvajata zaporedno.

3. **Domenska dekompozicija** (angl. *domain decomposition*): Izvedemo več poskusov premika sočasno. Ponavadi prostor razdelimo na celice in simuliramo kratkosežne interakcije med atomi, da lahko hkrati premikamo delce v različnih celicah.

Nerodni paralelizem je v kontekstu simulacij Monte Carlo bolj primeren za večjedrne procesorje in računalniške gruče kot pa za grafične procesne enote. Domenska dekompozicija v našem primeru odpade, saj z Lennard-Jonesovim potencialom modeliramo interakcije med vsemi atomi, ne glede na njihovo medsebojno razdaljo. Omejeni smo le s kriterijem minimalne slike. Zato smo se odločili za energijsko dekompozicijo.

Za izvedbo na grafičnih procesnih enotah smo uporabili platformo CUDA, ki smo jo opisali v poglavju 4.1. Računanje energije smo na grafično procesno enoto prestavili s pomočjo vzporedne redukcije. Izvedbo smo opisali v poglavju 4.2. Pri dovolj velikem številu atomov smo dosegli solidne pohitritve. Na grafično procesno enoto pa smo hoteli prestaviti tudi generiranje poskusa premika molekul in odločanje. Zato smo nadaljevali z optimizacijo algoritma in nastala je druga, optimizirana izvedba. Opisali smo jo v poglavju 4.3.

4.1 CUDA

Grafična procesna enota (angl. *Graphics processing unit, GPU*) je specializirano integrirano vezje, ki se primarno uporablja pri prikazovanju računalniške grafike. Dandanes jih najdemo v skoraj vseh računalnikih. Njihova glavna lastnost, zaradi katere so bolj primerne za prikazovanje računalniške grafike kot centralne procesne enote, je njihova visoko paralelna arhitektura, ki jim omogoča sočasno izvajanje številnih izračunov. Zaradi svoje učinkovitosti se čedalje bolj uporabljajo tudi za računsko zahtevne probleme, ki niso neposredno povezani z računalniško grafiko. Govorimo o splošno-namenskem

računanju na grafičnih procesnih enotah (angl. *general-purpose computing on graphics processing units, GPGPU*) [28].

CUDA, *Compute Unified Device Architecture*, je strojna in programska platforma za vzporedno računanje na grafičnih procesnih enotah [26]. Razvili so jo pri podjetju NVIDIA. Čeprav gre za brezplačno platformo, je ta zaprta in na voljo le na grafičnih procesnih enotah podjetja NVIDIA. Razvijalcem ponuja neposreden dostop do nabora ukazov računskih elementov na grafičnih procesnih enotah in do njihovega pomnilnika. Programe lahko pišemo v razširitvah programskih jezikov C, C++ in Fortran, imenovanih CUDA C/C++ in CUDA Fortran. Na voljo so tudi knjižnice, ki omogočajo pisanje programov CUDA v drugih popularnih programskih jezikih.

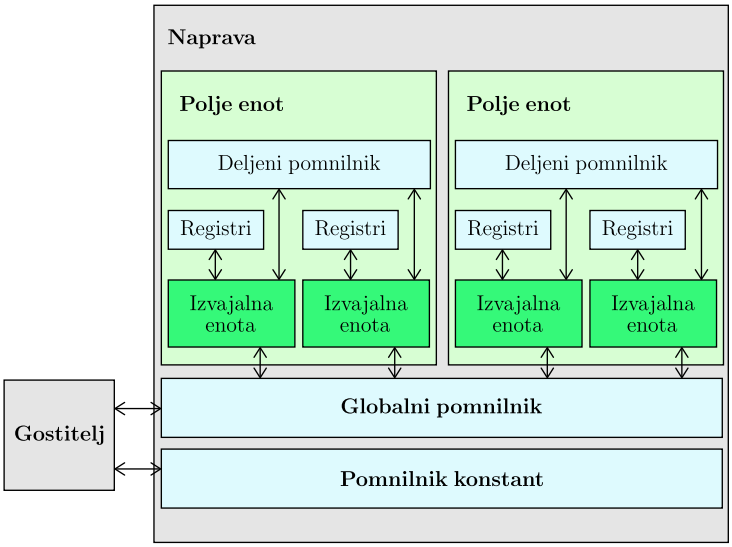
4.1.1 Arhitektura

Arhitektura CUDA (slika 4.1) je zgrajena okrog polj enot (angl. *Streaming Multiprocessors, SM*), ki so sestavljena iz več izvajalnih enot (angl. *CUDA Cores* ali *Streaming Processors, SP*). Polje enot je pravzaprav samostojen procesor, ki skrbi za sočasno izvajanje več sto niti. To mu omogoča arhitektura SIMT (angl. *Single-Instruction, Multiple-Thread*), kar pomeni, da več niti hkrati izvaja isti ukaz [25]. Skupino takšnih niti imenujemo snop (angl. *warp*). Na vseh trenutno podprtih napravah CUDA snopi vsebujejo 32 niti [27]. Vsaka nit se izvaja v eni izvajalni enoti.

Pomnilniški model (slika 4.1) je zasnovan na različnih tipih pomnilnika [33].

1. **Registri** (angl. *registers*)
   So najhitrejši tip pomnilnika, ki je na voljo vsaki niti. Njihovo število je zelo omejeno.

2. **Lokalni pomnilnik** (angl. *local memory*)
   Ko zmanjka prostora v registrih, se podatki shranjujejo v lokalni pomnilnik, ki je fizično del globalnega pomnilnika. Vsaka nit lahko dostopa le do svojega dela pomnilnika.



Slika 4.1: Arhitektura in pomnilniški model CUDA. Povzeto po [18].

3. **Deljeni pomnilnik** (angl. *shared memory*)
   Fizično se nahaja znotraj polja enot. Dostop do njega je zato precej hitrejši od dostopa do globalnega pomnilnika in le malce počasnejši od dostopov do registrov. Do njega lahko dostopajo le niti v istem bloku (poglavje 4.1.2), zato ga lahko uporabljamo za komunikacijo in deljenje podatkov med njimi.

4. **Globalni pomnilnik** (angl. *global memory*)
   Je največji izmed naštetih vrst pomnilnikov, vendar je tudi dostop do njega najpočasnejši, saj fizično ni na istem čipu kot grafična procesna enota. Do njega lahko dostopajo vse niti, ne glede na blok. Njegov glavni namen je komunikacija in prenos podatkov med gostiteljem in napravo.

5. **Pomnilnika konstant in tekstur** (angl. *constant and texture memory*)
   Oba sta specializirana predpomnjena bralna pomnilnika in sta ostanka grafičnih korenin grafičnih procesnih enot.

4.1.2 Izvajalni model

Program CUDA sestoji iz zaporednega programa, ki se izvaja na gostitelju (centralni procesni enoti), in iz enega ali več ščepcev (angl. *kernels*), ki se izvajajo na napravi (grafični procesni enoti). Ko ščepec v zaporedni kodi zaženemo, ga na napravi začne izvajati več vzporednih niti. Običajno je napisan tako, da ga vsaka nit izvaja nad različnimi podatki.

Niti so logično organizirane v bloke (angl. *blocks*), bloki pa v mrežo (angl. *grid*). Bloki in mreže so lahko eno-, dvo- ali tri-dimenzionalni, vsako nit pa identificiramo preko eno-, dvo- ali tri-dimenzionalnih indeksov. Dimenzionalnost in velikost blokov in mrež določi programer. Običajno jih prilagodi strojni opremi in dimenzijam problema, ki ga rešuje.

Celoten blok niti je dodeljen za izvajanje enemu polju enot. Na enem polju enot se lahko hkrati izvaja več blokov niti. V kolikor polje enot nima dovolj resursov za vzporedno izvajanje blokov, se ti izvajajo v poljubnem vrstnem redu. Niti znotraj enega bloka lahko med seboj komunicirajo preko deljenega pomnilnika, lahko pa jih med seboj tudi sinhroniziramo.

Izvajanje tipičnega programa CUDA poteka v štirih korakih. Najprej moramo iz pomnilnika gostitelja v pomnilnik naprave prenesti vhodne podatke. Nato gostitelj izda zahtevo za zagon ščepca. Ta se začne izvajati v več sočasnih nitih na napravi in pri tem dostopa in piše v pomnilnik naprave. Sledi še prenos rezultatov nazaj v pomnilnik gostitelja.

4.2 Osnovna izvedba s vzporedno redukcijo

Za paralelizacijo smo izbrali način z energijsko dekompozicijo, ki je dokaj enostaven za izvedbo. Celotna struktura algoritma ostane zelo podobna zaporednemu algoritmu. Paraleliziramo le računanje prispevkov energije interakcij med atomi, ki ga prestavimo na grafično procesno enoto. To velja tako za izračun začetne energije sistema kot tudi za izračun spremembe energije v vsakem koraku.

4.2.1 Priprava podatkov

Branje vhodnih podatkov in osnovna priprava podatkov ostane enaka kot pri zaporedni izvedbi. Dodatno pa moramo na grafično kartico prenesti podatke, ki jih potrebujemo za računanje energije in alocirati prostor za pomožne podatke. V izvedbi na centralni procesni enoti smo za polja uporabljali podatkovni tip `std:vector<T>`. Ta na grafičnih procesnih enotah ni na voljo, zato moramo uporabiti navadna polja in kazalce.

Osnovne podatke o modelu (struktura tipa `model_t`) in parametre za izračun radialne porazdelitvene funkcije smo prenesli v pomnilnik konstant, saj se tekom simulacije ne spreminjajo. Niti iz istega snopa bodo vedno dostopale do istega podatka iz teh struktur. To pa ne velja za izračunane parametre kombinacij tipov atomov (polje tipa `*energy_params_t`) in lokalne koordinate atomov, zato jih prenesemo v globalni pomnilnik. V globalni pomnilnik prenesemo še globalne koordinate atomov $\boldsymbol{x}_\mathrm{a}$ in pomožno polje `uint32_t *atom_to_particle`, ki preslika indeks atoma v indeks molekule. Tega polja v zaporedni izvedbi nismo potrebovali, saj smo vedno najprej iterirali po molekulah in nato po atomih znotraj molekule. V globalnem pomnilniku rezerviramo še prostor za histograme in nastavimo njihove vrednosti na nič. Rezerviramo tudi prostor za dve pomožni polji. Prvo polje bo hranilo premaknjene koordinate atomov $\boldsymbol{x}'_\mathrm{a}$, drugo polje pa vmesne vsote energije vsakega bloka niti.

Tekom simulacije nikoli ne potrebujemo podatkov o več kot eni molekuli naenkrat, zato podatke $i_{\mathrm{start},k}$, n_k, $\boldsymbol{x}_{\mathrm{m},k}$ in $\boldsymbol{R}_k$ ščepcem po potrebi podamo kot argument. Lokacijo in orientacijo molekule potrebujemo samo v ščepcu za izračun premaknjenih koordinat atomov.

4.2.2 Izračun začetne energije sistema

Sešteti moramo prispevek energije vseh parov atomov v sistemu. Delo najenostavneje razdelimo tako, da vsaka nit izračuna prispevek energije enega para atomov. Ko so izračunani vsi prispevki vseh parov atomov, jih sešte-

jemo. Do takrat so niti neodvisne in ne potrebujejo medsebojne sinhronizacije.

Vseh parov atomov je $(N_a)^2$. Zato smo niti organizirali v dvodimenzionalno mrežo in bloke. Tako lahko na podlagi dvodimenzionalnih identifikatorjev niti in blokov ugotovimo, za kateri par atomov je nit odgovorna:

```
uint32_t i = blockIdx.x * blockDim.x + threadIdx.x;
uint32_t j = blockIdx.y * blockDim.y + threadIdx.y;
```

Ščepec smo napisali tako, da ga lahko zaženemo tudi z mrežo niti, ki ima manj kot $(N_a)^2$ niti. Takrat vsaka nit po potrebi izračuna in sešteje prispevek energije za več parov atomov. To stori tako, da v zanki indeksoma `i` in `j` prišteva število niti v ustrezni dimenziji mreže.

