УНИВЕРЗИТЕТ У БЕОГРАДУ
ЕЛЕКТРОТЕХНИЧКИ ФАКУЛТЕТ

ОПТИМИЗАЦИЈА ЈАТОМ КОРИШЋЕЊЕМ ГРАФИЧКИХ КАРТИЦА

ДИПЛОМСКИ РАД

Ментор:

др Драган Олћан, в. проф.

Кандидат:
Мандић Филип
2015/0308

Београд, јул 2020.

Садржај

1 Увод

Алгоритам оптимизације јатом [1] један је од оптимизационих алгоритама примењивих на нелинеарне проблеме оптимизације (енглески: Nonlinear programming, скраћено NLP) [1]. Циљ овог рада је имплементација алгоритма оптимизације јатом на графичком процесору (енглески и скраћено GPU) и решавање два оптимизациона проблема из класе NLP, уз анализу перформанси и добијених резултата.

Уопштено, оптимизација представља тражење најбољег решења из неког скупа, при чему решење мора да задовољава одговарајуће критеријуме. Оптимизациона функција, односно функција грешке, је нумеричка мера разлике између жељеног и постигнутог решења [1]. На основу вредности оптимизационе функције можемо да упоредимо два могућа решења. Скуп могућих решења зваћемо још и оптимизациони простор. За NLP класу оптимизационих проблема, карактеристично је да оптимизациона функција има континуалне променљиве (на пример променљиве су из скупа реалних бројева), па је оптимизациони простор теоретски бесконачно велики. Самим тим, за систематско претраживање оваквог оптимизационог простора потребно је бесконачно много времена. Из тог разлога потребно је применити неки од оптимизационих алгоритама, а у овом раду биће коришћен алгоритам оптмизације јатом.

За алгоритам оптимизације јатом се показало да може да пронађе инжењерски прихватљиво решење у разумном временском оквиру. Овај алгоритам ради са скупом тачака у оптимизационом простору које посматра као једно јато. По одређеним правилима алгоритам ажурира позиције тачака и испитује вредност оптимизационе функције у њима. Тиме алгоритам претражује оптимизациони простор и конвергира ка бољем решењу. У овом раду оптимизација јатом ће бити коришћена за решавање два NLP проблема. Први је тражење минимума функције суме квадрата за задати број сабирака. Други, и значајно компликованији проблем, је проблем нумеричке интеграције.

Будући да алгоритам има особине које га чине погодним за паралелизацију, овај алгоритам може се ефикасно имплементирати помоћу популарних библиотека и платформи за развој паралелних програмских решења. Решење које је овде приказано, развијено је користећи CUDA (енглески Compute Unified Device Architecture) платформу [2]. Она пружа могућност да се NVIDIA графичке картице користе као додатни процесор у рачунару, односно пружа могућност програмеру да искористи предности графичке картице како би поправио перформансе свог програма.

Графичке картице имају битно другачију архитектуру од главне процесорске јединице (енглески и скраћено CPU). Оне располажу са великим бројем језгара и могу у паралели да изврше далеко више аритметичких операција. Овакве особине чине графичку картицу идеалним чипом за извршавање алгоритама који треба у паралели да изврше велики број математичких операција, попут алгоритма оптимизације јатом.

Са друге стране, употреба графичке картице намеће додатне одговорности програмеру приликом израде решења. Графичка картица има засебну процесорску јединицу и RAM меморију. Програмер мора да води рачуна о синхронизацији кода који се извршава на

графичкој картици и главном процесору, као и о управљању подацима који се налазе у две одвојене меморије – RAM меморији рачунара и RAM меморији графичке картице. Такође, на графичкој картици није могуће користити неке стандардне библиотеке и функције, попут стандардних метода за генерисање псеудослучајних бројева.

За потребе тестирања валидности и перформанси, развијено је и решење помоћу OpenMP API-ја (енглески Open Multi-Processing) [3]. Помоћу овог API-ја, могуће је паралелизовати обраду тако да се максимално искористе доступна језгра на главном процесору рачунара (CPU).

Решење је већински развијено помоћу програмског језика C [4], уз употребу неких библиотека из C++ стандардне библиотеке [5].

У овом раду биће приказан начин функционисања алгоритма оптимизације јатом, као и његове особине које га чине погодним за паралелизацију. Такође, приказано је како је извршена паралелизација и које су перформансе и резултати за дате оптимизационе функције постигнути. У опису проблема биће објашњени појмови који су коришћени у раду и изложен начин рада алгоритма оптимизације јатом. Потом, биће приказано како је извршена паралелизација алгоритма користећи CUDA платформу, односно OpenMP API. У поглављу резултати, приказане су оптимизационе функције и поређење добијених перформанси и резултата две имплементације. У закључку, дат је осврт на постигнуте резултате.

2 Опис проблема

У овом поглављу биће описан принцип функционисања алгоритма оптимизације јатом, као и технике које су примењене како би се успешно извршила паралелизација алгоритма помоћу CUDA платформе и OpenMP API-ја.

2.1 Терминологија

У наставку су наведени термини који ће бити коришћени у раду:

- **Димензија оптимизационог проблема** – број координата који има свака тачка у оптимизационом простору. Ову величину обележаваћемо са D.
- **Агент** – представља једну тачку у оптимизационом простору (једно могуће решење), односно један вектор координата решења (X_1, X_2...X_D). Овај вектор називаћемо још и **позицијом агента**.
- **Јато** – скуп агената помоћу којих се врши оптимизација.
- **Брзина агента** – представља D димензионални вектор (V_1, V_2...V_D) растојања за које је агент променио своју позицију у односу на претходно израчунавање оптимизационе функције, по свакој димензији.
- **Итерација** – једно израчунавање оптимизационе функције.
- **Ажурирање агената јата** – обухвата ажурирање података сваког агента јата, и рачунање оптимизационе функције за сваког агента у јату.
- **Најбољи агент** – агент који је у претходном ажурирању агената јата пронашао најбоље решење.
- **Ажурирање јата (Ажурирање целог јата)** – обухвата ажурирање агената јата, проналазак најбољег агента и његово евентуално памћење.
- **Локално најбоље пронађено решење** – најбоља позиција коју је агент пронашао у току оптимизационог поступка. Ова вредност ће бити обележена са p_{best}, и представља D димензионални вектор.
- **Глобално најбоље пронађено решење** – најбоља позиција коју је цело јато успело да пронађе у току оптимизације. Ова вредност ће бити обележена са g_{best}, и такође представља D димензиони вектор.

2.2 Приказ рада PSO алгоритма

Алгоритам оптимизације јатом инспирисан је понашањем животиња које живе у заједницама, па отуда и сам назив алгоритма. У домену оптимизација, овај алгоритам опонаша кретање животиња које живе у заједницама. Алгоритам ради са скупом агената (јединки) које су организоване у јато. Кретањем агената јата у оптимизационом простору функције, јато покушава да пронађе најповољнију могућу позицију, односно минимум оптимизационе функције.

Како би се постигао овакав ефекат, сваком агенту јата се ажурира брзина којом се креће, као и позиција на којој се налази. Након одређивања нове позиције, прелази се на израчунавање оптимизационе функције у тој позицији. Резултат овог израчунавања користи се за одржавање преосталих података које алгоритам користи. Овакав поступак ажурирања јата понавља се све док се не добије резултат којим је корисник задовољан, или се испуни критеријум који је корисник задао за крај оптимизације. У овом раду критеријум за завршетак оптимизације је број ажурирања јата који корисник задаје.

У наставку (листинг 1 и листинг 2) је приказан псеудокод који илуструје секвенцијалну имплементацију алгоритма оптимизације јатом. Функција update_agent се позива из главне функције (pso).

```
void update_agent(agents_velocity, agents_position, agents_lowest_error,
        agents_best_position, swarm_best_position, agent)
{
    //ažuriraj brzinu i poziciju agenta po svakoj dimenziji
    for (k = 0; k < dimension; k++)
    {
        update_velocity(agents_velocity, agent, k,
            agents_best_position, swarm_best_position)

        update_position(agents_position, agent, k)
    }

    error = calc_opt_function(agents_position[agent], opt_parameters)

    //sačuvaj novo najbolje rešenje agenta ukoliko je pronađeno
    if (error < agents_lowest_error[agent])
    {
        agents_lowest_error[agent] = error
        copy(agents_best_position[agent], agents_position[agent], dimension)
    }
}
```

Листинг 1 – Псеудокод секвенцијалне имплементације PSO алгоритма. Функција за ажурирање једног агента.

```
void pso(swarm_size, dimension, opt_parameters,
            number_of_turns, result_position, result_error)
{
    initialize(agents_velocity, agents_position, agents_lowest_error,
        agents_best_position, swarm_min_error, swarm_best_position)

    for (i = 0; i < number_of_turns; i++)
    {
        for (j = 0; j < swarm_size; j++)
        {
            update_agent(agents_velocity, agents_position, agents_lowest_error,
                            agents_best_position, swarm_best_position, j)
        }

        best_agent = find_best_agent(agents_best_position, swarm_size)

        //sačuvaj novo najbolje rešenje jata ukoliko je pronađeno
        if (agents_lowest_error[best_agent] < swarm_min_error)
        {
            swarm_min_error = agents_lowest_error[best_agent]
            copy(swarm_best_position, agents_best_position[best_agent],dimension)
        }
    }

    result_error = swarm_min_error //konačna greška
    copy(result_position, swarm_best_position, dimension) //najbolja pozicija

    free_memory(agents_velocity, agents_position, agents_lowest_error,
        agents_best_position, swarm_min_error, swarm_best_position)
}
```

Листинг 2 - Псеудокод секвенцијалне имплементације PSO алгоритма. Функција која заузима ресурсе и иницијализује податке, ажурира јато задати број пута и ослобађа заузете ресурсе.

