Tautvydas Petkus

Kauno Technologijos Universitetas

IFF-1

2013-12-14

Lygiagretaus programavimo Namų darbo ataskaita   
“Game of Life”

* 1. Užduotis

Užduoties tikslas – realizuoti matematiko John Horton Conway 1970m. aprašyto ląstelinio automato veikimą ir realizavimą, kai kiekviena algoritmo ląstelė yra lygiagretus procesas. Stebėti algoritmo veikimą ir priimti išvadas apie šio algoritmo racionalumą lygiagrečiuose procesuose bei patvirtinti arba paneigti išvadą, jog šis algoritmas yra racionalus, jeigu yra realizuotas lygiagrečiai

* 1. Užduoties analizė

“Game of Life” (gyvenimo žaidimas), yra algoritmas, paremtas tam tikromis taisyklėmis.

Algoritme tiriama būtybių populiacijos kaita einant laikui. Kiekvienas matricos taškas yra būtybė, kuri, einant laikui, gali išgyventi, arba mirti. Kiekvieno matricos taško kitimo taisyklės:

1. Jeigu tuščias matricos langelis turi tris kaimynus – jis užsipildo (atsiranda gyvybė)
2. Jeigu pilnas langelis turi vieną arba nei vieno kaimyno gretimais – šis langelis žūsta
3. Jeigu pilnas langelis turi 4 arba daugiau kaimynų gretimais – šis langelis žūsta
4. Jeigu pilnas langelis turi 2 arba 3 kaimynus gretimais – jis išgyvena

Šis ciklas yra nesibaigiantis – populiacija gali kisti amžinai. Taip pat gali būti atvejų, kai populiacija kinta cikliškai ir amžinai.

Sprendimo metodas

Šiai užduočiai realizuoti pasirinktos dvi platformos algoritmo realizavimui. CUDA ir OpenMP. Užduočiai realizuoti pasirinktos buvo 2 platformos būtent todėl, jog algoritmus realizavus, būtų galima rezultatus palyginti ir priimti išvadas, kuri platforma yra geriausia. Taip pat, šiam algoritmui reikia kuo didesnio skaičiaus procesų, todėl CUDA ir buvo pasirinkta viena iš technologijų, nes ji palaiko tiek procesų, kiek tų gali palaikyti vaizdo plokštė. Mano darbo kompiuterio vaizdo plokštė: Nvidia GeForce GT 525M 1024MB 900MHz

Algoritmo vizualinė dalis bus atlikta naudojant komandinę eilutę. Pati programa bus irgi paleidžiama tiktais sukompiliavus programą ir paleidus per komandinę eilutę. Pradinės algoritmo populiacijos duomenys bus generuojami atsitiktinai. Tiriama duomenų matrica: 50x50.

* 1. Programos aprašymas

Kadangi tiek CUDA, tiek OpenMP platformos buvo realizuotos naudojant C, programoje nenaudotos klasių struktūros. Žemiau aprašyti programos metodai su paaiškinimais

CUDA:

struct bool\_data kadangi CUDA nepalaiko dvimačių masyvų keliant duomenis iš CPU į GPU, buvo naudojamas struct tipo duomenų struktūra.

void evolution Šis procesas buvo naudojamas nuosekliam evoluciniam modeliui atlikti. Jame kiekvienam matricos langeliui yra įvykdomas žaidimo algoritmas.

void print šiame metode yra atspausdinama visa matrica į konsolę.

bool generation\_pass šis metodas naudojamas patikrinti visus matricos langelio gretutinius langelius. Juos įvertinus grąžinama reikšmė – arba langelis gyvena, arba miršta

\_\_device\_\_ bool generation\_pass\_paralell šis metodas daro tą patį kaip ir ankstesnis metodas, tik yra pritaikytas lygiagretiems procesams CUDA. Kadangi Cuda nepriima dvimačių masyvų iš CPU, šiame metode naudojamas ne dvimatis bool masyvas, bet struct tipo masyvas, kuriame yra bool masyvas

int check\_neighbour šis metodas skirtas patikrinti gretimam langeliui. Jeigu jisai gyvas, gražinamas 1. Priešingu atveju grąžinamas 0

cudaError\_t cudaEvolution – pagrindinis CUDA veikimo elementas, perkeliantis duomenis iš CPU į GPU ir atvirkščiai

