Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Anže Čuk

CUDA izvedba mehkega modela za simulacijo skupinskega vedenja

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Iztok Lebar Bajec

Ljubljana, 2016

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

V diplomski nalogi razvijte programsko opremo za realizacijo mehkega modela za simulacijo skupinskega vedenja v okolju CUDA. Pri tem se osredotočite na model, kjer je verjetnost interakcije med entitetami obratno sorazmerna z njihovo oddaljenostjo. Implementacijo testirajte in primerjajte z referenčno serijsko implementacijo. Rezultate kritično komentirajte.

Izjava o avtorstvu zaključnega dela

Spodaj podpisani Anže Čuk, vpisna številka 63090085, avtor zaključnega dela z naslovom:

CUDA izvedba mehkega modela za simulacijo skupinskega vedenja (angl. *CUDA implementation of a fuzzy collective behaviour model*)

IZJAVLJAM

1. da sem pisno zaključno delo študija izdelal samostojno pod mentorstvom doc. dr. Iztoka Lebarja Bajca;
2. da je tiskana oblika pisnega zaključnega dela študija istovetna elektronski obliki pisnega zaključnega dela študija;
3. da sem pridobil/-a vsa potrebna dovoljenja za uporabo podatkov in avtorskih del v pisnem zaključnem delu študija in jih v pisnem zaključnem delu študija jasno označil/-a;
4. da sem pri pripravi pisnega zaključnega dela študija ravnal/-a v skladu z etičnimi načeli in, kjer je to potrebno, za raziskavo pridobil/-a soglasje etične komisije;
5. soglašam, da se elektronska oblika pisnega zaključnega dela študija uporabi za preverjanje podobnosti vsebine z drugimi deli s programsko opremo za preverjanje podobnosti vsebine, ki je povezana s študijskim informacijskim sistemom članice;
6. da na UL neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve avtorskega dela v elektronski obliki, pravico reproduciranja ter pravico dajanja pisnega zaključnega dela študija na voljo javnosti na svetovnem spletu preko Repozitorija UL;
7. dovoljujem objavo svojih osebnih podatkov, ki so navedeni v pisnem zaključnem delu študija in tej izjavi, skupaj z objavo pisnega zaključnega dela študija.

V Ljubljani, dne 26. februarja 2016 Podpis študenta/-ke:

Zahvaljujem se družini za vso podporo v času študija. Zahvaljujem se tudi mentorju doc. dr. Iztoku Lebarju Bajcu za vso pomoč pri izdelavi diplomske naloge.

Kazalo

Povzetek

Naslov: CUDA izvedba mehkega modela za simulacijo skupinskega vedenja

V diplomskem delu je izvedena implementacija mehkega modela na grafični kartici v okolju CUDA. Model sestavlja ena vrsta entitet. Interakcija poteka med sosednjimi entitetami, kjer je verjetnost interakcije obratno sorazmerna z oddaljenostjo, zato je bil uveden optimizacijski postopek za pridobivanje sosednjih entitet s pomočjo delitve prostora. Proces ugotavljanja interakcije je razdeljen na številne ščepce, ki jih v diplomskem delu podrobneje opišemo. Za zagotavljanje pravilnosti vmesnih in končnih rezultatov smo uporabili JUnit teste in referenčno okolje jFuzzyLogic. Preverjali smo pravilno izbiro soseda, vrednosti posameznih pripadnostnih funkcij ter končno ostro vrednost. Na koncu prikažemo rezultate, kjer smo preverjali čas izvajanja različnih implementacij ter pohitritev rešitve s pomočjo implementacije na grafični kartici.

Ključne besede: mehka logika, CUDA, skupinsko vedenje, optimizacija.

Abstract

Title: CUDA implementation of a fuzzy collective behaviour model

This thesis consists of an implementation of a fuzzy model using the graphics card and the CUDA environment. Our fuzzy model consists of entities, which are all the same type. Entity interaction occurs only between neighbouring entities, where the probability of an interaction is inversely proportional to the distance between the neighbouring entities. For this reason an optimisation of the process of gathering neighbouring entities has been made by dividing the simulation area into bins. The algorithm that drives the entity interaction is implemented with multiple kernels, which are described in detail in this thesis. For the purpose of ensuring the correctness of our results, JUnit tests were used along with the jFuzzyLogic library, which serves as a reference fuzzy logic system. For each entity we tested if the choice of the neighbouring entity was valid. We also tested the individual membership functions and the final crisp output. We conclude this thesis by comparing the execution times of the individual implementations and measuring the speed up value of our graphic card implementation.

Keywords: fuzzy logic, CUDA, collective behaviour, optimisation.

Poglavje 1

Uvod

Človeštvo je skozi zgodovino razvilo različne matematične modele, s katerimi opisujemo svet okoli nas. Eno izmed zanimivih poglavij v človeški enciklopediji znanja je opisovanje skupinskega vedenja. Ta pojav lahko zasledimo tako v naravi, kjer lahko opazujemo zanimive strukture, ki nastajajo ob gibanju skupine živali (npr. jate ptic), kot v človeški družbi (npr. gibanje ljudi pri evakuaciji). Implementacijo skupinskega vedenja lahko dosežemo z različnimi pristopi [14, 3], v diplomskem delu pa se osredotočimo na pristop z mehko logiko.

Simulacije ponavadi potekajo na velikem številu elementov, nad katerimi je potrebno izvršiti različne računske operacije. Sekvenčna implementacija na centralni procesni enoti (CPE) se izkaže za zamudno, zato ponavadi iščemo načine, s katerimi lahko pohitrimo posamezen korak v izvajanju simulacije. Pri simulaciji skupinskega vedenja (kot velja tudi za večino drugih simulacij) imamo v posamezni časovni rezini prisotno podatkovno neodvisnost oz. tako imenovani podatkovni paralelizem, kar pomeni, da dokler ne prepisujemo trenutnih podatkov, lahko določeno operacijo izvajamo nad vsemi elementi v simulaciji sočasno. Za doseganje pohitritev obstajajo različne rešitve, npr. nitenje na procesorju z uporabo OpenMP [1], uporaba več-računalniških sistemov z MPI (angl. Message Passing Interface) ..., mi pa se bomo osredotočili na paralelizacijo z implementacijo na grafično procesni enoti (GPE) z upo-

rabo platforme CUDA[1].

V diplomskem delu imamo nabor entitet, ki se nahajajo v prostoru dane velikosti. Želimo simulirati njihovo gibanje, ki je v odvisnosti od sosednosti entitet. Interakcijo med njimi zapišemo z mehko logiko, implementacijo pa izvedemo na GPE. Izvedba je sestavljena iz številnih ščepcev. Na začetku prostor gibanja razdelimo na posamezne celice, kar pripomore k pohitritvi iskanja sosedov. Postopek delitve prostora smo povzeli po [8]. Nato za posamezno entiteto izračunamo nabor kandidatov za njeno sosedo. Iz nabora kandidatov naključno izberemo le eno entiteto. Verjetnost izbire je obratno sorazmerna z oddaljenostjo med obravnavano entiteto ter njenim kandidatom. Temu sledi postopek mehkega računanja, kjer vhodnim ostrim podatkom priredimo ustrezne mehke množice, z uporabo baze znanja in mehkega sklepanja pridobimo izhodne mehke podatke, ter na koncu, z uporabo metode za ostrenje, izračunamo končno ostro vrednost, s pomočjo katere lahko opišemo spremembo gibanja posamezne entitete. Veljavnost rešitve preverimo z uporabo JUnit testov.

[1]https://developer.nvidia.com/cuda-zone

Poglavje 2

Cilji in orodja

Diplomsko delo temelji na obstoječih delih [7, 4], kjer je za simulacijo skupinskega vedenja uporabljena mehka logika. Z namenom poenostavitve smo se omejili le na en tip entitet (plen), interakcijo pa omejili tako, da je obnašanje odvisno le od enega soseda, če je ta na voljo. Uporabili smo obstoječa pravila za interakcijo med entitetami ter ustrezne pripadnostne funkcije. Cilj diplomskega dela je izkoristiti paralelno naravo problema tako, da prenesemo računanje na GPE, kjer imamo, v primerjavi s CPE, na voljo mnogo več vzporednih niti, ki bodo izvajale računanje. Pri iskanju sosedov smo implementirali tudi postopek delitve prostora, ki pripomore k večji pohitritvi. Želeli smo preveriti hipotezo, ki pravi, da se bo pri večanju števila osebkov v simulaciji večala tudi časovna razlika med izvedbo na CPE in GPE. Predpostavljamo, da bo pri manjšem številu entitet časovna razlika izvedbe razmeroma majhna in bo postala bolj očitna šele z naraščanjem števila entitet.

2.1 Orodja

V tem sklopu bomo naredili pregled orodij in tehnologij, ki jih uporabljamo. Posamezno orodje oz. tehnologijo bomo na kratko opisali ter pojasnili, kako jo uporabljamo v našem diplomskem delu.

2.1.1 CUDA toolkit

CUDA toolkit je skupek orodji, ki so potrebna za programiranje v okolju CUDA. Vsebuje NVCC (angl. NVIDIA CUDA compiler) prevajalnik za grafične kartice podjetja nVIDIA, matematične knjižnice ter orodja za razhroščevanje in optimizacijo programov. Vsebuje tudi programske vodiče, dokumentacijo ter različne primere, ki služijo kot pomoč pri razvijanju programske opreme.

2.1.2 Thrust

Thrust[1] je C++ knjižnica za okolje CUDA, ki temelji na STL (angl. Standard Template Library) ter pripomore k lažjemu programiranju na GPE. Za naše diplomsko delo je predvsem pomemben sortirni algoritem `sort_by_key`, ki smo ga uporabili pri optimizaciji iskanja sosedov.

2.1.3 GLM

GLM je C++ knjižnica z naborom matematičnih operacij, ki temelji na GLSL (angl. OpenGL Shading Language) specifikacijah. Omogoča tudi uporabo v okolju CUDA, ter tako pripomore k lažji prenosljivosti problema med CPE in GPE.