Функција pso као улазне параметре прима величину јата (swarm_size), број димензија (dimension), параметре који се користе приликом рачунања оптимизационе функције (opt_parameters) и број ажурирања јата које функција треба да уради (number_of_turns).

На почетку функција иницијализује податке које користи. Потом следи ажурирање јата задати број пута. Овде се врши кретање агената јата у оптимизационом простору и самим тим проналазе све боља решења. Коначни резултати се чувају у два излазна параметра – најбоља пронађена позиција (result_position) и минимална пронађена грешка (result_error). На крају функција ослобађа сву меморију коју је заузела.

Агенти током кретања узимају у обзир и ажурирају два одвојена скупа података:

1. **Локални подаци** – ове податке сваки агент одржава за себе. У ову групу спадају информације о брзини којом се агент креће (V), позицији на којој се налази (X), локалном најбољем пронађеном решењу (p_{best}) и минималној постигнутој грешци (error). У псеудокоду променљиве agents_velocity, agents_position, agents_best_position, agents_lowest_error, представљају низове у којима се чувају ови подаци за сваког агента јата.
2. **Глобални подаци** – односно подаци који су заједнички за све агенте јата. Конкретно, алгоритам води рачуна о глобално најбољем пронађеном решењу и вредности оптимизационе функције у њему. У псеудокоду ове вредности се памте у подацима swarm_best_position, swarm_min_error. У наставку обележаваћемо их са g_{best} и min_error.

2.2.1 Ажурирање локалних података агената

На основу ова два скупа података, сваки агент ажурира своје локалне податке и рачуна оптимизациону функцију. Тачан редослед корака изгледа овако:

1. рачунање нове брзине агената,
2. рачунање нове позиције агената,
3. рачунање оптимизационе функције у новој позицији и
4. по потреби, ажурирање најбољег локалног пронађеног решења.

Брзине кретања једног агента у i-том ажурирању јата (v_i), рачуна се по следећој формули:

$$v_i = w * v_{(i-1)} + rand * c_1 * \left(p_{best} - x\right) + rand * c_2 * \left(g_{best} - x\right) \quad (1)$$

Ову формулу можемо да поделимо у три целине. Први сабирак ($w * v_{(i-1)}$) представља утицај инерције, односно тера агента да се креће на начин на који је то чинио до сада. Из тог разлога брзина којом се до тада агент кретао се множи са коефицијентом инерције w. Овом коефицијенту најчешће се додељује вредност 0,729 (ова вредност се генерално препоручује). Други и трећи сабирак представљају производ по три вредности, где је rand случајни број из опсега [0, 1], c_1 и c_2 су коефицијенти за чије се вредности најчешће узима 1,494 за оба коефицијента. Трећи чинилац у другом и трећем сабирку представља удаљеност агента од локално најбоље пронађеног решења и глобално најбоље пронађеног решења, респективно. Коефицијент c_1 се назива и когнитивни коефицијент, будући да одређује са коликом мером агент тежи да се врати у најбољу позицију коју је он до тада

пронашао. Са друге стране c_2 се назива социјални коефицијент, будући да од њега зависи "жеља" агента да се помери према решењу за које читаво јато сматра да је најбоље.

Након што се одреди брзина, прелази се на рачунање нове позиције агента. Позиција агента у i-том ажурирању јата (X_i), рачуна се по формули:

$$X_i = X_{(i-1)} + D_t * V \tag{2}$$

Из формуле се види да нова позиција зависи од претходне позиције агента и помераја, који се рачуна као брзина помножена са временским фактором Dt. Ради једноставности за временски фактор се најчешће узима вредност 1.

Приликом рачунања брзине уводи се ограничење да њен интензитет не би требало да прелази једну петину величине домена оптимизационе функције (односно оптимизационог простора). Овакво ограничење за последицу има да су агенти углавном груписани у једном делу домена, односно претражују део домена близак најбољем глобално пронађеном решењу. Уколико се деси да због брзине агента, нова позиција агента изађе из домена, алгоритам симулира еластични судар са ивицом домена. Другим речима, нова позиција агента биће у домену, онолико удаљена од ивице домена за колико је требала да буде ван.

Следећи корак јесте итерација, односно израчунавање оптимизационе функције у новој позицији агента. Овом приликом имплементиране су две различите оптимизационе функције – сума квадрата и нумеричка интеграција. Оне ће бити изложене у поглављу резултати, заједно са добијеним решењима и перформансама.

Након што агент заврши са израчунавањем оптимизационе функције, неопходно је да упореди нову вредност оптимизационе функције и вредност функције у најбољем локалном пронађеном решењу (error). Уколико је ново решење боље, потребно је сачувати тренутну позицију као локално најбоље пронађено решење (p_{best}), као и вредност коју је вратила оптимизациона функција у променљиву error.

Када се ова четири корака заврше, завршава се ажурирање једног агената. Ова четири корака се понављају за сваког агента у јату, и када се они заврше за сваког агента у јату, кажемо да је готово једно ажурирање свих агената јата.

2.2.2 Ажурирање заједничких података за цело јато

Када читаво јато заврши своје ажурирање, неопходно је одредити ново глобално најбоље пронађено решење (g_{best}). Потребно је пронаћи најбољег агента у јату и упоредити вредност оптимизационе функције у тој тачки са вредношћу актуелног глобално најбољег пронађеног решења. Уколико је вредност у најбољем агенту боља, потребно је запамтити га као ново глобално најбоље решење (g_{best}) и сачувати вредност оптимизационе функције у њему (min_error).

Битно је нагласити да овај корак не обавља сваки агент у јату, већ се он ради једном на нивоу целог јата независно од свих агената. Након тога, агенти јата су спремни да пређу на наредно ажурирање.

2.2.3 Иницијализација података

Почетне позиције агената одређују се помоћу генератора псеудослучајних бројева, у складу са доменом оптимизационе функције. Неопходно је иницијализовати и вредности за локално и глобално најбоље пронађено решење у складу са доменом, како би се приликом првог рачунања брзине агената добиле смислене вредности.

Приликом одређивања почетне брзине агената, вредности се такође одређују помоћу генератора псеудослучајних бројева. Неопходно је водити рачуна да интензитет не буде већи од једне петине оптимизационог простора.

Вредности error и min_error потребно је поставити на максималне могуће вредности, како би се њихово стање променило већ приликом првог ажурирања јата вредностима које је вратила оптимизациона функција.

2.2.4 Могућност паралелизације

Алгоритам има неколико особина који га чине погодним за паралелизацију. Прво, приликом ажурирања агената јата глобални подаци се само читају, због чега није неопходно уводити додатну политику приступа тим подацима. Такође, за ажурирање једног агента не користе се локални подаци других агената, па не постоји потреба ни за синхронизацијом на локалним подацима агената. Другим речима, ажурирање локалних података агената могуће је обављати у паралели, у потенцијално онолико нити колико јато има агената, без било какве потребе за синхронизацијом нити.

Како се очекује да израчунавање оптимизационе функције буде временски најзахтевнији корак у алгоритму, и како се оно ради по једном за сваког агента у једном ажурирању јата, очекује се да би оваква паралелизација у великој мери могла да убрза извршавање алгоритма оптимизације јатом.

Приликом ажурирања глобалних података, неопходно је пронаћи најбољег агента. Овај корак се своди на тражење минимума међу вредностима које је вратила оптимизациона функција. Тражење минимума у низу података се такође може паралелизовати, и штавише, у питању је једна варијанта проблема паралелне редукције, познатог проблема који се може ефикасно имплементирати помоћу OpenMP API-ја и CUDA платформе. Будући да за велика јата овај корак може постати временски захтеван, од интереса је да се изврши и његова паралелизација.

2.3 Имплементација помоћу CUDA платформе

У овом подпоглављу биће приказано како је извршена паралелизација претходно описаних корака. Имплементирано решење има структуру типичну за једну CUDA апликацију. Ток контроле програма креће од main() функције која се извршава на CPU. Ту програмер има прилику да заузме све потребне ресурсе и изврши све потребне иницијализације. Након тога следе вишеструки позиви функција које се извршавају на графичком процесору, задати број пута. Када се заврши са обрадом на графичком процесору, из кода који извршава CPU врши се ослобађање заузетих ресурса.

У наставку ће бити описана организација података у меморији, позиви функција који се извршавају на графичкој картици и детаљи имплементације тих функција који су последица коришћења CUDA платформе. На крају, биће дат осврт на ограничења програма који су последица коришћења одговарајућих концепта и библиотека.

2.3.1 Организација података у меморији

CUDA пружа више начина на који програмер може да користи расположиву меморију графичке картице. У решењу је коришћен концепт назван Unified memory [6]. Овај концепт омогућава да подаци буду видљиви у оба адресна простора, односно да постоји копија података и у оперативној меморији рачунара и у оперативној меморији графичке картице. Оперативни систем је задужен да ове податке одржава валидним кроз механизам page fault-a, па је приступ подацима униформан, без обзира са којег уређаја се приступа.

Након што се меморија алоцира потребно је попунити је иницијалним вредностима. Ови подаци биће уписани у део RAM меморије рачунара у којој је мапиран адресни простор графичке картице. Приликом првог приступа овој меморији од стране графичке картице, доћи ће до копирања података у њену оперативну меморију. То ће проузроковати да прво ажурирање јата траје дуже него што би то иначе било случај. Како се након првог ажурирања локалним подацима агената неће приступати од стране CPU-a, употреба оваквог типа меморије неће имати осетан утицај на перформансе решења.

За сваког агента памте се подаци о брзини, позицији и најбољој позицији коју је он до тада пронашао. Како су ове вредности све D димензиони вектори, подаци за све димензије целог јата се чувају у матрицама, при чему су димензије матрице величина јата и број димензија D. Поставља се питање линеаризације ових матрица у меморију, односно да ли је ефикасније смештати податке о једном агенту у узастопне меморијске локације или податке о једној димензији. Овом приликом изабрано је да се подаци о једној димензији смештају један до другог. Распоред података је такав да се на почетку налази податак о првој димензији првог агента, па податак о првој димензији другог агента итд.

