Fakultet tehničkih nauka

Univerzitet u Novom Sadu

Računarski sistemi visokih performanis

Paralelizacija računanja fraktala - Mandelbrot set

Autor:
Aleksandar Ignjatijević

Indeks:
E2 12/2021

25. januar 2022.

Sažetak

Mandelbrot set predstavlja osnovu fraktalne analize. Implementacija algoritama za generisanje Mandelbrot seta, u zavisnosti od željene veličine, nivoa detalja i preciznosti može zauzeti veoma veliku količinu kompjuterskih resursa. Samom paralelizacijom algoritama i njihovim izvršavanjem na većem broju jezgara ili na GPU-u se postiže daleko bolja efikasnost pri brzini algoritama. Realizovana su računanja u paraleli sa deljenom i distribuiranom memorijom, kao i paralelno računanje na GPU-u. Za paralelno računanje sa deljenom memorijom korišćen je OpenMP, za paralelno računanje sa distribuiranom memorijom korišćena je biblioteka Open MPI, dok je za GPU korišćena CUDA. U svim slučajevima postignuto je zavidno poboljšanje vremena izvršavanja algoritma.

Sadržaj

Spisak izvornih kodova

Spisak slika

Spisak tabela

1 Uvod

Predmet ovog rada je paralelizacija algoritama za računanje Mandelbrot seta uz C programskom jeziku uz pomoć *OpenMP*, *OpenMPI* i *CUDA*.

U poglavlju broj dva biće reči o samoj geometriji fraktala, poznatim setovima fraktala, kao i o samom Mandelbrot setu fraktala. Biće detaljno opisana matematička pozacina Mandelbrot seta, kao i algoritam za njegovo računanje.

U poglavlju tri biće opisane tehnologije koje su korišćene pri implementaciji paralelnih rešenja. Biće reči o osnovama *OpenMP* i osnovnim principima na kojima se zasniva paralelno programiranje sa deljenom memorijom. Pored *OpenMP*, biće detaljno opisan *MPI*, kao i konkretna implementacija tog standarda u vidu *OpenMPI* biblioteke. Nakon toga, biće objašnjen *CUDA Toolkit* koji je korišćen za spuštanje paralelnog rešenja na GPU.

Poglavlje 4 je posvećeno detaljnijem prikazu implementiranih paralelnih rešenja u *OpenMP*, *OpenMPI* i *CUDA*.

Poglavlje 5 posvećeno je rekapitulaciji rada.

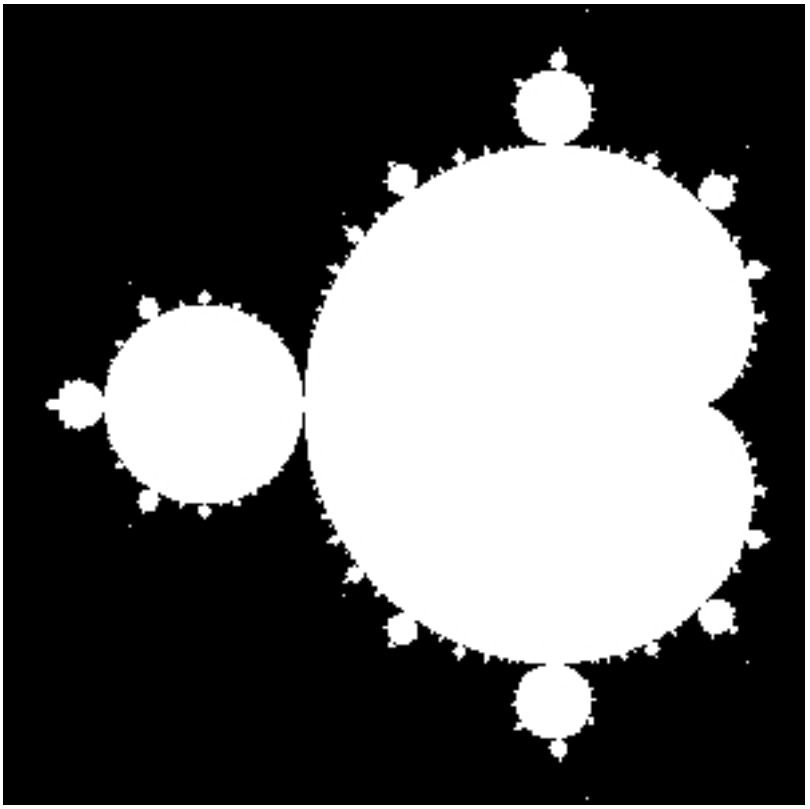
2 Geometrija fraktala

Reč "fraktal" se prvi put pojavila u radu Benoa Manderblota, nastala od latinske reči *fractus*, što znači polomljen. Mandelbrot je pokušao time da opiše objekte koji su toliko iregularni da ne mogu da pripadaju tradicionalnoj geometriji. [2]