\_\_device\_\_ int check\_neighbour\_paralell – toks pats metodas kaip ir int check\_neighbour, tik naudojams GPU (CPU procesai negali būti naudojami GPU)

\_\_global\_\_ void startEvolution – metodas, skirtas įvykdyti populiacijos pakitimo žingsnį vienam matricos taškui. Nauojamas lygiagrečiuose procesuose. Šiame metode gaunamas proceso bloko numeris bei bloko proceso numeris (taip galime sakyti sudarome procesų matricą, su kuria galime paskirti kiekvieną procesą kiekvienam populiacijos elementui)

\_\_global\_\_ void startEvolution10 – analogiškas ankstesniam procesui, tik vietoj vieno pakitimo žingsnio įvykdomi 10 žingsnių kiekvienam proceso elementui.

\_\_global\_\_ void startEvolution100 – analogiškai kaip ankstesni metodai. Įvykdoma 100 žingsnių

int main() – pagrindinis programos veikimo metodas. Jame sugeneruojama populiacija atsitiktine tvarka.

const int MAX\_ROW = 20;

const int MAX\_COL = 20;

Šios konstantos nurodo, kokio dydžio bus mūsų populiacija. Pirmoji konstanta nurodo matricos ilgį, antroji – plotį.

OpenMP:

void evolution\_paralell(int (\*pop)[MAX\_COL]); - funkcija, kurios metu sukuriamos gijos ir jos įvykdomos. Kiekvienas procesas skirtas vienam matricos langeliui

void evolution\_paralell\_chaos(int (\*pop)[MAX\_COL]); - veikimas toks pats kaip ir ankstesnės funkcijos, tik padaromos 100 iteracijų vietoj vienos

Programos kodas:

OpenMP:

//IFF-1 Tautvydas Petkus

//L-ND - OpenMP

//Dabartiniai nustatymai: stulpelių skaičius - 100, eilučių skaičius - 100

#include <omp.h>

#include <time.h>

#include <stdlib.h>

#include <stdio.h>

const int MAX\_ROW = 50; //MAX 108

const int MAX\_COL = 50; //MAX 330

void evolution(int (\*pop)[MAX\_COL], int generation);

void evolution\_paralell(int (\*pop)[MAX\_COL]);

void evolution\_paralell\_chaos(int (\*pop)[MAX\_COL]);

void print(int (\*pop)[MAX\_COL], int generation);

int generation\_pass(int (\*pop)[MAX\_COL], int i, int j);

int check\_neighbour(int pop);

int main(int argc, char \*\*argv)

{

char ch;

int population[MAX\_ROW][MAX\_COL]; //C nepalaiko bool. Reikia naudoti int

srand(time(NULL));

int i = 0; //Sugeneruojama atsitiktinė populiacijos matrica

for (i = 0; i < MAX\_ROW; i++)

{

int j = 0;

for (j = 0; j < MAX\_COL; j++)

{

if (rand() % 2)

{

population[i][j] = 0;

}

else

{

population[i][j] = 1;

}

}

}

int generation = 0;

print(population, generation); //atspausdinama pradinė populiacija

printf("Press 1 for sequental game, press 2 parallel game, press 3 for chaotic game\n"); //pasirenkamas simuliacijos rėžimas

ch = getchar();

printf("Press any key to continue...");

getchar(); getchar();

i = 0;

if (ch == '1')

{

while (i == 0)

{

evolution(population, generation); //Nuoseklus algoritmo veikimas

generation += 1;

}

}

else if (ch == '2')

{

while (i == 0)

{

evolution\_paralell(population); //Lygiagretus algoritmo veikimas (1 iteracija per procesą)

generation += 1;

print(population, generation);

}

}

else if (ch == '3') //Lygiagretus algoritmo veikimas ( 10 iteracijų per procesą)

{

while (i == 0)

{

evolution\_paralell\_chaos(population);

generation += 100;

print(population, generation);

}

}

return 0;

}

void evolution(int (\*pop)[MAX\_COL], int generation)

{

int new\_pop[MAX\_ROW][MAX\_COL];

int i = 0;

for (i = 0; i < MAX\_ROW; i++)

{

int j = 0;

for (j = 0; j < MAX\_COL; j++) //Einama per visus matricos taškus

{

new\_pop[i][j] = generation\_pass(pop, i, j); //keičiama langelio būsena. reikia saugoti į naują masyvą