2.1.4 jFuzyyLogic

jFuzzyLogic [6, 5] je odprtokodna knjižnica za implementacijo mehke logike v programskem jeziku Java na osnovi FCL (angl. Fuzzy Control Language). V diplomskem delu knjižnico uporabimo pri testiranju pravilnosti rešitev z JUnit[2] testi.

[1] https://developer.nvidia.com/thrust
[2] http://junit.org/

2.1.5 Visual Studio

Visual Studio je zelo napredno in razširjeno razvojno okolje podjetja Microsoft. Omogoča razvoj programske opreme v številnih programskih jezikih za okolje Windows ter omogoča razvoj spletnih strani in aplikacij. Vsebuje zelo uporaben razhroščevalnik ter C in C++ prevajalnike za okolje Windows. V diplomski nalogi uporabljamo brezplačno različico Visual Studio 2013.

2.1.6 Eclipse

Eclipse je brezplačno odprtokodno razvojno okolje, primarno namenjeno razvijanju v programskem jeziku Java. Omogoča integracijo vtičnikov, preko katerih ponuja možnost razvijanja v drugih programskih jezikih, kot so C, C++, PHP, Python, itd. Na uradni spletni strani so na voljo različni paketi, ki vsebujejo osnovna orodja za delo v določenem programskem jeziku. V diplomskem delu uporabljamo paket Eclipse IDE for Java Developers (Mars release) z namenom pisanja JUnit testov.

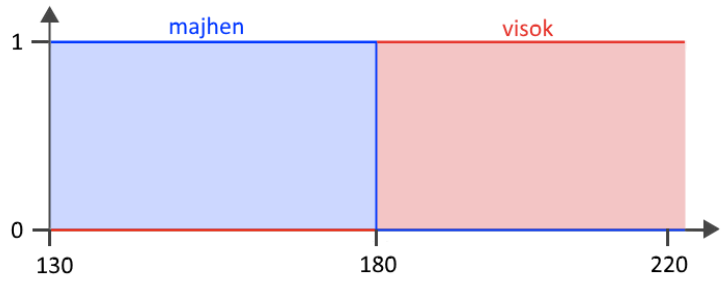
Poglavje 3

Mehka logika

V vsakdanjem življenju si pri opisovanju lastnosti pomagamo z besedami (npr. velik, rdeč ...), ki pa jih ne moremo neposredno uporabiti v matematičnih izračunih. Potrebujemo način, kako lahko združimo izmerjene vrednosti ter opisne kvalifikatorje in z njimi računamo. Računalniški svet praviloma temelji na Boolovi logiki, kar poenostavljeno pomeni, da vsak primer lahko popolnoma drži oziroma ne drži. Taka definicija je za nekatere probleme povsem primerna, vendar pa lahko hitro opazimo pomanjkljivosti pri opisovanju problemov, ki niso tako strogo, "jasno" definirani. Za lažje razumevanje si poglejmo naslednji primer. Imamo sistem, kjer kot vhodni podatek dobimo posameznikovo višino, na podlagi katere ga klasificiramo. Z namenom poenostavitve problema recimo, da sta na voljo dve stanji: majhen in visok. Predpostavimo, da je naša meja, kjer ocenimo osebo kot visoko, 180 cm. Takšno opredelitev prikazuje slika 3.1, matematično pa bi jo zapisali takole:

$$majhen(x) = \begin{cases} 0: & \text{if } x \geq 180 \\ 1: & \text{if } x < 180, \end{cases} \tag{3.1}$$

$$visok(x) = \begin{cases} 1: & \text{if } x \geq 180 \\ 0: & \text{if } x < 180. \end{cases} \tag{3.2}$$



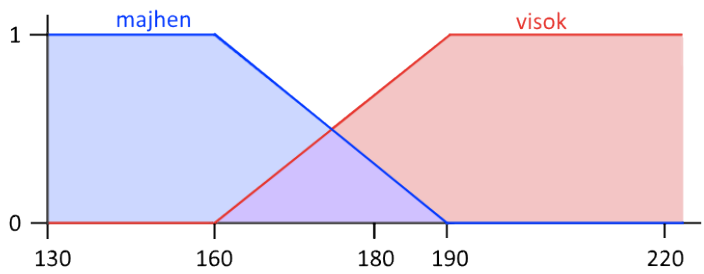
Slika 3.1: Ostri pripadnostni funkciji razredov majhen in visok.

Vzemimo sedaj osebo, ki je visoka 179 cm. Hitro lahko opazimo pomanjkljivosti našega pristopa, saj bo oseba izvzeta iz množice visokih ljudi, pa čeprav je njena višina zelo blizu naše meje. Na tem mestu vstopi mehka logika, ki razširi obstoječo množico pripadnosti iz $\{0, 1\}$ na interval $[0, 1]$, kar pomeni, da lahko neka trditev drži oz. ne drži tudi le delno. Za naš primer lahko sedaj pripadnost izrazimo kot:

$$majhen(x) = \begin{cases} 1: & \text{if } x < 160 \\ -\dfrac{(x-160)}{190-160}: & 160 \leq x \leq 190 \\ 0: & sicer, \end{cases} \tag{3.3}$$

$$visok(x) = \begin{cases} 1: & \text{if } x \geq 180 \\ \dfrac{(x-160)}{190-160}: & 160 \leq x \leq 190 \\ 0: & sicer. \end{cases} \tag{3.4}$$

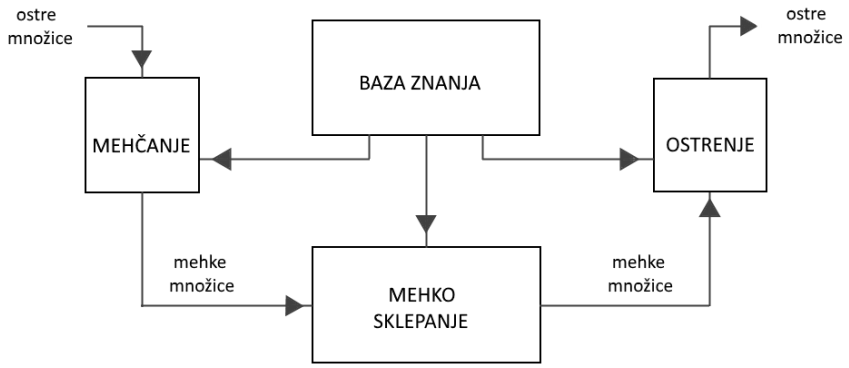
Opredeljevanje višine je tako razširjeno z vmesnimi vrednostmi, ki segajo od 0 do 100%. V našem primeru to pomeni, da je delež pripadnosti k množici visokih ljudi za osebo, ki meri 179 cm, približno enak 63%. Prav tako velja, da je delež pripadnosti k množici majhnih ljudi 37%. Tako opredelitev prikazuje slika 3.2.



Slika 3.2: Mehki pripadnostni funkciji razredov majhen in visok.

3.1 Mehki sistem

Mehka logika torej razširi ostre množice, definirane na podlagi konvencionalne računalniške logike tako, da omogoča delno pravilnost. V tem poglavju si bomo ogledali postopek pretvarjanja in računanja z mehkimi množicami. Pregled nad sistemom nam prikazuje slika 3.3, vrednosti, uporabljene v opisu, pa izhajajo iz diplomske naloge.



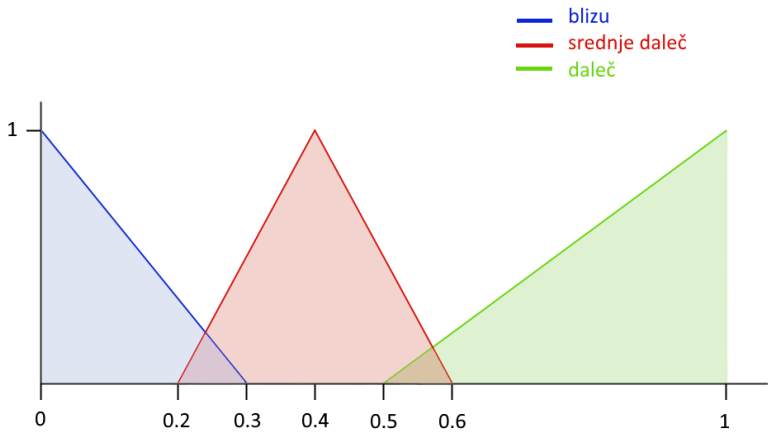
Slika 3.3: Mehki sistem.

3.2 Mehčanje ostrih vrednosti

V mehki sistem dobimo kot vhod ostre vrednosti, ki jih je potrebno pretvoriti v mehke vrednosti. Temu postopku pravimo mehčanje (angl. fuzzification) in poteka preko pripadnostnih funkcij, ki slikajo ostre vrednosti v mehke. To storimo v odvisnosti od našega predznanja glede natančnosti vhodnih podatkov. V našem primeru privzamemo natančne vhodne podatke, zato je mehčanje v obliki ostre vrednosti (angl. singleton). To je pripadnostna funkcija, ki vrne vrednost 1 zgolj v primeru ostre vrednosti x, drugače vrne vrednost 0.

3.3 Baza znanja

Baza znanja vsebuje opise dvoumnih termov, ki se lahko pojavijo v mehkih pravilih. Poglejmo primer. Opisati želimo oddaljenost entitete glede na drugo entiteto. Oddaljenost opišemo kot blizu, srednje daleč in daleč. Temu sovpadajo pripadnostne funkcije, ki so prikazane na sliki 3.4.



Slika 3.4: Pripadnostne funkcije za oddaljenost.

V našem primeru pravimo, da so pripadnostne funkcije trikotne oblike. Poleg trikotne so pogoste oblike še trapezoidna, linearna po delih, gausova ter v obliki zvonca [11]. Hkrati baza znanja vsebuje tudi druge parametre, pomembne za vršenje mehkega sklepanja, kot so interpretiranje posameznih logičnih operacij. V našem primeru je konjunkcija izražena kot produkt, disjunkcija kot algebraična vsota, implikacija kot produkt in akumulacija kot algebraična vsota.

