УНИВЕРЗИТЕТ У БЕОГРАДУ
ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ

ЕВАЛУАЦИЈА И ВИЗУЕЛИЗАЦИЈА ПРОБЛЕМА ИНТЕРАКЦИЈЕ ЧВРСТИХ ТЕЛА КОРИШЋЕЊЕМ *BARNES-HUT* АЛГОРИТМА НА ГРАФИЧКОМ ПРОЦЕСОРУ

Дипломски рад

Ментор
доц. др Марко Мишић

Студент
Вељко Рвовић, 2018/0132

Београд, август 2022.

Садржај

Увод

Проблем интеракције чврстих тела(енг. *N-Body problem*) представља проблем предвиђања кретања тела у систему у којем тела међусобно интерагују. Те интеракције се најчешће посматрају као гравитационе силе и сви примери у овом раду ће се бавити искључиво таквим интеракцијама. Симулације оваквих система се широко користе у астрофизици ради анализе различитих система, од мањих као што су Земља-Месец-Сунце до великих као што су еволуција и формација читавих галаксија.

Фокус овог рада је део проблема који се тиче рачунских перформанси оваквих симулација. С обзиром да желимо да симулирамо изузетно велике системе, односно, системе са великим бројем тела, желимо оптимизовано решење. Један начин решавања овога јесте коришћење *Barnes-Hut* [1] алгоритма. Овај алгоритам је апроксимациони алгоритам који смањује сложеност са $O(n^2)$ на $O(n \log n)$. Неки од најзахтевнијих *high-performance computing* пројеката, као што је *DEGIMA(Nagasaki University)* раде компутациону астрофизику користећи *Barnes-Hut* алгоритам[2]. Секвенцијална имплементација овог алгоритма нам доноси значајно убрзање, али је и даље ограничена рачунским капацитетима централног процесора и с тога се доводи у питање могућност паралелизације.

Циљ је да искористимо капацитет графичког процесора како би добили још већа убрзања и тиме омогућили извршавања већих симулација. Пошто је овај алгоритам базиран на структури стабла, што је ирегуларна структура, његова примена на графичком процесору захтева другачији приступ од стандардних алгоритама који користе регуларне структуре података. С тога ћемо обратити посебну пажњу на делове имплементације који служе да омогуће ефикасно извршавање над ирегуларном структуром података као што је стабло. Многи од тих делова може бити корисно и за имплементацију других алгоритама који се базирају на структури стабла. Тиме ћемо показати да чак и за алгоритме базиране на стаблу можемо добити значајна убрзања уколико пажљиво имплементирамо алгоритам на графичком процесору.

Додатно је имплементирана визуелизација како би се валидирала имплементација алгоритма и боље демонстрирало његово функционисање. Визуелизација сама по себи не представља прави научни пример, већ је више естетичког карактера и служи да илуструје резултат.

У поглављу 2. ће бити наведена историја и опис проблема уз теоријске основе. Уз

то ће бити дато наивно решење проблема, као и мотивација за оптимизацију и паралелизацију. У поглављу 3. биће дат преглед коришћених технологија и објашњења потребна за њихово разумевање. То укључује OpenGL и његове библиотеке коришћене за визуелизацију, као и објашњења појмова у CUDA програмирању. У поглављу 4. биће објашњен сам алгоритам и његова секвенцијална имплементација. У поглављу 5. ће бити представљена визуелизација алгоритма и начин на који је имплементирана. У поглављу 6. биће објашњена паралелна имплементација алгоритма по његовима деловима са свим оптимизацијама које су примењене. У поглављу 7. ће бити приказани резултати мерења све три верзије алгоритма и дискусија. У последњем поглављу ће бити наведен закључак рада са идејом за даља истраживања у области имплементације паралелних алгоритама на графичком процесору.

О проблему интеракције чврстих тела

У овом поглављу ће прво бити описана историја проблема, као и сам *N-Body problem* кроз класичан пример који ће се користити касније у имплементацији. Затим ће се навести ограничења при његовој наивној имплементацији и мотивација за оптимизацију, односно паралелизацију.

2.1 Историја проблема

Познавајући три орбиталне позиције неке планете, Њутн је произвео једначину користећи директну аналитичку геометрију како би предвидео кретање планете, односно како би израчунао њене орбиталне елементе. Учинивши то, он и други су убрзо открили током наредних година да те једначине нису добро предвиделе неке орбите. Њутн је схватио да је то зато што су гравитационе интерактивне силе међу свим планетама утицале на све њихове орбите.

Ово откриће улази у срж онога што *N-body* проблем представља физички. Као што је Њутн схватио, морају бити познате и гравитационе интерактивне силе, како би се одредила орбита планете. Тако је дошло до свести о *N-body* проблему почетком 17. века. Ове гравитационе силе јесу у складу са Њутновим законима, али многе вишеструке интеракције су чиниле свако егзактно решење немогућим [3]. Данас постоје решења *two-body* проблема као и *three-body* проблема.

У општем случају, *n-body* проблеми се решавају или симулирају користећи нумеричке методе. За случај великог броја тела што ћемо гледати у овом раду, користе се апроксимационе методе како би се смањила временска сложеност.

2.2 Формулација проблема

N-Body problem у астрофизици представља проблем одређивања кретања космичких тела која интерагују међусобно гравитационим силама. Посматраћемо следећи пример: у неком простору налази се N тела, свако тело делује на свако друго тело гравитационом силом. Нека та гравитациона сила буде описана Њутновим законом гравитације[4],

а то је сила којом тело j делује на тело i:

$$\vec{F_{ji}} = \frac{Gm_i m_j}{|\vec{r_i} - \vec{r_j}|^3}(\vec{r_i} - \vec{r_j}), \tag{2.1}$$

Како би се одредило кретање неког тела, треба да сумирамо све силе које делују на њега[5]

$$F_i = G \sum_{j=1;j\neq i}^{N} m_i m_j \frac{\vec{r_i} - \vec{r_j}}{|\vec{r_i} - \vec{r_j}|^3} \tag{2.2}$$

где је

m_i, m_j — масе тела i и j

$\vec{r_i}, \vec{r_j}$ — вектори помераја тела

G — гравитациона константа

Након што добијемо укупну силу која делује на неко тело, можемо да израчунамо његово убрзање.

$$m_i a_i = F_i, \quad a_i = \frac{F_i}{m_i} \tag{2.3}$$

Сада можемо ажурирати брзину тела тако што јој додамо тренутну вредност убрзања, а онда исто тако ажурирати тренутну позицију тела тако што јој додамо тренутну вредност брзине.

2.3 Наивна имплементација

Наивна имплементација би подразумевала да само саберемо све силе које делују на тело и на основу тога срачунамо његово кретање. Пошто по трећем Њутновом закону знамо да интензитети сила F_{ij} И F_{ji} су једнаки, довољно је да се изврши $\frac{N(N-1)}{2}$ израчунавања. То значи да ће сложеност овакве имплементације бити $O(N^2)$. Оваква сложеност је много ограничавајућа у погледу перформанси и скалабилности, јер време извршавања квадратно расте са бројем тела у симулацији. Тај број у реалним симулацијама може бити значајно велики. Број звезда у глобуларним звезданим јатима може бити и неколико стотина хиљада(нпр *Messier 80*[6]).

2.4 Мотивација за оптимизацију

За симулације са хиљадама, десетинама хиљада, стотинама хиљада тела наивна имплементација није рачунски прихватљива. С тога морамо некако да смањимо број ових рачунања како би добили бржи алгоритам. За ово се користе апроксимационе методе. Ми ћемо користити *Barnes-Hut* алгоритам, који примењује хијерархијски приступ

користећи стабло(*quad-tree*). Овај алгоритам и слични принципи хијерархијске поделе простора се користе у разним областима као што су обрада слике, генерисање *mesh*-ева, просторно индексирање у базама података. У наредном поглављу је описан овај алгоритам и његова секвенцијална имплементација.