Sam izračun prispevka energije para atomov je povsem enak kot v zaporedni izvedbi. Paziti moramo le, da ignoriramo prispevke energije med atomi, ki so del iste molekule. V ta namen uporabimo polje `atom_to_particle`, da dobimo indeksa molekul, ki ju nato primerjamo.

Nato moramo vse prispevke energij še sešteti. Najbolj učinkovito je, če tudi to storimo na grafični procesni enoti. Tako je nazaj na centralno procesno enoto potrebno prenesti le eno vrednost. Uporabimo vzporedno redukcijo, opisano v naslednjem poglavju.

Na koncu moramo rezultat še deliti z dve, saj smo prispevek energije vsakega para atomov šteli dvakrat. Namesto tega bi lahko sešteli le prispevke energije unikatnih parov atomov, kot smo to storili v zaporedni izvedbi. Če bi uporabili še preslikovalni algoritem, bi potrebovali le polovico niti kot sicer [12]. Ker pa se izračun celotne energije sistema izvede samo enkrat tekom celotne simulacije, se za ta pristop nismo odločili. Raje smo se posvetili optimizaciji računanja sprememb energije v vsakem koraku simulacije.

4.2.3 Vzporedna redukcija

Vzporedna redukcija (angl. *parallel reduction*) je način, kako zreduciramo seznam vrednosti v eno samo vrednost. Če seznam vsebuje n vrednosti, ga zreducira v $\lceil \log_2 n \rceil$ korakih. Za primerjavo, navadno reduciranje z zapore-

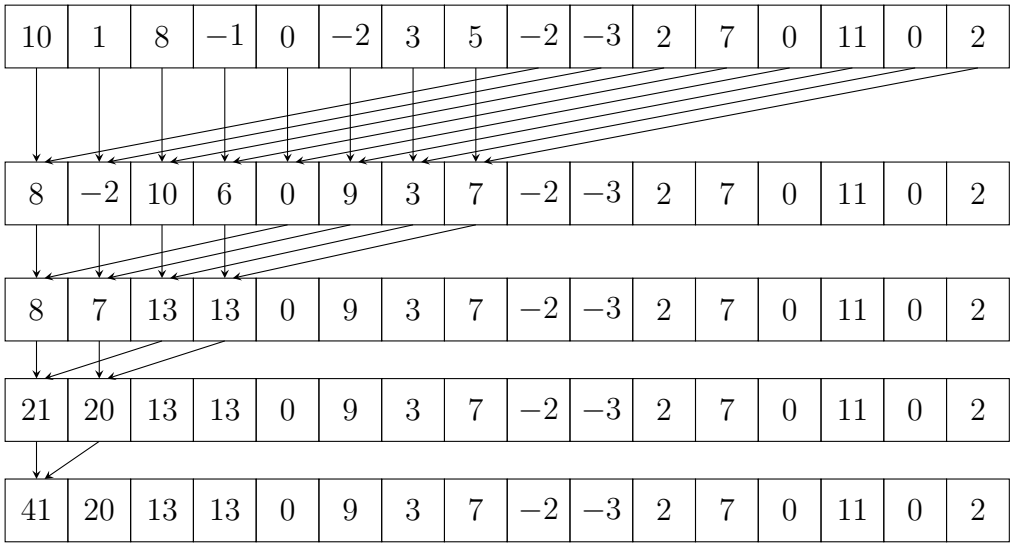
dnim iteriranjem čez elemente seznama potrebuje $n - 1$ korakov.

Izsek kode 4.1: Vzporedna redukcija. Izsek kode je del ščepca za izračun začetne energije sistema.

```
real_t energy; // The result of this thread-s computation
uint32_t n_threads_in_block = blockDim.x * blockDim.y;
uint32_t tid = threadIdx.y * blockDim.x + threadIdx.x;

// Reduce the calculated energy in each block
extern __shared__ real_t sh_energy[]; // shared memory
reduce_block(sh_energy, energy, tid, n_threads_in_block);

// Save sum of this block's energy to global memory
uint32_t bid = blockIdx.y * gridDim.x + blockIdx.x;
if (tid == 0)
  block_energies[bid] = sh_energy[0];

// Reduce energy across all blocks
__device__ uint32_t retired_cnt = 0;
uint32_t n_blocks = gridDim.x * gridDim.y;
if (n_blocks > 1)
  reduce_global(block_energies, sh_energy, &retired_cnt, tid,
    n_threads_in_block, n_blocks);
```

Vzporedno redukcijo, ki smo jo uporabili v naši simulaciji, lahko vidimo na izsekih kode 4.1 in 4.2. Najprej izvedemo redukcijo znotraj blokov niti. Pri tem vmesne rezultate shranjujemo v deljeni pomnilnik in uporabljamo sinhronizacijo niti v bloku (`__syncthreads()`). Uporabili smo način redukcije z zaporednim naslavljanjem, prikazan na sliki 4.2 [13]. S tem se izognemo razhajanju niti in konfliktom pri dostopu do deljenega pomnilnika. Ko je ta postopek končan, je vsota energije v vsakem bloku shranjena v prvem elementu njegovega deljenega pomnilnika, `sh_energy[0]`.

Sedaj moramo sešteti še rezultate vseh blokov. Tu nastopi težava, saj CUDA omogoča le sinhronizacijo med nitmi v istem bloku, ne pa tudi sinhronizacije med bloki. Problem bi lahko rešili z zagonom ločenega ščepca s samo enim blokom niti ali pa prenosom podatkov na centralno procesno enoto in navadno redukcijo. Mi smo uporabili način sinhronizacije blokov z globalno spremenljivko in atomskimi ukazi [27]. Najprej vse rezultate blokov prenesemo iz deljenega v globalni pomnilnik. Nato prva nit v vsakem bloku z



Slika 4.2: Shema vzporedne redukcije. Vsoto 16-ih vrednosti v seznamu izračunamo v $log_2 16 = 4$ korakih. Povzeto po [13].

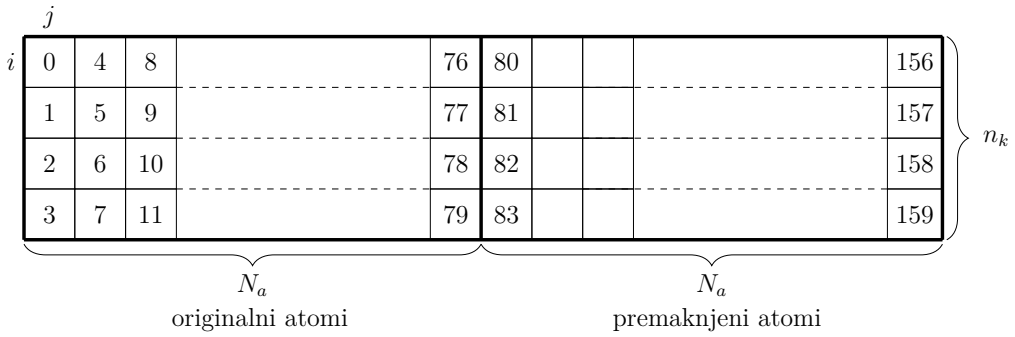
atomskim ukazom `atomicInc` poveča števec zaključenih blokov. Če je vrednost števca enaka številu vseh blokov, je blok, ki se trenutno izvaja, zadnji. Zadnji blok nato prenese rezultate iz globalnega pomnilnika v svoj deljeni pomnilnik in jih zreducira po že znanem postopku. Rezultat na koncu shrani v globalni pomnilnik, da ga lahko prenesemo nazaj na centralno procesno enoto. V tem postopku je še posebej pomembna pravilna uporaba pregrad. Takšen algoritem je najučinkovitejši vzporedni način za izračun vsote velikega seznama vrednosti na grafični procesni enoti [12].

4.2.4 Glavni del algoritma

Glavni del algoritma ostane zelo podoben algoritmu 3.3. Gnezdeni zanki in generiranje naključnega premika so nespremenjeni. Izračun premaknjenih koordinat atomov pa smo prestavili na grafično procesno enoto. V ločenem ščepcu skopiramo globalne koordinate vseh atomov $\boldsymbol{x}_\mathrm{a}$ v pomožno polje $\boldsymbol{x}'_\mathrm{a}$. Pri tem za atome iz premaknjene molekule izračunamo nove koordinate.

Izsek kode 4.2: Pomožne funkcije za bločno in globalno vzporedno redukcijo.

```
template<class T> __device__ void
reduce_block(volatile T* sh_data, T my_sum, uint32_t tid, uint32_t
    n_threads_in_block)
{
  sh_data[tid] = my_sum; // Save this thread's sum to shared memory
  __syncthreads();

  for (uint32_t stride = n_threads_in_block/2; stride > 0; stride >>= 1){
    if (tid < stride) {
      my_sum += sh_data[tid + stride];
      sh_data[tid] = my_sum;
    }
    __syncthreads();
  }
}

template<class T> __device__ void
reduce_global(volatile T* gl_data, volatile T* sh_data, unsigned*
    retirement_count, uint32_t tid, uint32_t n_threads_in_block, uint32_t
    n_blocks)
{
  __shared__ bool am_last;

  // Wait for all outstanding memory instructions in this thread
  __threadfence();

  // Thread 0 takes a ticket
  if (tid == 0) {
    uint32_t ticket = atomicInc(retirement_count, n_blocks) + 1;
    am_last = ticket == n_blocks;
  }
  __syncthreads();

  // The last block sums the results of all other blocks
  if (am_last) {
    // Each thread now handles the sum of a block.
    // Handle the case where there are fewer threads than blocks.
    T my_sum = 0;
    for (uint32_t i = tid; i < n_blocks; i += n_threads_in_block)
      my_sum += gl_data[i];

    reduce_block(sh_data, my_sum, tid, n_threads_in_block);

    if (tid == 0) {
      gl_data[0] = sh_data[0]; // Save result to global memory
      *retirement_count = 0; // Reset retirement count for next run
    }
  }
}
```



Slika 4.3: Shema razdelitve računanja prispevkov energije med niti. V kvadratih so zapisani globalni identifikatorji niti *gtid*.

Na grafično procesno enoto smo premaknili tudi računanje sprememb energije sistema. Izračunati moramo prispevke energije med atomi v premaknjeni molekuli in vsemi preostalimi atomi. Da lahko izračunamo spremembo energije, moramo to storiti dvakrat, z originalnimi in s premaknjenimi koordinatami atomov. Uporabili smo $2\,N_a\,n_k$ niti, ki smo jih organizirali kar v enodimenzionalno mrežo in bloke. Globalni identifikator niti je definiran z $gtid = bid \cdot n_{threads} + tid$, kjer je *bid* indeks bloka, *tid* indeks niti v bloku in $n_{threads}$ število niti v bloku. Shemo razdelitve računanja prispevkov energije lahko vidimo na sliki 4.3. Da v ščepcu ugotovimo, za kateri par atomov je odgovorna vsaka nit, uporabimo enačbe

$$moved = gtid \geq (N_a \cdot n_k) \quad , \tag{4.1}$$

$$i = (gtid \bmod n_k) + i_{\mathrm{start},k} \quad \text{in} \tag{4.2}$$

$$j = (gtid/n_k) \bmod N_a \quad . \tag{4.3}$$

Število niti je lahko v vsakem koraku drugačno, saj je odvisno od števila atomov v premaknjeni molekuli.

Razdaljo med atomoma in prispevek energije izračunamo na enak način kot v zaporedni izvedbi. Koordinate atomov vzamemo iz ustreznega polja, bodisi polja $\boldsymbol{x}_{\mathrm{a}}$ bodisi pomožnega polja $\boldsymbol{x}'_{\mathrm{a}}$. Prispevke energije med atomi, ki so del iste molekule, ignoriramo. Če računamo prispevek energije pred premikom molekule, ga zaradi enačbe (2.1) pomnožimo z -1. Ko bomo na

koncu prispevke sešteli, bo zato rezultat že vseboval spremembo energije. Če se atoma prekrivata, prispevek energije nastavimo na vrednost NaN, kot v zaporedni izvedbi. Računanja pa ne moremo takoj prekiniti, saj se sočasno izvaja veliko niti. Vrednost NaN se pri seštevanju propagira, zato lahko pri odločanju še vedno preverimo, ali je pri premiku prišlo do prekrivanja.