3.4 Mehko sklepanje

Vedenje naših entitet v sistemu narekujejo mehka pravila, ki se upoštevajo pri fazi mehkega sklepanja in slikajo vhodne mehke podatke v izhodne mehke podatke. Ta pravila so IF-THEN oblike, posamezno pravilo lahko zapišemo kot:

$P^{(i)}$: IF (x_1 IS $X_1^{(i)}$) AND (x_2 IS $X_2^{(i)}$) AND ... AND (x_n IS $X_n^{(i)}$) THEN (y IS $Y^{(i)}$),

kjer posamezni x_i predstavlja mehko vhodno spremenljivko, y pa mehko izhodno spremenljivko. Pravilo ima lahko na vhodni strani več vrednosti, ki jih je potrebno združiti v eno samo mehko vrednost, ki jo bomo upoštevali v postopku mehke implikacije. Temu združevanju pravimo agregacija in temelji na operacijah definiranih nad mehkimi množicami [12]. V naslednjem koraku je potrebno v postopku akumulacije združiti izhodne vrednosti posameznih pravil. Tako kot pri agregaciji in mehki implikaciji, so tudi tu na voljo različni načini implementacije.

3.5 Ostrenje mehkih vrednosti

Kot smo omenili, je izhod mehkega sklepanja zopet mehka množica z ustrezno funkcijo pripadnosti, zato jo je potrebno pretvoriti v ostro množico. Obstaja

več metod ostrenja (angl. defuzzification), predstavili bomo le najbolj pomembne [15]. Izbira metode ostrenja je odvisna od karakteristike problema, zato mora razvijalec presoditi, katera metoda najbolj ustreza njegovemu problemu.

3.5.1 Težiščna metoda

Težiščna metoda (angl. COG, center of gravity) sodi med najbolj uporabljene metode za ostrenje. V prvem koraku izračuna ploščino pod krivuljo unije skaliranih izhodnih pripadnostnih funkcij. Nato izračuna geometrijsko sredino dobljene ploščine. Prednost te metode je, da upošteva vsa pravila, ki imajo izpolnjenost pogojev večjo od 0, slabost težiščne metode pa predstavlja računska zahtevnost.

3.5.2 Metoda središča vsot

Pri metodi središča vsot (angl. COS, center of sums) se v prvem koraku izračuna geometrijska sredina ploščine posamezne skalirane izhodne pripadnostne funkcije. Nato se izračuna uteženo povprečje geometrijskih sredin ploščin posameznih pripadnostnih funkcij. Razlika s težiščno metodo je v tem, da se pri metodi središča vsot obravnava ploščina posamezne pripadnostne funkcije, medtem ko težiščna metoda temelji na uniji pripadnostnih funkcij. Metoda središča vsot je, v okviru procesiranja, hitrejša od težiščne, kljub temu, pa smo kot metodo ostrenja izbrali težiščno metodo, saj je le-ta referenčna.

3.5.3 Metode maksimumov

V to skupino spadajo tri različne metode maksimumov: metoda najnižjega maksimuma (ang. FOM, first of maxima), metoda najvišjega maksimuma (angl. LOM, last of maxima) ter metodo srednjega maksimuma (angl. MOM, middle of maxima). Pri vseh treh metodah se upošteva mehko pravilo, ki

najbolj učinkuje na rezultat. Skupno vsem trem metodam je tudi to, da so hitre s stališča procesiranja.

Poglavje 4

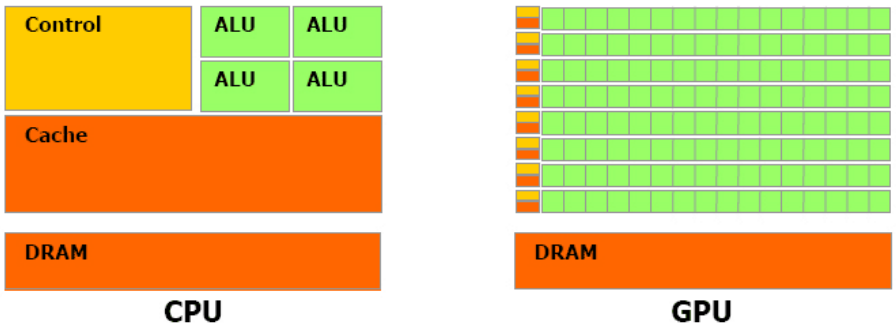
CUDA

CUDA (angl. Compute Unified Device Architecture) je platforma za splošno procesiranje na grafičnih karticah, ki jo je izdalo in ureja podjetje nVIDIA. V preteklosti so programerji za namen izvajanja računskih problemov na GPE uporabljali senčilne jezike, kot sta GLSL (angl. OpenGl Shading Language) in HLSL (angl. High Level Shading Language). CUDA je bila narejena z namenom olajšanja tega postopka, saj njena arhitektura vsebuje poenoten senčilni cevovod, ki omogoča dostop programa do vsake aritmetično logične enote na GPE [13]. Uporaba je omejena le na grafične kartice podjetja nVIDIA, natančen spisek podprtih kartic lahko najdemo na njihovi spletni strani[1], dostopen pa je tudi seznam njihovih starejših modelov[2]. V nadaljevanju bomo kot grafične kartice poimenovali kartice podjetja nVIDIA, ki podpirajo arhitekturo CUDA.

4.1 Arhitektura

Izvedbe vzporednih rešitev na grafičnih karticah prinašajo precejšne pohitritve od implementacij na CPE. Poglaviten razlog za take pohitritve se nahaja v arhitekturni razliki med CPE in GPE, prikazani s sliko 4.1. Zgradba CPE

[1] https://developer.nvidia.com/cuda-gpus
[2] https://developer.nvidia.com/cuda-legacy-gpus



Slika 4.1: Razlika v arhitekturi med CPE in GPE.

je zasnovana z namenom optimizacije izvrševanja sekvenčne kode [9] in povdarkom na velikem predpomnilniku, medtem ko imamo na GPU poudarjeno veliko število aritmetično logičnih enot (ALE). Od začetka z letom 2006 do danes so nastale različne iteracije mikroarhitekture. Znane so pod imeni Tesla, Fermi, Kepler ter najnovejša Maxwell. Vsaka kartica je zgrajena iz številnih enot SM (angl. Streaming Multiprocesors). Posamezen SM pa vsebuje nabor izvajalnih, SP (angl. Stream Processors), elementov, ki so kasneje dobili naziv CUDA jedra (angl. CUDA cores). Število SM in SP se razlikuje med posameznimi modeli GPE.

4.2 Pomnilnik

Pri snovanju programa za GPE je zelo pomembno, da razmislimo, kako bomo hranili svoje podatke, saj se časovni dostopi do različnih tipov pomnilnika zelo razlikujejo (glej tabelo 4.1), kar pa ima lahko velike posledice za uspešnost našega programa.

4.2.1 Registri

Predstavljajo časovno najbolj optimalen prostor za hranjenje podatkov. Za razliko od CPE ima GPE na tisoče registrov na voljo za posamezni SM. Število registrov na posamezno nit v ščepcu (glej poglavje 4.3.2) je določeno

Tabela 4.1: Dostopni časi do posameznih pomnilniških struktur.

tip	Registri	Skupni pomnilnik	Teksturni pomnilnik	Konstantni pomnilnik	Globalni pomnilnik
Pasovna širina	∼8 TB/s	∼1.5 TB/s	∼200 MB/s	∼200 MB/s	∼200 MB/s
Latenca	1 cikel	1 - 32 ciklov	∼400 - 600	∼400 - 600	∼400 - 600

ob času prevajanja programa. Uporaba registrov je za programerja transparentna, vendar lahko ščepec, ki potrebuje preveliko število registrov na posamezno nit, omejuje število blokov, ki se lahko izvajajo na posameznem SM in posledično skupno število niti.

4.2.2 Skupni pomnilnik

Predpomnilnik se je na GPE pojavil šele z arhitekturo Fermi. Do takrat je skupni pomnilnik služil kot edina alternativa predpomnilnika, ki je upravljan s strani programerja. Opisuje naslovni prostor, ki je viden nitim znotraj skupnih SM enot. Spremenljivki, za katero želimo, da se nahaja v skupnem pomnilniku, dodamo predpono `__shared__`. Kot smo že prikazali s tabelo 4.1, je skupni pomnilnik veliko hitrejši od globalnega, zato ga je zaželeno uporabiti, kjer problem to dopušča. Ponavadi ga uporabimo pri vmesnih izračunih, ko problem razdelimo med posamezne bloke. Pri uporabi skupnega pomnilnika mora programer paziti na tvegano stanje, kjer si pomaga z vgrajenimi ukazi za sinhronizacijo niti.

4.2.3 Lokalni pomnilnik

Lokalni pomnilnik fizično ne obstaja, temveč gre za abstraktni pojem. Gre za rezerviran prostor v globalnem pomnilniku, ki se zasede v primeru, ko posamezna nit porabi vse svoje razpoložljive registre. Dostop do tega pomnilnika je veliko počasnejši kot dostop do registrov.

4.2.4 Globalni pomnilnik

Globalni pomnilnik predstavlja največjo ter najpočasnejšo obliko pomnilnika na grafični kartici. Ponavadi v programu do glavnega pomnilnika dostopamo vsaj dvakrat. Prvič pri prenašanju podatkov iz gostitelja, s katerimi želimo, da ščepec operira. Drugi dostop pa je potreben, če želimo podatke uporabiti zunaj ščepca, saj je glavni pomnilnik edini pomnilnik, ki hrani podatke med različnimi zagoni ščepcev.

4.2.5 Konstantni pomnilnik

Konstantni pomnilnik je del glavnega pomnilnika in je v večini primerov velik 64 KB. Med izvajanjem ščepca so vanj omogočeni le bralni dostopi. Ti so hitrejši kot pa dostopi do navadnega glavnega pomnilnika, ker omogočajo predpomnenje. Bralni dostop za niti (glej poglavje 4.3.1), ki predstavljajo prvo ali drugo polovico zvitka niti, dosega podobne hitrosti, kot pri dostopu do registrov, dokler dostopajo do enakega pomnilniškega naslova. V programski kodi konstanto spremenljivko deklariramo s predpono `__constant__`.