Одлука да се подаци распореде на овај начин подстакнут је архитектуром саме графичке картице. На графичкој картици језгра на којима се врши обрада су организована у скупове који се називају мултипроцесори. Један мултипроцесор увек извршава скуп од 32 нити који се назива Warp. Карактеристика једног Warp-a је да све нити у датом тренутку

извршавају исту инструкцију. Меморијски модули су тако организовани да могу да доставе податке за све нити из Warp-а у једном приступу, уколико су ти подаци узастопни. Оваквом организацијом постиже се да цео Warp добије податак по цени читања податка за једну нит, и такав приступ се назива поравнати приступ подацима (coalescing memory access). Како у овом програму нити из Warp-а углавном раде обраду над подацима из исте димензије, одлучено је да се ти подаци сместе у узастопне локације.

Мана оваквог приступа је код операција где свака нит из Warp-а ради са подацима из различитих димензија једног агента. Уколико су ти подаци довољно размакнути, за такво читање или упис једном Warp-u ће бити потребно да одради 32 пута читање или упис, односно по једном за сваку нит. Овакво понашање слично је читању матрице по колонама у секвенцијалним програмима и може озбиљно да деградира перформансе. Срећом, приликом имплементације на свега једном месту је било потребе за оваквим типом приступа.

Сваки мултипроцесор располаже са одређеном количином меморије на чипу, која има далеко бржи одзив. Ова меморија се назива дељена меморија [7] и може да се користи за побољшање перформанси решења. Дељена меморија је коришћена као мали програмабилни кеш за убрзавање извршавања функције нумеричке интеграције, као и алгоритма претраге најбољег агента. Капацитет дељене меморије на коришћеном хардверу је 48 kB. За функцију суме квадрата коришћење дељене меморије не доноси велико убрзање, пошто се сваком сабирку приступа по једном. Такође, ова функција је тестирана и за број димензија преко 500. При већим димензијама број података би премашивао величину дељене меморије, што би довело до пада перформанси програма.

2.3.2 Позив функције за ажурирање локалних података агената

Будући да графичка картица може да покрене велики број нити, могуће је радити ажурирање свих агената у паралели. За то је потребно покренути онолико нити на графичкој картици колико јато има агената.

CUDA уводи специјалну синтаксу у језик C којим се врши покретање нити на графичкој картици. Код који се извршава на графичкој картици потребно је изместити у засебну функцију која има модификатор __global__ на почетку свог потписа. Приликом позива овакве функције иза њеног имена задају се два скупа аргумената.

Унутар ознака <<< >>> наводе се аргументи које оперативни систем користи за покретање нити на графичкој картици. Могуће је навести три аргумента. Прва два се користе да се одреди број нити који се покреће на графичкој картици. На графичкој картици нити које су покренуте су организоване у блокове. Карактеристика једног блока је да се све његове нити извршавају на једном мултипроцесору, па је могуће вршити њихову синхронизацију. Први аргумент који се наводи представља број блокова који се покреће, а други представља број нити који ће сваки блок имати. За број нити по блоку треба узети неку вредност која представља умножак броја 32 (величина једног warp-а), будући да оперативни систем увек алоцира цео број warp-ова. У супротном, неки warp-ови ће имати нити које ће заузети језгра, али неће радити користан посао. У овом раду за величину једног блока је узето фиксно 256 нити по блоку. Број потребних блокова се одређује у зависности од величине јата.

Трећи аргумент је опционалан и представља количину дељене меморије коју сваки блок захтева. На основу овог аргумента одређује се колико блокова може у једном тренутку да се извршава на једном мултипроцесору. Количина дељене меморије која је тражена зависи од тога који оптимизациони проблем се решава. За функцију суме квадрата овај параметар није коришћен, док за функцију нумеричке интеграције сваки блок је узимао по D * sizeof(double) бајтова по нити. Узимајући у обзир количину доступне дељене меморије и потребе по блоку, највише два блока су могла у паралели да се извршавају на једном мултипроцесору. Уколико би постојала потреба за више блокова на једном мултипроцесору, њихово извршавање би се одвијало секвенцијално (два по два).

Други скуп аргумената приликом покретања су аргументи које функција прима. Синтакса за њихово навођење је иста као и код обичних C функција, у обичним заградама. Помоћу њих је могуће проследити податке над којима ће се вршити обрада.

Псеудокод за ажурирање јата је приказан у наставку (листинг 3):

```
__global__ update_agent(swarm_size, dimension, agents_velocity, agents_position,
    agents_lowest_error, agents_best_position, swarm_best_position)
{
    //ažuriranje jednog agenta
}

void pso()
{
    //...

    for(i = 0; i < number_of_turns; i++)
    {
        //poziv koda koji se izvršava na GPU
        update_agent <<<broj_blokova, broj_niti, deljena_memorija_po_bloku>>>
            (swarm_size, dimension, agents_velocity, agents_position,
                agents_lowest_error, agents_best_position, swarm_best_position)

        cudaDeviceSyncronize()

        //pronadji agenta koji ima najbolje rešenje
        best_agent = find_best_agent(agents_best_position, swarm_size)

        //sačuvaj novo najbolje rešenje jata ukoliko je pronađeno
        if (agents_lowest_error[best_agent] < swarm_min_error)
        {
            swarm_min_error = agents_lowest_error[best_agent]
            copy(swarm_best_position, agents_best_position[best_agent],dimension)
        }
    }

    //...
}
```

Листинг 3 – Псеудокод функције која позива обраду на графичком процесору.

Треба напоменути да је покретање обраде на графичком процесору асинхрона операција. То значи да нит на CPU наставља да се извршава паралелно са покренутим нитима на графичком процесору. Пошто је за исправан рад алгоритма неопходно да се ажурирање свих агената заврши комплетно пре него што се крене на наредне кораке, додаје се позив функције cudaDeviceSyncronize() [8]. Ова функција ће блокирати извршавање на CPU све док се не заврши покренуто извршавање на графичкој картици.

Поред аргумената који су прослеђени из главног програма, CUDA аутоматски прослеђује још неколико промењивих у функцију која се извршава на графичком процесору. У питању су промењиве које означавају индекс покренутог блока којем нит припада, индекс те нити у блоку, као и величину блока. На основу ове три вредности могуће

је одредити глобални индекс (односно идентификатор) једне нити. Глобални индекс нити се користи за индексирање агента чије вредности дата нит треба да ажурира.

У наставку (листинг 4) дат је приказ функције која ажурира једног агента јата.

```
__global__ update_agent(swarm_size, dimension, agents_velocity, agents_position,
    agents_lowest_error, agents_best_position, swarm_best_position)
{
    id = blockIdx.x * blockDim.x + threadIdx.x

    if (id < swarm_size)
    {
        for (k = 0; k < dimension; k++)
        {
            update_velocity(agents_velocity, id, k,
                agents_best_position, swarm_best_position)

            update_position(agents_position, id, k)
        }

        error = calc_opt_function(agents_position, id, opt_parameters)

        //sačuvaj novo najbolje rešenje agenta ukoliko je pronađeno
        if (error < agents_lowest_error[id])
        {
            agents_lowest_error[id] = error
            for (k = 0; k < dimension; k++)
            {
                agents_best_position[k][id] = agents_position[k][id]
            }
        }
    }
}
```

Листинг 4 – Псеудокод функција која ажурира једног агента на графичком процесору.

Ова функција се извршава за сваког агента по једном у једном ажурирању јата. Функција прво заузима дељену меморију уколико јој је потребна (код функције нумеричке интеграције). Кључном речи extern наводи се преводиоцу да је количина заузете меморије прослеђена приликом позива функције. Затим се рачуна глобални индекс нити и одређује да ли је у дозвољеном опсегу. Ово је неопходно пошто може доћи до ситуације да се из главног програма покрене више нити него што јато има агената. На пример, уколико је блок 256 нити, а јато има 1000 агената, биће неопходно да се покрену 4 блока, односно 1024 нити. Уколико је то случај, потребно је пре било какве даље обраде проверити да ли је добијени индекс у опсегу од 0 до величине јата. Након тога се извршава исти посао као код секвенцијалне имплементације, ажурирају се брзина агента и позиција, потом се рачуна оптимизациона функција и по потреби ажурира најбоље локално пронађено решење.

Приликом рачунања брзине постоји потреба за генерисањем псеудослучајних бројева у опсегу (0, 1]. При томе, доња граница овде није укључена јер коришћени генератор псеудослучајних бројева не генереришe нулу. За ту потребу коришћена је библиотека CURAND, која има неколико имплементираних генератора случајних бројева. Генератор који је овде коришћен је Mersenne Twister генератор [9], за кога је карактеристично да има добре стохастичке особине и дугачку периоду псеудослучајних бројева које генерише.

2.3.3 Проналазак најбољег агента

Како је већ речено, проналазак најбољег агента своди се на проналазак минимума у низу. Велики број доступних језгара може да се користи за испитивање великог броја елемената у паралели. На тај начин може да се скрати време извршавања функције.

Основна идеја је да се низ подели на два једнака подниза, при чему се њихови елементи у паралели упореде и мања вредност запише у први подниз. Добијени резултати се поново пореде на исти начин, све док се не добије да је број елемената који треба упоредити 0.

На пример, низ дужине 8 потребно је поделити на два подниза дужине 4. Након тога, у паралели се пореде први, други, трећи и четврти елементи оба подниза. Мање вредности се смештају у први подниз, који се потом поново дели на два дела. Тада се у паралели пореде први и други елементи нових поднизова. У наредном кораку након половљења, поредиће се први и једини елементи нових поднизова, након чега ће минимална вредност бити записана у првом елементу целог низа. Напомена, низ произвољне дужине третиран је као низ чији је број елемената први следећи степен броја два.