V istem ščepcu izvedemo tudi posodabljanje histograma. Uporabiti moramo atomske ukaze, saj lahko več niti sočasno dostopa do istega intervala histograma. Da zmanjšamo število konfliktov, histogram najprej posodabljamo v deljenem pomnilniku. Pred koncem ščepca pa vrednosti tega histograma prištejemo k vrednostim histograma v globalnem pomnilniku.

Na koncu ščepca izvedemo še vzporedno redukcijo, da seštejemo rezultate vseh niti. Uporabimo povsem enak postopek kot pri računanju začetne energije sistema. Rezultat prenesemo nazaj na centralno procesno enoto.

Med simulacijo na in z grafične procesne enote prenašamo le najnujnejše podatke. To so predvsem izračunana sprememba energije in podatki o premaknjeni molekuli. Vsi ostali prenosi se izvedejo bodisi ob začetku bodisi ob zaključku simulacije.

Sprejemanje oziroma zavračanje poskusa premika je povsem enako kot v zaporedni izvedbi. Pri sprejetem premiku posodobimo globalne koordinate atomov tako, da zamenjamo kazalca na polji $\boldsymbol{x}_\mathrm{a}$ in $\boldsymbol{x}'_\mathrm{a}$ v globalnem pomnilniku.

Ko je simulacije konec, moramo iz grafične procesne enote prenesti še histogram in globalne koordinate atomov, da jih lahko shranimo.

4.3 Optimizirana izvedba

Izračuna začetnega stanja energije nismo spreminjali, saj se izvede samo enkrat tekom simulacije in ne vpliva bistveno na čas izvajanja. Osredotočili smo se na optimizacijo glavnega dela algoritma, saj ta traja najdlje. V osnovni vzporedni izvedbi smo v vsakem koraku algoritma izvedli dva klica ščepca, in sicer enega za posodobitev koordinat premaknjenih atomov in drugega za

izračun spremembe energije. Zelo dobro bi bilo, če bi lahko izvedli samo en ščepec na vsak cikel algoritma. To pomeni, da bi v ščepec prestavili celotno notranjo zanko algoritma 3.3, z generiranjem poskusov premikov molekul in odločanjem o sprejetju premikov vred.

Ta zanka v ščepcu ne more biti popolnoma vzporedna, saj je vsak korak algoritma odvisen od rezultata prejšnjega koraka. Lahko bi v vsakem koraku uporabili podoben način globalne sinhronizacije kot pri vzporedni redukciji. To ni najbolj učinkovito, saj bi vzporedno računali le spremembo energije, odločanje pa bi opravljala le ena nit. Liang *et al.* so razvili boljši način [20]. Ugotovili so, da lahko nekatere prispevke energije seštejemo že preden izvedemo odločitev o sprejetju poskusa premika iz prejšnjega koraka.

Predpostavimo, da simuliramo samo atome. Ker uporabljamo zaporedno izbiranje, v koraku i poskušamo premakniti atom i. Nove koordinate atoma lahko generiramo vzporedno. Da izračunamo spremembo energije, ki jo povzroči ta premik, moramo izračunati razdalje med atomom i in vsemi preostalimi atomi j. Izračunati moramo razdalje pred premikom, r_{ij}, in razdalje po premiku, r'_{ij}. Uporabimo enačbi

$$r_{ij} = \begin{cases} dist(\boldsymbol{x}_i, \boldsymbol{x}_j), & j > i \\ dist(\boldsymbol{x}_i, \widetilde{\boldsymbol{x}}_j), & j < i \end{cases} \quad \text{in} \quad r'_{ij} = \begin{cases} dist(\boldsymbol{x}'_i, \boldsymbol{x}_j), & j > i \\ dist(\boldsymbol{x}'_i, \widetilde{\boldsymbol{x}}_j), & j < i \end{cases} . \tag{4.4}$$

Z $\boldsymbol{x}_i$ in $\boldsymbol{x}_j$ smo označili nepremaknjene koordinate atomov i in j, z $\boldsymbol{x}'_i$ premaknjene koordinate atoma i in z $\widetilde{\boldsymbol{x}}_j$ koordinate atoma j, ki so rezultat odločitve iz prejšnjih korakov algoritma. Pomembna ugotovitev je, da za $j > i$ razdalje r_{ij} in r'_{ij} niso odvisne od rezultatov prejšnjih korakov. Zato jih lahko takoj vzporedno izračunamo. Vzporedno lahko izračunamo tudi delne vsote spremembe energij za vsak korak, $\Delta E_i = \sum_{j>i} \left[U_{\mathrm{LJ}}(r'_{ij}) - U_{\mathrm{LJ}}(r_{ij}) \right]$. Za prvi korak, $i = 0$, je ta delna vsota že dejanska sprememba energije. Zato lahko sprejmemo odločitev o poskusu premika in dobimo $\widetilde{\boldsymbol{x}}_0$. Nato lahko vzporedno izračunamo še vse razdalje $r_{i\,0}$ in $r'_{i\,0}$, kjer je $i > 0$. Na podlagi teh razdalj izračunamo spremembe energije in jih prištejemo ustrezni delni vsoti ΔE_i. S tem ΔE_1 postane dejanska sprememba energije v koraku 1 in lahko sprej-

memo odločitev o poskusu premika atoma 1. Postopek ponavljamo, dokler niso sprejete odločitve o poskusih premikov vseh atomov.

Ta algoritem smo prilagodili tako, da je možno simulirati tudi molekule. Odločanje o premiku vsake molekule mora namreč izvesti le ena od niti, odgovornih za atome v njej. Ta mora od ostalih niti pridobiti spremembe energij, ki so jih izračunale, in z njimi deliti rezultat odločitve. Paziti je potrebno tudi, da ne upoštevamo interakcij med atomi v isti molekuli. Dodatno oviro predstavlja razdelitev niti na bloke. Niti, ki so odgovorne za atome v eni molekuli, morajo namreč biti v istem bloku, da jih lahko med seboj sinhroniziramo.

4.3.1 Priprava podatkov

Priprava podatkov je podobna pripravi podatkov naše prve vzporedne izvedbe iz poglavja 4.2. Tokrat ne potrebujemo pomožnih polj za premaknjene koordinate atomov $\boldsymbol{x}'_{\mathrm{a}}$ in vmesne vsote energij blokov. Potrebujemo pa nekatere druge podatkovne strukture.

Ker lahko vsaka molekula vsebuje različno število atomov, je lahko vsak blok niti odgovoren za različno število molekul. Zato izračunamo in na grafično kartico prenesemo polji B_{start} in B_{n}. Polje B_{start} za vsak blok niti vsebuje indeks prvega atoma v prvi molekuli, za katero je ta blok niti odgovoren. Polje B_{n} za vsak blok niti vsebuje število atomov, za katere je ta blok niti odgovoren. Za sinhronizacijo med bloki niti rezerviramo še polje B_{state} in vse njegove elemente nastavimo na nič.

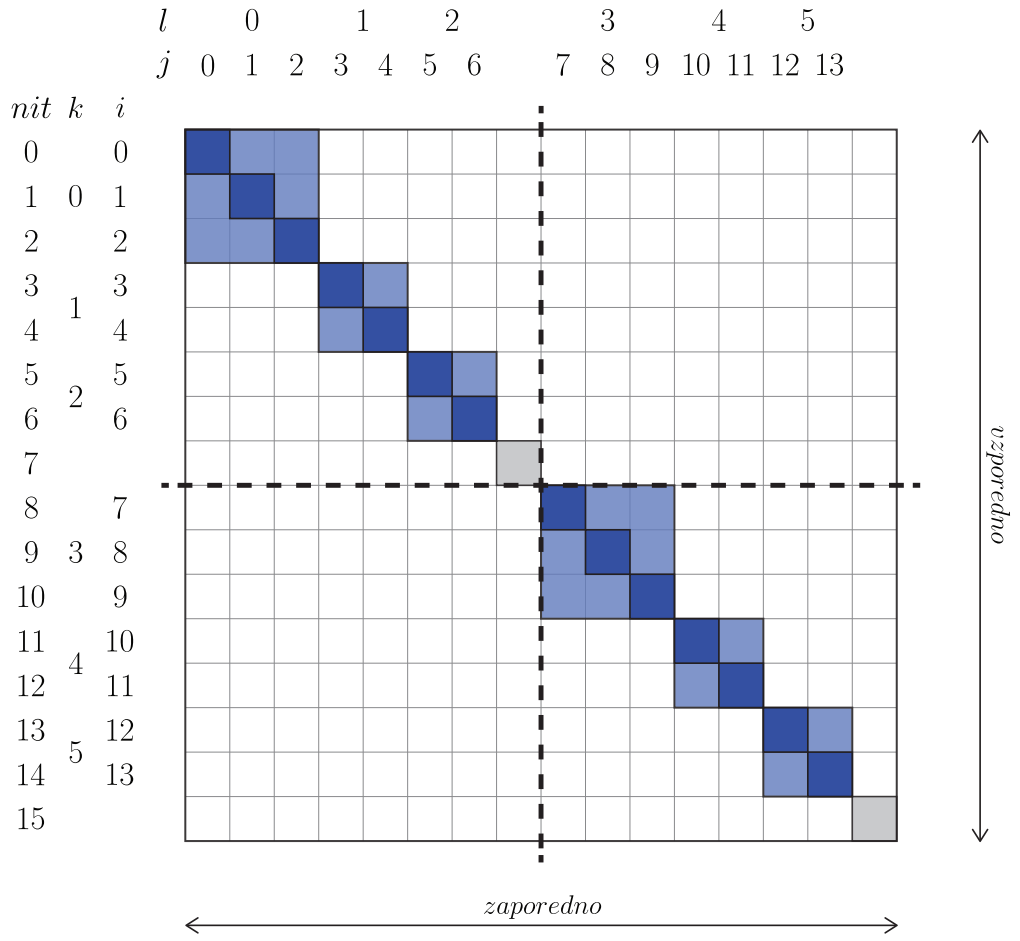
Ker smo premikanje molekul in odločanje o sprejetju premikov prestavili na grafično kartico, moramo tja prenesti tudi vse koordinate molekul $\boldsymbol{x}_{\mathrm{m}}$ in vse orientacije molekul $\boldsymbol{R}$. Rezerviramo tudi prostor za naključna števila, ki jih potrebujemo za generiranje naključnih premikov in odločanje. V globalnem pomnilniku potrebujemo tudi prostor za spremenljivke, v katere shranjujemo rezultat. To so E, E_{sum}, n_{acc} in n_{rej}.

4.3.2 Glavni del algoritma

Večino računanja smo prestavili na grafično procesno enoto, s čimer se znebimo notranje zanke v algoritmu 3.3. Na centralni procesni enoti se izvajata samo še priprava podatkov in zunanja zanka, ki šteje cikle simulacije. V vsakem obhodu te zanke najprej pripravimo polje naključnih števil in jih prenesemo v globalni pomnilnik grafične procesne enote. Za vsako molekulo potrebujemo sedem naključnih vrednosti: tri za translacijo, tri za rotacijo in eno za odločanje. Nato zaženemo ščepec, ki z algoritmom 4.6 izvede poskus premika in odločanje za vseh N molekul. Med izvajanjem tega ščepca lahko že pripravimo polje naključnih števil za naslednji cikel. CUDA namreč podpira sočasno izvajanje kode na centralni procesni enoti in grafičnih procesnih enotah. Zaradi kompleksnosti algoritma in sinhronizacije med nitmi smo posodabljanje histogramov premaknili v ločen ščepec, ki ga zaženemo vsakih ν_H ciklov.