}

}

memcpy(pop,new\_pop, MAX\_ROW\*MAX\_COL\*sizeof(int)); //išsaugomas naujas masyvas ant senojo

print(pop, generation);

}

void evolution\_paralell(int (\*pop)[MAX\_COL])

{

int gijosNr = omp\_get\_thread\_num();

int i = 0;

omp\_set\_num\_threads(MAX\_ROW \* MAX\_COL); //Nustatoma tiek gijų, kiek yra langelių matricoje

#pragma omp parallel private(gijosNr)

{

gijosNr = omp\_get\_thread\_num();

int row = gijosNr / MAX\_ROW;

int col = gijosNr - MAX\_ROW \* row; //surandame matricos eilę ir stulpelį pagal proceso numerį

pop[row][col] = generation\_pass(pop, row, col); //Pakeičiame matricos būseną

}

}

void evolution\_paralell\_chaos(int (\*pop)[MAX\_COL])

{

int gijosNr = omp\_get\_thread\_num();

int i = 0;

omp\_set\_num\_threads(MAX\_ROW \* MAX\_COL);

#pragma omp parallel private(gijosNr)

{

gijosNr = omp\_get\_thread\_num();

int row = gijosNr / MAX\_ROW;

int col = gijosNr - MAX\_ROW \* row;

for (i = 0; i < 100; i++) //100 iteracijų

{

pop[row][col] = generation\_pass(pop, row, col);

}

}

}

void print(int (\*pop)[MAX\_COL], int generation)

{

system("clear");

char row[MAX\_ROW\*MAX\_COL+MAX\_ROW];

printf("Generation: %5d\n", generation);

int offset = 0;

int i = 0;

for (i = 0; i < MAX\_ROW; i++)

{

int j = 0;

for (j = 0; j < MAX\_COL; j++)

{

if (pop[i][j] == 1)

{

row[offset] = '0'; //simbolis "0" matosi geriausiai konsolės lange. Todėl jis ir yra naudojamas pavaizduoti matricos užpildytus langelius

}

else{

row[offset] = ' ';

}

offset += 1;

}

row[offset] = '\n';

offset += 1;

}

printf("%s\n", row);

}

int generation\_pass(int (\*pop)[MAX\_COL], int i, int j) //algoritmo logika. Tikriname, kaimynus. Jeigu jie egzistuoja, pridedame skaičius prie counter elemento

{

int counter = 0;

if (i != 0) {counter += check\_neighbour(pop[i-1][j]); }

if (i != 0 && j != MAX\_COL - 1) {counter += check\_neighbour(pop[i-1][j + 1]); }

if (j != MAX\_COL - 1) {counter += check\_neighbour(pop[i][j + 1]); }

if (i != MAX\_ROW - 1 && j != MAX\_COL - 1) {counter += check\_neighbour(pop[i + 1][j + 1]); }

if (i != MAX\_ROW - 1) {counter += check\_neighbour(pop[i + 1][j]); }

if (i != MAX\_ROW - 1 && j != 0) {counter += check\_neighbour(pop[i+1][j-1]); }

if (j != 0) {counter += check\_neighbour(pop[i][j-1]); }

if (i != 0 && j != 0) {counter += check\_neighbour(pop[i-1][j-1]); }

if (pop[i][j])

{

if (counter < 2) //Jei kaimynų mažai, langelis miršta

{

return 0;

}

else if (counter > 3) //Jeigu per daug, irgi miršta

{

return 0;

}

else //priešingais atvejais išgyvena

{

return 1;

}

}

else

{

if (counter == 3) //Jeigu 3 kaimynai - langelis užsipildo

{

return 1;

}

}

return 0;

}

int check\_neighbour(int pop)

{

if (pop == 1)

{

return 1;

}

else

{

return 0;

}

}

CUDA:

//IFF-1 Tautvydas Petkus

//L-ND - CUDA

//Failo dydis - 100x100 duomenų

//Dabartiniai nustatymai: stulpelių skaičius - 100, eilučių skaičius - 100

#include "cuda\_runtime.h"

#include "device\_launch\_parameters.h"

#include <time.h>

#include <stdlib.h>

#include <cuda.h>

#include <stdio.h>

#include <iostream>

#include <fstream>

#include <sstream>

#include <thrust/host\_vector.h>

#include <thrust/device\_vector.h>

using namespace std;

const int MAX\_ROW = 50; //MAX 108

const int MAX\_COL = 50; //MAX 330

struct bool\_data{

bool data[MAX\_COL];