Да би алгоритам исправно радио, неопходно је увести одговарајућу синхронизацију међу нитима које пореде елементе. Наиме, да би се извршило неко поређење из другог корака алгоритма, неопходно је да се претходно заврше оба поређења из првог корака која дају улазне податке. Како је на графичком процесору синхронизација нити могућа искључиво на нивоу блока, није могуће имплементирати алгоритам проналаска најмањег елемента тако да да коначну вредност у једном покретању.

Идеја је да се овакав алгоритам покрене за више блокова, при чему сваки блок обради велики број елемената. Резултат овакве обраде је низ елемената који представљају минимуме у својим блоковима. Овај низ могуће је поново пропустити кроз исти алгоритам, при чему се тада добија минимум целог улазног низа на месту првог елемента. Међутим, како је алгоритам тестиран за величину јата до 50000 агената, после првог покретања алгоритма добијао се низ који је имао релативно мали број елемената. Емпиријски се показало да је низ ових димензија ефикасније било претражити помоћу CPU, па је на крају узета следећа имплементација. Сваки блок је проналазио минимум у поднизовима који су садржали 2048 елемената. Резултујући низ је претражен помоћу CPU, алгоритмом који тражи минимум у сложености O(n).

У наставку (листинг 5) је дат код којим се илуструје извршавање на CPU приликом тражења минимума. Овде су приказани позив ка графичкој картици, позив функције која тражи минимум на CPU, провера да ли је добијено ново најбоље решење и његово евентуално памћење.

```
void pso()
{
    //...

    for(i = 0; i < number_of_turns; i++)
    {
        update_agent <<<...>>> (...)
        cudaDeviceSyncronize()

        find_best_agent_gpu
            <<<broj_blokova, broj_niti, deljena_memorija_po_bloku>>>
            (errors, indexs, result_errors, result_indexs, swarm_size)

        cudaDeviceSyncronize()

        tmp = find_best_agent_cpu(result_errors, swarm_size)
        best_agent = result_indexs[tmp]

        //sačuvaj novo najbolje rešenje jata ukoliko je pronađeno
        if (agents_lowest_error[best_agent] < swarm_min_error)
        {
            swarm_min_error = agents_lowest_error[best_agent]

            copy_best <<<broj_blokova, broj_niti>>>
                (swarm_best_position, agents_position, best_agent, dimension)
            cudaDeviceSyncronize()
        }
    }

    //...
}
```

Листинг 5 – Псеудокод позива функције која на графичкој картици проналази минимуме.

У главној функцији pso, након што се заврши ажурирање агената и изврши синхронизација, поново се покреће извршавање на графичкој картици са циљем да се добију минимални елементи у сваком поднизу. На главном процесору неопходно је сачекати да графичка картица заврши пре него што се настави са обрадом, тако да се додаје још један позив функције cudaDeviceSyncronize(). Функцији која се извршава на графичком процесору се прослеђује пет параметара. Први параметар errors представља копију низа agents_lowest_error. Ову копију је потребно направити пошто функција мења прослеђене податке. Како функција треба да врати и индекс агента који је пронашао најмању вредност, неопходно је водити рачуна и о индексима агената који ће се чувати у низу indexs. Пре

позива, неопходно је иницијализовати овај низ вредностима од 0 до величине јата минус један. Наредна два параметра result_error и result_indexs чувају пронађене минимуме, односно индексе агената који су постигли те минимуме. Последњи параметар представља величину низа који се претражује.

Након што ова функција врати минимуме, низ се прослеђује функцији која на главном процесору проналази позицију најмањег елемента међу њима. Ова функција је обележена са find_best_agent_cpu и њена имплементација је приложена (листинг 6).

```
int find_best_agent_cpu(values, swarm_size)
{
    best = values[0]
    agent = 0
    for (i = 1; i < swarm_size; i++)
    {
        if (values[i] < best)
        {
            best = values[i]
            agent = i
        }
    }
    return agent
}
```

Листинг 6 – Псеудокод функције која проналази минимум у низу на CPU.

Уколико је пронађено боље решење од најбољег пронађеног решења до тог тренутка, потребно је запамтити позицију и вредност оптимизационе функције најбољег агента. Копирање позиције ефикасније је урадити на графичком процесору, будући да се и вредности које се копирају и локација где се копирају налазе у RAM меморији графичке картице. За потребе копирања позиције могуће је покренути онолико нити колико износи димензија проблема. Имплементација је једноставна, проверава се да ли је нит у одговарајућем опсегу, и ако јесте, копира податак за једну димензију. Оваква имплементација има предност у односу да се копирање ради са CPU-а, међутим како подаци у меморији који треба да се копирају нису поравнати, овај корак ће имати утицај на перформансе решења.

У листингу 7 дата је имплементација функције која тражи минимум у делу низа користећи графички процесор.

```
__global__ void find_best_agent_gpu
(double* errors, unsigned* indexs, double* result_errors,
      unsigned* result_indexs, unsigned len)
{
    start_index = blockIdx.x * blockDim.x * 2
    id = threadIdx.x
    step = blockDim.x
    pair = 0

    block_indexs = &indexs[start_index + id]
    extern __shared__ shared_memory[]

    if (start_index + id < len)
    {
        shared_memory[id] = errors[start_index + id]
    }

    if (start_index + id + step < len)
    {
        shared_memory[id + step] = errors[start_index + id + step]
    }

    __syncthreads()

    while (step > 0)
    {
        pair = id + step

        if ((start_index + pair < len) && (id < step)
            && (shared_memory[id] > shared_memory[pair]))
        {
            shared_memory[id] = errors[pair]
            block_indexs[id] = indexs[pair]
        }
        __syncthreads()

        step >>= 1
    }

    if (!threadIdx.x)
    {
        result_errors[blockIdx.x] = shared_memory[id]
        result_indexs[blockIdx.x] = block_indexs[id]
    }
}
```

Листинг 7 – Псеудокод функције која тражи минимум у делу низа на графичком процесору.

Функција на почетку рачуна позицију подниза у коме тражи минимум. Након тога, врши се копирање елемената подниза у дељену меморију, уколико је нит у валидном опсегу. Ту је потребно извршити синхронизацију нити пре него што се пређе на главни део програма. Променљива step памти тренутну величину поднизова, и када њена вредност падне на 0, најмања вредност у низу ће се наћи на првој локацији. На крају, потребно је записати минималну вредност и позицију на којој се она налазила у низове намењене за резултате. Тај посао препуштен је првој нити у блоку, оној која има индекс 0. Индекс у низовима на којима треба запамтити ове вредности је одређен индексом блока који тражи минимум.

Ова функција је имплементирана уз одређене оптимизације које не мењају логику рада, али чине код мање читљивим. Како би се постигло убрзање, тело петље у којој се ради поређење је развијено, односно њено тело је копирано одговарајући број пута. Том приликом вредност променљиве step је замењена одговарајућим константама. За величину блока је узето 512 нити. Такође, додато је да сваки блок обрађује 2048 елемената, односно свака нит прво пронађе минимум између четири елемената па тек онда ту вредност упише у дељену меморију. За то време информације о позицијама на којима се налазе решења су се чувала у оперативној меморији. Када програм сведе поднизове на величину 16, један warp је одговоран да заврши посао до краја. Од тог тренутка, могуће је учитати вредности о индексима у регистре нити и искористити warp shuffle [10] комуникацију између нити. Warp shuffle омогућава да нити из једног warp-а прослеђују вредности регистара међу собом, тако да се додатно смањује број одлазака до меморије.

2.3.4 Ограничења имплементације

За реализацију Unified меморије неопходна графичка картица са одређеним хардверским особинама, па се уводи ограничење да је за рад програма неопходна графичка картица која подржава API 3.0 или новији. Warp shuffle је такође доступан тек од верзије 3.0.

Употреба Mersenne Twister генератора псеудослучајних бројева из CURAND библиотеке намеће неколико ограничења у виду броја блокова и броја нити по блоковима. Овај генератор користи велики број параметара како би генерисао случајне бројеве. Могуће је или користити предефинисане скупове параметара, или их задавати у програму. У решењу су коришћени предефинисани скупови параметара, будући да они дају добре стохастичке особине, а да за потребе имплементације PSO алгоритама статистика генератора случајних бројева није од пресудног значаја. CURAND библиотека има ограничење да највише 200 оваквих скупова параметара може да се користи. Такође, највише 256 нити из једног блока могу безбедно да користе један скуп параметара, а да не долази до проблема међусобног утркивања. Из овог разлога узето је да један блок нити приликом ажурирања агената има тачно 256 нити. Такође, максималан број нити који може да обавља ажурирање агената без потребе за синхронизацијом износи 200 * 256 нити, односно 51200 нити. Како су рађени тестови за величине јата до 50000 агената, није било потребе да се уводи додатна синхронизација, или алоцира још генератора.

2.4 Имплементација помоћу OpenMP API-ја

У овом потпоглављу биће приказано како је извршена паралелизација помоћу OpenMP API-ја. Као и код претходно приказаног решења, паралелизовани су кораци ажурирања агената јата и тражења најбољег агента.

Паралелизација се извршава користећи специјалне претпроцесорске директиве. Услов је да компајлер подржава OpenMP, како би могао да разуме ове директиве и генерише код који ће се извршавати у паралели на доступним језгрима процесора.

За разлику од графичког процесора, CPU има мање језгара и може да извршава далеко мање нити у паралели. Последица тога је да се неће сваки агент ажурирати паралелно у посебној нити, већ ће доступне нити добити да ажурирају подједнаке делове јата. Такође, код тражења најбољег агента низ са решењима ће бити подешен на поднизове једнаких димензија, и свака нит ће добити да претражи један подниз.