4.2.6 Teksturni pomnilnik

Podobno kot konstantni pomnilnik je tudi teksturni pomnilnik del glavnega pomnilnika, do katerega so znotraj ščepca mogoči le bralni dostopi ter podpira predpomnenje. Kljub temu, da je bil teksturni pomnilnik narejen z namenom izrisovanja grafike, omogoča tudi splošno računanje na GPE. Za dober izkoristek je pomembno, da so pomnilniški dostopi prostorsko lokalizirani, kar v grobem pomeni, da bodo zaporedne niti dostopale do naslovov, ki so si medsebojno blizu. Ta lastnost za nas pomeni, da bodo niti znotraj enakega zvitka, ki berejo prostorsko lokalizirane pomnilniške naslove, nudile najboljši strojni izkoristek. V primeru, da želimo računske podatke preko grafičnega cevovoda tudi izrisati, je uporaba te vrste pomnilnika obvezna.

4.3 Pisanje programov

Struktura programov, napisanih z uporabo platforme CUDA, se deli v dve kategoriji. Prva zajema ukaze, ki jih bo izvajal CPE, in tipično zajema opravila kot so inicializacija struktur na CPE, inicializacija struktur na GPE, prenašanje podatkov iz CPE na GPE ter obratno, zaganjanje ščepcev, sprostitev pomnilnika, itd. Druga kategorija zajema ukaze, ki se bodo izvajali na grafični kartici (angl. device), kamor sodijo implementacije ščepcev ter morebitnih funkcij, ki se kličejo znotraj le-teh. Funkcijo, za katero želimo, da se izvede na grafični kartici, označimo s predpono `__device__` , s predpono `__host__` pa označimo funkcijo, za katero želimo, da se izvede na CPE.

Kodo, v kateri uporabljamo CUDA ukaze, shranjujemo v ".cu" oziroma ".cuh" datotekah, prevajamo jih pa z NVCC prevajalnikom.

4.3.1 Zvitek niti

CUDA grafične kartice niti združujejo v zvitke po tipično 32 niti. Delitev se izvede tako, da indeks niti ostaja zvezen. Vsaka nit, kot del zvitka, izvaja isti programski ukaz na različnih podatkih. V primeru, ko operand, do katerega nit znotraj zvitka dostopa, ni pripravljen za uporabo, se napredovanje v tem zvitku ustavi. Možna je menjava zvitka z menjavo konteksta. Na vsakem SP se izvede celoten zvitek, na vsakem SM pa se sočasno lahko izvede maksimalno število zvitkov, ki ga narekuje arhitektura grafične kartice.

4.3.2 Ščepec

Ščepec je ime za funkcijo, ki se lahko izvede le na napravi (GPE) in ne na strani gostitelja (CPE). Klic je možen le s strani gostitelja. V kodi jo označimo s predpono `__global__`. Funkcija mora vračati tip `void`, njen klic izvedemo z naslednjo sintakso:

```
KernelName<<< 1, N >>> (A, B, C);
```

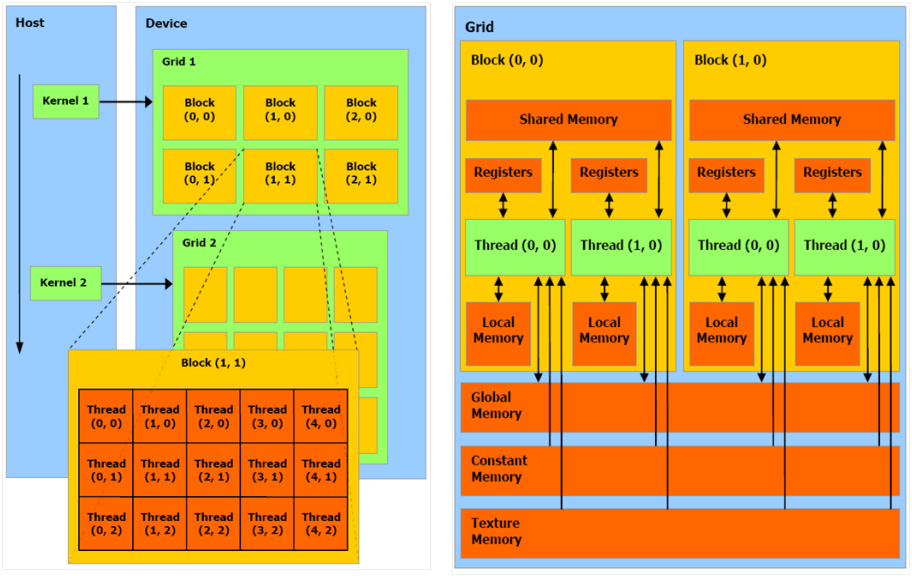
Kot vidimo, se jedra poganja na nekoliko poseben način. V prostoru, ki se nahaja med <<< in >>>, imamo definirana dva zagonska parametra. Prvi parameter predstavlja število blokov (glej poglavje 4.3.3) znotraj posamezne mreže (glej poglavje 4.3.4). Drugi parameter pa predstavlja število niti znotraj enega bloka. Nato v navadnih oklepajih sledijo parametri funkcije, ki smo jih morali pred klicem ustrezno inicializirati na strani gostitelja (CPE).

4.3.3 Blok

Blok je skupek niti, katerih število je odvisno od parametra, ki smo ga podali pri zagonu ščepca. Predstavlja kopijo ščepca, ki se bo izvedla na teh nitih. Za lažje razumevanje si poglejmo sliko 4.2. Posamezni blok lahko identificiramo s pomočjo x, y in z indeksov. Na grafični kartici se lahko sočasno izvaja toliko blokov, kolikor imamo SM enot. V primeru, da želimo zagnati več blokov, kot pa imamo SM enot, za njihovo preklapljanje skrbi posebna čakalna vrsta. Maksimalno število niti v posameznem bloku je definirano s strani arhitekture GPE. Ravno tako je definirano tudi maksimalno število blokov, ki jih lahko uporabimo pri zagonu ščepca. Vredno je tudi opomniti, da se število lahko spremeni, glede na uporabljeno dimenzijo (1D, 2D, 3D). Z namenom optimizacije naj bo število niti v bloku večkratnik velikosti zvitka niti (števila niti v zvitku).

4.3.4 Mreža

Mrežo predstavlja združevanje niti na najvišjem nivoju. Sestavljajo jo posamezni bloki. Mreža je lahko predstavljena v 1D, 2D ali 3D, vendar nekatere grafične kartice omogočajo le 1D in 2D strukturo. Podobno kot pri blokih lahko mrežo identificiramo z x, y in z indeksi.



Slika 4.2: Gradbeni elementi v okolju CUDA.

4.4 Alternative

Pohitritev danega algoritma je možno izvesti na številne načine. Za namene nitenja na procesorju so programerjem na voljo različne rešitve, od katerih bomo na kratko opisali nitenje z uporabo OpenMP. Nekaj besed bomo namenili tudi ogrodju OpenCL.

4.4.1 OpenMP

OpenMP (angl. Open Multi-Processing)[3] je programski vmesnik, ki podpira multiprocesiranje s skupnim pomnilnikom preko različnih platform. Sestavljajo ga številni ukazi za prevajalnik, nabor funkcij in okoljske spremenljivke, ki vplivajo na dogajanje v času izvajanja. Omogoča enostavno pisanje paralelnih programov.

[3] http://openmp.org/wp/

4.4.2 OpenCL

OpenCL (angl. Open Computing Language)[4] predstavlja ogrodje za pisanje vzporednih programov, ki se lahko izvajajo na centralno procesnih enotah, grafičnih procesnih enotah, procesorjih, namenjenih za procesiranje digitalnih signalov ter drugih arhitekturah. Način pisanja programov v OpenCL je podoben tistemu v okolju CUDA, glavna prednost pa je omogočeno izvajanje na arhitekturi različni od GPE ter večje število podprtih grafičnih kartic; CUDA je na voljo le na grafičnih karticah podjetja nVIDIA, med tem ko je OpenCL podprt tudi na karticah drugih podjetji (npr. AMD).

[4]https://www.khronos.org/opencl/

Poglavje 5

Algoritem

V diplomskem delu želimo na GPE implementirati algoritem, ki za posamezno entiteto na podlagi mehke logike izračuna njeno spremembo smeri gibanja. Sprememba smeri je odvisna od entitet, ki so od trenutne oddaljene največ za radij vidljivosti. V tem poglavju bomo podrobneje opisali posamezne korake izvajanja algoritma.

Algoritem se v grobem sestoji iz treh delov:

- inicializacije,
- postopkov za iskanje entitete, s katero pride do interakcije,
- ter postopkov mehkega računanja.

Iskanje sosednje entitete, s katero pride do interakcije, smo pohitrili z delitvijo prostora na posamezne celice. Kako potekajo koraki iskanja, podrobneje opišemo v poglavju 5.2. V primeru, ko ima trenutno obravnavana entiteta veljavno sosedo (glede na parametre nastavljene v datoteki "Settings.h"), nastopi mehko računanje. Tu z uporabo baze znanja vhodnim ostrim podatkom priredimo ustrezne mehke množice, izvedemo mehko sklepanje ter preko ostrenja izračunamo končno ostro vrednost, s katero lahko ustrezno spremenimo gibanje entitete.

5.1 Inicializacija

Inicializacijski postopek je identičen za CPE rešitev kot za CUDA rešitev. V njem generiramo N entitet in jim določimo identiteto, položaj v prostoru ter smer gibanja. V tem koraku inicializiramo tudi strukture, ki jih bomo kasneje uporabili pri mehkem računanju.

5.2 Postopki iskanja soseda

Kot smo že omenili, je sprememba smeri gibanja entitete odvisna od njene sosede. Pri naivni rešitvi, bi pri iskanju ustreznega soseda morali iterirati skozi celoten seznam entitet, kar nam pri N entitetah predstavlja časovno zahtevnost $O(N^2)$. Z namenom optimizacije, se iskanja sosedov lotimo z delitvijo prostora, kar opisujejo naslednji štirje koraki.