};

void evolution(bool (\*pop)[MAX\_COL], int generation);

void print(bool (\*pop)[MAX\_COL], int generation);

bool generation\_pass(bool (\*pop)[MAX\_COL], int i, int j);

bool generation\_pass\_paralell(bool\_data \*pop, int i, int j);

int check\_neighbour(bool pop);

cudaError\_t cudaEvolution(bool (\*pop)[MAX\_COL], int generation, char mode);

\_\_device\_\_ int check\_neighbour\_paralell(bool pop)

{

if (pop)

{

return 1;

}

else

{

return 0;

}

}

\_\_device\_\_ bool generation\_pass\_paralell(bool\_data \*pop, int i, int j)

{

int counter = 0;

if (i != 0) {counter += check\_neighbour\_paralell(pop[i-1].data[j]); }

if (i != 0 && j != MAX\_COL - 1) {counter += check\_neighbour\_paralell(pop[i-1].data[j + 1]); }

if (j != MAX\_COL - 1) {counter += check\_neighbour\_paralell(pop[i].data[j + 1]); }

if (i != MAX\_ROW - 1 && j != MAX\_COL - 1) {counter += check\_neighbour\_paralell(pop[i + 1].data[j + 1]); }

if (i != MAX\_ROW - 1) {counter += check\_neighbour\_paralell(pop[i + 1].data[j]); }

if (i != MAX\_ROW - 1 && j != 0) {counter += check\_neighbour\_paralell(pop[i+1].data[j-1]); }

if (j != 0) {counter += check\_neighbour\_paralell(pop[i].data[j-1]); }

if (i != 0 && j != 0) {counter += check\_neighbour\_paralell(pop[i-1].data[j-1]); }

if (pop[i].data[j])

{

if (counter < 2)

{

return false;

}

else if (counter > 3)

{

return false;

}

else

{

return true;

}

}

else

{

if (counter == 3)

{

return true;

}

}

return false;

}

\_\_device\_\_ void print\_paralell(bool\_data \*pop)

{

//system("cls");

char row[MAX\_ROW\*MAX\_COL+MAX\_ROW];

int offset = 0;

for (int i = 0; i < MAX\_ROW; i++)

{

for (int j = 0; j < MAX\_COL; j++)

{

if (pop[i].data[j])

{

row[offset] = '0';

}

else{

row[offset] = ' ';

}

offset += 1;

}

row[offset] = '\n';

offset += 1;

}

//printf("%s\n", row);

}

\_\_global\_\_ void startEvolution(bool\_data \*p, bool\_data \*new\_p){ //1 iteracija

int row = blockIdx.x;

int col = threadIdx.x;

new\_p[row].data[col] = generation\_pass\_paralell(p, row, col);

}

\_\_global\_\_ void startEvolutionCHAOS(bool\_data \*p, bool\_data \*new\_p){ //100 iteraciju

int row = blockIdx.x;

int col = threadIdx.x;

for (int i = 0; i < 100; i++)

{

new\_p[row].data[col] = generation\_pass\_paralell(p, row, col);

}

}

int main()

{

char ch;

bool population[MAX\_ROW][MAX\_COL];

printf("Press 1 for random generator, press 2 for file input...\n");

ch = fgetchar();

if (ch == '1')

{

srand(time(NULL));

for (int i = 0; i < MAX\_ROW; i++)

{

for (int j = 0; j < MAX\_COL; j++)

{

if (rand() % 2)

{

population[i][j] = false;

}

else

{

population[i][j] = true;

}

}

}

}

else if (ch == '2')

{

ifstream in("PetkusT.txt");

for(string line; getline(in, line);){

for (int i = 0; i < MAX\_ROW; i++)

{

getline(in, line);

if (line != ""){

stringstream ss;

ss << line;

int k = 0;

for (int j = 0; j < MAX\_COL; j++)

{

ss >> k;

if (k == 1){

population[i][j] = true;

}

else{

population[i][j] = false;

}

}

}

}

}

}

int generation = 0;

print(population, generation);

printf("Press 1 for sequental game, press 2 parallel game, press 3 for CHAOS mode...\n");

ch = fgetchar(); ch = fgetchar();

printf("Press any key to continue...");

getchar(); getchar();

if (ch == '1')