У наставку (листинг 8) је приказано како је коришћена omp parallel for директива за паралелизацију ажурирања јата.

```
void pso()
{
    //...
    for (i = 0; i < number_of_turns; i++)
    {
        // za shared parametre treba navesti sve parametre funkcije update_agent
        // osim j
        #pragma omp parallel for shared(...) private(j)
        for (j = 0; j < swarm_size; j++)
        {
            update_agent(swarm_size, dimension, agents_velocity, agents_position,
                agents_lowest_error, agents_best_position, swarm_best_position, j
)
        }

        best_agent = find_best_agent(agents_best_position, swarm_size)

        //sačuvaj novo najbolje rešenje jata ukoliko je pronađeno
        if (agents_lowest_error[best_agent] < swarm_min_error)
        {
            swarm_min_error = agents_lowest_error[best_agent]
            copy(swarm_best_position, agents_best_position[best_agent],dimension)
        }
    }
    //...
}
```

Листинг 8 – Псеудокод који приказује паралелизацију ажурирања агената помоћу OpenMP API-ја.

Директива се односи на петљу изнад које је написана. У позадини, свакој нити ће бити прослеђен један опсег у тој петљи који нит треба да изврши. На пример, уколико имамо јато величине 60 агената и процесор може да извршава 12 нити у паралели, прва нит ће добити да изврши итерације ове петље за опсег бројача од 0 до 5, друга нит од 5 до 10 и тако даље.

Коришћена су два типа аргумената приликом коришћења директиве. За дељене (shared) параметре наводе се сви подаци чије вредности треба да буду заједничке за све нити. Вредност приватних (private) аргумената ће се разликовати за сваку нит. Унутар тела врши се позив функције која ажурира локалне податке једног агента, при чему се аргументом j специфицира који агент се ажурира.

Претрага најбољег агента илустрована је следећим псеудокодом (листинг 9):

```
int find_best_agent(errors, swarm_size)
{
    global_best_value = errors[0]
    global_best_agent = 0

    local_best_value, local_best_agent, i = 0

    #pragma omp parallel shared(errors, swarm_size, global_best_agent,
        global_best_value) private(i, local_best_agent, local_best_value)
    {
        local_best_value = errors[0]
        local_best_agent = 0

        #pragma omp for private(i)
        for (i = 0; i < swarm_size; i++)
        {
            if (errors[i] < local_best_value)
            {
                local_best_value = errors[i]
                local_best_agent = i
            }
        }

        #pragma omp critical
        {
            if (local_best_value < global_best_value)
            {
                global_best_value = local_best_value
                global_best_agent = local_best_agent
            }
        }
    }
    return global_best_agent
}
```

Листинг 9 – Псеудокод функције која тражи најбољег агента помоћу OpenMP API-ја.

На почетку дефинишу се промењљиве у функцији које ће бити коришћене. Прва директива специфицира наредбе које ће бити рађене у паралели. Друга директива служи да се раздели посао на расположиве нити, као што је то био случај код ажурирања агената. Свака нит за себе тражи минимално решење у поднизу који је добила. Када заврши, потребно је да провери да ли је њено решење мање од оног које је до тада важило за најмање. Уколико јесте, нит уписује своје решење као најбоље. Како се овом приликом читају и модификују заједнички подаци за све нити, неопходно је увести одговарајућа ограничења за приступ тим подацима. Трећом директивом (omp critical) означава се да је овај сегмент кода потребно извршити атомично. На тај начин спречава се потенцијално међусобно утркивање између нити. На крају, када све нити заврше обраду, функција враћа индекс најбољег агента.

3 Резултати

У овом поглављу биће дат приказ добијених резултата помоћу имплементираних решења. Биће приказане оптимизационе функције коришћене за тестирање развијеног решења. Након тога дато је поређење добијених резултата оптимизације и поређење перформанси ових решења за обе оптимизационе функције.

Све симулације су рађене на истом хардверу. Имплементација помоћу OpenMP API-ја покретана је на 9th Generation Intel® Core™ i7-9750H процесору [11]. Овај процесор располаже са 12 MB кеш меморије, 6 језгара, при чему брзина такта иде до 4.50 GHz. Имплементација помоћу CUDA платформе је покретана на NVIDIA® GeForce® GTX 1650 графичкој картици [12]. Ова графичка картица располаже са 4 GB GDDR5 RAM меморије, такт је у границама 1485 - 1665 MHz, а сама картица има 896 језгара.

3.1 Сума квадрата као оптимизациона функција

Функција суме квадрата представља пример једноставне аналитички задате оптимизационе функције. Формула по којој се рачуна оптимизациона функција се математички може записати:

$$f(x) = \sum_{k=1}^{D} x_k^2 \tag{3}$$

Јато треба да пронађе минимум ове функције, за вредности X_k у опсегу [0, 1). Решење оптимизационог проблема је такође у опсегу [0, 1). Оно је аналитички познато и једнако је 0. Тежина решавања оптимизационог проблема сразмерна је броју димензија (D), односно броју сабирака које јато треба да сведе до нуле.

За овај оптимизациони проблем поређено је коначно решење добијено након 1000 ажурирања јата. Како је у питању лака оптимизациона функција, очекује се да алгоритам да добре резултате. Такође, поређено је време потребно за једно ажурирање агената, као и време потребно за проналазак најбољег агента. Након тога, графички је представљено време потребно за једно ажурирање јата.

У наставку дата је Табела 1 у којој се налазе постигнути резултати оптимизације за проблем суме квадрата. Опсег у коме се налазе решења је [0, veličina_jata), при чему решења ближа нули представљају боља решења. Алгоритам оптимизације јатом је покретан по једном за девет различитих величина јата (100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000) и за шест различитих вредности димензија оптимизационог проблема (10, 20, 50, 100, 200, 500).

За свих девет величина јата и свих шест димензија обе имплементације су дале инжењерски добра и слична решења. Из табеле се може видети да је алгоритам оптимизације

јатом погодан за решавање овог проблема, будући да је у већини случајева успевао да пронађе решење са прецизношћу до 5 значајних цифара.

Табела 1 – резултати оптимизације за функцију суме квадрата
(0 је тачно решење, мања вредности у табели представља бољи резултат)

	100		200		500	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
10	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
20	0,00000	0,00819	0,00000	0,00000	0,00000	0,00000
50	0,00000	0,71839	0,00000	0,00000	0,00000	0,00000
100	0,00614	2,25217	0,00005	0,00019	0,00000	0,00000
200	0,35760	5,30591	0,31710	0,12539	0,02444	0,00229
500	9,25448	13,93755	8,45640	6,52406	3,59048	2,44339
	1000		2000		5000	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
10	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
20	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
50	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
100	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
200	0,00143	0,00155	0,00078	0,00010	0,00003	0,00014
500	2,53527	3,01016	1,47841	1,45983	1,29385	1,69142
	10000		20000		50000	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
10	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
20	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
50	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
100	0,00000	0,00000	0,00000	0,00000	0,00000	0,00000
200	0,00001	0,00000	0,00001	0,00000	0,00000	0,00000
500	0,91753	0,41364	0,86295	0,58753	0,47623	0,50840

У наставку су дате Табела 2 и Табела 3 у којима су приказане перформансе приликом покретања алгоритма за овај проблем. Табела 2 приказује потребно време да се изврши једно ажурирање свих агената јата. Табела 3 приказује време потребно за проналазак најбољег агента, као и његово памћење уколико је потребно. Сви резултати изражени су у милисекундама.

Табела 2 – Време једног ажурирања свих локалних података агената за функцију суме квадрата. Времена су изражена у ms.

	100		200		500	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
10	0,028	0,169	0,041	0,196	0,109	0,220
20	0,059	0,225	0,086	0,267	0,174	0,261
50	0,106	0,434	0,167	0,492	0,364	0,505
100	0,171	0,734	0,303	0,829	0,816	0,868
200	0,297	1,371	0,552	1,564	1,173	1,643
500	0,676	3,306	1,280	3,756	3,125	3,979
	1000		2000		5000	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
10	0,313	0,206	0,556	0,229	1,037	0,221
20	0,425	0,308	0,778	0,265	1,194	0,324
50	0,937	0,521	1,639	0,544	2,790	0,654
100	1,748	0,917	3,374	0,947	5,563	1,194
200	3,048	1,697	6,184	1,750	10,789	2,263
500	7,095	4,043	13,305	4,162	26,517	5,454
	10000		20000		50000	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
10	1,186	0,422	3,343	0,531	7,292	1,016
20	2,674	0,579	6,401	0,868	14,441	1,745
50	5,507	1,059	14,894	1,600	33,065	3,851
100	10,979	2,003	26,693	2,939	64,390	7,454
200	22,201	3,795	52,609	5,827	126,366	14,645
500	54,262	7,240	115,018	13,894	302,030	32,125

За мали број агената у јату и мали број димензија може се видети да OpenMP постиже бољи резултат. Овакав резултат се може објаснити чињеницом да CPU ради на значајно већој фреквенцији сигнала такта и располаже са разним софистицираним технологијама попут предиктора скокова. Односно, CPU је ефикаснији када треба да се брже одради мали број послова.

Са повећањем броја агената и димензија имплементација помоћу графичке картице почиње да постиже значајно боље резултате. Овде до изражаја долази значајно већи број језгара на којима се врши обрада, као и специфична меморијска организација, која може да пружи податке без потребе да језгра дуго чекају. За очекивати је да се даљим повећањем броја агената додатно повећа ова разлика у перформансама у корист имплементације на графичкој картици.