2.5 Мотивација за паралелизацију

Типичне примене програмирања на графичким процесорима укључују паралелизацију алгоритама који користе регуларне структуре података. Обично се избегавају алгоритми који се базирају на рекурзији и ирегуларним структурама. Међутим, са напредовањем графичких процесора и њиховим рачунским способностима, све више оваквих алгоритама постаје погодно за имплементацију на графичком процесору, доводећи до значајних убрзања. У овом раду ћемо показати један такав пример где се паралелизација може показати погодна и за ирегуларне структуре података. Имплементацијом на графичком процесору омогућићемо извршавање симулација које су за ред величине веће у односу на симулације на централном процесору. Како би омогућили ово морамо водити рачуна о проблемима који се јављају при имплементацији оваквих алгоритама на графичком процесору.

Преглед паралелних платформи и коришћених технологија

Имплементација је писана у програмском окружењу Microsoft Visual Studio 2019 на програмском језику C++. За визуелизацију је коришћен OpenGL графички интерфејс са библиотекама GLEW и GLFW. За имплементацију паралелног алгоритма је коришћена CUDA платформа верзије 11.7, *compute capability:* 6.1.

У наставку је наведено шта је то OpenGL и које су библиотеке коришћене за визуелизацију и изглед једног програма. Након тога су објашњени најбитнији концепти потребни за програмирање на графичком процесору. Ти концепти ће нам бити потребни и за боље схватање имплементације алгоритма на графичком процесору.

3.1 OpenGL и библиотеке

OpenGL(енгл *Open Graphics Library*) је стандардна спецификација која описује вишеплатформски програмски интерфејс за писање програма који раде са дводимензионалном и тродимензионалном рачунарском графиком. Иако у имену садржи реч библиотека, OpenGL представља само спецификацију. То значи да нуди низ операција кроз свој API, али не и имплементацију ових операција. Произвођачи хардвера, они који производе графичке процесоре су ти који су дужни да имплементирају ове операције. Те имплементације се налазе у драјверима и преводе OpenGL API команде у одговарајуће команде драјвера. OpenGL API се тиче само рачунарске графике, а не и анимација, GUI и слично. Због тога се уз OpenGL често користе разне библиотеке, као што је случај и у овом пројекту. Даље наводимо коришћене библиотеке.

GLEW(OpenGL Extension Wrangler Library) је вишеплатформска C/C++ библиотека за учитавање екстензија. Омогућава ефикасан механизам за одређивање које су све OpenGL екстензије подржане за време извршавања.

GLFW је вишеплатформска C/C++ библиотека за развој апликација на рачунару. Пружа једноставан API за креирање прозора, контекста, површи, као и за примање улаза са тастатуре или миша, и обраду догађаја.

Навешћемо пример програма који креира прозор, иницијализује библиотеке, проверава да ли је све успешно иницијализовано и ако јесте прелази на петљу која врши рендеровање графике. Овакав програм је имплементиран у делу за визуелизацију. Овде су приказани само делови кода који се тичу OpenGL и библиотека.

```
int main(int argc, char* argv[])
{
    GLFWwindow* window = 0;
    /* Initialize the library */
    if (!glfwInit())
        return -1;
    /* Create a windowed mode window and its OpenGL context */
    window = glfwCreateWindow(980, 980, "NBodySim", NULL, NULL);
    if (!window)
    {
        glfwTerminate();
        return -1;
    }
    /* Make the window's context current */
    glfwMakeContextCurrent(window);
    GLenum err = glewInit();
    if (GLEW_OK != err)
    {
        /* Problem: glewInit failed, something is seriously wrong. */
        fprintf(stderr, "Error: %s\n", glewGetErrorString(err));
    }

    /* Loop until the user closes the window or until all iterations are done */
    int iter = 0;
    while (!params.visualize || !glfwWindowShouldClose(window))
    {
        if (params.numIters > 0 && ++iter > params.numIters) break;
        // Clear the screen to black
        glClearColor(0.0f, 0.0f, 0.0f, 0.5f);
        glClear(GL_COLOR_BUFFER_BIT);

        // Rendering something . . .

        /* Swap front and back buffers */
        glfwSwapBuffers(window);
        /* Poll for and process events */
        glfwPollEvents();
    }
    glfwTerminate();

    return 0;
}
```

3.2 Појмови у CUDA програмирању

У овој секцији ће бити укратко објашњени релевантни појмови и концепти који се користе у CUDA програмирању.[7]

- Кернел — функција која се извршава на графичком процесору
- *Warp* — назив за нити унутар истог блока које се групишу заједно у групе фиксне величине за истовремено извршавање инструкција
- Дивергенција нити — уколико се деси промена контроле тока за нити из истог *warp-a*, долази до серијализације ових нити. То значи да се извршавају све могуће путање једна по једна. Ово доводи до деградације перформанси. Најчешћи пример за ово је условно гранање у коду.
- Хијерархија нити — CUDA нити формирају блокове који могу имати од 1 до 3 димензије и свака нит поседује индекс свог блока и свој индекс у блоку. Ово постиже две ствари. Прво, често је поседовање димензија корисно у обради елемената као што су низ или матрица јер се нити могу лако мапирати на елементе низа или матрице. Друго, блокови нити се извршавају међусобно незасивно, било којим редом, паралелно или секвенцијално. Та независност им омогућава скалабилност са бројем језгара.
- Меморијска хијерархија — Како би оптимално искористили ресурсе графичког процесора, потребно нам је добро знање његове меморијске хијерархије. CUDA нити могу приступати подацима у више различитих меморијских простора током свог извршавања. Од највећег значаја су нам локална, дељена и глобална меморија. Постоје још и два меморијска простора(константни и меморија за текстуре) али они нису релевантни за овај рад. Свака нит има своју приватну локалну меморију. Сваки блок нити поседује дељену меморију видљиву само нитима тог блока и која траје док и блок. Глобална меморија је видљива свим нитима и перзистира кроз више кернел позива од стране апликације. Глобална и локална меморија се налазе на уређају, док се дељена меморија налази на чипу, што је чини бржом јер поседује значајно већи пропусни опсег и мању латенцију од локалне и глобалне меморије. Због тога је најчешћи случај да кад год то можемо користимо дељену меморију уместо глобалне.
- *Memory coalescing* — техника којом се обезбеђује оптимално искоришћење пропусног опсега меморије. Дешава се када нити које истовремено извршавају исту инструкцију приступају узастопним меморијским локацијама. Ови захтеви за приступ меморији ће се онда груписати у један захтев и тиме смањити број приступа меморији.

Секвенцијална имплементација Barnes-Hut алгоритма

У овом поглављу ће бити објашњен алгоритам и приказана имплементација урађена као што је описана у [8]. Циљ алгоритма је да смањи број рачунања потребан за одређивање укупне силе која делује на неко тело. Одавде па надаље говоримо искључиво о верзији алгоритма у дводимензионом простору. Аналогно све важи за случај тродимензионог простора, осим што се користи $octree$ уместо $quadtree$.

4.1 Идеја алгоритма

Идеја алгоритма је да искористи чињеницу да уколико на неко тело i делује гравитационом силом група од x тела која су сличне удаљености у односу на тело i и тела унутар групе су међусобно близу, они се(гледано за тело i) могу апроксимирати у једно тело које делује гравитационом силом на тело i. Односно, желимо да групишемо тела која су међусобно близу у једно тело, уколико су довољно далеко од тела на које делују. Тела апроксимирамо користећи центар масе. Центар масе неке групе тела је средња позиција тела у тој групи са тежинским фактором масе тела. На пример, ако два тела имају позиције (x_1, y_1) и (x_2, y_2) и масе m_1 и m_2 респективно, њихово апроксимирано тело има следеће вредности:

$m = m_1 + m_2$

$x = (x_1 m_1 + x_2 m_2)\frac{1}{m}$

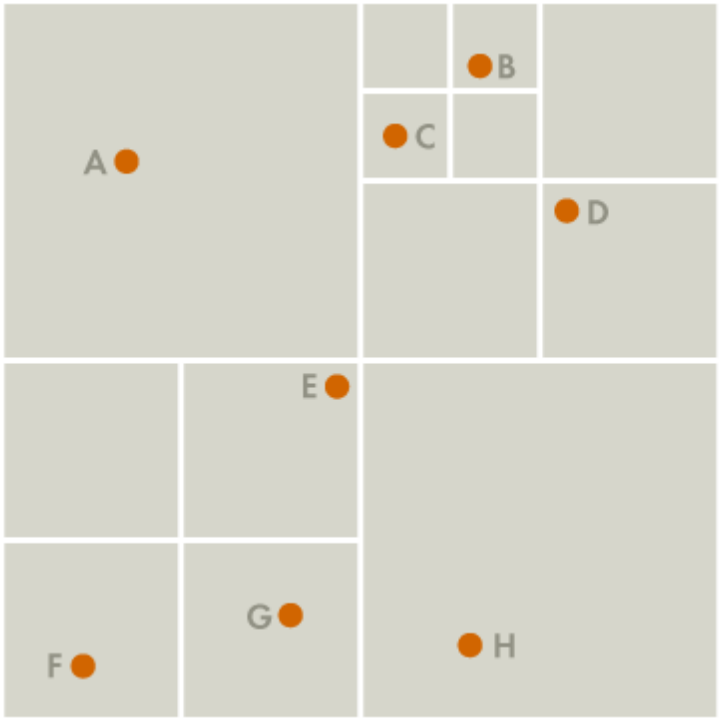
$y = (y_1 m_1 + y_2 m_2)\frac{1}{m}$

У наставку је објашњем сам алгоритам. Прво су објашњени основни појмови и структура података која се користи при његовој имплементацији. Након тога су наведени редом сви делови алгоритма и демонстрирани користећи пример.

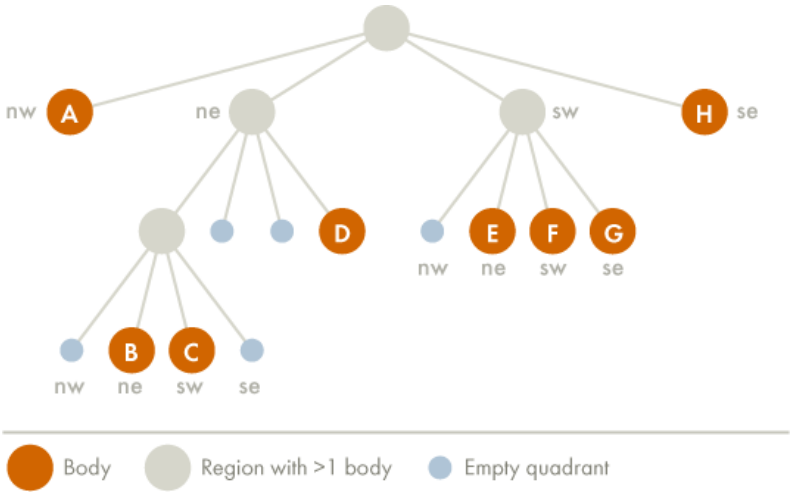
4.2 Алгоритам

4.2.1 Основни појмови и објашњења

Barnes-Hut алгоритам користи стабло(*quad-tree*) како би рекурзивно поделио простор. Ово стабло има чворове где сваки одговара одређеном делу простора и сваки чвор има тачно 4 потомка, како би поделио простор на 4 једнака дела. Чвор стабла одговара читавом простору, док сваки његов потомак одговара одређеном квадранту(четвртини простора). Простор се рекурзивно дели на квадранте док сваки квадрант не садржи 1 или 0 тела у себи(као на 4.1). Добија се резултујуће стабло 4.2(слике преузете са [8]).



Слика 4.1: Пример простора подељеног на квадранте



Слика 4.2: Резултујуће стабло

Сваки спољашњи чвор стабла представља тело, док сваки унутрашњи чвор представља групу тела која се налази у њему, чувајући центар масе и укупну масу групе.

За срачунавање укупне силе на неко тело i треба обавити обилазак стабла од корена. Уколико је центар масе унутрашњег чвора довољно удаљен од тела i, треба апроксимирати сва тела која се налазе у том делу стабла као једно тело чија је позиција једнака центру масе и чија је маса једнака укупној маси те групе. Уколико унутрашњи чвор ипак није довољно удаљен од тела i, онда треба обићи све његове потомке. За одређивање да ли је чвор довољно удаљен од неког тела, треба израчунати s/d, где је s димензија квадранта који је представљен унутрашњим чвором, а d удаљеност између тела i и центра масе унутрашњег чвора. Добијену вредност треба упоредити са неком граничном вредношћу θ. Уколико $s/d < \theta$, онда је унутрашњи чвор довољно далеко и може се вршити апроксимација. Вредност параметра θ одређује прецизност симулације(мања вредност - већа прецизност). За вредност 0 добили би идентичан резултат као наивна имплементација. Barnes и Hut користе $\theta = 1$, што ће бити коришћено и овде.

4.2.2 Креирање стабла

При креирању стабла прво се одређују ивице стабла, односно налазе се тела која су највише удаљена десно, лево, горе и доле респективно. Она одређују димензије простора које је репрезентовано стаблом. Након тога треба само додавати тела једно по једно користећи следећи рекурзивни алгоритам који ради додавање тела p у стабло са кореном у r:

1. Уколико r не садржи тело, додели му тело p
2. Уколико је чвор r унутрашњи чвор, ажурирај његов центар масе и укупну масу и настави рекурзивно додавање тела у одговарајући квадрант

3. Уколико је чвор r спољашњи чвор који садржи неко тело s, то значи да су сада оба ова тела у истом квадранту, па га треба разбити на још 4 квадранта, односно потомка у стаблу. Након тога треба додати тела p и s у одговарајуће квадратне. Могуће је да и након овог разбијања ова два тела буду у истим квадрантима, те треба поново рекурзивно разбијати простор. На крају треба ажурирати центар масе и укупну масу чвора r.

Формирање стабла је корак који се ради на почетку сваке итерације алгоритма и његова сложеност је $O(NlogN)$, где је *N* број чворова.

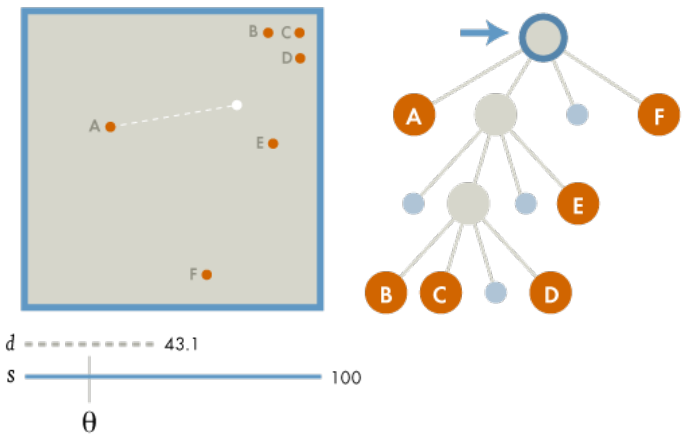
4.2.3 Рачунање укупне силе која делује на тело

За рачунање укупне силе која делује на неко тело i, користи се следећа рекурзивна функција која почиње од корена стабла

1. Уколико је тренутни чвор спољашњи чвор који представља неко тело j(различито од i), израчунати силу којом j делује на i и додати у суму
2. У супротном(тренутни чвор је унутрашњи) израчунати однос s/d. Уколико $s/d < \theta$, апроксимирај сва тела унутар овог чвора као тело представљено тим чвором, израчунај силу којом делује на i и додај је у суму.
3. У супротном, позови ову функцију за све потомке тренутног чвора

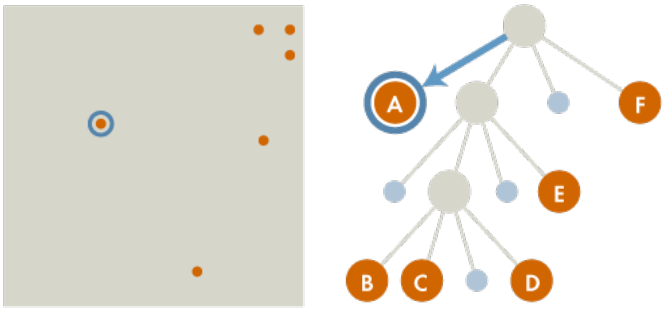
Демонстрираћемо на следећем примеру са телима *a, b, c, d, e, f* која имају масе 1, 2, 3, 4, 5, 6, респективно. Рачунамо укупну силу која делује на тело *a*. За параметар θ користићемо вредност 0.5.