{

while (true)

{

evolution(population, generation); //Nuosekliai

generation += 1;

}

}

else if (ch == '2' || ch == '3')

{

while (true)

{

cudaError\_t cudaStatus = cudaEvolution(population, generation, ch); //Ivykdome funkcija, kurioje algoritmas bus atliekamas lygiagreciai

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaEvolution failed!");

fgetchar();

return 1;

}

cudaStatus = cudaDeviceReset();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaDeviceReset failed!");

return 1;

}

generation += 1;

}

}

return 0;

}

void evolution(bool (\*pop)[MAX\_COL], int generation)

{

bool new\_pop[MAX\_ROW][MAX\_COL];

for (int i = 0; i < MAX\_ROW; i++)

{

for (int j = 0; j < MAX\_COL; j++)

{

new\_pop[i][j] = generation\_pass(pop, i, j);

}

}

memcpy(pop,new\_pop, MAX\_ROW\*MAX\_COL\*sizeof(bool));

print(pop, generation);

}

void print(bool (\*pop)[MAX\_COL], int generation)

{

system("cls");

char row[MAX\_ROW\*MAX\_COL+MAX\_ROW];

printf("Generation: %5d\n", generation);

int offset = 0;

for (int i = 0; i < MAX\_ROW; i++)

{

for (int j = 0; j < MAX\_COL; j++)

{

if (pop[i][j])

{

row[offset] = '0';

}

else{

row[offset] = ' ';

}

offset += 1;

}

row[offset] = '\n';

offset += 1;

}

printf("%s\n", row);

}

bool generation\_pass(bool (\*pop)[MAX\_COL], int i, int j)

{

int counter = 0;

if (i != 0) {counter += check\_neighbour(pop[i-1][j]); }

if (i != 0 && j != MAX\_COL - 1) {counter += check\_neighbour(pop[i-1][j + 1]); }

if (j != MAX\_COL - 1) {counter += check\_neighbour(pop[i][j + 1]); }

if (i != MAX\_ROW - 1 && j != MAX\_COL - 1) {counter += check\_neighbour(pop[i + 1][j + 1]); }

if (i != MAX\_ROW - 1) {counter += check\_neighbour(pop[i + 1][j]); }

if (i != MAX\_ROW - 1 && j != 0) {counter += check\_neighbour(pop[i+1][j-1]); }

if (j != 0) {counter += check\_neighbour(pop[i][j-1]); }

if (i != 0 && j != 0) {counter += check\_neighbour(pop[i-1][j-1]); }

if (pop[i][j])

{

if (counter < 2)

{

return false;

}

else if (counter > 3)

{

return false;

}

else

{

return true;

}

}

else

{

if (counter == 3)

{

return true;

}

}

return false;

}

int check\_neighbour(bool pop)

{

if (pop)

{

return 1;

}

else

{

return 0;

}

}

cudaError\_t cudaEvolution(bool (\*pop)[MAX\_COL], int generation, char mode)

{

struct bool\_data p[MAX\_ROW];

struct bool\_data new\_p[MAX\_ROW];

for (int i = 0; i < MAX\_ROW; i++)

{

for (int j = 0; j < MAX\_COL; j++)

{

p[i].data[j] = pop[i][j];

}

}

struct bool\_data \*dev\_p;

struct bool\_data \*dev\_new\_p;

cudaError\_t cudaStatus;

cudaStatus = cudaSetDevice(0);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");

goto Error;

}

cudaStatus = cudaMalloc((void\*\*)&dev\_p, MAX\_ROW \* sizeof(bool\_data)); //skiriame atminti tiek pradinei matricai

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMalloc failed!");

goto Error;

}

cudaStatus = cudaMalloc((void\*\*)&dev\_new\_p, MAX\_ROW \* sizeof(bool\_data)); //tiek naujai matricai

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMalloc failed!");

goto Error;

}

cudaStatus = cudaMemcpy(dev\_p, p, MAX\_ROW \* sizeof(bool\_data), cudaMemcpyHostToDevice); //perduodam duomenis

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

cudaStatus = cudaMemcpy(dev\_new\_p, new\_p, MAX\_ROW \* sizeof(bool\_data), cudaMemcpyHostToDevice);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

if (mode == '2')