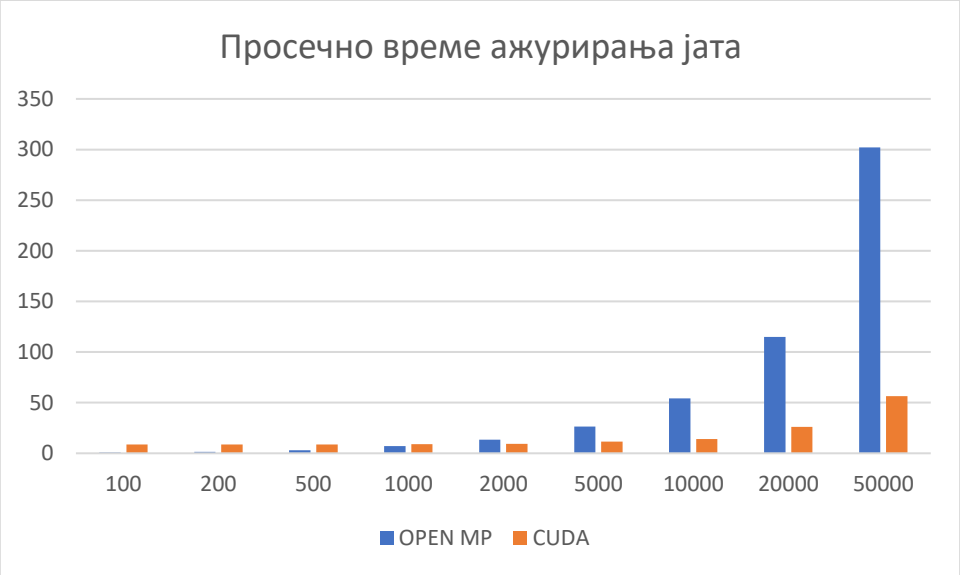
Табела 3 – Време проналаска најбољег решења и његовог чувања за функцију суме квадрата. Времена су изражена у ms.

	100		200		500	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
10	0,009	1,517	0,016	1,441	0,010	1,313
20	0,010	1,604	0,016	1,471	0,003	1,435
50	0,008	1,792	0,005	1,703	0,011	1,628
100	0,011	2,220	0,006	2,057	0,010	2,036
200	0,003	2,799	0,007	2,796	0,006	2,739
500	0,008	5,255	0,011	4,824	0,012	4,747
	1000		2000		5000	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
10	0,021	1,344	0,024	1,281	0,024	1,534
20	0,018	1,469	0,008	1,526	0,013	1,603
50	0,021	1,660	0,018	1,719	0,013	1,844
100	0,017	2,079	0,012	2,080	0,012	2,351
200	0,015	2,789	0,015	2,904	0,016	3,233
500	0,025	5,085	0,022	5,275	0,023	6,085
	10000		20000		50000	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
10	0,007	1,960	0,031	1,945	0,078	2,411
20	0,016	2,055	0,030	2,564	0,075	3,160
50	0,016	2,487	0,048	2,714	0,077	4,997
100	0,021	3,540	0,042	3,866	0,092	8,256
200	0,015	5,337	0,049	6,291	0,093	14,293
500	0,022	6,783	0,052	12,198	0,093	24,335

У кораку проналаска најбољег агента и његовог евентуалног чувања решење помоћу OpenMP API-ја постиже значајно боље резултате, чак и преко 200 пута боље. Постоји неколико могућих објашњења зашто је овде велика разлика у перформансама. Прво, будући да се са низом у коме се памте вредности оптимизационе функције ради пред крај ажурирања агента, велика је вероватноћа да су подаци за овај корак већ спремни у кеш меморији. Даље, приликом имплементације на графичком процесору постоји неколико корака који могу да деградирају перформансе. Из кода који се извршава на CPU потребно је читати одређене податке из меморије графичке картице. Оперативни систем ће вршити копију читаве странице у којој се податак налази. Такође, као што је већ наведено у опису проблема, овде долази до непоравнатих читања из меморије када се ради копирање нове најбоље позиције,

што за последицу има пад перформанси. Негативан утицај овог корака сразмеран је повећању димензије проблема.

Збир времена потребног за ажурирање агената јата и ажурирање најбољег решења представља време једног ажурирања јата. На наредном графику (слика 1) дато је поређење просечних времена ажурирања јата у зависности од величине јата, при чему је за димензију проблема узето 500. Времена на у-оси су изражена у милисекундама (ms).



Слика 1 – просечно време ажурирања јата у ms

Из дијаграма се јасно види да за већа јата коришћење графичког процесора у великој мери поправља перформансе. На овакав резултат највећи утицај је имало велико убрзање које је постигнуто на графичкој картици приликом ажурирања локалних података јата. Иако је корак ажурирања најбољег агента јата био преко 260 пута спорији код имплементације на графичкој картици (24,335 ms на према 0,093 ms), показало се да он ипак има значајно мањи утицај на перформансе целог програма.

Уколико се посматра време извршавања целог програма, потребно је размотрити и време потребно за иницијализацију података и враћање заузетих података. За претходно коришћене параметре (величина јата 50000 агената, број димензија 500) на заузимање ресурса и иницијализацију података потрошено је значајно више времена у имплементацији на графичком процесору (9730,000 ms у односу на 2304,000 ms колико је потрошено код OpenMP имплементације). Слично је и за враћање заузетих ресурса, при чему су апсолутне вредности мање (259,000 ms у односу на 153,000 ms). Иако на ове кораке треба потрошити незанемарљиво време, у овом раду нису изложени детаљни подаци о перформансама за њих. Разлог је што се они извршавају свега једном по покретању, па је њихов удео у укупном времену лимитиран и опада са повећањем броја ажурирања јата које треба да се уради.

3.2 Нумеричка интеграција као оптимизациони проблем

У овом потпоглављу биће приказан оптимизациони проблем нумеричке интеграције и дате перформансе и решења која су добијена.

Приликом поређења биће прво приказан однос перформанси два имплементирана решења. Приказано је просечно време потребно за ажурирање агената, време потребно за проналазак и евентуално памћење најбољег агента и биће дат графички приказ времена потребног за једно ажурирање јата. Након тога дат је преглед укупног времена извршавања за 2000 ажурирања јата.

За сваку конфигурацију (величина јата, m_{max}) рађено је по 100 покретања. Добијене вредности перформанси су просечне вредности за свих 100 извршавања. Добијени резултати оптимизације су искоришћени како би се израчунали кумулативни минимуми (просечно најбоље постигнуто решење) за сваку конфигурацију. Кумулативни минимуми за имплементацију помоћу графичке картице ће бити графички приказани.

На крају, имплементација помоћу графичке картице је покретана како би се пронашло најбоље могуће решење за овај проблем. Коначно решење биће упоређено са решењем које је постигнуто помоћу алгоритма диференцијалне еволуције (алтернативни алгоритам за решавање нелинеарних проблема).

Овај оптимизациони проблем се може дефинисати на следећи начин. Тражи се вредност интеграла чија је подинтегрална функција:

$$f(x) = x^m \ln(x), m \in \mathrm{N}_0 \tag{4}$$

у границама интеграције од 0 до 1. Интеграл се рачуна нумерички на следећи начин:

$$\int_0^1 f(x)dx \approx \sum_{k=1}^{N} \mathrm{w_k} * f(\mathrm{X_k}) \tag{5}$$

Оптимизациони проблем представља одређивање N парова (w_k, x_k) тако да дата апроксимација буде што тачнија. Треба напоменути да је решење овог интеграла за дату функцију аналитички познато, односно важи:

$$\int_0^1 x^m \ln(x)\, dx = -\frac{1}{(m+1)^2}, m \in \mathrm{N}_0 \tag{6}$$

На основу једначина (5) и (6) оптимизациону функцију можемо дефинисати и математички записати као:

$$f(x_1, x_2 \dots x_n, w_1, w_2 \dots w_n) = \frac{1}{m_{max}} * \sum_{m=1}^{m_{max}} \frac{\left| \frac{1}{(m+1)^2} + \sum_{k=1}^{N} w_k * f(x_k) \right|}{\frac{1}{(m+1)^2}} \quad (7)$$

Како јато прилази тачном решењу, вредност бројиоца се приближава нули. Када се за све сабирке спољашње суме добије вредност нула, тачно решење је пронађено. Димензија проблема зависи од параметра N за који ће бити узета вредност 5. Димензију проблема је могуће израчунати као N * 2. Други важан параметар је m_{max}. Повећавањем ове вредности расте тежина оптимизационог проблема.

Проблеми који су слични овом појављују се у нумеричкој анализи електромагнетских система и од изузетне су важности за прецизну нумеричку анализу.

У наставку следе добијене перформансе и резултати. Прво ће бити дато поређење перформанси обе имплементације. У Табела 4 дат је преглед времена потребног за ажурирање агената јата. Тестирање је вршено за исте величине јата као раније (100, 200, 500, 1000, 2000, 5000, 10000, 20000, 50000). За параметар m_{max} је узимана вредност од 1 до 10.

Табела 4 – Време једног ажурирања свих локалних података агената за проблем нумеричке интеграције. Времена су изражена у ms.