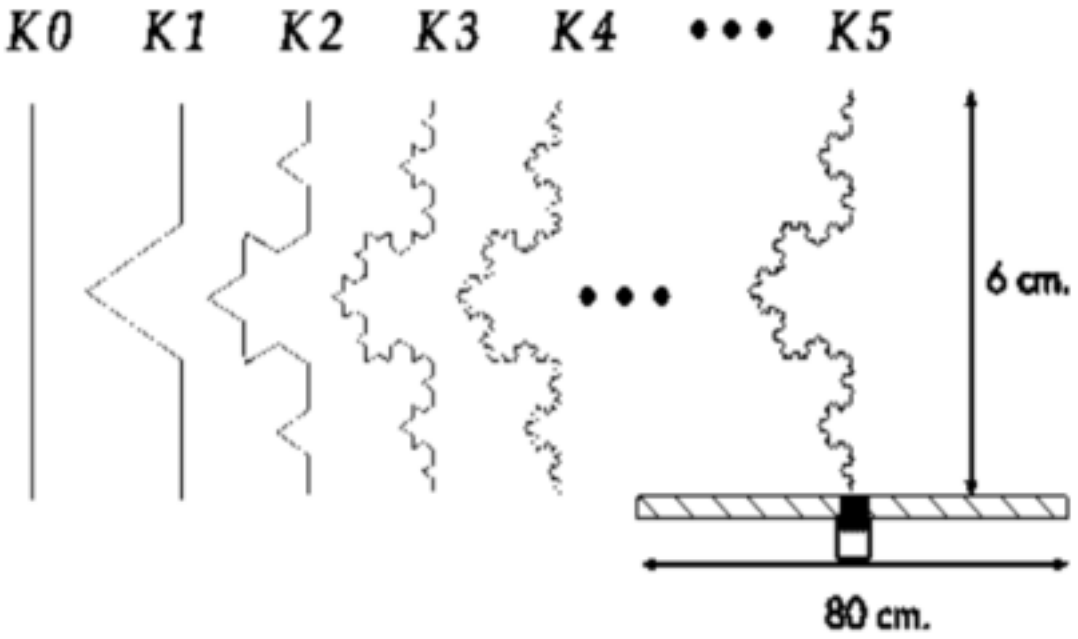
Osnovna alatka u geometriji fraktala je dimenzija. Bitno je istaći da će svaki fraktal imati veoma detaljnu i delikatnu strukturu na svim svojim nivoima skaliranja, a i veliki broj fraktala ima i određeni stepen fizičkih sličnosti sa celinom. [2]

Mandelbrot definiše fraktal kao set sa Hausdorf (eng. *Hausdorff*) dimenzijom, koja je uvek veća od topološke dimenzije. Hausdorf dimenzija predstavlja meru grubosti ili glatkoće za vremenske serije ili prostorne podatke. [3] Sa druge strane, topološka dimenzija seta je uvek prirodni broj, odnosno, nula u slučaju da je u potpunosti isključen ili jedan u slučaju da pripada proizvoljno maloj okolini isključene tačke. [2]

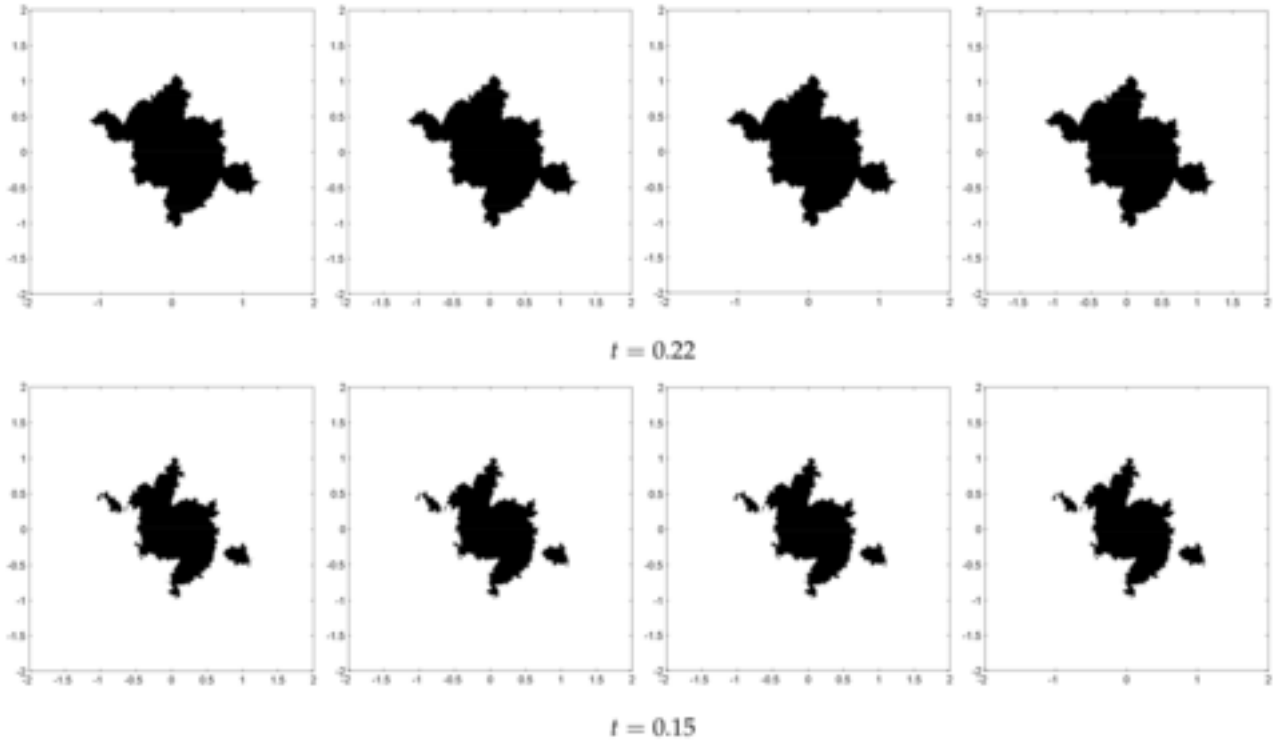
Poznatiji setovi fraktala su: Mandelbrotov set (npr. slika 1), Kochov set (npr. slika 2), Julia set (npr. slika 3) i trougao Sjerpinskog (npr. slika 4).