Рачунање почиње од корена и гледа се удаљеност d између тела *a* и центра масе корена(бела тачка). Пошто је однос $\frac{s}{d} = 100/43.1 > \theta = 0.5$, обављамо процес рекурзивно за све потомке корена.



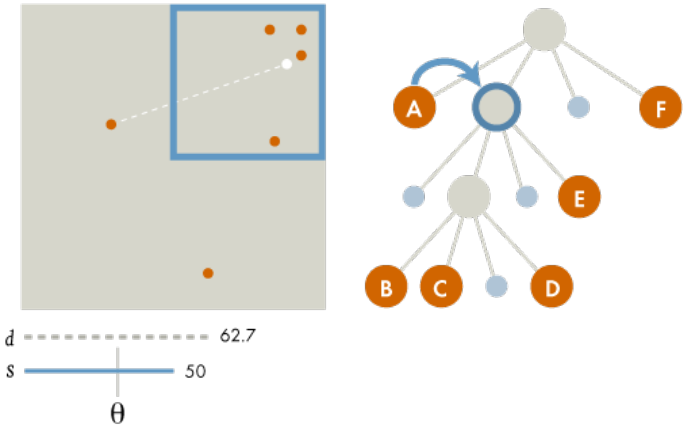
Слика 4.3: Пример рачунања силе

Први потомак је само тело *a*. Пошто тело не делује силом на самог себе, њега прескачемо.



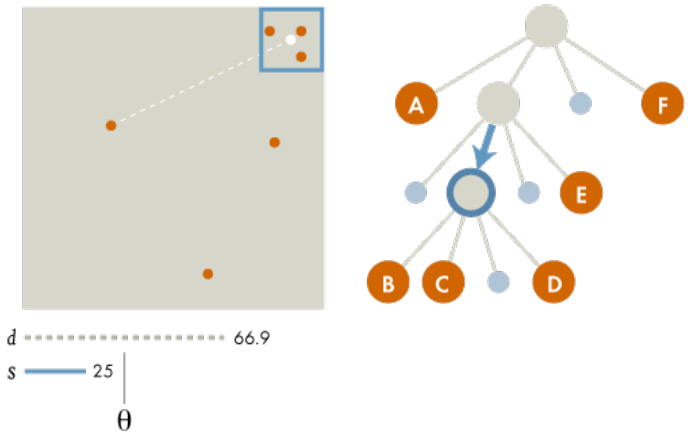
Слика 4.4: Пример рачунања силе

Следећи чвор представља североисточни квадрант и садржи центар масе тела *b, c, d, e*. Пошто је $\frac{s}{d} = 50/62.7 > \theta$, опет рекурзивно вршимо обраду за потомке овог чвора.



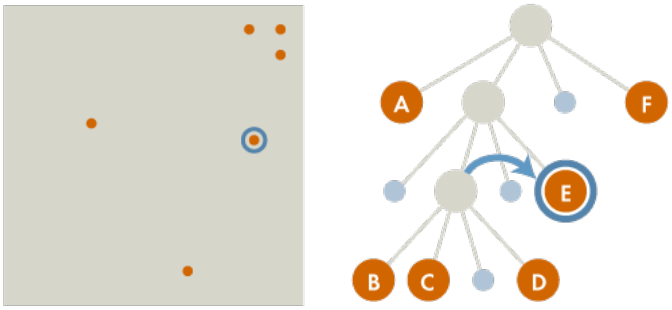
Слика 4.5: Пример рачунања силе

Следећи чвор је такође унутрашњи чвор и садржи тела *b, c, d*. Сада добијамо $\frac{s}{d} = 25/66.9 < \theta$. Пошто задовољава услов апроксимације, третирамо унутрашњи чвор као једно тело, рачунамо силу којом делује на тело *a* и додајемо га укупној сили која делује на *a*. Пошто смо извршили апроксимацију, не посећујемо потомке овог унутрашњег чвора, већ настављамо рачунање, посећујући чвор са телом *e*.



Слика 4.6: Пример рачунања силе

Следећи чвор је спољашњи чвор који садржи тело *e*. За њега рачунамо силу којом делује на тело *a* и додајемо у укупну силу која делује на *a*. Аналогно после тога исто радимо за чвор који садржи тело *f* и завршавамо рачунање.



Слика 4.7: Пример рачунања силе

Следећи фрагмент кода представља имплементацију рекурзивног рачунања укупне силе која делује на тело i.

```
void QuadTree::calcForceRecursive(Node* t, int i)
{
    if (t == 0) return;
    if (t->leaf)
    {
        if(t->id != i) calcForce(t, i);
    }
    else {
        float s = t->right - t->left;
        float d = dist(&sim->positions[i], t->pos);
        // if s/d < theta then treat this as a single body
        if (s / d < sim->theta)
        {
                calcForce(t, i);
```

```
        }
        else {
                calcForceRecursive(t->nw, i);
                calcForceRecursive(t->ne, i);
                calcForceRecursive(t->sw, i);
                calcForceRecursive(t->se, i);
        }
    }
}
```

4.3 Имплементација

Имплементирана је класа *NBodySeq* која чува све податке потребне за симулацију и имплементира функције за вршење симулације. За креирање инстанце ове класе прослеђује се број тела у симулацији након чега се генеришу насумично распоређена тела која представљају почетно стање симулације. Ова класа поседује две методе за извршавање једне итерације симулације: *runBruteForce()* за извршавање једне итерације директном(наивном) методом и *runBarnesHut()* за извршавање једне итерације *Barnes-Hut* методом.

Функција која извршава итерацију *Barnes-Hut* алгоритма налази се у наставку и састоји се од претходно описаног креирања стабла, након чега се за свако тело *i* позива метода која користећи стабло рачуна укупну силу која делује на *i* и ажурира његову позицију.

```
void NBodySeq::runBarnesHut()
{
        buildQuadTree();

        for (int i = 1; i < numParticles; ++i) {
                tree->forceCalculations(i);

                positions[i * 2] += velocities[i * 2];
                positions[i * 2 + 1] += velocities[i * 2 + 1];
        }
}
```

Класа *NBodySeq* користи класу *QuadTree* која енкапсулира стабло потребно за имплементацију *Barnes-Hut* методе. Класа *QuadTree* садржи показивач на корен стабла и имплементира све методе потребне за рад са стаблом, чији су описи дати у наставку:

```
class QuadTree{
public:
    ...

    // unos novog tela
```

```
    void insert(Node& n);

    // vizuelizacija stabla
    void displayLines();

    void clear();

    // racunanje ukupne sile koja deluje na telo i
    void forceCalculations(int i);

    int getNumNodes();

private:
    ...
}
```

4.4 Покретање програма

Програм се може покретати преко командне линије и поседује конфигурационе параметре.

Конфигурациони параметри се чувају у структури *Params*. Ти параметри јесу број тела у симулацији, број итерација симулације(вредност 0 представља неограничен број итерација), вредност параметра θ, опција за визуелизацију, опција за мерење времена извршавања, опција за *debug*.

```
struct Params {
        int num_particles = 10000;
        int numIters = 0;
        float theta = 0.5f;
        bool visualize = true;
        bool display_tree = false;
        bool display_times = false;
        bool display_debug = false;
};
```

Поједини параметри се могу проследити при покретању програма путем аргумената конзолне линије. За покретање са опцијом визуелизације додаје се аргуменат '-v'. За покретање са одређеним бројем тела додаје се аргуменат '-num' након којег се додаје жељени број тела.

Визуелизација алгоритма

Визуелизација је додатак овом пројекту како би се лепше приказала а и самим тим валидирала добијена симулација. Имплементирана је користећи високоперформантни OpenGL графички интерфејс уз библиотеке GLEW и GLFW. Програм се покреће са визуелизацијом додавајући '-v' као аргумент командне линије.