	100		200		500	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
1	0,040	0,242	0,074	0,276	0,161	0,290
2	0,044	0,253	0,090	0,293	0,179	0,302
3	0,050	0,280	0,074	0,341	0,197	0,370
4	0,051	0,310	0,086	0,399	0,199	0,429
5	0,047	0,344	0,102	0,447	0,253	0,487
6	0,055	0,369	0,106	0,501	0,244	0,558
7	0,058	0,393	0,103	0,558	0,271	0,620
8	0,059	0,430	0,117	0,611	0,283	0,672
9	0,064	0,467	0,133	0,664	0,312	0,738
10	0,078	0,493	0,154	0,718	0,320	0,801
	1000		2000		5000	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
1	0,313	0,288	0,638	0,291	1,497	0,380
2	0,375	0,308	0,723	0,305	1,622	0,409
3	0,384	0,364	0,796	0,371	1,747	0,535
4	0,438	0,421	0,893	0,426	1,968	0,655
5	0,445	0,488	0,857	0,479	2,030	0,781
6	0,451	0,543	0,870	0,548	2,164	0,906
7	0,553	0,609	1,085	0,608	2,290	0,983
8	0,560	0,676	1,038	0,672	2,488	1,090
9	0,578	0,736	1,083	0,731	2,648	1,097
10	0,641	0,798	1,158	0,792	2,801	1,118
	10000		20000		50000	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
1	2,933	0,474	5,912	0,702	14,372	1,476
2	3,166	0,528	6,312	0,803	15,488	1,515
3	3,411	0,710	6,721	1,117	17,113	2,049
4	3,674	0,902	7,337	1,156	17,938	2,693
5	3,959	1,047	7,836	1,355	19,024	3,312
6	4,196	1,092	8,378	1,585	20,650	3,948
7	4,519	1,134	8,944	1,823	22,482	4,554
8	4,765	1,274	9,417	2,079	23,599	5,206
9	5,081	1,409	10,088	2,292	25,098	5,807
10	5,367	1,546	10,497	2,551	26,255	6,410

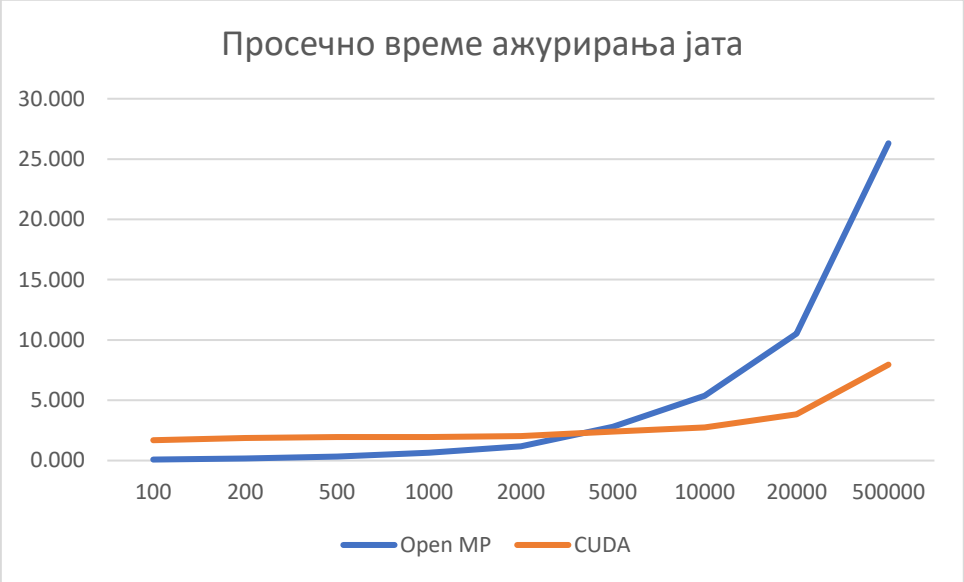
Из табеле се може видети да је имплементација помоћу OpenMP API-ја постигла боље резултате за јата до 2000 агената. Након тога могућности графичке картице долазе до изражаја. Са даљим повећавањем броја агената повећава се и разлика у перформансама у корист имплементације помоћу графичке картице.

Табела 5 - Време проналаска најбољег решења и његовог чувања за функцију нумеричке интеграције. Времена су изражена у ms.

	100		200		500	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
1	0,007	1,060	0,008	1,070	0,008	1,064
2	0,008	1,076	0,009	1,072	0,009	1,084
3	0,009	1,095	0,008	1,097	0,007	1,088
4	0,006	1,132	0,007	1,132	0,008	1,123
5	0,007	1,118	0,008	1,130	0,007	1,131
6	0,007	1,176	0,008	1,174	0,009	1,138
7	0,009	1,199	0,009	1,140	0,008	1,131
8	0,008	1,134	0,006	1,149	0,008	1,136
9	0,007	1,141	0,008	1,163	0,010	1,135
10	0,008	1,196	0,009	1,157	0,008	1,143
	1000		2000		5000	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
1	0,008	1,066	0,009	1,072	0,008	1,078
2	0,007	1,074	0,011	1,083	0,010	1,095
3	0,008	1,094	0,009	1,093	0,012	1,103
4	0,011	1,117	0,011	1,143	0,012	1,169
5	0,008	1,131	0,010	1,183	0,010	1,198
6	0,011	1,161	0,009	1,198	0,012	1,251
7	0,009	1,138	0,010	1,206	0,012	1,153
8	0,010	1,145	0,010	1,193	0,010	1,294
9	0,010	1,156	0,008	1,215	0,012	1,301
10	0,010	1,154	0,012	1,253	0,012	1,283
	10000		20000		50000	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
1	0,013	1,096	0,014	1,166	0,043	1,338
2	0,013	1,125	0,016	1,183	0,041	1,345
3	0,013	1,134	0,016	1,187	0,053	1,369
4	0,013	1,155	0,020	1,188	0,045	1,386
5	0,014	1,197	0,022	1,281	0,049	1,464
6	0,015	1,233	0,017	1,263	0,050	1,438
7	0,015	1,206	0,018	1,232	0,054	1,497
8	0,013	1,217	0,019	1,306	0,054	1,499
9	0,014	1,213	0,017	1,301	0,053	1,555
10	0,015	1,197	0,020	1,288	0,051	1,544

Из Табела 5 могу се видети перформансе за корак тражења најбољег агента. Као и код функције суме квадрата, овај корак се обавља значајно брже помоћу OpenMp API-ја. Разлози за овакво понашање су исти као и код функције суме квадрата, будући да имплементација овог корака не зависи од оптимизационе функције.

У наставку (слика 2) приказан је график који илуструје потребно време да се изврши цела имплементација у зависности од величине јата. За тежину проблема узето је da je m_{max} = 10. Као и до сада, приказана времена су у милисекундама.



Слика 2 – просечно време једног ажурирања јата у ms

Добијени резултати су у складу са резултатима добијеним за функцију суме квадрата. Имплементација на графичком процесору у почетку постиже лошији резултат. Тај резултат је последица мало споријег ажурирања јата и доста лошијих перформанси приликом тражења најбољег агента. Са друге стране, функција има значајно спорији раст код имплементације на графичком процесору, па се за велика јата добијају осетно боље перформансе.

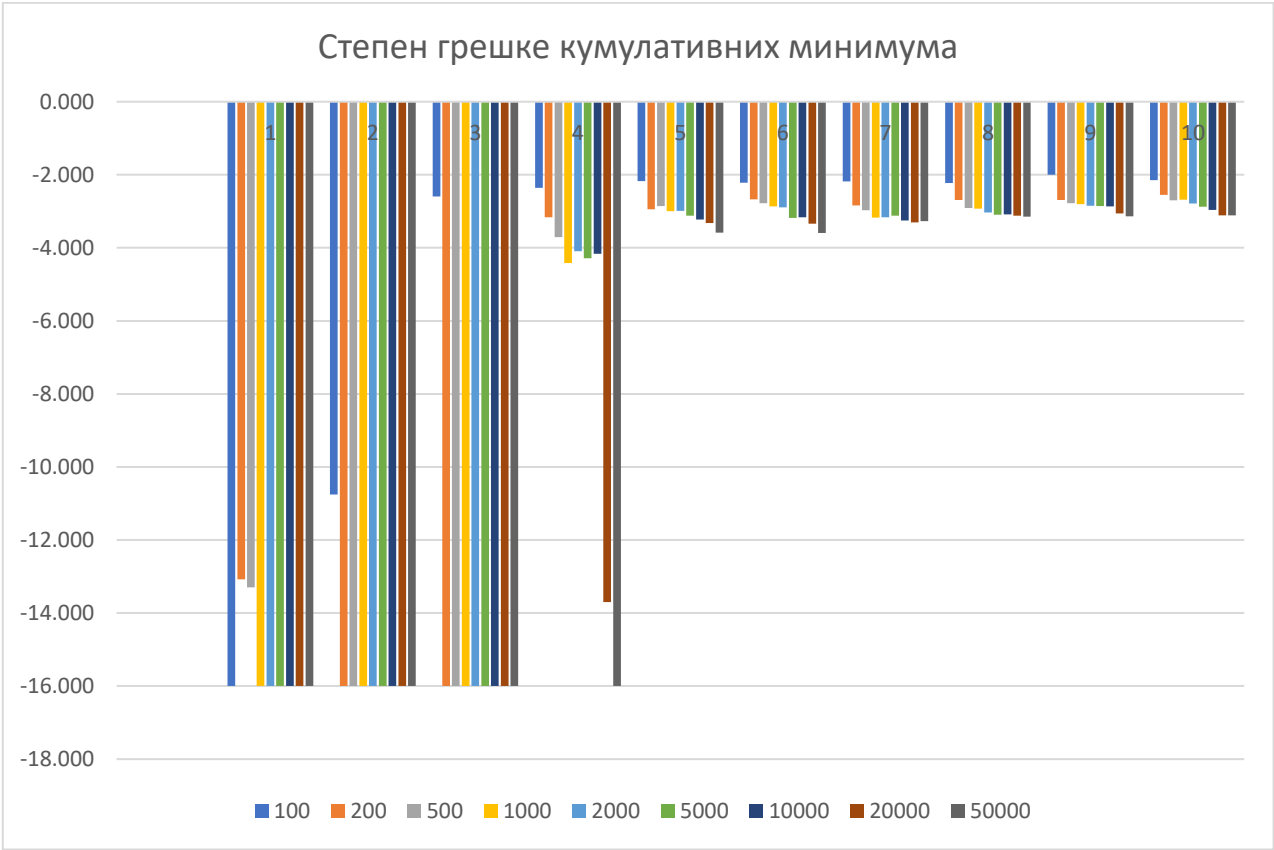
У наставку је дата Табела 6 у којој је приказано укупно време потрошено на оптимизацију помоћу обе имплементације. Притом укључена су времена потребна за заузимање меморије и иницијализацију података, 2000 ажурирања јата и деалокацију заузете меморије. Све вредности у табели су изражене у секундама.