{

startEvolution<<<MAX\_ROW, MAX\_COL>>>(dev\_p, dev\_new\_p); //Ivykdom gijas

}

if (mode == '3')

{

startEvolutionCHAOS<<<MAX\_ROW + 1, MAX\_COL>>>(dev\_p, dev\_new\_p);

}

cudaStatus = cudaGetLastError();

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "startEvolution launch failed: %s\n", cudaGetErrorString(cudaStatus));

goto Error;

}

cudaStatus = cudaMemcpy(p, dev\_p, MAX\_ROW \* sizeof(bool\_data), cudaMemcpyDeviceToHost); //Susigrazinam rezultatus

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

cudaStatus = cudaMemcpy(new\_p, dev\_new\_p, MAX\_ROW \* sizeof(bool\_data), cudaMemcpyDeviceToHost);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaMemcpy failed!");

goto Error;

}

for (int i = 0; i < MAX\_ROW; i++)

{

for (int j = 0; j < MAX\_COL; j++)

{

pop[i][j] = new\_p[i].data[j]; //perrasom is naujo pradine matrica

}

}

print(pop, generation);

Error:

cudaFree(dev\_p);

return cudaStatus;

}

* 1. Testavimas

Kadangi populiacijos matrica yra generuojama atsitiktinai, nėra apibrėžtų ir tikslių metodų, kaip ją galime ištestuoti. Vienas iš galimų variantų testavime – sužinoti sistemos apribojimus nustatant matricos ilgį ir plotį.  
CUDA:

200x200 – Sistema viekia sėkmingai

1000x1000 – matrica nesugeneruojama

500x500 – matrica sugeneruojama

800x800 – matrica nesugeneruojama

OpenMP:

100x100 – sistema veikia sėkmingai

200x200 – programa neveikia – OpenMP nepalaiko 40.000 procesų.

150X150 – programa veikia

Instaliavimas:

CUDA: Paleiskite projektą per Visual Studio 2012. Paleiskite programą.  
 Įveskite populiacijos generavimo tipą. 1 – sugeneruoti atsitiktinę populiaciją, 2 – nuskaityti populiaciją iš failo PetkusT.txt  
 Įveskite algoritmo pobūdį. 1- nuoseklus algoritmas. 2 – lygiagretus algoritmas po vieną iteraciją procesui. 3 – lygiagretus algoritmas po 100 iteracijų procesui.

OpenMP: Susikompiliuokite failą PetkusT\_ND\_OpenMP.c per komandinę eilutę komanda “gcc –fopenmp PetkusT\_ND\_OpenMP.c”

Paleiskite programą komandine eilute “./a.out”

Įveskite algoritmo pobūdį. 1- nuoseklus algoritmas. 2 – lygiagretus algoritmas po vieną iteraciją procesui. 3 – lygiagretus algoritmas po 100 iteracijų procesui.

* 1. Vykdymo laikas. Iki 100 iteracijų

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Matricos dydis | Nuoseklus | CUDA 1it. | CUDA 100it. | OpenMP 1it. | OpenMP 100it. |
| 50x50 | 0,771 | 7,397 | 0,142 | 5,32 | 0,09 |
| 100x100 | 1,553 | 9,246 | 0,174 | 19,52 | 0,36 |
| 200x200 | 4,221 | 11,527 | 0,232 | - | - |
| 300x300 | 9,102 | 16,141 | 0,25 | - | - |
| 500x500 | 23,86 | 31,119 | 0,479 | - | - |

Išvados:

Atlikus algoritmus ir analizes prieita išvada, jog nuoseklus algoritmo realizavimas yra pats racionaliausias. CUDA, atlikdama darbą lygiagrečiai, darbą atlieka 5 sekundėm ilgiau, negu nuosekliai, tačiau išlaiko algoritmo logiką. Atlikus 100 iteracijų CUDA platformoje – algoritmo logika išsikreipia ir po 100 iteracijų jokių didelių skirtumų nepastebima, nesulaukiama rezultatų, kokių tikimasi.  
 Atlikus algoritmą OpenMP platformoje – platform ne tik kad nepalaiko daugybės gijų, ji taip pat ir žymiai ilgiau atlieka darbą, negu OpenMP. Taip pat, ji iškreipia programinio kodo logiką ir yra visiškai netinkama algoritmo realizavimui.

Algoritmo realizavimas lygiagrečiai yra **neracionalus**.