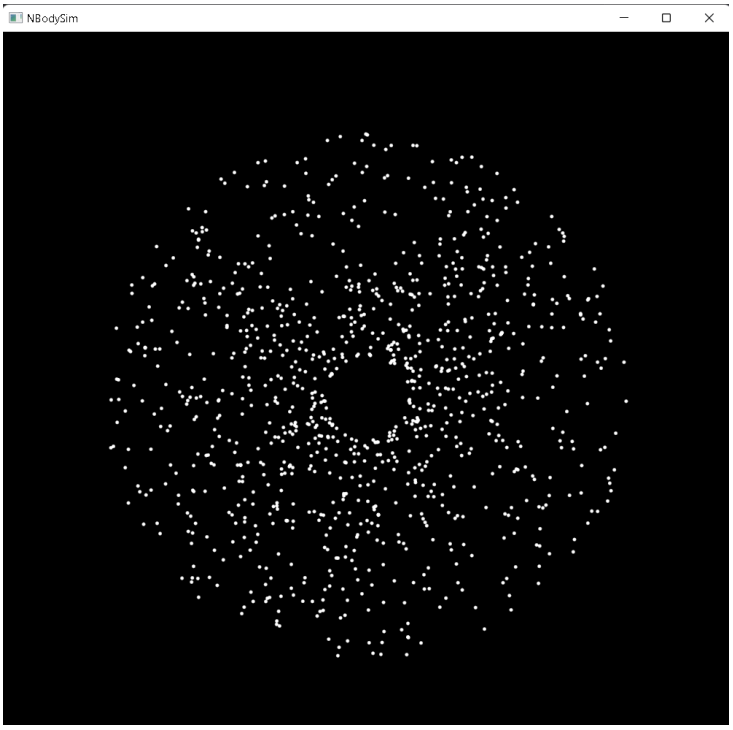
5.1 Визуелизација простора и тела

Главни део визуелизације се састоји од прозора димензија 980x980 и нацртаних тела са одређеним почетним распоредом у простору(видети слику 5.1). Ова група тела ће нам представљати звезде. Звезде формирају такозвана звездана јата. Постоје разни модели за опис глобуларних звезданих јата(нпр *Plummer model*[9]). Овде је узет прост модел где се униформно распоређују тела унутар кружног прстена око неког центра. Свако тело има свој почетни смер кретања и почетну брзину, која је таква да би тело наставило да се креће кружно око центра. У центру се налази тело које представља црну рупу и које се не креће. Црна рупа делује много јачом силом на остала тела како би задржала њихово кружно кретање. За резултат се добија систем у којем се тела углавном крећу у кругу око центра, са путањама које се мењају као резултат међусобне интеракције гравитационим силама.

За исцртавање било чега на OpenGL платформи потребан нам је *Shader* програм. Ови програми представљају корисничке програме написане за покретање на графичком процесору. Суштински, представљају код који се извршава у неком стадијуму графичког процесирања[10]. За цртање тела користећи OpenGL интерфејс написани су следећи *Shader* програми:

```
// Vertex shader
const GLchar* vertexSource =
    "#version 130\n"
    "in vec2 position;"
    "uniform mat4 model;"
    "uniform mat4 view;"
    "uniform mat4 projection;"
    "void main()"
```

```
    "{"
    "    gl_Position = projection * view * model *vec4(position, 0.0, 1.0);"
    "}";
// Fragment shader
const GLchar* fragmentSource =
"#version 130\n"
"void main()"
"{"
"    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);"
"}";
```



Слика 5.1: Визуелизација са 1000 тела

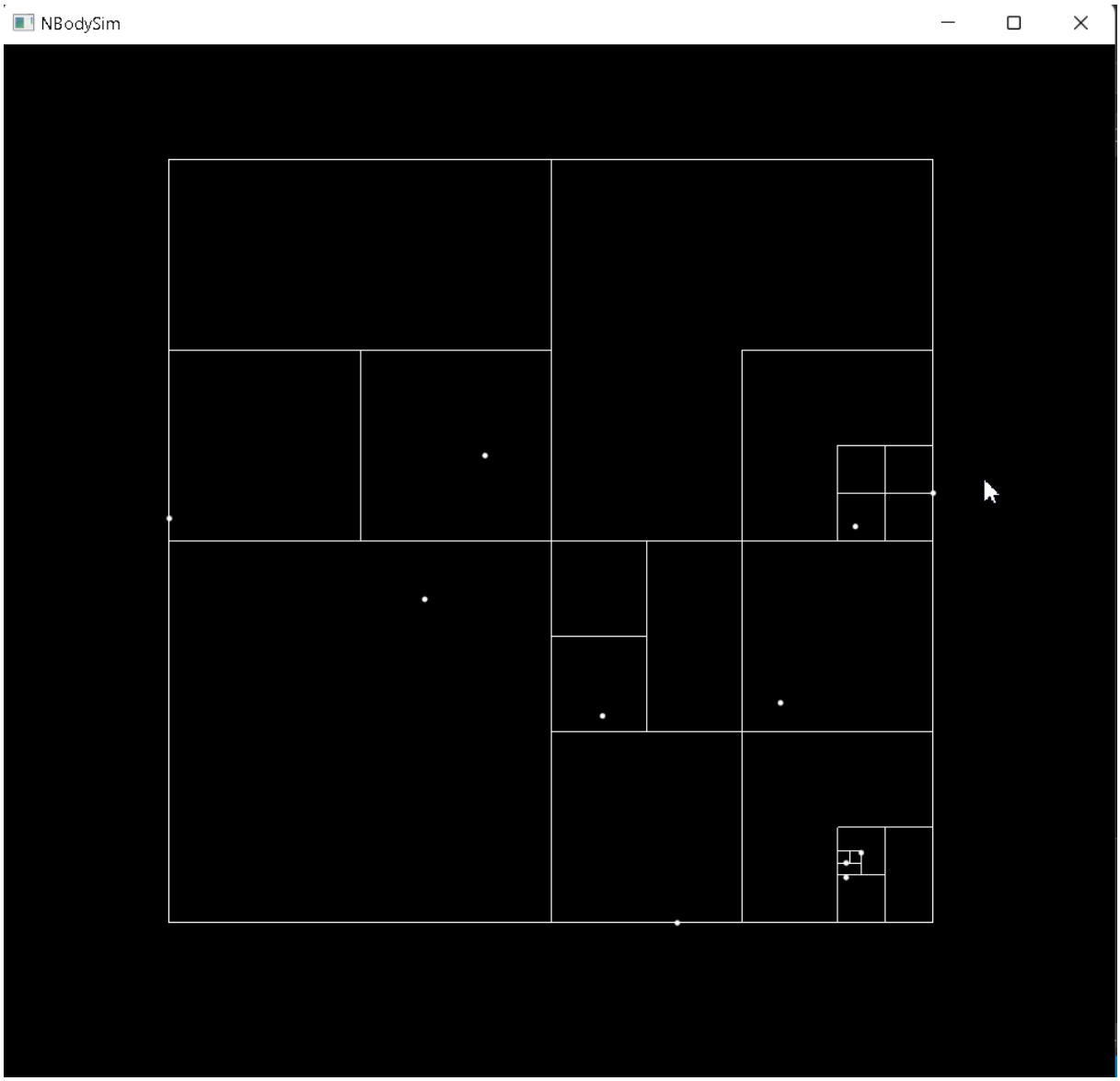
5.2 Приказ стабла

Визуелизација још додатно може да прикаже стабло(објашњено у 4.2.1). Ово је имплементирано рекурзивним обиласком стабла и исцртавањем ивица сваког квадранта који у себи садржи тело(слика 5.2). Имплементација овога се налази у методама *QuadTree.displayLines()* и *QuadTree.displayLinesRecursive()*:

```
void QuadTree::displayLines() {
    if (root != 0)
    {
        glColor3f(1.0, 0, 0);
        glBegin(GL_LINES);
```