Табела 6 – укупно време трајања оптимизације за 2000 ажурирања. Сва времена су дата у секундама.

	100		200		500	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
1	0,096	2,679	0,168	2,745	0,343	2,759
2	0,104	2,721	0,199	2,793	0,380	2,836
3	0,119	2,838	0,167	2,955	0,411	2,990
4	0,115	3,001	0,189	3,178	0,419	3,211
5	0,111	3,024	0,222	3,265	0,524	3,341
6	0,124	3,243	0,231	3,498	0,510	3,513
7	0,136	3,381	0,229	3,524	0,561	3,609
8	0,135	3,242	0,249	3,652	0,586	3,739
9	0,144	3,325	0,285	3,801	0,651	3,874
10	0,174	3,552	0,329	3,878	0,661	4,015
	1000		2000		5000	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
1	0,649	2,760	1,303	2,779	3,026	2,984
2	0,770	2,830	1,475	2,840	3,279	3,081
3	0,791	2,987	1,620	3,001	3,535	3,354
4	0,902	3,183	1,815	3,261	3,977	3,781
5	0,912	3,341	1,744	3,477	4,096	4,129
6	0,931	3,536	1,766	3,652	4,366	4,528
7	1,130	3,608	2,198	3,807	4,619	4,396
8	1,144	3,769	2,106	3,906	5,012	5,026
9	1,182	3,923	2,191	4,081	5,336	5,048
10	1,307	4,024	2,347	4,305	5,641	5,031
	10000		20000		50000	
	OpenMP	CUDA	OpenMP	CUDA	OpenMP	CUDA
1	5,925	3,216	11,918	3,833	28,968	5,807
2	6,393	3,395	12,720	4,071	31,193	5,897
3	6,880	3,788	13,540	4,716	34,470	7,024
4	7,406	4,234	14,779	4,808	36,103	8,350
5	7,980	4,639	15,781	5,456	38,282	9,787
6	8,453	4,855	16,862	5,864	41,536	10,987
7	9,103	4,848	17,987	6,246	45,212	12,368
8	9,589	5,162	18,942	6,935	47,443	13,658
9	10,225	5,416	20,280	7,368	50,439	15,011
10	10,798	5,648	21,103	7,853	52,748	16,182

За сваки скуп улазних параметара вршено је по 100 покретања. За свако покретање запамћено је и најбоље постигнуто решење. На основу најбољих решења из сваког покретања израчунат је кумулативни минимум (средња вредност најбољих резултата сваког покретања). На наредном графику (слика 3) дат је преглед степена грешке кумулативних минимума за сваку комбинацију улазних параметара (величина јата, m_{max}). Како је машинска

тачност реда величине 10^{-15}, резултат -16 на дијаграму означава да је пронађен тачан резултат, односно аналитички познато решење и решење добијено оптимизацијом се поклапају на 15 значајних цифара. С обзиром на то су коришћени бројеви у двострукој прецизности, теоријски није могуће постићи бољу прецизност.



Слика 3 – ред величине грешке најбољег пронађеног решења током оптимизације за вредности m_{max} од 1 до 10, и свих 9 девет величина јата

Са датог графика можемо закључити да за мале вредности mmax оптимизациони проблем није тежак и може се решити помоћу оптимизације јатом за свега пар секунди. За вредност 4, овај проблем постаје доста тежи и успешно је решен уз помоћ великих јата. За вредности веће од 4, алгоритам оптимизације јатом није успевао да пронађе поклапање на више од 3 децимале у 2000 ажурирања јата. Том приликом величина јата није играла велику улогу, будући да је добијен исти степен тачности за све величине јата.

Након тога алгоритам је покренут са циљем да се пронађе тачно решење до машинске тачности. Том приликом покренут је алгоритам са 32678 агената и дато му је 1.200.000 ажурирања јата. Ово покретање је трајало мало мање од два сата и поновљено је више пута. Иако је урађено 6000 пута више ажурирања у сваком покретању, алгоритам није успео да пронађе много прецизније решење. Степен грешке остаје 10^{-4}, при чему је минимална добијена грешка 0,000155803307466.

Овај проблем решаван је и помоћу алгоритма диференцијалне еволуције [1], ради поређења добијеног резултата. За свега неколико минута алгоритам диференцијалне еволуције успео је да пронађе значајно боље решење. Након 23000 генерација, минимална грешка је износила 0,000007667772933. Степен грешке је 10^{-6}.

Овај проблем је намерно изабран као тежак оптимизациони проблем ради провере перформанси имплементираног алгоритма као и сагледавања разлика између PSO алгоритма и других оптимизационих алгоритама.

На основу поређења са решењем које је постигао алгоритам диференцијалне еволуције, можемо закључити да уколико се тражи решење са великом прецизношћу, алгоритам оптимизације јатом није најефикаснији алгоритам за решавање овог оптимизационог проблема.

У Табела 7 дате су координате у оптимизационом простору за које су алгоритми оптимизације јатом и диференцијалне еволуције пронашли најбоље вредности. У последњој колони, дате су и координате тачног решења.

Табела 7 – приказ координата добијених решења и тачног решења

Параметар:	Оптимизација јатом:	Диференцијална еволуција	Тачно решење
X1	0,976741041848277	0,360244169402227	0,070962713742682
X2	0,530034325759424	0,279876480518343	0,242854538403076
X3	0,246469591079672	0,891781161994079	0,477865040535688
X4	0,002456318458440	0,174312595352743	0,719992203868191
X5	0,793531426799322	0,605465318432087	0,909947523904315
W1	0,139689029642347	0,124894094897430	0,125608096118729
W2	0,282040216087830	0,112914131616367	0,211715949646026
W3	0,277355222591234	0,195102587609467	0,248711371709213
W4	0,897480202511278	0,687676626062062	0,225395652758139
W5	0,233793529533220	0,168024865242341	0,146438559921064

4 Закључак

У овом раду приказан је рад алгоритма оптимизације јатом, и начин на који он може да се паралелизује користећи OpenMP API и CUDA платформу. Алгоритам је коришћен за решавање два оптимизациона проблема. Први је проблем проналаска минимума функције суме квадрата за D варијабли. Други оптимизациони проблем је одређивање интеграла једне функције нумеричком методом. Након тога, поређене су перформансе обе имплементације, и имплементација на графичкој картици је коришћена како би се пронашло најбоље могуће решење за проблем нумеричке интеграције.

Обе имплементације су показале да имају своје предности. Као кључан параметар показала се величина јата, односно од броја агената зависи која имплементација ће имати боље перформансе. За мања јата, реда неколико стотина или хиљада агената, обе реализације се добро понашају, при чему имплементација помоћу OpenMP API-ја успева да изврши пар хиљада ажурирања јата неколико секунди брже. Приликом коришћења већих јата, имплементација помоћу графичке картице показује значајно боље перформансе и може у великој мери да скрати трајање оптимизације. Стога закључујемо да је из угла перформанси алгоритам оптимизације јатом погодан да се имплементира на графичком процесору.

Приликом одабира имплементације треба размотрити неколико фактора. Поред величине јата, треба напоменути да је за развој и тестирање имплементације на графичком процесору било потребно више времена, па је целокупан процес решавања проблема од његовог дефинисања, до добијања коначног решења трајао мало дуже. Такође, за ову имплементацију је потенцијално потребно уложити и ресурсе, будући да се захтева додатан хардвер (NVIDIA графичка картица). За лакше оптимизационе проблеме попут суме квадрата, поставља се питање оправданости развоја оваквог решења, будући да је за слично време OpenMP реализација дала прихватљив резултат. Код тежих оптимизација, попут решавања проблема нумеричке интеграције до машинске тачности, употреба овакве имплементације је значајно скратила време извршавања. Уколико би се наставило са процесом оптимизације, време потрошено за извршавање би се у довољној мери скратило да се развој оваквог решења исплати. Стога можемо закључити да је за тешке оптимизационе проблеме, где ће се користити већа јата, и где ће сам процес оптимизације дуже трајати, инжењерски оправдано разматрати имплементацију на графичком процесору.

5 Литература

[1] Д. Олћан, Инжењерски оптимизациони алгоритми – скрипте за предавања, 2020.

[2] Марко Мишић, Compute Unified Device Architecture (CUDA) - http://mups.etf.rs/vezbe/MPS%20-%20CUDA.pdf, (20.6.2020)

[3] Марко Мишић, Јанко Илић, OpenMP - http://mups.etf.rs/vezbe/MPS%20-%20OpenMP.pdf, (20.6.2020)

[4] Kernigham W. Brian, Ritchie M. Dennis, „Програмски језик C“, друго издање, CET, 2003.

[5] Краус Ласло, „Програмски језик C++ са решеним задацима“, 10. издање, Академска мисао, 2016.

[6] Cuda toolkit documentation – Unified memory, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd, (20.6.2020)

[7] Cuda toolkit documentation – Shared memory, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#shared-memory, (20.6.2020)

[8] Cuda toolkit documentation – cudaDeviceSyncronize, https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__DEVICE.html#group__CUDART__DEVICE_1g10e20b05a95f638a4071a655503df25d, (20.6.2020)

[9] Cuda toolkit documentation – Bit Generation with the MTGP32 generator, https://docs.nvidia.com/cuda/curand/device-api-overview.html#bit-generation-2, (20.6.2020)

[10] Cuda toolkit documentation – Warp Shuffle functions, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#warp-shuffle-functions, (20.6.2020)

[11] Intel® Core™ i7-9750H Processor, https://ark.intel.com/content/www/us/en/ark/products/191045/intel-core-i7-9750h-processor-12m-cache-up-to-4-50-ghz.html, (20.6.2020)

[12] GEFORCE GTX 1650 - https://www.nvidia.com/en-eu/geforce/graphics-cards/gtx-1650/, (20.6.2020)