Za izvedbo algoritma potrebujemo vsaj N_a niti, ki jih organiziramo v enodimenzionalno mrežo in bloke. Vsaka nit je odgovorna za izračun spremembe energije, ki jo povzroči premik enega atoma. Nit, ki je odgovorna za prvi atom v molekuli, je odgovorna še za odločanje o sprejetju poskusa premika molekule. Shemo razdelitve dela lahko vidimo na sliki 4.4. Niti so razdeljene na enako velike bloke. Ker se meje med bloki ne ujemajo vedno z mejami med molekulami, je lahko v vsakem bloku nekaj niti odveč. Molekule so med bloke namreč razdeljene tako, da so niti, odgovorne za atome v isti molekuli, vedno v istem bloku. Vsako nit identificiramo z indeksoma *bid* in *tid*, kjer je *bid* indeks bloka, ki mu nit pripada, *tid* pa indeks niti znotraj bloka *bid*.

Na začetku algoritma 4.6 vsaka nit v svoje registre naloži podatke o atomu, za katerega je odgovorna, in molekuli, ki ta atom vsebuje. V registre niti shranimo tudi spremenljivko ΔE, ki bo akumulirala vsoto spremembe energije premika atoma. Nato vsaka nit generira naključen premik svoje molekule, da dobimo $\boldsymbol{x}'_{\mathrm{m}}$ in $\boldsymbol{R}'$. Pri tem niti, ki pripadajo atomom iste molekule, uporabijo ista naključna števila. Tako je generiran premik enak za vse atome



Slika 4.4: Shema razdelitve dela in interakcij med atomi za 6 molekul z različnim številom atomov. Vsaka nit je odgovorna za atom i, ki je del molekule k. Izračuna in sešteje spremembe energije, ki so posledica interakcij z ostalimi atomi j v molekulah l. Črtkane črte predstavljajo razdelitev niti na bloke. Niti 7 in 15 ostaneta brez dodeljenih atomov.

Algoritem 4.6 Psevdokoda ščepca za en cikel Metropolisovega algoritma.

▷ Lokalne spremenljivke niti
1: $tid \leftarrow$ indeks niti v bloku, $bid \leftarrow$ indeks bloka niti
2: $n_{\text{blocks}} \leftarrow$ število blokov
3: $i \leftarrow B_{\text{start},\,bid} + tid$ ▷ indeks atoma, za katerega je ta nit odgovorna
4: $k \leftarrow$ indeks molekule, ki vsebuje atom i
5: $i_{\text{end}} \leftarrow i_{\text{start},k} + n_k - 1$ ▷ indeks zadnjega atoma v molekuli
6: $i'_{\text{end}} \leftarrow i_{\text{end}} - B_{\text{start},\,bid}$ ▷ i_{end}, relativen na začetek bloka
7: $i'_{\text{start}} \leftarrow i_{\text{start},k} - B_{\text{start},\,bid}$ ▷ $i_{\text{start},k}$, relativen na začetek bloka
8: $\boldsymbol{x}_{\text{a}} \leftarrow \boldsymbol{x}_{\text{a},i}$
9: $\Delta E \leftarrow 0$
10: $\boldsymbol{x}'_{\text{m}}$, $\boldsymbol{R}'$, $\boldsymbol{x}'_{\text{a}} \leftarrow$ generiraj naključni premik molekule k

11: CalculateUpperTriangle()
12: CalculateLowerTriangleBlocks()
13: DecisionAndLowerTriangle()

14: **if** $bid = n_{\text{blocks}} - 1$ **then**
15: nastavi vse elemente B_{state} na 0 ▷ ponastavimo B_{state} za naslednji zagon
16: **end if**

v molekuli. Ker lahko vse niti dejansko izvedejo isto kodo, se nismo odločili za enkraten izračun premikov in deljenja le teh preko deljenega pomnilnika. Vsaka nit nato izračuna še premaknjene koordinate $\boldsymbol{x}'_{\mathrm{a}}$ svojega atoma. Podatke $\boldsymbol{x}'_{\mathrm{m}}$, $\boldsymbol{R}'$ in $\boldsymbol{x}'_{\mathrm{a}}$ niti shranijo v svoje registre. Sledi vzporedno računanje sprememb energije.

Najprej lahko vzporedno izračunamo vse spremembe energije v zgornjem trikotniku na sliki 4.4. Vsaka nit i k ΔE prišteje spremembe energij ΔE_{ij}, ki so posledica interakcij med atomi i in j, kjer je $j > i$. Uporabimo algoritma 4.7 in 4.8.

Algoritem 4.7 Psevdokoda za izračun sprememb energij za zgornji trikotnik na sliki 4.4.

```
1:  procedure CalculateUpperTriangle()
2:      for b = bid + 1 to n_blocks − 1 do                ▷ interakcije v blokih b > bid
3:          ΔE ← ΔE + BlockDeltaEnergy(b)
4:      end for

5:      X[tid] ← x_a                                      ▷ prenos v deljeni pomnilnik
6:      sinhroniziraj niti v bloku
7:      W ← število niti v snopu
8:      for t = ⌊tid/W⌋ · W + 1 to B_{n, bid} − 1 do      ▷ interakcije v bloku bid
9:          if t > i'_end then                            ▷ z molekulami l > k
10:             ΔE ← ΔE + DeltaEnergy(x_a, x'_a, X[t])
11:         end if
12:     end for
13:     sinhroniziraj niti v bloku
14: end procedure
```

Najprej upoštevamo interakcije z atomi v blokih b, kjer je $b > bid$, nato pa še interakcije z atomi v bloku bid. Ko računamo interakcije z atomi v bloku b, prenesemo njihove lokacije iz globalnega pomnilnika v polje X v deljenem pomnilniku, saj bodo do njih dostopale vse niti v bloku. Nato z zanko v algoritmu 4.8 v vrstici 15 izračunamo in seštejemo spremembe energij, ki so posledica interakcij med atomom i in vsemi atomi v bloku b. Vse niti v bloku začnejo zanko pri istem indeksu in zato hkrati dostopajo do istih

Algoritem 4.8 Psevdokoda za izračun spremembe energije, ki je posledica premika atoma i. Prvi del upošteva interakcije atoma i z atomom j, drugi del pa interakcije atoma i z blokom atomov b.

```
1: function DeltaEnergy(x_{a,i}, x'_{a,i}, x_{a,j})
2:     σ ← σ(τ_i, τ_j), ϵ ← ϵ(τ_i, τ_j)
3:     r^2 ← dist^2(x'_{a,i}, x_{a,j})
4:     E_new ← LJ(r^2, σ, ϵ) if r^2 > r^2_min(τ_i, τ_j) else NaN
5:     E_old ← LJ(dist^2(x_{a,i}, x_{a,j}), σ, ϵ)
6:     return E_new − E_old
7: end function

8: function BlockDeltaEnergy(b)                ▷ b je indeks bloka, b ≠ bid
9:     j ← B_{start, b} + tid
10:    if j < N_a then
11:        X[tid] ← x_{a,j}                    ▷ prenos v deljeni pomnilnik
12:    end if
13:    sinhroniziraj niti v bloku
14:    ΔE ← 0
15:    for t = 0 to B_{n, b} − 1 do            ▷ za vsak atom v bloku b
16:        ΔE ← ΔE + DeltaEnergy(x_a, x'_a, X[t])
17:    end for
18:    sinhroniziraj niti v bloku
19:    return ΔE
20: end function
```

elementov polja X v deljenem pomnilniku. Tako se izognemo konfliktom pri dostopu do deljenega pomnilnika in izkoristimo njegovo funkcijo raztrosa (angl. *broadcast*). Če sta si atoma preblizu, ponovno uporabimo vrednost NaN.

Nato na podoben način upoštevamo še interakcije znotraj bloka *bid*, kjer moramo paziti na meje med molekulami. Koordinate atomov v tem bloku prenesemo v deljeni pomnilnik. Z zanko v algoritmu 4.7 v vrstici 8 izračunamo in seštejemo spremembe energij, ki so posledica interakcij med atomom i in atomi v molekulah l v bloku *bid*, kjer je $l > k$. Zanka se začne z indeksom $\lfloor tid/W \rfloor \cdot W + 1$ zato, da vse niti v istem snopu dostopajo do istih podatkov v deljenem pomnilniku.

Preostane nam še izračun sprememb energije v spodnjem trikotniku na sliki 4.4 in odločanje o sprejetju poskusa premika. Ker je vsak korak algoritma odvisen od rezultata prejšnjega koraka, moramo počakati, da se predhodni bloki niti zaključijo. Uporabimo polje B_{state} v glavnem pomnilniku, kjer vrednost 1 v $B_{\text{state},\,b}$ pomeni, da je blok niti b zaključil z računanjem in lahko uporabimo njegove rezultate. Prispevke energije, ki so posledica interakcij z atomi v blokih b, kjer je $b < bid$, izračunamo z algoritmom 4.9. V zanki iteriramo čez te bloke in vsakič preverimo B_{state}, da ugotovimo če so rezultati bloka že na voljo. Če niso, počakamo. Preverjanje izvaja v gnezdeni zanki prva nit v bloku, čakajo pa vse niti v bloku. Prvemu bloku niti tega seveda ni potrebno storiti, saj nima predhodnih blokov.

Za odločanje in izračun preostalih interakcij znotraj bloka *bid* uporabimo algoritem 4.10. Najprej v deljeni pomnilnik v polji X in Y prenesemo lokacije atomov v tem bloku in delne vsote sprememb energije, ki jih je izračunala vsaka nit. Lokacije potrebujemo za izračun preostalih interakcij, delne vsote sprememb energije pa za sprejem odločitve. Pravilen vrstni red odločanja znotraj bloka zagotovimo z zanko čez vse atome v bloku in sinhronizacijo niti na primernih mestih v algoritmu.

Algoritem 4.9 Psevdokoda za izračun sprememb energij za bloke v spodnjem trikotniku na sliki 4.4.

```
1: procedure CalculateLowerTriangleBlocks()
2:     for b = 0 to bid − 1 do                    ▷ interakcije v blokih b < bid
3:         if tid = 0 then
4:             repeat wait until B_state, b = 1
5:         end if
6:         sinhroniziraj niti v bloku
7:         ΔE ← ΔE + BlockDeltaEnergy(b)
8:     end for
9: end procedure
```

Ker vsaka molekula lahko vsebuje več atomov, moramo sešteti delne vsote sprememb energije, ki pripadajo atomom v tej molekuli. To stori nit, ki je odgovorna za prvi atom v molekuli. Uporabi delne vsote iz polja Y v deljenem pomnilniku. Tako dobimo končno spremembo energije, ki nastane zaradi premika molekule. Ista nit lahko nato sprejme odločitev o sprejetju poskusa premika molekule na enak način kot v zaporedni izvedbi. Glede na rezultat odločitve, nit v globalnem pomnilniku ustrezno posodobi spremenljivke E, E_{sum}, n_{acc} in n_{rej}, ki hranijo rezultat simulacije. Če je bil poskus premika molekule sprejet, nit shrani novo lokacijo in orientacijo molekule v globalni pomnilnik. Rezultat odločitve shranimo v spremenljivko v deljenem pomnilniku, ker ga potrebujejo ostale niti, odgovorne za atome v tej molekuli. Te, če je bil poskus premika molekule sprejet, shranijo nove koordinate svojega atoma na ustrezno mesto v polju X.