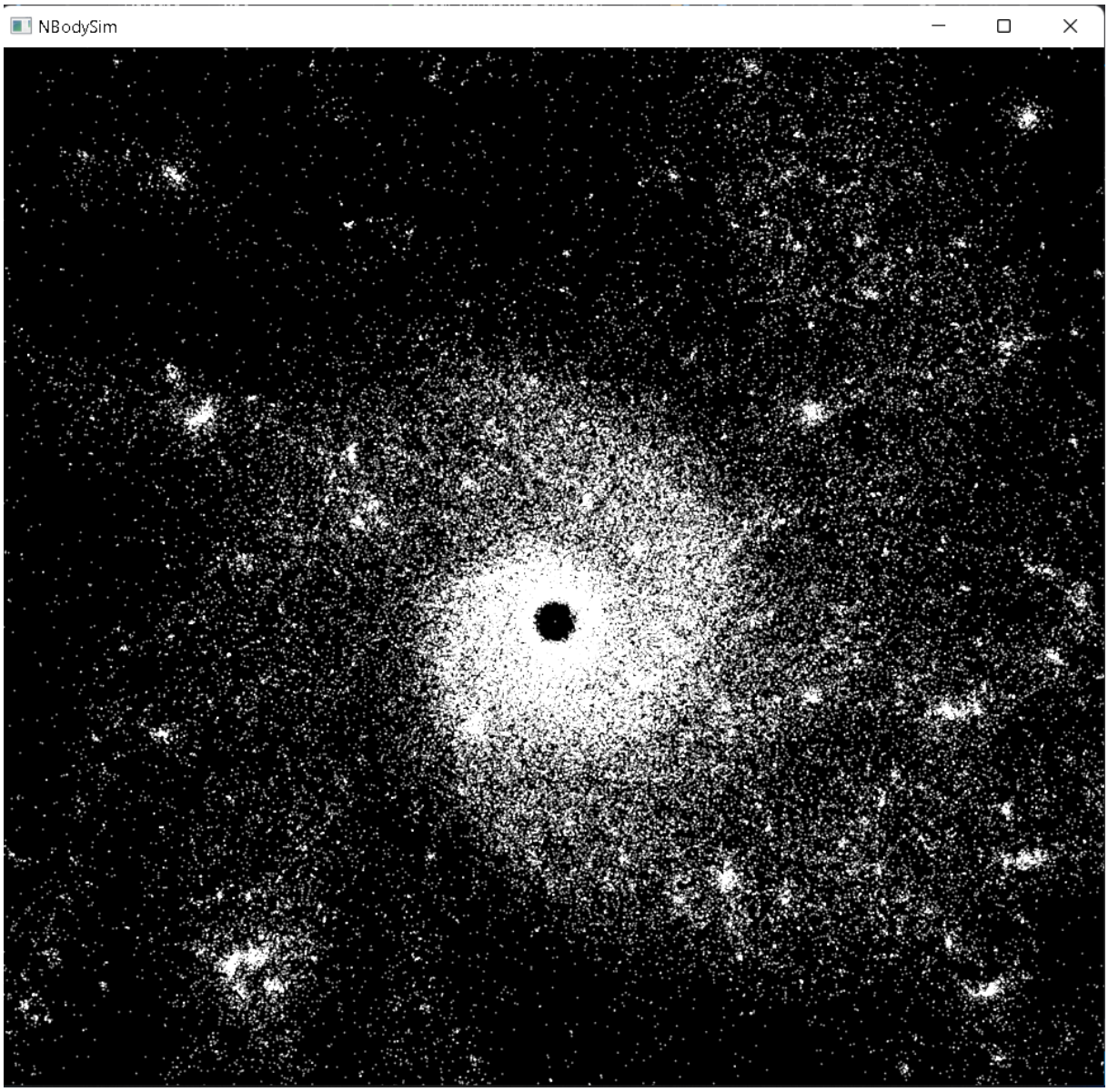
```
        displayLinesRecursive(root);

        glEnd();
    }
}

void QuadTree::displayLinesRecursive(Node* root)
{
    if (root)
    {
            glVertex2f(root->left, root->top);
            glVertex2f(root->right, root->top);

            glVertex2f(root->left, root->top);
            glVertex2f(root->left, root->bottom);

            glVertex2f(root->right, root->top);
            glVertex2f(root->right, root->bottom);

            glVertex2f(root->left, root->bottom);
            glVertex2f(root->right, root->bottom);

            displayLinesRecursive(root->nw);
            displayLinesRecursive(root->ne);
            displayLinesRecursive(root->sw);
            displayLinesRecursive(root->se);
    }
}
```

Приказ стабла у визуелизацији се ради тако што се додаје '-tree' као аргумент командне линије при покретању програма.



Слика 5.2: Визуелизација стабла

Након имплементације алгоритма на графичком процесору у стању смо да покренемо веће симулације и добијемо визуелизацију са 100.000 тела као на слици 5.3.



Слика 5.3: Визуелизација са 100000 тела после 20 секунди симулације

Имплементација на графичком процесору

У овом поглављу ћемо објаснити и дискутовати имплементацију *Barnes-Hut* алгоритма на графичком процесору урађену по [11] на CUDA платформи[12]. Паралелна имплементација се садржи из следећих CUDA кернела:

1. Рачунање димензије стабла
2. Формирање стабла
3. Ажурирање информација за сваки унутрашњи чвор
4. Сортирање тела по удаљености
5. Рачунање силе за свако тело
6. Ажурирање позиције и брзине сваког тела

Слика 6.1: Кернели који чине алгоритам

У даљим секцијама ћемо објаснити оптимизације које су уведене како би се овај алгоритам ефикасно извршавао на графичком процесору. Прво ћемо навести глобалне оптимизације које важе за све кернеле, а онда ћемо дискутовати посебне оптимизације унутар самих кернела. Притом ћемо само обраћати пажњу на оне оптимизације које се истичу од других паралелних алгоритама.

6.1 Глобалне оптимизације

Стабла се обично имплементирају користећи показиваче који се алоцирају динамичко на *heap*-у. То значи да је приступ овако алоцираних података много спорији јер се они не алоцирају на узастопним локацијама, те не долази до *memory coalescing*, који је објашњен у 3.2. Због тога ћемо користити структуру низа да репрезентујемо стабло. Пошто приступамо чворовима стабла који у себи садрже више поља, како би приступ неком пољу *X* био ефикаснији, треба обезбедити да се поља *X* налазе на узастопним локацијама. То ћемо обезбедити тако што ћемо уместо *низа структура*, користити *структуру низова* да би представили наша тела. Сваки низ ове структуре ће садржати вредности неког поља за све чворове стабла. Вредност на индексу *i* низа *X* би представљала вредност поља *X* за елемент *i*. Због униформности, користићемо исту структуру да представимо унутрашњи чвор стабла и тело и смештаћемо их у исти низ

Тела ћемо алоцирати с почетка низа, док ћемо унутрашње чворове алоцирати с краја низа. Индекс са вредношћу -1 ћемо користити као *null* показивач. Оваква организација нам је згодна јер тело/чвор сада представљамо индексом и можемо лако проверити да ли се ради о телу или о чвору простим упоређивањем индекса са бројем тела.

6.2 Рачунање димензије стабла

Први кернел врши редукцију над низом који садржи координате тела, налазећи минималне и максималне вредности за обе димензије. Прво се копирају подаци из главне меморије у дељену меморију и над њом се врши редукција на начин који обезбеђује највећи ниво истовременог учитавања(*coalesced*).

6.3 Формирање стабла

Други кернел имплементира итеративни алгоритам за креирање стабла користећи браве. Тела су додељена блоковима и нитима унутар блока. Свака нит додаје своја тела тако што обиђе стабло од корена до жељеног чвора и онда покуша да закључа одговарајућег потомка атомским уписом недефинисане вредности. Уколико упис ове вредности(закључавање браве) успе, нит додаје ново тело и откључава браву. Уколико тело већ постоји на тој локацији, нит прво креира нови чвор тако што атомски тражи следећи слободан индекс у низу, дода већ постојано и ново тело, уради операцију *__threadfence* која обезбеђује то да све остале нити виде ново подстабло, и ослобађа браву.

Нити покушавају да приступе брави све док не успеју. Овим се смањује контенција за главном меморијом и успорило напредовање успешних нити. Докле год једна нит унутар *warp*-а успе да приступи брави, остале нити ће се зауставити док успешна нит не заврши свој процес додавања тела и не откључа браву. Ова појава, специфична за

рекурзивне алгоритме, овде је корисна јер драстично смањује број приступа, од којих ће већина бити неуспешна јер ће наићи на закључану браву. Како би спречили и *warp*-ове у којима ниједна нит није успела да приступи својој брави да оптерећују меморију поновљеним приступима на крају сваке итерације додајемо баријеру(__*syncthreads*).