5.2.1 Delitev prostora na celice

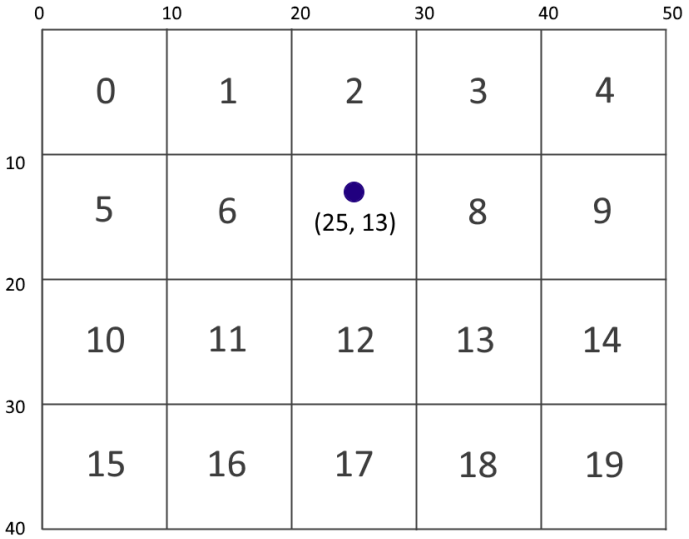
Prostor, po katerem se lahko gibljejo naše entitete, je omejen, zato ga lahko razdelimo na posamezne kvadratne celice. Velikost stranic je podana kot konstanta in jo lahko spreminjamo v datoteki "Settings.h". Delitev izvedemo v ščepcu, kjer vsaka nit izračuna indeks celice, v katero sodi glede na svoj položaj. Pri računanju indeksa celice upoštevamo, da začnemo celice šteti pri indeksu 0. Denimo, da pos_x in pos_y predstavljata x in y koordinato entitete, M predstavlja število celic v vrstici ter $cell_{size}$ velikost stranice mrežne celice. Potem lahko izračunamo indeks celice c_{id}, v kateri se trenutno nahajamo:

$$x = \lceil \frac{pos_x}{cell_{size}} \rceil - 1, \tag{5.1}$$

$$y = \lceil \frac{pos_y}{cell_{size}} \rceil - 1, \tag{5.2}$$

$$c_{id} = y \times M + x. \tag{5.3}$$

Pri izračunih vedno uporabljamo podatke v 1D, zato smo temu ustrezno prilagodili izračun indeksa. Z namenom lažje predstave, si poglejmo izračun



Slika 5.1: Primer entitete v prostoru.

na primeru. Recimo, da imamo naš prostor 50×40 razdeljen na 20 enakih kvadratnih celic, v vsaki vrstici po 5. Velikost stranice celice je tako 10. Za entiteto iz slike 5.1 potem indeks izračunamo kot:

$$x = \lceil \frac{25}{10} \rceil - 1 = 2, \tag{5.4}$$

$$y = \lceil \frac{13}{10} \rceil - 1 = 1, \tag{5.5}$$

$$c_{id} = 1 \times 5 + 2 = 7. \tag{5.6}$$

Indeksi celic se shranijo v seznam $c_{unsorted}$. Za posamezno entiteto z indeksom e_{id} velja, da je njen indeks celice zapisan v $c_{unsorted}[e_{id}]$. Poleg umeščanja entitete v celico, v posamezni niti z indeksom t_{id} zapišemo še kopijo indeksa trenutno izračunane celice v $c_{sorted}[t_{id}]$ ter kopijo indeksa entitete v $e_{sorted}[t_{id}]$, ki ju bomo potrebovali pri sortiranju entitet.

5.2.2 Sortiranje entitet

V tem koraku sortiramo indekse entitet zapisane v e_{sorted} na podlagi naraščajočih indeksov celic. Sortiranje izvedemo nad kopijami podatkov, ki smo jih pripravili pri vmeščanju entitet v celice. Implementacijo izvedemo s pomočjo knjižnice Thrust s sintakso, prikazano na izpisu 5.1. Uporabljen sortirni algoritem je CUDA implementacija algoritma radix sort. Ob končanem koraku imamo v c_{sorted} zapisane naraščajoče indekse celic in v e_{sorted} temu ustrezno prirejene indekse entitet.

Izpis 5.1: Sortiranje s knjižnico Thrust

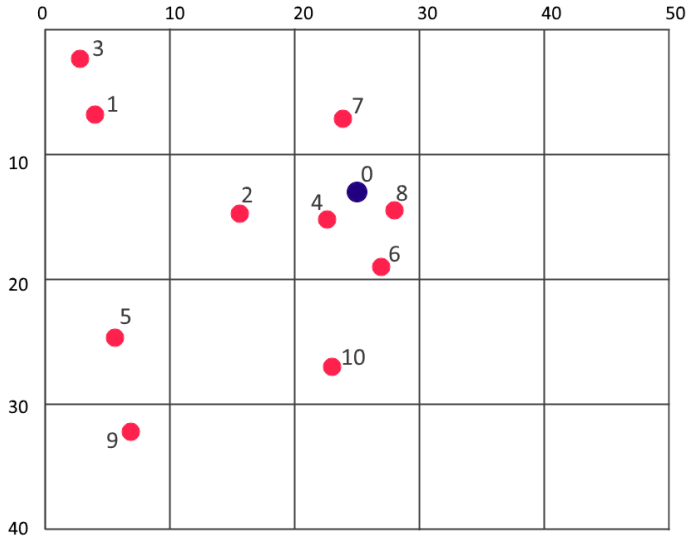
```
thrust::sort_by_key
(
  RandomAccessIterator1 keys_first,
  RandomAccessIterator1 keys_last,
  RandomAccessIterator2 values_first
);
```

5.2.3 Število entitet v posamezni celici

Z namenom lažje razlage dopolnimo primer s slike 5.1 tako, da v prostor dodamo še 10 entitet, kot prikazuje slika 5.2. Za posamezno polno celico preštejemo, koliko entitet se v njej nahaja. To izvedemo tako, da se sprehodimo skozi niz indeksov urejenih celic c_{sorted}. Naj t_{id} predstavlja trenutni indeks niti v jedru. Vsaka nit v jedru prebere vrednost $v = c_{sorted}[t_{id}]$. V primeru, da se v razlikuje od vrednosti $c_{sorted}[t_{id}-1]$, velja da je $c_{start}[v] = t_{id}$. Podobno v primeru, ko se vrednost v razlikuje od vrednosti $c_{sorted}[t_{id}+1]$, velja da je $c_{end}[c_{id}] = t_{id}$. Za primer s slike 5.2 je omenjena rešitev prikazana s tabelo 5.1.

Tabela 5.1: Primer struktur za iskanje sosednjih entitet.

c_{id}	$e_{unsorted}$	$c_{unsorted}$	c_{sorted}	e_{sorted}	c_{start}	c_{end}
0	0	7	0	3	0	1
1	1	0	0	1	-1	-1
2	2	6	2	7	2	2
3	3	0	6	2	-1	-1
4	4	7	7	0	-1	-1
5	5	10	7	4	-1	-1
6	6	7	7	8	3	3
7	7	2	7	6	4	7
8	8	7	10	5	-1	-1
9	9	15	12	10	-1	-1
10	10	12	15	9	8	8
11	/	/	/	/	-1	-1
12	/	/	/	/	9	9
13	/	/	/	/	-1	-1
14	/	/	/	/	-1	-1
15	/	/	/	/	10	10
16	/	/	/	/	-1	-1
17	/	/	/	/	-1	-1
18	/	/	/	/	-1	-1
19	/	/	/	/	-1	-1



Slika 5.2: V prostor dodamo še 10 entiet.

5.2.4 Iskanje soseda

Na primeru s slike 5.3 podrobneje razložimo uporabnost dodatnih struktur. Trenutna entiteta, z indeksom $e_{id} = 0$, je prikazana z modro barvo ter se nahaja v celici z indeksom $c_{id} = 7$. Zelen krog predstavlja radij dosega njenega pogleda, torej entitete znotraj tega kroga so primerni kandidati za interakcijo. Algoritem iskanja sosedov izvedemo tako, da kot kandidate upoštevamo entitete, ki so znotraj pripadajočih celic, ki jih seka krog vidljivosti. Te celice označimo z oranžno barvo. Entitete, ki se nahajajo v eni izmed obarvanih celic in se ne nahajajo znotraj kroga vidljivosti, bomo kasneje odstranili oziroma jih ne bomo upoštevali pri računanju. Oglejmo si, kako rešitev poteka v našem programu.