Po sprejetju odločitve o poskusu premika molekule k so na voljo nove koordinate vseh atomov v molekuli k. Niti, ki so v tem bloku odgovorne za atome v molekulah l, kjer je $l > k$, lahko te koordinate sedaj uporabijo. Izračunajo in seštejejo še preostale spremembe energij, ki so posledica interakcij med atomi molekul k in l. S temi rezultati posodobijo delne vsote v polju Y v deljenem pomnilniku. V naslednjem obhodu zanke ima nit, odgovorna za prvi atom v molekuli $k+1$, vse potrebne podatke, da lahko sešteje relevantne delne vsote in sprejme odločitev. Ta postopek se nato ponavlja, dokler ne

Algoritem 4.10 Psevdokoda za odločanje o poskusu premika in izračun sprememb energij za spodnji trikotnik na sliki 4.4 znotraj bloka *bid*.

```
 1: procedure DecisionAndLowerTriangle()
 2:     X[tid] ← x_a, Y[tid] ← ΔE
 3:     sinhroniziraj niti v bloku
 4:     for t = 0 to B_{n,bid} − 1 do
 5:         if t = i'_start and t = tid then                ▷ prvi atom v molekuli k
 6:             for u = 1 to n_k − 1 do
 7:                 ΔE ← ΔE + Y[i'_start + u]
 8:             end for
 9:             accept ← Decide(ΔE)          ▷ odločitev shranimo v deljeni pomnilnik
10:             if accept then
11:                 E ← E + ΔE
12:                 n_acc ← n_acc + 1
13:                 x_{m,k}, R_k ← x'_m, R'
14:             else
15:                 n_rej ← n_rej + 1
16:             end if
17:             E_sum ← E_sum + E
18:         end if
19:         sinhroniziraj niti v bloku
20:         if t = i'_start and tid ≥ i'_start and tid ≤ i'_end and accept then
21:             X[tid] ← x'_a
22:         end if
23:         sinhroniziraj niti v bloku
24:         if t < i'_start and t < tid then
25:             ΔE ← ΔE + DeltaEnergy(x_a, x'_a, X[t])
26:             Y[tid] ← ΔE
27:         end if
28:         sinhroniziraj niti v bloku
29:     end for
30:     if tid < B_{n,bid} then
31:         x_{a,i} ← X[tid]
32:     end if
33:     sinhroniziraj niti v bloku
34:     if tid = 0 then
35:         B_{state,bid} = 1
36:     end if
37: end procedure
```

sprejmemo vseh odločitev o poskusih premikov molekul v tem bloku.

Ko je ta postopek zaključen, moramo še prenesti nove koordinate atomov iz deljenega pomnilnika v globalni pomnilnik in nastaviti $B_{\text{state},\,bid}$ na vrednost 1. Tako bodo naslednji bloki lahko nadaljevali računanje. Ko zadnji blok zaključi z računanjem, le-ta ponastavi vrednosti v polju B_{state} na nič in ga s tem pripravi za naslednji zagon ščepca.

Uporabljena sinhronizacija med bloki niti s poljem zastavic B_{state} predstavlja nevarnost mrtvih zank (angl. *deadlock*). Do teh lahko pride, če vsi bloki, ki se trenutno izvajajo na grafični procesni enoti, čakajo na ustrezno stanje B_{state}. Blok, ki bi nastavil ustrezno stanje B_{state}, pa se ne more začeti izvajati, ker je grafična procesna enota zasedena s čakajočimi bloki. V naši izvedbi se temu lahko izognemo, saj je vsak blok odvisen zgolj od prejšnjih blokov. Opozoriti velja, da CUDA ne zagotavlja vrstnega reda pri razporejanju blokov. Le-ti se izvajajo v poljubnem vrstnem redu [27]. Izkaže pa se, da trenutne naprave bloke v enodimenzionalnih mrežah razporejajo zaporedno, v naraščajočem vrstnem redu. Zato v naši izvedbi ni težav z mrtvimi zankami. Če se razporejanje blokov v prihodnosti spremeni, bi morali ustrezno prilagoditi indeks *bid*. Namesto `blockIdx.x` bi morali uporabiti globalni števec, ki bi ga v vsakem bloku povečali z atomskimi ukazi. Podoben način smo uporabili pri vzporedni redukciji v poglavju 4.2.

Poglavje 5

Meritve in rezultati

5.1 Testno okolje

Vse čase izvajanja smo merili na delovni postaji s procesorjem Intel Core i7-5960X, 32 GB delovnega pomnilnika in grafično kartico NVIDIA GeForce GTX 1080 Ti. Procesor ima osem fizičnih jeder in šestnajst niti z osnovno frekvenco 3 GHz in maksimalno frekvenco 3,5 GHz. Podrobnosti o grafični kartici lahko vidimo na izseku kode 5.1.

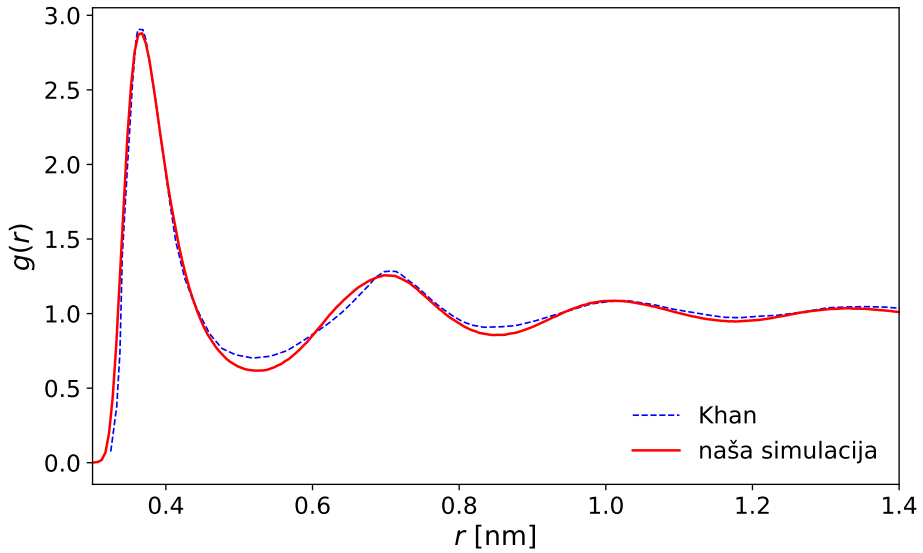
Izsek kode 5.1: Izsek izpisa programa CUDA Device Query, ki prikazuje lastnosti grafične kartice NVIDIA GeForce GTX 1080 Ti.

```
Device 0: "GeForce GTX 1080 Ti"
  CUDA Driver Version / Runtime Version          9.1 / 9.1
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 11264 MBytes
  (28) Multiprocessors, (128) CUDA Cores/MP:     3584 CUDA Cores
  GPU Max Clock rate:                            1633 MHz (1.63 GHz)
  Memory Clock rate:                             5505 Mhz
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
```

5.2 Pravilnost delovanja

Da smo se prepričali o pravilnosti delovanja naših izvedb, smo rezultate naših simulacij primerjali z rezultati iz literature in z rezultati simulacij, izvedenih z referenčnim programom MOLSIM [29].

V vseh treh naših izvedbah uporabljamo isto seme za generiranje psevdonaključnih števil. Pri istih vhodnih podatkih z vsemi izvedbami dobimo enake rezultate. Zato smo simulacije, potrebne za primerjavo, izvajali le z optimizirano vzporedno izvedbo. Na začetku vseh simulacij so molekule v simulacijsko škatlo razporejene povsem naključno.

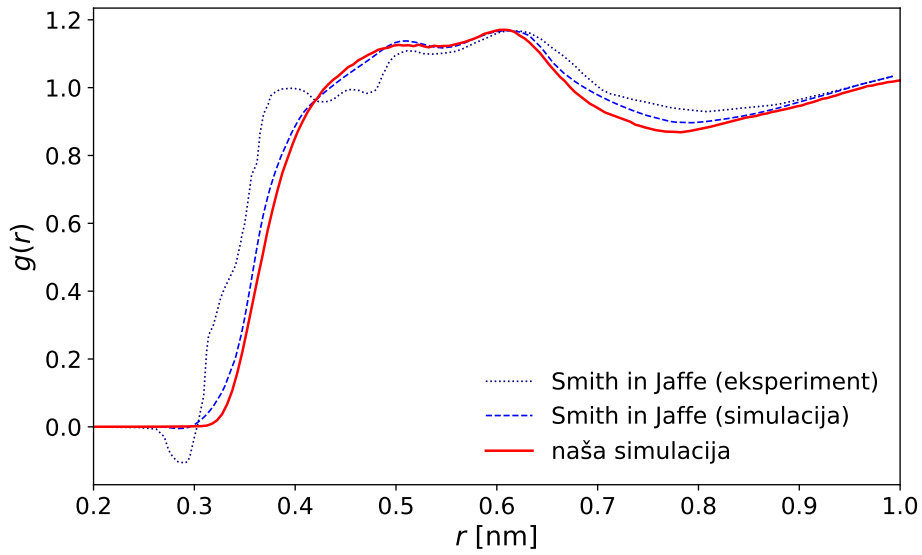


Slika 5.1: Graf radialne porazdelitvene funkcije argona. Primerjamo rezultate naše simulacije in rezultate Khana [17]. Parametri naše simulacije: $T = 84{,}4\,\mathrm{K}$, $\sigma = 0{,}3345\,\mathrm{nm}$ [36], $\epsilon = 1{,}045\,\mathrm{kJ/mol}$ [36], $L = 3{,}62\,\mathrm{nm}$, $N_a = 1000$ in $N_{MC} = 20000$.

Najprej smo preverili delovanje simulacije enoatomnih molekul. Khan je izračunal radialne porazdelitvene funkcije za tekoči argon pri različnih temperaturah [17]. Primerjavo z rezultati naše simulacije lahko vidimo na sliki 5.1. Radialni porazdelitveni funkciji sta si zelo podobni. Njuni ekstremi so na istih mestih, vrednosti v ekstremih pa so pri naši simulaciji malce

nižje. Manjša odstopanja smo zaradi možnih razlik v simulacijskih modelih pričakovali.

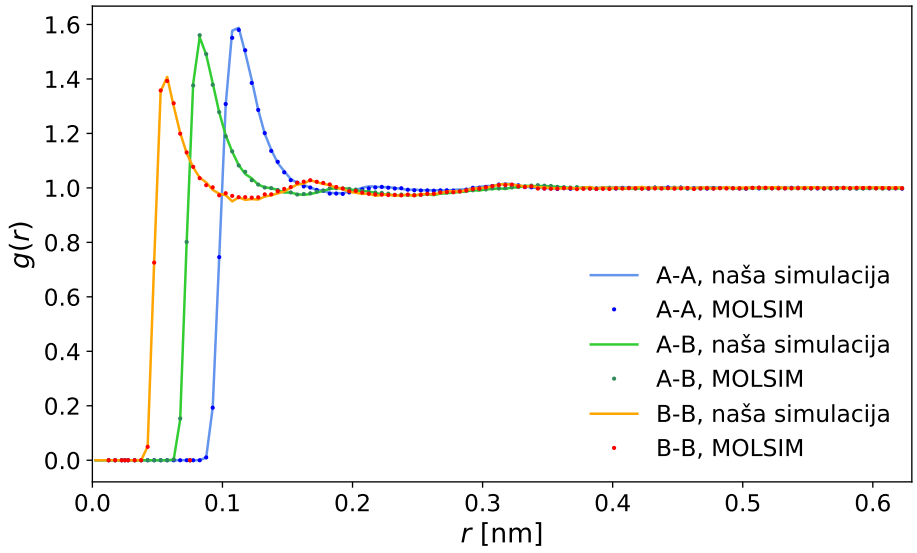
Nato smo preverili še pravilnost delovanja simulacije molekul, sestavljenih iz več atomov. Smith in Jaffe sta izvedla simulacijo molekulske dinamike za tekoči benzen in primerjala rezultate z eksperimentalnimi rezultati [31]. Primerjavo z rezultati naše simulacije lahko vidimo na sliki 5.2. Vsako skupino CH v molekuli benzena smo modelirali kot en atom s $\sigma = 0{,}375\,\mathrm{nm}$ in $\epsilon = 0{,}46\,\mathrm{kJ/mol}$ [16]. Vsaka molekula je sestavljena iz šest takšnih skupin, ki so razporejene v oglišča pravilnega šestkotnika s stranico $0{,}139\,\mathrm{nm}$. Radialni porazdelitveni funkciji obeh simulacij sta si zelo podobni, čeprav sta uporabljeni različni metodi. Malce bolj pa pričakovano odstopa radialna porazdelitvena funkcija, ki je pridobljena eksperimentalno.