6.4 Ажурирање информација за сваки унутрашњи чвор

Овај кернел не поседује никакве посебне оптимизације. Једина оптимизација је та што су се у претходном кораку одмах рачунали тежински збирови позиција и масе. Сада се вредност позиције сваког чвора добијеног након формирања стабла дели са масом тог чвора.

6.5 Сортирање тела по удаљености

Овај кернел не утиче на коректност решења, али значајно убрзава извршавање главног кернела. То успева сортирањем које групише блиско удаљена тела близу једно другог у меморији, обезбеђујући ефикаснији приступ. Сортирање врши конкурентним уношењем тела у нови низ тако да се тела појављују истим редоследом као да је обављен *in-order* обилазак. Свакој нити се додељују чворови стабла. Нит обрађује све потомке тог чвора тако што за сваки проверава да ли је у питању спољашњи чвор и ако јесте додаје га у нови низ на позицији на којој би се налазио да је извршен *in-order* обилазак. Ова позиција може да се срачуна јер се при креирању стабла за свако подстабло чувала информација о броју спољашњих чворова које садржи.

6.6 Рачунање силе за свако тело

Ово је главни кернел јер захтева највише времена за извршавање, самим тим је и најбитнији за оптимизацију. Додељује редом сва тела нитима. За свако њој додељено тело, нит треба да обради одређени префикс стабла како би израчунала силу која делује на то тело. Ови префикси су слични за тела која су физички близу једна другоме, а различита за удаљена тела. Приликом учитавања ових префикса, уколико се они доста разлику за нити унутар истог *warp*-а, десиће се то да ће се велики број нити деактивирирати и остати неискоришћено као резултат дивергенције нити. С обзиром на дивергенцију нити унутар *warp*-а, од изузетне је важности да се ови префикси унутар *warp*-а разликују што је мање могуће. То омогућавамо сортирањем у претходном кернелу.

Можемо у потпуности елиминисати дивергенцију ако кажемо да све нити унутар *warp*-а имају исти префикс стабла, који је једнак унији свих њихових префикса.

Додатне оптимизације које су уведене се тичу кеширања података како би се смањили приступи главној меморији, као и коришћење стека на нивоу *warp*-а.

6.7 Ажурирање позиције и брзине сваког тела

Последњи кернел само ради ажурирање позиција и брзина сваког тела на начин који ефикасно приступа меморији. То је омогућено коришћењем *структуре низова*. Позиције и брзине се налазе у својим одвојеним низовима и приступање њима се врши на потпуно *coalesced* начин јер се налазе на узастопним локацијама.

6.8 Имплементација

Имплементација на графичком процесору урађена као у [12] представљена је класом *BarnesHutCUDA*. Ова класа чува све податке потребне за симулацију и реализује обраду једне итерације у методи *update()*. Објекат класе се инстанцира са бројем тела као аргументом. Кратак приказ класе дат је у наставку.

```
class BarnesHutCUDA
{
public:
    ...

    BarnesHutCUDA(int n);
    ~BarnesHutCUDA();

    // obrada jedne iteracije simulacije
    void update();
    // inicijalizacija simulacije
    void init();
    // rezultujuce pozicije tela
    const float* getOutput() { return h_output; }
private:
    // model za pocetni raspored tela
    void diskModel();
}
```

Функција *update()* врши једну итерацију симулације тако што позива редом све потребне кернеле. Можемо приметити кернеле 1-6 описане у поглављу 6. Они одговарају следећим кернелима редом: ComputeBoundingBox(), BuildQuadTree(), ComputeCentreOfMass(), SortParticles(), CalculateForces(), IntegrateParticles(). Додатно још постоје два кернела, један за иницијализацију низова и други за копирање коначног резултата. Главни делови ове функције дати су у наставку:

```
void BarnesHutCUDA::update()
{
    . . .

    ResetArrays(d_mutex, d_x, d_y, d_mass, d_count, d_start, d_sorted, d_child,
```

```
    d_index, d_left, d_right, d_bottom, d_top, numParticles, numNodes);
    ComputeBoundingBox(d_mutex, d_x, d_y, d_left, d_right, d_bottom, d_top,
    numParticles);
    BuildQuadTree(d_x, d_y, d_mass, d_count, d_start, d_child, d_index, d_left,
    d_right, d_bottom, d_top, numParticles, numNodes);
    ComputeCentreOfMass(d_x, d_y, d_mass, d_index, numParticles);
    SortParticles(d_count, d_start, d_sorted, d_child, d_index, numParticles);
    CalculateForces(d_x, d_y, d_vx, d_vy, d_ax, d_ay, d_mass, d_sorted, d_child,
    d_left, d_right, numParticles, G);
    IntegrateParticles(d_x, d_y, d_vx, d_vy, d_ax, d_ay, numParticles, 2, G);
    FillOutputArray(d_x, d_y, d_output, numNodes);

    ...
    cudaMemcpy(h_output, d_output, 2 * numNodes * sizeof(float),
    cudaMemcpyDeviceToHost);

    step++;
}
```

6.9 Дискусија

Приликом ове имплементације треба највише да се обрати пажња на:

- Балансирање посла: битно је доделити што је могуће више уједначене количине посла свакој нити. Ово смо постигли користећи структуру низа и равномерно додељивање посла блоковима и нитима унутар блокова.
- Смањивање дивергенције: за случај рекурзивних алгоритама ово је тачка на којој се највише губе перформансе. Груписање сличног посла и података који су блиски по неком вектору је од великог значаја за ефикасан приступ меморији. Треба водити рачуна о контроли тока. Често се испостави да је неко решење боље иако нити раде више посла него што је потребно, како се не би блокирале. Ово смо постигли сортирањем тела по међусобној удаљености и груписањем њихових префикса обраде.
- Приступи главној меморији: генералан принцип у свим програмима на графичким процесорима јесте да се што више смањи број приступа главној меморији користећи различите технике кеширања или ефикасног приступа сукцесивној меморији на нивоу *warp*-а. Ово смо реализовали користећи дељену меморију и друге технике кеширања темо где је то могуће.

6.10 Недостаци решења

Ово решење поседује и неке недостатке које ћемо навести у овом поглављу. Због омогућавања визуелизације вршили смо копирање података назад на централни процесор на крају сваке итерације. Постоји могућност преноса података између CUDA и OpenGL која се врши директно на графичкој плочици и која би нам омогућила визуелизацију без споријег преноса података између графичког и централног процесора. Уколико се желе постићи максималне перформансе при покретању програма са визуелизацијом, згодно би било реализовати ову могућност.

Резултати и дискусија

У овом поглављу прво ће бити описан хардвер на коме су добијени резултати и како су мерени, затим ће бити приказани и упоређивани резултати наивне имплементације, секвенцијалне имплементације и паралелне имплементације алгоритма.

7.1 Тест платформа

Програми су били покретани на рачунару са централним процесором Intel Core i5-1035G1 @ 1.00GHz и графичким процесором NVIDIA Geforce MX330. Спецификације графичког процесора су дате у табели 7.1.