Naj t_{id} predstavlja indeks niti. Vsaka nit v ščepcu izvaja postopek za svojo entiteto, ki jo prebere iz seznama neurejenih entitet, kar pomeni, da bo $e_{id} = t_{id}$. V našem primeru obravnavamo entiteto $e_{id} = 0$. Želimo izvedeti, v kateri celici se naša entiteta nahaja. To informacijo dobimo tako, da iz

seznama neurejenih celic $c_{unsorted}$ (glej poglavje 5.2.1) preberemo vrednost $c_{id} = c_{unsorted}[t_{id}]$, kjer c_{id} predstavlja indeks celice, v kateri se naša obravnavana entiteta nahaja. Za primer iz slike 5.3 torej velja $c_{id} = c_{unsorted}[0] = 7$ (glej tabelo 5.1). Sedaj, ko vemo v kateri celici se trenutno nahajamo, želimo izvedeti, koliko sosedno ležečih celic pride v poštev pri iskanju kandidatov za sosedno entiteto (oranžne celice pri sliki 5.2). Kot smo že omenili, so te celice v preseku s krogom vidljivosti entitete, torej lahko doseg *range*, ki predstavlja število celic, ki jih zajema radij kroga, izračunamo kot

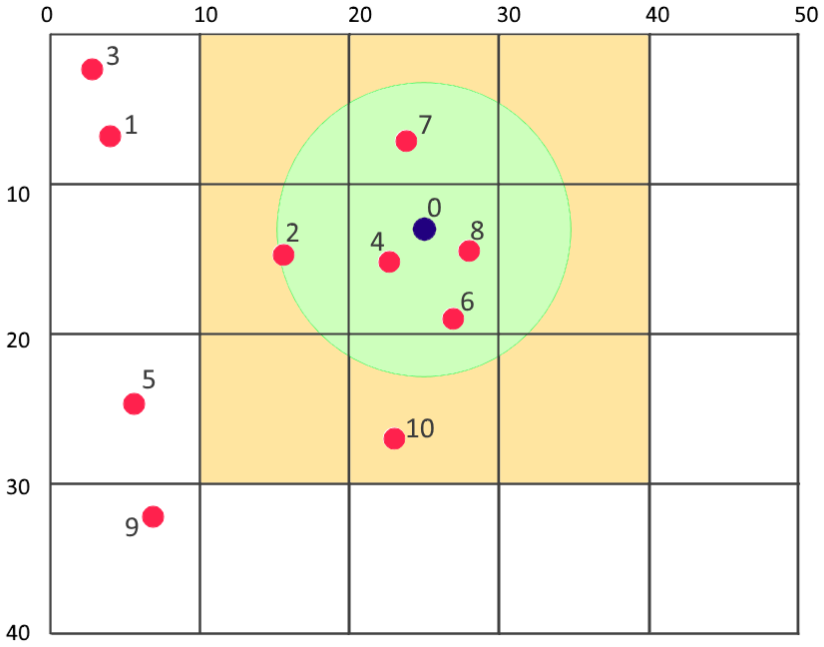
$$range = \lceil \frac{visibility}{cell_{size}} \rceil, \tag{5.7}$$

kjer *visiblity* predstavlja radij našega pogleda ter $cell_{size}$ velikost stranice celice. V našem primeru s slike 5.3 je $range = 1$. Sosednje celice določimo tako, da se iz trenutne celice premaknemo v vsaki smeri od skupnih 8 (levo, desno, navpično gor, navpično dol ter štiri diagonale) za *range* število celic.

Sedaj, ko imamo določene celice oz. njihove indekse, kjer se lahko nahajajo morebitni kandidati za sosede, pričnemo s preiskovanjem posamezne celice. Za trenutno celico, v kateri se nahajamo, ter vsako celico v nizu sosednjih celic preverimo, če vsebuje kandidate za izbor soseda trenutni entiteti. To naredimo tako, da preberemo vrednost $i_{start} = c_{start}[t_{id}]$ (glej poglavje 5.2.3). V kolikor je ta vrednost večja od -1, vemo, da se v celici nahaja vsaj ena entiteta. V tem primeru do posameznih entitet $entity_i$ znotraj celice dostopamo s for zanko:

```
for (i = i_start; i <= i_end; i++)
{
        entity_i = e_sorted[i];
}
```

kjer e_{sorted} predstavlja niz indeksov urejenih entitet ter velja $i_{end} = c_{end}[t_{id}]$. Pri vsaki entiteti $entity_i$ moramo tudi preveriti, če je njen indeks različen od indeksa trenutne entitete ter če leži znotraj našega vidnega kroga, kar storimo z vektorskimi operacijami. Za primer s slike 5.2 z uporabo opisanega postopka in s pomočjo tabele 5.1 dobimo seznam kandidatov za sosedo



Slika 5.3: Iskanje sosedov.

$\{7, 2, 4, 8, 6\}$.

Naš algoritem predpostavlja interakcijo le med dvema sosednjima entitetama, zato bomo izmed kandidatov za sosede izbrati le eno. To storimo tako, da izmed kandidatov naključno izberemo eno intetiteto, kjer je verjetnost izbire obratno sorazmerna z oddaljenostjo od trenutno obravnavane entitete. Torej bližje kot je naši entiteti, večjo verjetnost ima, da bo izbrana za sosedo.

5.3 Mehko računanje

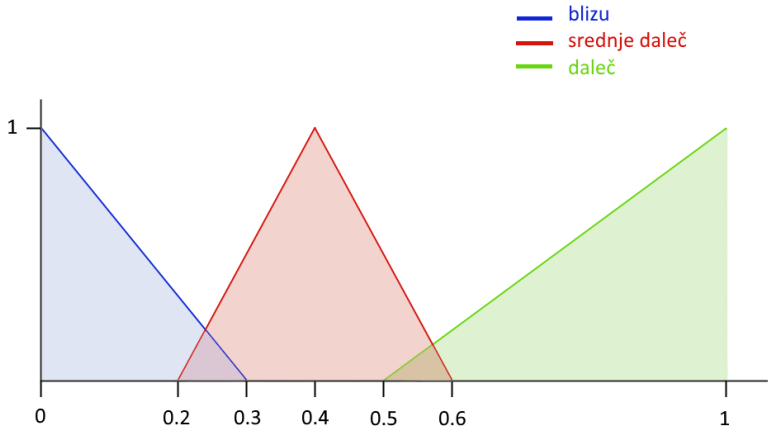
Zadnji korak našega algoritma predstavlja mehko računanje. V tem jedru vsaka nit za pripadajočo entiteto izračuna njeno spremembo gibanja, glede na interakcijo z izbrano sosedo. V prvi fazi je potrebno na osnovi ostrih vre-

dnosti entitete z uporabo pripadnostnih funkcij ugotoviti stopnjo pripadnosti posameznemu dvoumnemu tremu. Vhodni termi v obravnavi se nanašajo na oddaljenost od soseda (slika 5.4), pozicijo (slika 5.5) glede na soseda ter usmerjenost (slika 5.6). Izhodni term predstavlja spremembo usmerjenosti (slika 5.7). Pripadnostne funkcije v koraku inicializacije definiramo kot strukture. Primer strukture za parameter pozicije:

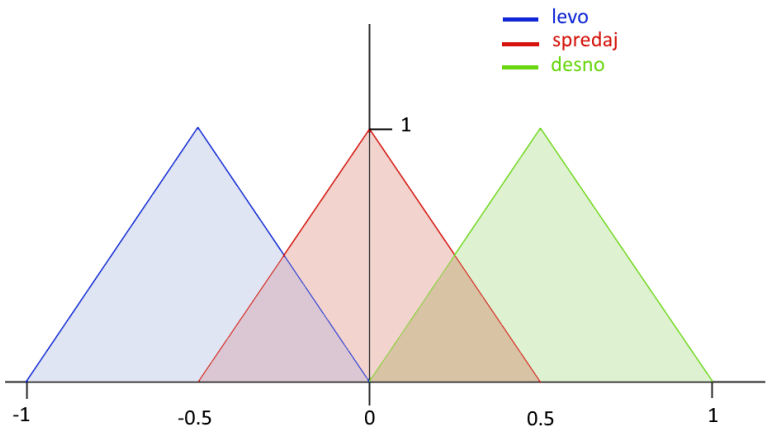
```
struct PreyP
{
        glm::vec2 df;
        glm::vec2 infront[3];
        int infrontLength;
        glm::vec2 right[3];
        int rightLength;
        glm::vec2 left[3];
        int leftLength;
        ...
};
```

Kot vidimo, so posamezne funkcije termov (infront, right, left) definirane kot 2D vektor, kamor shranimo robne točke trikotne pripadnostne funkcije. Pri določanju pripadnosti se potem sprehodimo skozi točke, kjer preverimo, ali ostra vrednost leži med dvema sosednjima paroma točk. V kolikor to velja, si zabeležimo stopnjo pripadnosti termu. Zatem sledi mehko sklepanje. V naši bazi znanja se nahaja naslednjih 9 pravil:

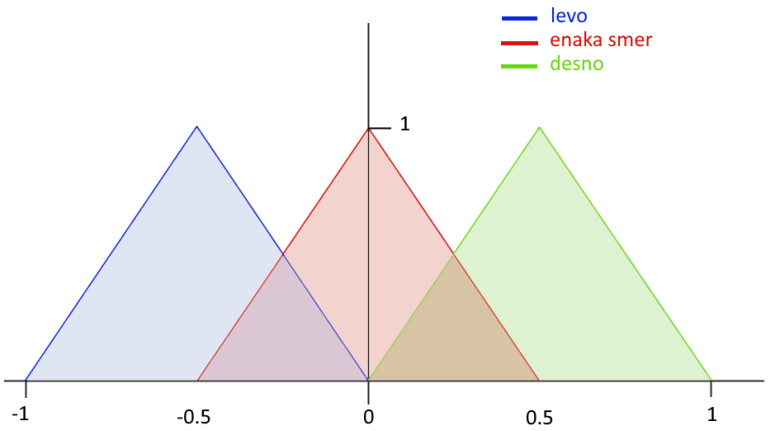
$P^{(1)}$: IF ($preyD$ IS $near$) AND ($preyP$ IS $left$) THEN (Hc IS $right$)
$P^{(2)}$: IF ($preyD$ IS $near$) AND ($preyP$ IS $right$) THEN (Hc IS $left$)
$P^{(3)}$: IF ($preyD$ IS $near$) AND ($preyP$ IS $infront$) THEN (Hc IS $samedir$)
$P^{(4)}$: IF ($preyD$ IS med) AND ($preyH$ IS $left$) THEN (Hc IS $left$)
$P^{(5)}$: IF ($preyD$ IS med) AND ($preyH$ IS $right$) THEN (Hc IS $right$)
$P^{(6)}$: IF ($preyD$ IS med) AND ($preyH$ IS $samedir$) THEN (Hc IS $samedir$)



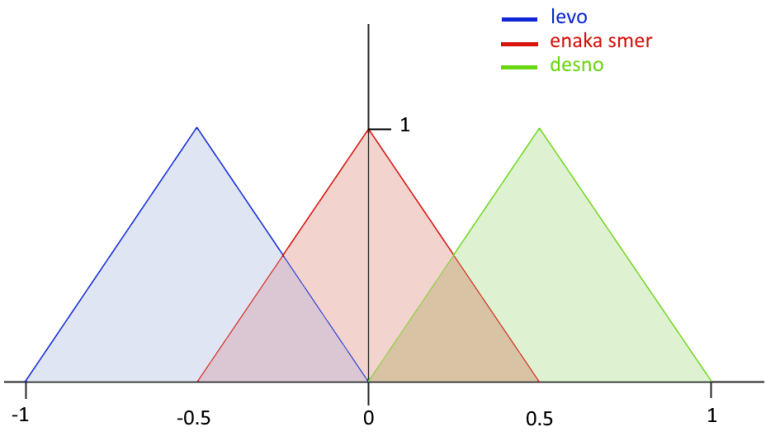
Slika 5.4: Pripadnostne funkcije za parameter oddaljenosti.