Slika 5.2: Graf radialne porazdelitvene funkcije benzena. Primerjamo rezultate naše simulacije in rezultate Smitha in Jaffeja [31]. Parametri naše simulacije: $T = 298\,\mathrm{K}$, $\sigma = 0{,}375\,\mathrm{nm}$ [16], $\epsilon = 0{,}46\,\mathrm{kJ/mol}$ [16], $L = 2{,}8132\,\mathrm{nm}$, $N = 150$, $N_a = 900$ in $N_{MC} = 20000$.

Izvedli smo tudi simulacijo, kjer so molekule sestavljene iz različnih tipov atomov. Uporabili smo dva tipa atomov, A in B, s parametri $\sigma_A = 2\ \sigma_B$ in

$\epsilon_A = 2\ \epsilon_B$. Vsaka molekula je sestavljena iz enega atoma tipa A in enega atoma tipa B, razdalja med njima pa je 0,243 nm. Simulacijo smo z enakimi vhodnimi podatki izvedli še s programom MOLSIM. Primerjavo radialnih porazdelitvenih funkcij lahko vidimo na sliki 5.3. Le-te so skoraj povsem enake. Povprečna napaka radialnih porazdelitvenih funkcij je 0,003. Tudi povprečni energiji atoma tekom simulacije sta skoraj enaki. E_{avg} naše simulacije je $-1{,}2876\,\mathrm{kJ/mol}$, E_{avg} simulacije s programom MOLSIM pa $-1{,}2884\,\mathrm{kJ/mol}$.



Slika 5.3: Graf radialnih porazdelitvenih funkcij za sistem molekul z dvema različnima atomoma. Primerjamo rezultate naše simulacije in rezultate, pridobljene s programom MOLSIM. Parametri simulacije: $T = 240\,\mathrm{K}$, $\sigma_A = 1\,\mathrm{nm}$, $\epsilon_A = 1\,\mathrm{kJ/mol}$, $\sigma_B = 0{,}5\,\mathrm{nm}$, $\epsilon_B = 0{,}5\,\mathrm{kJ/mol}$, $L = 2{,}8132\,\mathrm{nm}$, $N = 500$, $N_a = 1000$ in $N_{MC} = 10000$.

Rezultati naših simulacij so zelo podobni rezultatom iz literature in rezultatom, pridobljenih s programom MOLSIM. Sklepamo lahko, da naše izvedbe metode Monte Carlo za simulacijo fluidov delujejo pravilno.

5.3 Analiza pohitritev

Naš glavni namen pri prilagoditvi metode Monte Carlo za simulacije fluidov, da se ta lahko izvaja na grafičnih procesnih enotah, je skrajšanje časa izvajanja simulacij. Zato smo opravili meritve časov izvajanja za vse naše izvedbe pri različnih vhodnih podatkih. Na čas izvajanja simulacije najbolj vplivata število atomov N_a v simulacijskem modelu in število ciklov simulacije N_{MC}. Za vsako kombinacijo vhodnih podatkov smo meritev opravili trikrat in izračunali povprečje. Merili smo celoten čas izvajanja simulacij, ki vključuje čas priprave podatkov in podatkovnih struktur, potrebnih za simulacijo. Naše izvedbe podpirajo računanje s podatki v plavajoči vejici tako v enojni kot v dvojni natančnosti. Meritve smo izvedli v obeh formatih natančnosti.

Izvajalne čase osnovne vzporedne izvedbe, t_{GPU1}, in optimizirane vzporedne izvedbe, t_{GPU2}, smo primerjali z izvajalnimi časi zaporedne izvedbe, t_{CPU}. Pohitritve smo izračunali po enačbah

$$S_{\mathrm{GPU1}} = \frac{t_{\mathrm{CPU}}}{t_{\mathrm{GPU1}}} \qquad \text{in} \qquad S_{\mathrm{GPU2}} = \frac{t_{\mathrm{CPU}}}{t_{\mathrm{GPU2}}} \quad , \tag{5.1}$$

kjer sta S_{GPU1} in S_{GPU2} pohitritvi osnovne in optimizirane vzporedne izvedbe v primerjavi z zaporedno izvedbo.

Najprej smo izvedli meritve pri različnih vrednostih N_{MC}. Uporabili smo simulacijski model iz poglavja 5.2, ki vsebuje 500 molekul, sestavljenih iz dveh različnih atomov. Število vseh atomov v sistemu je torej $N_a = 1000$.

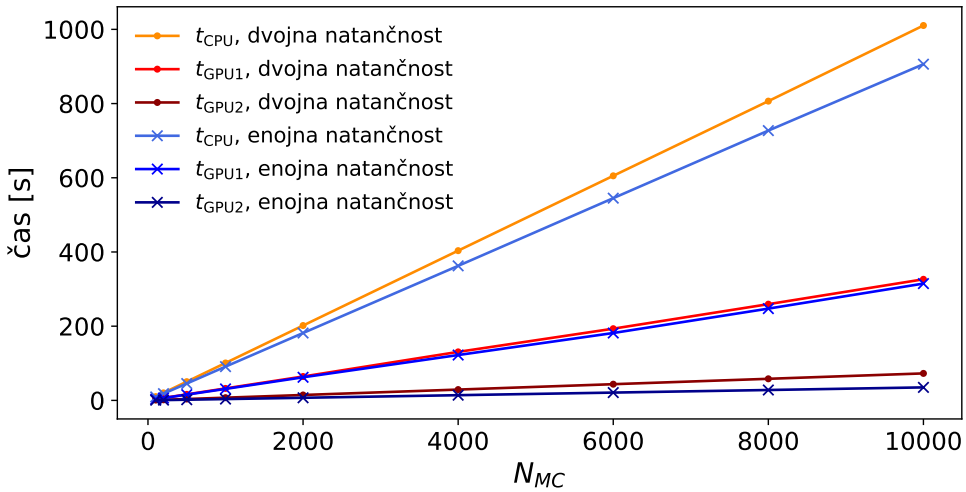
Rezultate meritev lahko vidimo v tabelah 5.1 in 5.2. Kot smo pričakovali, sta vzporedni izvedbi hitrejši od zaporedne. Osnovna vzporedna izvedba doseže trikratne pohitritve, optimizirana izvedba pa pri enojni natančnosti doseže kar 26-kratno pohitritev. Število ciklov ne vpliva na čas izvajanja posameznega cikla, zato časi izvajanja naraščajo linearno z N_{MC}. To je lepo razvidno z grafa na sliki 5.4. Posledično se pohitritve z večanjem N_{MC} bistveno ne spreminjajo, kar lahko vidimo na grafu na sliki 5.5. Pohitritve naraščajo le pri manj kot 500 ciklih, saj tam bolj pride do izraza čas, potreben za izračun začetne energije sistema ter pripravo podatkov in podatkovnih struktur ob začetku simulacije. Ta je pri vzporednih izvedbah daljši kot

N_{MC}	t_{CPU} [s]	t_{GPU1} [s]	S_{GPU1} [s]	t_{GPU2} [s]	S_{GPU2}
100	9,3	3,2	2,9	0,5	18,6
200	18,2	6,2	2,9	0,8	22,7
500	45,4	15,5	2,9	1,8	25,2
1000	91,0	31,0	2,9	3,6	25,3
2000	181,6	62,3	2,9	7,0	25,9
4000	362,6	122,2	3,0	14,0	25,9
6000	545,0	181,6	3,0	21,0	26,0
8000	727,2	247,4	2,9	28,0	26,0
10000	906,1	314,8	2,9	35,0	25,9

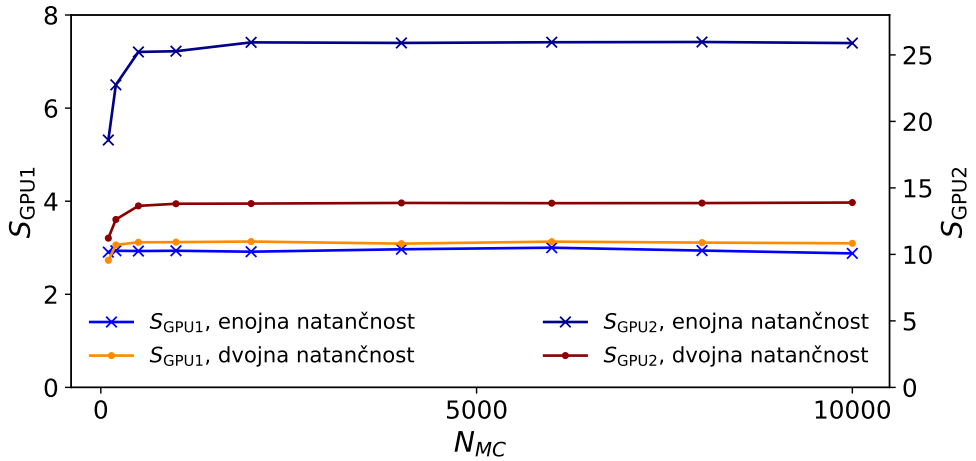
Tabela 5.1: Izvajalni časi in pohitritve simulacije molekul iz dveh različnih atomov pri enojni natančnosti in različnih N_{MC}.

N_{MC}	t_{CPU} [s]	t_{GPU1} [s]	S_{GPU1} [s]	t_{GPU2} [s]	S_{GPU2}
100	10,1	3,7	2,7	0,9	11,2
200	20,2	6,6	3,1	1,6	12,6
500	50,5	16,2	3,1	3,7	13,6
1000	100,8	32,3	3,1	7,3	13,8
2000	201,8	64,4	3,1	14,6	13,8
4000	403,6	130,7	3,1	29,1	13,9
6000	605,3	193,3	3,1	43,7	13,9
8000	806,6	259,3	3,1	58,2	13,9
10000	1010,4	326,4	3,1	72,7	13,9

Tabela 5.2: Izvajalni časi in pohitritve simulacije molekul iz dveh različnih atomov pri dvojni natančnosti in različnih N_{MC}.



Slika 5.4: Graf časov izvajanja simulacije molekul iz dveh različnih atomov pri različnih N_{MC}.



Slika 5.5: Graf pohitritev simulacije molekul iz dveh različnih atomov pri različnih N_{MC}.

pri zaporedni izvedbi, saj vključuje tudi prenašanje podatkov na grafično procesno enoto in ostale operacije, povezane s tem. Z večanjem števila ciklov ta čas postane zanemarljiv, zato se pohitritve pri več kot 500 ciklih ustalijo.

Izvedli smo tudi meritve pri različnih vrednostih N_a. Ta parameter namreč vpliva na čas izvajanja posameznega cikla simulacije. Uporabili smo enak simulacijski model. Število ciklov smo nastavili na le 100, saj smo ugotovili, da časi izvajanja naraščajo linearno z N_{MC}. Pri večjih N_{MC} in velikem številu atomov bi lahko nekatere meritve zaporedne izvedbe namreč trajale več kot 24 ur. Rob simulacijske škatle smo ustrezno prilagajali, da smo ohranili enako številsko gostoto pri vseh vrednostih N_a.