CUDA driver version	11.7
CUDA compute capability	6.1
Number of multiprocessors	3
GPU clock rate	1556 MHz
Memory clock rate	3421MHz
Device global memory	2047MB
Shared memory per block	48KB
Constant memory	64KB
Maximum number of threads per block	1024
Maximum thread dimension	[1024, 1024, 64]
Maximum grid size	[2147483647, 65535, 65535]

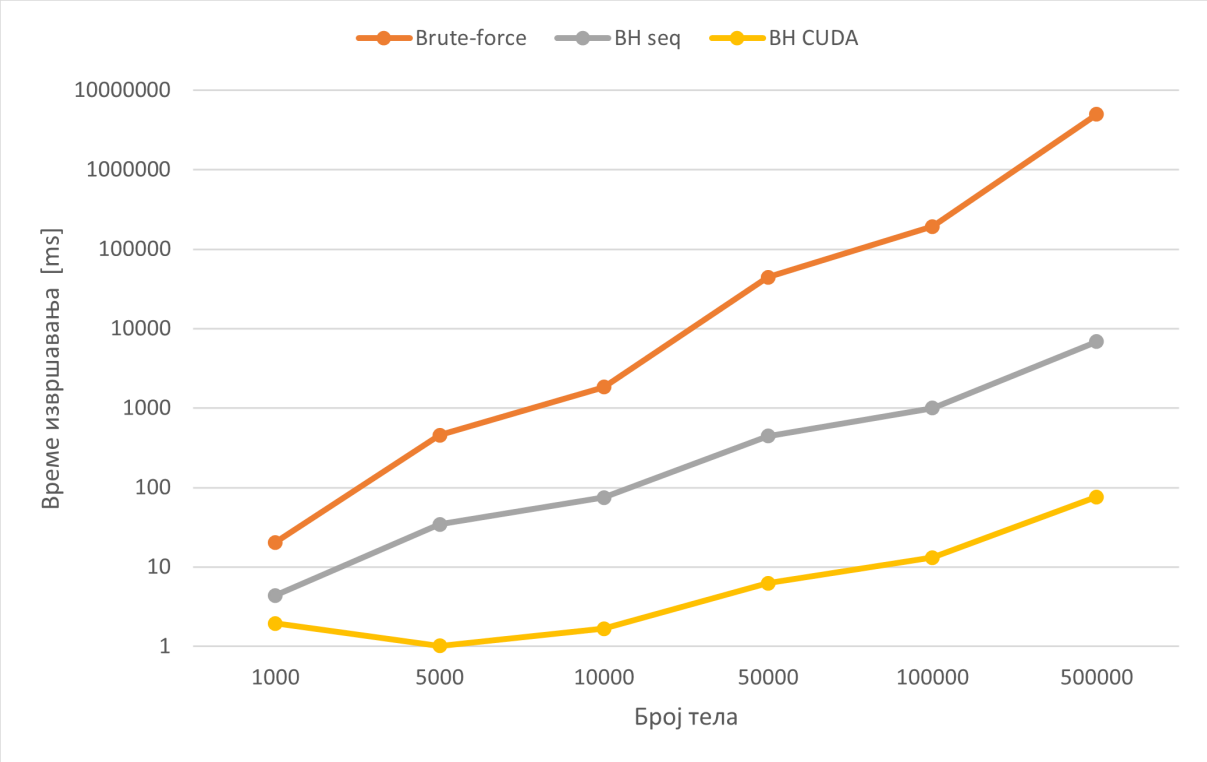
Табела 7.1: Спецификације графичког процесора

7.2 Методологија тестирања

Мерења су вршена покретањем програма без опције за визуелизацију и мерено је просечно време потребно за рачунање једног корака симулације. Ово је урађено за сваку од три имплементације(наивна, секвенцијална, паралелна) и за различите бројеве тела.

7.3 Резултати и дискусија

График 7.1 приказује однос времена извршавања корака симулације и броја тела симулације за сва три алгоритма у log-log скали. Мерене су симулације са 1000, 5000, 10000, 50000, 100000, и 500000 тела. У сваком од ових случајева се добија да је *Barnes-Hut* алгоритам значајно бољи од наивне имплементације. Такође се добија да је паралелна имплементација најбоља од све три. За мањи број тела, убрзање паралелне у односу на секвенцијалну имплементацију није толико велико(2.25X за 1000 тела), док за веће симулације су убрзања много већа, редом: 33x, 45x, 71x, 76x, и чак 90x за највећу симулацију од 500000 тела. Разлог томе је што за мање величине симулације велики део нити буде блокирано и тиме смањује искоришћење. Треба имати у виду да ови резултати нису добијени на најновијим графичким и централним процесорима, али показују итекако значајно убрзање.



Слика 7.1: Време извршавања корака симулације за различит број тела

Закључак

Овај рад је кроз различите имплементације истог алгоритма демонстрирао пар ствари. Прво смо анализирали *N-body* проблем, објаснили један могућ приступ његовом решавању: *Barnes-Hut* алгоритам. Реализовали смо класу која имплементира овај метод као и наивни метод за извршавање симулације. Овај алгоритам базиран на структури стабла нам смањује временску сложеност са $O(n^2)$ на $O(n \log n)$. Иако на први поглед не делује као погодан алгоритам за паралелизацију, показали смо да се и за алгоритме са ирегуларним структурама, као што је то овде био случај са стаблом, могу добити значајна убрзања уколико се добро имплементира паралелизација. Међутим, овакве имплементације су захтевне и потребно је доста труда и солидно знање програмирања на графичким процесорима. Због саме тежине и времена потребног за имплементацију, као и компликованог одржавања оваквих програма, доводи се у питање исплативост оваквог приступа уколико бисмо и многе друге алгоритме овим начином да убрзамо. Значајно би било креирати решења која се могу лако примењивати на више проблема.

Рад такође укључује и визуелизацију, како би приказао шта се добија симулацијом. Пример који је урађен је група тела насумично расподељена у кругу око центра који делује својом гравитационом силом на сва тела, одржавајући њихове кружне путање. На слици 5.3 можемо видети стање оваквог система након 20 секунди и интересантне формације тела које настају као резултат међусобног интераговања гравитационим силама. Међутим, симулације овог типа се појављују у разним областима, од биологије и хемије, па до разних физичких симулација из научних области или индустрије(нпр. физика у видео играма).

У овом раду бавили смо се класичним *Barnes-Hut* алгоритмом [1]. Може се напоменути да постоји и модификована варијанта алгоритма од стране *Barnes* [13]. Ова варијанта алгоритма не гледа да групише блиска тела довољно далеко удаљена од неког другог тела, већ да групише тела која се налазе међусобно близу и да за све њих искористи исту листу интераговања. То би резултирало значајно мањим стаблом, што би могло да значи да је овај алгоритам бољи кандидат за паралелну имплементацију. За овај рад је изабрана класична метода јер је познатија, а и јер је фокус више био ка томе да се демонстрира да је могуће убрзање паралелизацијом алгоритама са ирегуларним структурама података.

Литература

[1] J. Barnes and P. Hut, "A hierarchical o (n log n) force-calculation algorithm," *nature*, vol. 324, no. 6096, pp. 446–449, 1986.

[2] "Barnes–hut simulation." https://en.wikipedia.org/wiki/Barnes%E2%80%93Hut_simulation. последњи приступ: 17.07.2022.

[3] "N-body problem." https://en.wikipedia.org/wiki/N-body_problem. последњи приступ: 17.07.2022.

[4] "Newton's law of universal gravitation." https://en.wikipedia.org/wiki/Newton%27s_law_of_universal_gravitation. последњи приступ: 17.07.2022.

[5] "The barnes-hut galaxy simulator." https://beltoforion.de/en/barnes-hut-galaxy-simulator/. последњи приступ: 17.07.2022.

[6] "Messier 80." https://en.wikipedia.org/wiki/Messier_80. последњи приступ: 17.07.2022.

[7] "Cuda programming guide." https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#introduction. последњи приступ: 17.07.2022.

[8] K. W. Tom Ventimiglia, "The barnes-hut algorithm." http://arborjs.org/docs/barnes-hut. последњи приступ: 17.07.2022.

[9] "Plummer model." https://en.wikipedia.org/wiki/Plummer_model. последњи приступ: 17.07.2022.

[10] O. Wiki, "Shader — opengl wiki." https://www.khronos.org/opengl/wiki/Shader. последњи приступ 17.07.2022.

[11] M. Burtscher and K. Pingali, "An efficient cuda implementation of the tree-based barnes hut n-body algorithm," in *GPU computing Gems Emerald edition*, pp. 75–92, Elsevier, 2011.

[12] J. Sandham, "Nbodycuda." https://bitbucket.org/jsandham/nbodycuda/src/master/, 2016. последњи приступ 17.07.2022.

[13] J. E. Barnes, “A modified tree code: Don’t laugh; it runs,” *Journal of Computational Physics*, vol. 87, no. 1, pp. 161–170, 1990.