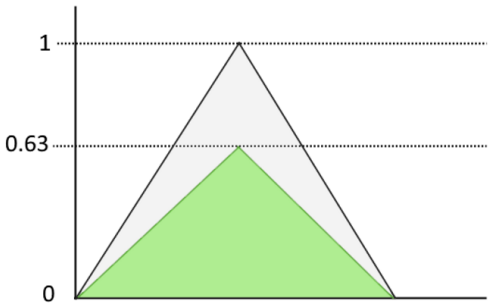
Slika 5.5: Pripadnostne funkcije za parameter pozicije.



Slika 5.6: Pripadnostne funkcije za parameter usmerjenosti.



Slika 5.7: Pripadnostne funkcije za parameter spremembe smeri gibanja.



Slika 5.8: Primer skaliranja izhodne pripadnostne funkcije.

$P^{(7)}$: IF ($preyD$ IS far) AND ($preyP$ IS $left$) THEN (Hc IS $left$)
$P^{(8)}$: IF ($preyD$ IS far) AND ($preyP$ IS $right$) THEN (Hc IS $right$)
$P^{(9)}$: IF ($preyD$ IS far) AND ($preyP$ IS $infront$) THEN (Hc IS $samedir$)

kjer $preyD$ predstavlja oddaljenost glede na soseda, $preyP$ položaj glede na soseda, $preyH$ smer gibanja ter Hc spremembo usmerjenosti. Za vsako izmed pravil izračunamo izpolnjenost pogojev glede na v predhodnjem koraku izračunane pripadnosti termov. Kot primer vzemimo, da je stopnja pripadnosti termu blizu parametra oddaljenosti ($preyD = near$) enaka 0.9, stopnja pripadnosti termu levo parametra položaja ($preyP = left$) pa 0.7. Konjunkcijo smo definirali kot produkt, kar pomeni, da je izpolnjenost pogojnega dela pravila $P^{(1)}$ v tem primeru enaka produktu $0.9 \times 0.7 = 0.63$.

Kot produkt pa je v diplomskem delu definirana tudi implikacija, kar pomeni, da izhodne pripadnostne funkcije ustrezno skaliramo tako, da vsako y vrednost osnovne pripadnostne funkcije, pomnožimo s prej izračunano izplonjenostjo pravila, kot prikazuje slika 5.8. V zadnjem koraku je potrebno na podlagi vseh mehkih izhodnih vrednosti izračunati končno ostro vrednost.

V diplomskem delu smo kot metodo ostrenja izbrali težiščno metodo, opisano v [2]. Kot smo že omenili, za vsako pravilo izračunamo skalirano obliko izhodne pripadnostne funkcije (v našem primeru so to enakostranični trikotniki). Implementacija težiščne metode, ki jo uporabljamo v diplomskem

delu, za posamezno skalirano pripadnostno funkcijo izračuna produkt centroida trikotnika $c^{(i)}$ ter vrednost izpolnjenosti $Hc^{(i)}$ pogojnega dela ustreznega pravila $P^{(i)}$. Te produkte nato sešteje in dobljeno vsoto deli z vsoto $Hc^{(i)}$. Matematično bi metodo zapisali kot:

$$y = \frac{\sum_{i=1}^{N} c^{(i)} \times Hc^{(i)}}{\sum_{i=1}^{N} Hc^{(i)}}. \tag{5.8}$$

5.4 Testiranje

Kot metodo testiranja vmesnih ter končnih rezultatov smo uporabili testiranje v programskem jeziku Java z JUnit. Rešitve programa smo primerjali z rezultati, pridobljenimi s pomočjo jFuzzyLogic knjižnice. Preverili smo pravilnost določitve parov sosedov ter rezultate, pridobljene pri mehkem računanju.

5.4.1 Preverjanje veljavnosti sosedov

Veljavnost parov sosedov smo preverili glede na njuno lokacijo ter radij vidnega polja. V naši rešitvi se na zahtevo ustvari tekstovna datoteka, kjer sta v vrstici shranjena lokacija posamezne entitete ter indeks njenega soseda. Podatki v vrstici so ločeni s podpičjem (;) vrstica pa izgleda tako:

pos_x;pos_y;$index$

kjer pos_x in pos_y predstavljata x in y koordinato entitete v prostoru, $index$ pa predstavlja indeks njene sosede. V tekstovni datoteki številka vrstice predstavlja indeks osebka (začnemo šteti od 0). V testu preverimo, če par točk, ki ju predstavljata lokaciji entitet, leži v razdalji vidljivosti drug od druge. To preverimo s preprostim računom:

$$\|pos_1 - pos_2\| <= visibility \tag{5.9}$$

kjer *visibility* predstavlja maksimalno razdaljo, pri kateri lahko entiteta vpliva na drugo entiteto. V primeru, ko entiteta nima sosede, se v datoteko, kot indeks njene sosede, zapiše število -1. Takrat se v testu preveri, da

$$\forall(e_i, e_j); i, j = 1...N \wedge i \neq j \tag{5.10}$$

velja:

$$\|pos_i - pos_j\| > visibility \tag{5.11}$$

kjer e_i predstavlja trenutno entiteto ter e_j poljubno drugo entiteto v nizu. Torej za trenutno entiteto preverimo, če je od vseh ostalih entitet oddaljena za več kot razdaljo vidljivosti *visibility*.

5.4.2 Preverjanje rezultatov mehkega računanja

Na željo uporabnika se rezultati, pridobljeni pri mehkem računanju, zapišejo v tekstovno datoteko imenovano "hc_output.txt". Pri tem se upošteva format vrstice:

$preyD_{med}$,$preyD_{near}$,$preyD_{far}$,$preyD_{originalValue}$;
$preyP_{infront}$,$preyP_{left}$,$preyP_{right}$,$preyP_{originalValue}$;
$preyH_{samedir}$,$preyH_{left}$,$preyH_{right}$,$preyH_{originalValue}$;
$preyHc_{samedir}$,$preyHc_{left}$,$preyHc_{right}$,$preyHc_{crispValue}$;

kjer *preyD* predstavlja skupino pripadnostnih funkcij oddaljenosti, *preyP* skupino pripadnostnih funkcij pozicije, *preyH* skupino pripadnostnih funkcij usmerjenosti ter *Hc* skupino pripadnostnih funkcij spremembe smeri. Vsaka vrstica predstavlja vrednosti posamezne entitete, ki se primerjajo z vrednostmi, pridobljenimi s pomočjo jFuzzyLogic knjižnice. Za posamezno entiteto se preko začetnih (vhodnih) parametrov izračuna:

- vrednosti pripadnostnih funkcij preyP
- vrednosti pripadnostnih funkcij preyD

- vrednosti pripadnostnih funkcij preyH
- končno ostro vrednost Hc

Vsaka izračunana vrednost $value_{calc}$ se primerja z ustrezno vrednostjo, zapisano v datoteki, $value_{dat}$, tako da se izračuna

$$|value_{dat} - value_{calc}| <= \varepsilon \tag{5.12}$$

kjer ε predstavlja mero natančnosti.

Poglavje 6

Rezultati

V diplomskem delu smo postavili hipotezo, ki pravi, da se bo pri povečanem številu entitet v simulaciji, povečala tudi vrzel med CPE in GPE izvedbenim časom. Primerjave smo opravili pri različnem številu entitet in sicer za:

- implementacijo na CPE (nadalno kot CPE-N),
- implementacijo na CPE z deljenjem prostora (v nadaljevanju CPE-DP),
- implementacijo na GPE z deljenjem prostora (v nadaljevanju GPE-DP),
- implementacijo na GPE z deljenjem prostora in uporabo teksturnega pomnilnika (v nadaljevanju GPE-DP-TP).

Vsako meritev smo opravili 20-krat. Prostor je bil velikosti 1920×1080 enot, posamezna entiteta pa velikosti 1 enote. Stranica posamezne kvadratne celice $cell_{size}$ meri 200 enot, kar sovpada z velikostjo radija vidljivosti *visibility*. Z izjemo števila entitet, so ostali parametri konstantni skozi celotno testiranje. Podatki so bili pridobljeni na računalniku s procesorjem Intel i7-4702MQ (2.20 GHz), pomnilnikom DDR3 8GB, ter grafično kartico GeForce 820M. Za izvajanje je skrbel operacijski sistem Windows 10 Pro 64bit. Velikost bloka na grafični kartici (glej poglavje 4.3.3) smo nastavili na 512.

Tabela 6.1: Primerjava časov izvajanja v milisekudanh ($mean \pm sd$) med CPE-N, CPE-DP ter GPE-DP.

Št. entitet	CPE-N	CPE-DP	GPE-DP
1000	6.234 ± 0.118	2.502 ± 0.136	1.622 ± 0.005
5000	162.120 ± 0.831	40.031 ± 0.282	32.331 ± 0.389
10000	643.721 ± 3.470	159.965 ± 0.918	130.827 ± 1.749
20000	2534.735 ± 4.258	633.074 ± 4.216	547.328 ± 3.300
40000	10718.422 ± 22.308	2636.852 ± 6.564	2280.874 ± 3.079
60000	31796.058 ± 34.323	7222.657 ± 28.113	5258.173 ± 3.078
80000	62663.805 ± 65.970	14383.727 ± 33.669	9524.250 ± 5.245
100000	109269.324 ± 74.310	25413 ± 36.031	15105.867 ± 8.737

V prvem koraku smo izvedli primerjavo med CPE-N, CPE-DP ter GPE-DP. V tabeli 6.1 so za posamezno rešitev prikazani časovi izvajanja v milisekundah ($mean \pm sd$) pri danem številu entitet. Iz dane tabele je razviden doprinos deljenja prostora k optimizaciji.