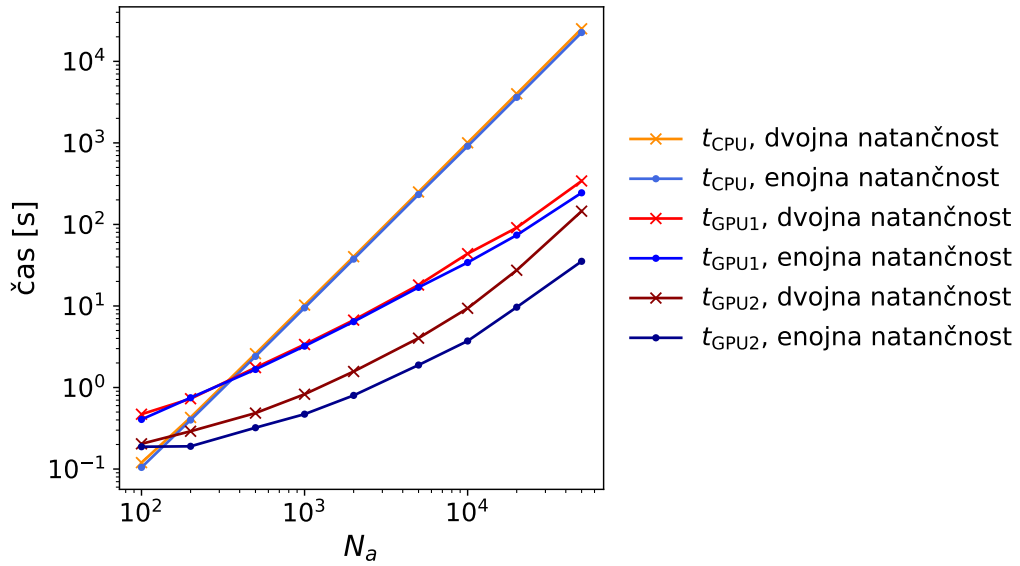
Rezultate meritev lahko vidimo v tabelah 5.3 in 5.4. Kot smo pričakovali, sta vzporedni izvedbi precej hitrejši od zaporedne izvedbe. Osnovna vzporedna izvedba je od zaporedne hitrejša pri 500 atomih, optimizirana izvedba pa že pri 200 atomih. Za simulacijo 50000 atomov osnovna vzporedna izvedba pri dvojni natančnosti doseže 74-kratno, pri enojni natančnosti pa 92-kratno pohitritev. Optimizirana vzporedna izvedba je še hitrejša, saj pri dvojni natančnosti doseže kar 172-kratno, pri enojni natančnosti pa skoraj 640-kratno pohitritev. Na grafu na sliki 5.6 lahko vidimo, da izvajalni časi zaporedne izvedbe z večanjem števila atomov naraščajo eksponentno. V vsakem ciklu simulacije je namreč potrebno upoštevati ${N_a}^2$ interakcij med atomi. Pri vzporednih izvedbah pa izvajalni časi naraščajo precej počasneje, saj se računanje interakcij med atomi porazdeli med niti, ki tečejo vzporedno. Na grafu na sliki 5.7 lahko vidimo pohitritve v odvisnosti od števila atomov. Opazimo, da so pohitritve večje pri večjem številu atomov v sistemu. Razlog je v tem, da več kot je atomov, več vzporednih niti uporabimo. Grafična procesna enota je zato bolj izkoriščena. Seveda ne moremo uporabiti neskončnega števila niti, saj se grafična procesna enota slej ko prej zasiči. Z grafa je lepo razvidno, da pohitritve pri večjem številu atomov naraščajo počasneje kot pri manjšem številu atomov.

N_a	t_{CPU} [s]	t_{GPU1} [s]	S_{GPU1} [s]	t_{GPU2} [s]	S_{GPU2}
100	0,1	0,4	0,2	0,2	0,5
200	0,4	0,7	0,6	0,2	2,0
500	2,4	1,7	1,4	0,3	8,0
1000	9,5	3,2	3,0	0,5	19,0
2000	37,5	6,4	5,9	0,8	46,9
5000	232,7	16,9	13,8	1,9	122,5
10000	911,0	34,1	26,7	3,7	246,2
20000	3619,7	74,0	48,9	9,7	373,2
50000	22574,8	244,3	92,4	35,3	639,5

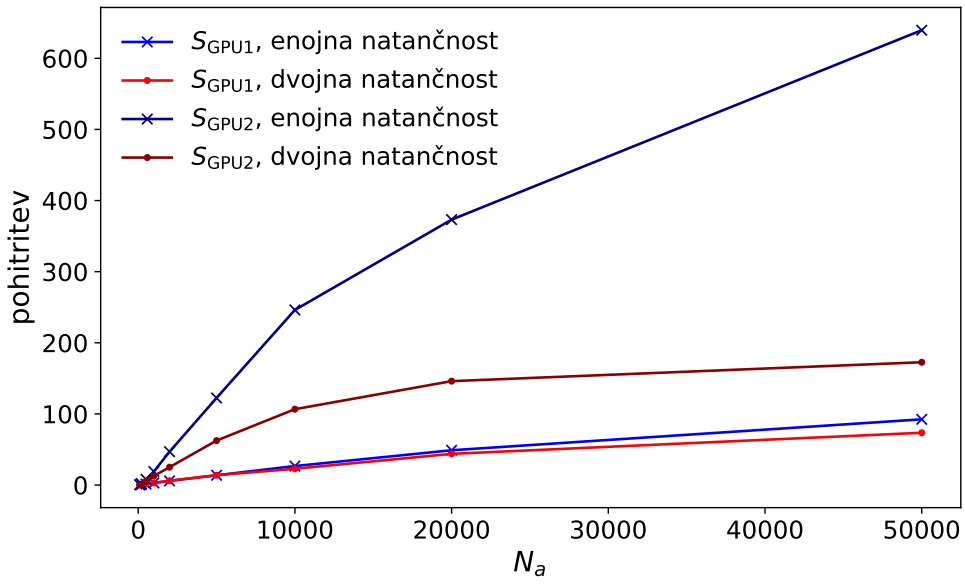
Tabela 5.3: Izvajalni časi in pohitritve simulacije molekul iz dveh različnih atomov pri enojni natančnosti in različnih N_a.

N_a	t_{CPU} [s]	t_{GPU1} [s]	S_{GPU1} [s]	t_{GPU2} [s]	S_{GPU2}
100	0,1	0,5	0,2	0,2	0,5
200	0,4	0,7	0,6	0,3	1,3
500	2,6	1,8	1,4	0,5	5,2
1000	10,2	3,4	3,0	0,8	12,7
2000	40,2	6,7	6,0	1,6	25,1
5000	250,1	18,1	13,8	4,0	62,5
10000	1002,5	43,8	22,9	9,4	106,6
20000	4002,5	91,2	43,9	27,4	146,1
50000	25087,6	341,3	73,5	145,3	172,7

Tabela 5.4: Izvajalni časi in pohitritve simulacije molekul iz dveh različnih atomov pri dvojni natančnosti in različnih N_a.



Slika 5.6: Graf časov izvajanja simulacije molekul iz dveh različnih atomov pri različnih N_a. Skala je logaritemska.



Slika 5.7: Graf pohitritev simulacije molekul iz dveh različnih atomov pri različnih N_a.

Grafične procesne enote s števili v plavajoči vejici računajo počasneje v formatu dvojne natančnosti kot v formatu enojne natančnosti [32]. Glavni razlog je razmerje med številom izvajalnih enot za računanje v dvojni natančnosti in številom izvajalnih enot za računanje v enojni natančnosti. Pri profesionalnih grafičnih karticah, namenjenih splošnemu računanju, je to razmerje tipično 1:2. Pri potrošniških grafičnih karticah pa je to razmerje slabše. Pri grafični kartici, ki smo jo uporabili pri naših meritvah, je to razmerje 1:32.

Iz rezultatov meritev lahko vidimo, da so izvajalni časi vseh izvedb res večji pri uporabi dvojne natančnosti. Razlika je največja pri optimizirani vzporedni izvedbi, kjer so izvajalni časi pri $N_a = 50000$ in enojni natančnosti kar štirikrat krajši kot pri dvojni natančnosti. Precej manjše razlike so pri osnovni vzporedni izvedbi, kjer je pri $N_a = 50000$ format dvojne natančnosti le $1,4$-krat počasnejši od formata enojne natančnosti. To nakazuje, da je ozko grlo te izvedbe drugje.

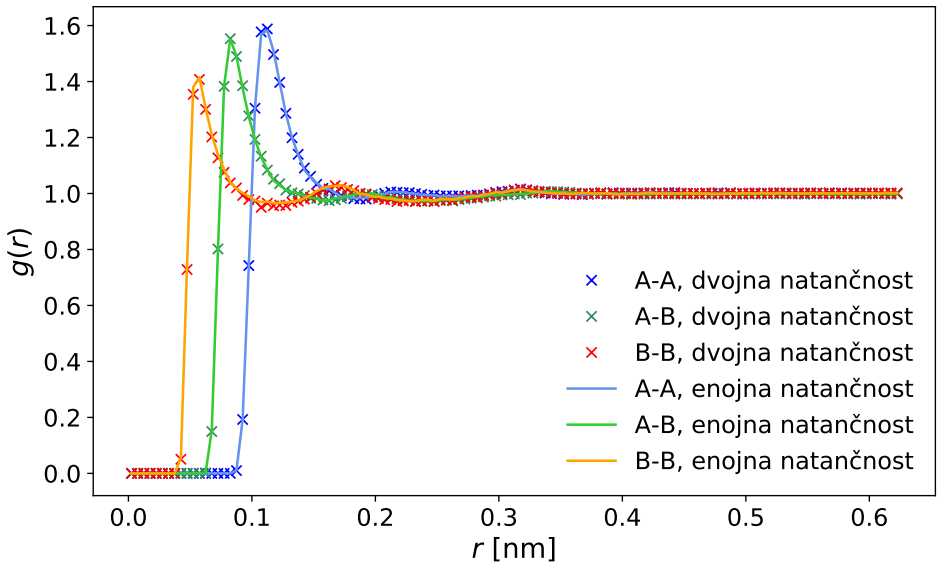
5.4 Vpliv predstavitve števil na izračune

V poglavju 5.3 smo ugotovili, da naše izvedbe metode Monte Carlo hitreje računajo, če za podatke v plavajoči vejici uporabimo enojno natančnost namesto dvojne. Takšne pohitritve seveda niso brez slabosti, saj uporaba enojne natančnosti pomeni hitrejše akumuliranje napak pri zaokroževanju in manjšo natančnost rezultatov. Če so te napake sprejemljive, lahko z uporabo enojne natančnosti prihranimo precej časa, pa tudi denarja. Potrošniške grafične kartice so namreč precej cenejše od profesionalnih.

Že tekom razvoja naših izvedb smo ugotovili, da te ne delujejo pravilno, če za spremenljivko E uporabimo enojno natančnost. Spremenljivka namreč tekom celotne simulacije akumulira energijo sistema. Pri dovolj dolgi simulaciji postane vrednost te spremenljivke tako velika, da prištevanje relativno majhnih prispevkov energije posameznega koraka simulacije nanjo ne vpliva več. Zato za to spremenljivko vedno uporabimo dvojno natančnost.

Z optimizirano vzporedno izvedbo smo izvedli nekaj simulacij v obeh formatih natančnosti. Simulirali smo dvajset tisoč ciklov pri različnem številu atomov s simulacijskimi modeli, opisanimi v poglavju 5.2. Rezultate simulacij v enojni in dvojni natančnosti smo primerjali med seboj.

Radialne porazdelitve funkcije so v vseh primerih skoraj enake. Eno od primerjav porazdelitvenih funkcij med simulacijami z enojno in dvojno natančnostjo lahko vidimo na sliki 5.8.



Slika 5.8: Primerjava porazdelitvenih funkcij med simulacijama z enojno in dvojno natančnostjo pri $N_{MC} = 20000$ in $N_a = 1000$. Simulirali smo molekule iz dveh različnih atomov.

Primerjali smo tudi vrednosti izhodnega podatka E_{avg}. Vrednosti E_{avg} pri različnih simulacijah in obeh formatih natančnosti lahko vidimo v tabeli 5.5. Pri simulacijah argona in benzena se vrednosti E_{avg} začnejo razlikovati na tretjem decimalnem mestu, pri simulaciji molekul iz dveh različnih atomov pa na drugem decimalnem mestu.

N_a	1000	5000	10000
E_{avg}, enojna nat.	-1,293 ± 0,020	-1,295 ± 0,008	-1,297 ± 0,007
E_{avg}, dvojna nat.	-1,289 ± 0,018	-1,295 ± 0,009	-1,287 ± 0,007

(a) molekule iz dveh različnih atomov

N_a	1000	5000	10000
E_{avg}, enojna nat.	-6,016 ± 0,017	-6,036 ± 0,008	-6,039 ± 0,006
E_{avg}, dvojna nat.	-6,013 ± 0,016	-6,037 ± 0,007	-6,039 ± 0,006

(b) argon

N_a	996	4998	9996
E_{avg}, enojna nat.	-5,304 ± 0,033	-5,371 ± 0,014	-5,378 ± 0,010
E_{avg}, dvojna nat.	-5,305 ± 0,034	-5,371 ± 0,014	-5,376 ± 0,010

(c) benzen

Tabela 5.5: Primerjava izhodnega podatka E_{avg} med enojno in dvojno natančnostjo pri različnih simulacijskih modelih in N_a.

Poglavje 6

Sklepne ugotovitve

V magistrski nalogi smo se osredotočili na metodo Monte Carlo za simulacijo fluidov. Ta je po naravi stohastična in zaporedna. Kljub temu smo jo uspešno paralelizirali in prilagodili za izvajanje na grafičnih procesnih enotah. Uporabili smo platformo CUDA in princip energijske dekompozicije. S pomočjo vzporedne redukcije smo najprej paralelizirali in na grafično procesno enoto prestavili računanje sprememb energije v simuliranem sistemu. Z optimizacijo smo nadaljevali in na grafično procesno enoto prestavili tudi generiranje poskusov premikov molekul in odločanje o teh premikih. To se je izkazalo za precej zahteven proces, saj je vsak korak simulacije odvisen od rezultatov prejšnjega koraka. Uporabili smo posebno tehniko sinhronizacije med bloki niti in pazili na pravilno uporabo pregrad. Poudariti velja, da smo podprli tudi simulacijo molekul, sestavljenih iz več različnih atomov. Obstoječe vzporedne izvedbe na grafičnih procesnih enotah namreč podpirajo le simulacijo enoatomnih delcev. Podpora molekulam še malenkost bolj zaplete sinhronizacijo in deljenje podatkov med nitmi, pri premikanju molekul pa je poleg translacije potrebno upoštevati tudi rotacijo.

Interakcije med atomi smo modelirali s potencialom Lennard-Jones. Tekom simulacije računamo energijo sistema in radialno porazdelitveno funkcijo. O pravilnem delovanju naših izvedb smo se prepričali s primerjanjem naših rezultatov z rezultati iz literature in rezultati referenčnega programa

MOLSIM.

Naš glavni cilj pri prilagajanju te metode za izvajanje na grafičnih procesnih enotah je bil skrajšanje časa izvajanja simulacij. Zato smo opravili meritve časov izvajanja simulacij pri različnih vhodnih podatkih in analizirali pohitritve. Izvajalne čase smo primerjali z izvajalnimi časi zaporedne izvedbe, ki se izvaja na centralni procesni enoti. Rezultati so presegli naša pričakovanja, saj smo že z osnovno vzporedno izvedbo dosegli odlične pohitritve. Z optimizirano vzporedno izvedbo pa smo dosegli skoraj 640-kratne pohitritve. Pohitritve so največje pri simulacijah večjih sistemov. Na primer, za le 100 ciklov dolgo simulacijo 50 tisoč atomov z zaporedno izvedbo na centralni procesni enoti potrebujemo šest ur. Do enakih rezultatov lahko z našo optimizirano vzporedno izvedbo na grafični procesni enoti pridemo v 35 sekundah.

Pri izvajanju simulacij na grafičnih procesnih enotah format predstavitve števil v plavajoči vejici bistveno vpliva na čas izvajanja. Naša optimizirana vzporedna izvedba je pri uporabi enojne natančnosti do štirikrat hitrejša, kot pri uporabi dvojne natančnosti, sami rezultati simulacije pa so si zelo podobni.

Optimizirano vzporedno izvedbo je seveda možno še izboljšati. V trenutni izvedbi je izračun spremembe energije, ki jo povzroči premik enega atoma, dodeljen eni niti. Za izvedbo simulacije potrebujemo torej le toliko niti, kot je atomov v modelnem sistemu. To pomeni, da je pri majhnih sistemih grafična procesna enota zaradi majhnega števila vzporednih niti neizkoriščena. Zato bi lahko uporabili več niti in izračun spremembe energije enega atoma porazdelili na dve ali več sosednjih niti [20].

Literatura

[1] M. P. Allen in D. J. Tildesley, *Computer simulation of liquids*. Oxford university press, 2017.

[2] J. A. Anderson, M. E. Irrgang, in S. C. Glotzer, "Scalable Metropolis Monte Carlo for simulation of hard shapes," *Computer Physics Communications*, št. 204, str. 21–30, 2016.

[3] J. A. Anderson, E. Jankowski, T. L. Grubb, M. Engel, in S. C. Glotzer, "Massively parallel Monte Carlo for many-particle simulations on GPUs," *Journal of Computational Physics*, št. 254, str. 27–38, 2013.

[4] J. Arvo, "Fast random rotation matrices," v *Graphics Gems III (IBM Version)*. Elsevier, 1992, str. 117–120.

[5] A. Becker, "Sampling a uniformly random rotation, wolfram demonstrations project," 2012. Dostopno na: http://demonstrations.wolfram.com/SamplingAUniformlyRandomRotation/ (pridobljeno 26. 6. 2018).

[6] J. Chen, J.-G. Mi, in K.-Y. Chan, "Comparison of different mixing rules for prediction of density and residual internal energy of binary and ternary Lennard–Jones mixtures," *Fluid Phase Equilibria*, št. 178, zv. 1-2, str. 87–95, 2001.

[7] "CMake." Dostopno na: https://cmake.org/ (pridobljeno 25. 8. 2018).

[8] U. K. Deiters, "Efficient coding of the minimum image convention," *Zeitschrift für Physikalische Chemie*, št. 227, zv. 2-3, str. 345–352, 2013.

[9] S. Deublein, B. Eckl, J. Stoll, S. V. Lishchuk, G. Guevara-Carrion, C. W. Glass, T. Merker, M. Bernreuther, H. Hasse, in J. Vrabec, "ms2: A molecular simulation tool for thermodynamic properties," *Computer Physics Communications*, št. 182, zv. 11, str. 2350–2367, 2011.

[10] D. Frenkel in B. Smit, *Understanding molecular simulation: From algorithms to applications*. New York: Academic Press, 2002, št. 1.

[11] W. R. Gilks, "Markov Chain Monte Carlo," v *Encyclopedia of Biostatistics*. Wiley Online Library, 2005.

[12] E. Hailat, V. Russo, K. Rushaidat, J. Mick, L. Schwiebert, in J. Potoff, "Parallel Monte Carlo simulation in the canonical ensemble on the graphics processing unit," *International Journal of Parallel, Emergent and Distributed Systems*, št. 29, zv. 4, str. 379–400, 2014.

[13] M. Harris, "Optimizing parallel reduction in CUDA," Nvidia Corporation, Tehnično poročilo, 2008. Dostopno na: http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf (pridobljeno 12. 7. 2018).

[14] R. L. Harrison, "Introduction to Monte Carlo simulation," v *AIP conference proceedings*, št. 1204, zv. 1. AIP, 2010, str. 17–21.

[15] W. K. Hastings, "Monte Carlo sampling methods using Markov chains and their applications," *Biometrika*, št. 57, zv. 1, str. 97–109, 1970.

[16] W. L. Jorgensen, J. D. Madura, in C. J. Swenson, "Optimized intermolecular potential functions for liquid hydrocarbons," *Journal of the American Chemical Society*, št. 106, zv. 22, str. 6638–6646, 1984.

[17] A. A. Khan, "Radial distribution functions of fluid argon," *Physical Review*, št. 134, zv. 2A, str. A367, 1964.

[18] D. B. Kirk in W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[19] D. P. Kroese, T. Brereton, T. Taimre, in Z. I. Botev, "Why the Monte Carlo method is so important today," *Wiley Interdisciplinary Reviews: Computational Statistics*, št. 6, zv. 6, str. 386–392, 2014.

[20] Y. Liang, X. Xing, in Y. Li, "A GPU-based large-scale Monte Carlo simulation method for systems with long-range interactions," *Journal of Computational Physics*, št. 338, str. 252–268, 2017.

[21] V. I. Manousiouthakis in M. W. Deem, "Strict detailed balance is unnecessary in Monte Carlo simulation," *The Journal of chemical physics*, št. 110, zv. 6, str. 2753–2756, 1999.

[22] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, in E. Teller, "Equation of state calculations by fast computing machines," *The journal of chemical physics*, št. 21, zv. 6, str. 1087–1092, 1953.

[23] J. Mick, E. Hailat, V. Russo, K. Rushaidat, L. Schwiebert, in J. Potoff, "GPU-accelerated Gibbs ensemble Monte Carlo simulations of Lennard-Jonesium," *Computer Physics Communications*, št. 184, str. 2662–2669, 2013.

[24] "Visual Studio Community," Microsoft. Dostopno na: https://visualstudio.microsoft.com/vs/community/ (pridobljeno 25. 8. 2018).

[25] J. Nickolls in W. J. Dally, "The GPU computing era," *IEEE micro*, št. 30, zv. 2, str. 56–69, 2010.

[26] "CUDA Zone," NVIDIA, Sep 2017. Dostopno na: https://developer.nvidia.com/cuda-zone (pridobljeno 25. 8. 2018).

[27] "CUDA C Programming guide v9.2," NVIDIA, 2018. Dostopno na: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (pridobljeno 12. 7. 2018).

[28] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, in J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, št. 96, zv. 5, str. 879–899, 2008.

[29] J. Reščič in P. Linse, "MOLSIM: A modular molecular simulation software," *Journal of computational chemistry*, št. 36, zv. 16, str. 1259–1274, 2015.

[30] J. K. Shah, E. Marin-Rimoldi, R. G. Mullen, B. P. Keene, S. Khan, A. S. Paluch, N. Rai, L. L. Romanielo, T. W. Rosch, B. Yoo *in sod.*, "Cassandra: An open source Monte Carlo package for molecular simulation," *Journal of Computational Chemistry*, št. 38, str. 1727–1739, 2017.

[31] G. D. Smith in R. L. Jaffe, "Comparative study of force fields for benzene," *The Journal of Physical Chemistry*, št. 100, zv. 23, str. 9624–9630, 1996.

[32] V. Strbac, J. Vander Sloten, in N. Famaey, "Analyzing the potential of GPGPUs for real-time explicit finite element analysis of soft tissue deformation using CUDA," *Finite Elements in Analysis and Design*, št. 105, str. 79–89, November 2015.

[33] C.-L. Su, P.-Y. Chen, C.-C. Lan, L.-S. Huang, in K.-H. Wu, "Overview and comparison of OpenCL and CUDA technology for GPGPU," v *Circuits and Systems (APCCAS), 2012 IEEE Asia Pacific Conference on.* IEEE, 2012, str. 448–451.

[34] W. F. van Gunsteren, X. Daura, N. Hansen, A. E. Mark, C. Oostenbrink, S. Riniker, in L. J. Smith, "Validation of molecular simulation: An overview of issues," *Angewandte Chemie International Edition*, št. 57, zv. 4, str. 884–902, 2018.

[35] J. Wei in F. Kruis, "GPU-accelerated Monte Carlo simulation of particle coagulation based on the inverse method," *Journal of Computational Physics*, št. 249, str. 67–79, 2013.

[36] J. A. White, "Lennard-jones as a model for argon and test of extended renormalization group calculations," *The Journal of chemical physics*, št. 111, zv. 20, str. 9352–9356, 1999.

[37] "Molecular simulation/Monte Carlo methods," Wikibooks, 2018. Dostopno na: https://en.wikibooks.org/w/index.php?title=Molecular_Simulation/Monte_Carlo_Methods&oldid=3382330 (pridobljeno 26. 6. 2018).

[38] "Molecular simulation/The Lennard-Jones potential," Wikibooks, 2018. Dostopno na: https://en.wikibooks.org/w/index.php?title=Molecular_Simulation/The_Lennard-Jones_Potential&oldid=3417110 (pridobljeno 26. 6. 2018).