Pri izpisu povprečnih časov izvajanja posameznega ščepca lahko opazimo, da se čas pri ščepcu, namenjenemu iskanju sosedov, poglavitno razlikuje od ostalih, saj skoraj celotni čas rešitve zaseda računanje sosedov. Povprečen čas izvajanja posameznega ščepca pri 60000 entitetah je prikazan s tabelo 6.2.

Znatno povečanje časa izvajanja omenjenega ščepca je posledica večanja števila možnih kandidatov za sosede, saj pri konstantni velikosti prostora, stranice celice ter radija vidljivosti, večanje števila entitet pomeni večjo nasičenost. Pri pregledu kode ščepca smo opazili ozko grlo pri številnih dostopih do globalnega pomnilnika. Dostopi se velikokrat vršijo do enakih naslovov, zato smo globalne dostope, ki imajo možnost večkratne ponovitve, nadomestili s teksturnim pomnilnikom ter ponovili meritve. Novi izmerjeni podatki, prikazani v tabeli 6.3, prikazujajo izboljšanje GPE rešitve.

Primerjava povprečnih časov ščepcev obeh GPE rešitev je prikazana s tabelo 6.4. Na sliki 6.1 je prikazan graf pohitritev CPE-DP, GPE-DP in GPE-DP-TP glede na CPE-N, kjer smo pri različnem številu entitet primerjali količnike povprečnih časov izvedbe.

Tabela 6.2: Povprečen čas izvajanja posameznega ščepca GPE-DP v milisekundah pri 60000 entitetah.

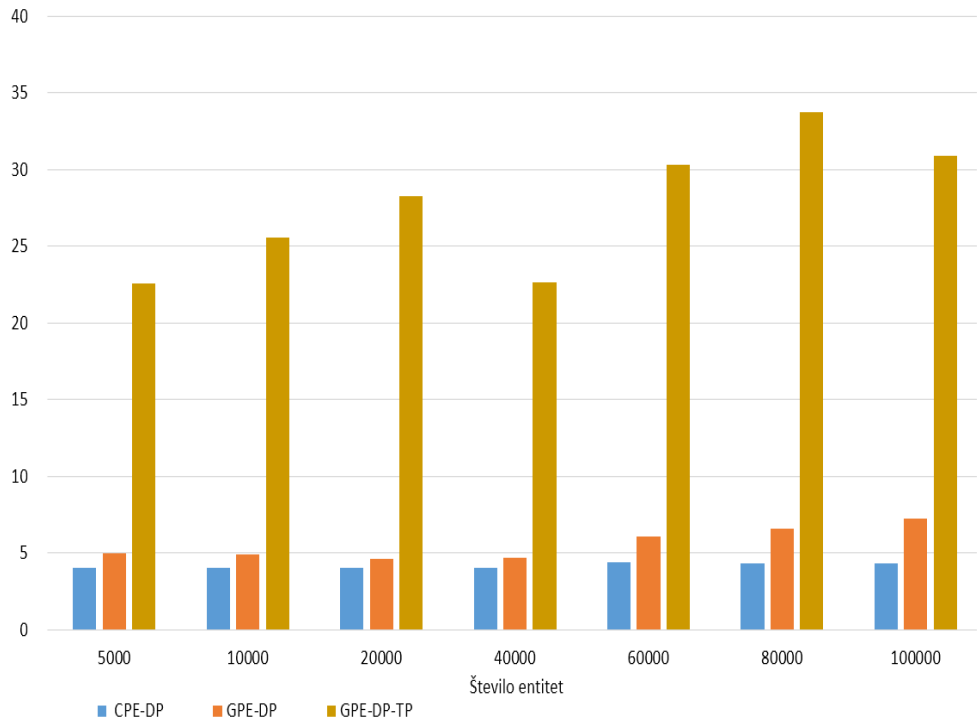
Ščepec	Čas izvajanja (v ms)
Delitev na celice	0.559
Sortiranje	1.045
Priprava seznama zaštetje entitet v celicah	0.009
Štetje entitet v celicah	0.028
Seznam naključnih števil	0.499
Iskanje soseda	5251.494
Mehko računanje	4.540

Tabela 6.3: Primerjava časov izvajanja v milisekudanh ($mean \pm sd$) med CPE-N, CPE-DP ter GPE-DP-TP.

Št. entitet	CPE-N	CPE-DP	GPE-DP-TP
1000	6.234 ± 0.118	2.502 ± 0.136	0.889 ± 0.008
5000	162.120 ± 0.831	40.031 ± 0.282	7.173 ± 0.003
10000	643.721 ± 3.470	159.965 ± 0.918	25.153 ± 0.262
20000	2534.735 ± 4.258	633.074 ± 4.216	89.743 ± 1.240
40000	10718.422 ± 22.308	2636.852 ± 6.564	473.866 ± 2.650
60000	31796.058 ± 34.323	7222.657 ± 28.113	1047.597 ± 3.545
80000	62663.805 ± 65.970	14383.727 ± 33.669	1854.793 ± 3.420
100000	109269.324 ± 74.310	25413 ± 36.031	3537.082 ± 4.772

Tabela 6.4: Povprečen čas izvajanja posameznega ščepca GPE-DP in GPE-DP-TP v milisekundah pri 60000 entitetah.

Ščepec	GPE-DP	GPE-DP-TP
Delitev na celice	0.558	0.670
Sortiranje	1.052	0.755
Priprava seznama zaštetje entitet v celicah	0.008	0.008
Štetje entitet v celicah	0.028	0.028
Seznam naključnih števil	0.499	0.499
Iskanje soseda	5255.481	1041.094
Mehko računanje	4.544	4.542



Slika 6.1: Graf pohitritev glede na CPE-N.

Poglavje 7

Sklepne ugotovitve

V diplomskem delu smo izvedli implementacijo mehkega modela na grafični kartici v okolju CUDA. Uporabljen mehki model temelji na interakciji med sosednjimi entitetami, zato smo vpeljali optimizacijski postopek za pridobivanje sosednjih entitet s pomočjo delitve prostora. Implementirali smo preprosto CPE rešitev, CPE rešitev z deljenjem prostora, GPE rešitev z deljenjem prostora ter GPE rešitev z deljenjem prostora in uporabo teksturnega pomnilnika. Primerjava povprečnih izvedbenih časov rešitev pri različnem številu entitet je potrdila uspešnost optimizacije deljenja prostora. Razvidna je tudi pohitritev obeh GPE implementacij, ki je toliko bolj očitna pri uporabi teksturnega pomnilnika.

Diplomsko delo bi bilo mogoče nadgraditi na različne načine. V sklopu implementacije na GPE bi lahko še bolje izkoristili pomnilniško hierarhijo, saj v trenutni implementaciji ne uporabljamo skupnega pomnilnika. Tako bi dosegli še hitrejše izvajalne čase. Pohitritev bi lahko prinesla tudi nadgradnja obstoječega algoritma za iskanje sosedov, saj ta še vedno zajema večinski delež izvajalnega časa. Zanimivo bi bilo tudi nadomestiti algoritem za deljenje prostora s kakšnim bolj naprednim algoritmom iskanja sosedov (npr. [10]).

V sklopu mehke logike bi lahko razširili naš sistem tako, da uvedemo tudi drugi tip entitet (npr. plenilce). Trenutno interakcijo z enim samim sosedom

bi lahko nadgradili na interakcijo z več sosedi.

Dobro bi bilo tudi omogočiti branje baze znanja, uporabljene v mehki logiki, iz datoteke ter razširiti program v simulacijo.

Literatura

[1] A. Amritkar, S. Deb, and D. Tafti. Efficient parallel CFD-DEM simulations using OpenMP. *Journal of Computational Physics*, 256:501–519, 2014.

[2] Y. Bai, H. Zhuang, and D. Wang. *Advanced Fuzzy Logic Technologies in Industrial Applications*. Advances in Industrial Control. Springer London, 2007.

[3] I. Lebar Bajec and F. H. Heppner. Organized flight in birds. *Animal Behaviour*, 78(4):777–789, 2009.

[4] I. Lebar Bajec, N. Zimic, and M. Mraz. Simulating flocks on the wing: the fuzzy approach. *Journal of Theoretical Biology*, 233(2):199–220, 2005.

[5] P. Cingolani and J. Alcala-Fdez. jFuzzyLogic: a robust and flexible fuzzy-logic inference system language implementation. In *Fuzzy Systems (FUZZ-IEEE), 2012 IEEE International Conference on*, pages 1–8. IEEE, 2012.

[6] P. Cingolani and J. Alcalá-Fdez. jFuzzyLogic: a Java library to design fuzzy logic controllers according to the standard for fuzzy control programming. In *International Journal of Computational Intelligence Systems*, pages 61–75. 2013.

[7] J. Demšar and I. Lebar Bajec. Simulated predator attacks on flocks: a comparison of tactics. *Artificial life*, 20(3):343–359, 2014.

[8] S. Green. Particle simulation using cuda. *NVIDIA whitepaper*, 2010.

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

[10] J. Moon Lee and H. K. Cho. A simple heuristic to find efficiently k-nearest neighbors in flocking behaviors. In *Proceedings of the 6th WSEAS international conference on Computer Engineering and Applications, and Proceedings of the 2012 American conference on Applied Mathematics, AMERICAN-MATH*, volume 12, pages 60–64, 2012.

[11] J.M. Mendel. *Uncertain Rule-based Fuzzy Logic Systems: Introduction and New Directions*. Prentice Hall PTR, 2001.

[12] T.J. Ross. *Fuzzy Logic with Engineering Applications*. Wiley, 2009.

[13] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-purpose GPU Programming*. Addison-Wesley, 2011.

[14] T. Vicsek and A. Zafeiris. Collective motion. *Physics Reports*, 517(3–4):71–140, 2012.

[15] J. Virant. *Čas v mehkih sistemih*. Didakta, 1998.