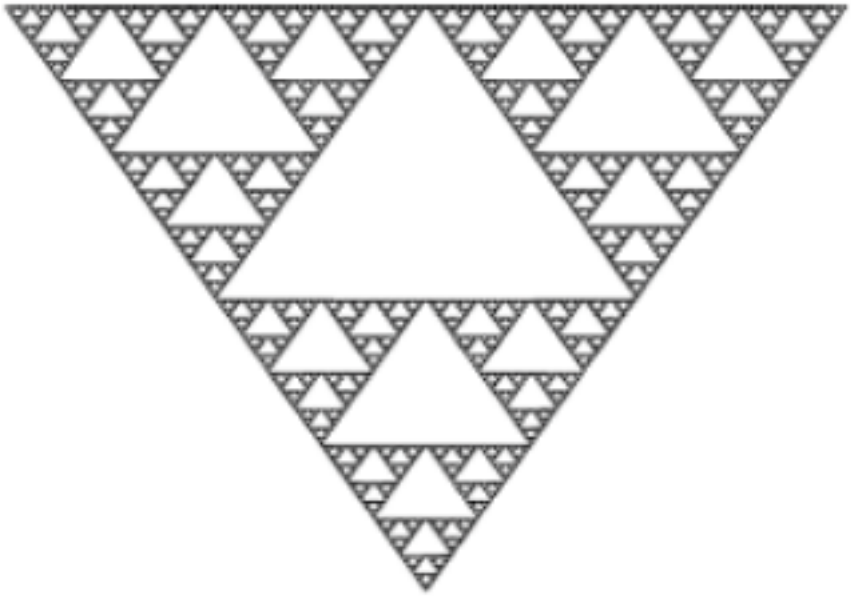
Slika 1: Prikaz Mandelbrot seta, generisan paralelnim algoritmom sa deljenom memorijom



Slika 2: Prikaz Koch seta [6]



Slika 3: Prikaz Julia seta [9]



Slika 4: Prikaz troublova Sjerpinskog seta [6]

2.1 Mandelbrot set

Mandelbrot je 1982. godine predložio M set. Početna vrednost bi bila kompleksnog broja z=0. N+1-i M set bi bio u formulaciji:[4]

$$z_{n+1} = {z_n}^2 + c$$

Postoje dva glavna principa za generisanje Manderblot fraktala. Prva metoda se zasniva na iterativnom sistemu funkcija (eng. *Iterated Function System*, skr. *IFS*), a druga na algoritmu vremenskog bekstva (eng. *Escape Time Algorithm*, skr. *ETA*).

Kako ETA algoritam koristi maksimalan broj iteracija pri izračunavanju približnih iteracionih trajektorija, daleko je pogodniji za generisanje slika kompleksnih fraktala od IFS algoritma. Svaki kompleksni broj čiji moduo nije veći od unapred određenog ograničenja i ukoliko je broj iteracija manji ili jednak sa unapred zadatim maksimalnim brojem iteracija, biva prikazan. Takođe, tim kompleksnim brojevima se dodeljuju različite broje na osnovu broja iteracija i na taj način se dobijaju veoma zanimljive figure.

3 Korišćene tehnologije

U nastavku biće reči o korišćenim tehnologijama pri implementaciji paralelnih rešenja.

3.1 Open MP

OpenMP je biblioteka za paralelno programiranje u *SMP* (eng. *symmetric multi-processors*, odnosno, *shared-memory processors*) modelu. Prilikom programiranje sa *OpenMP* bibliotekom, sve niti dele istu memoriju i podatke, iz tog razloga se *OpenMP* koristi za paralelno programiranje sa deljenom memorijom. *OpenMP* podržava implementacije u C, C++ i Fortranu. U C i C++, funkcije iz biblioteke *OpenMP* se uključuju uključivanjem zaglavlja `#include <omp.h>`.

Glavna jedinica izvršavanja u *OpenMP* je nit (eng. *thread*). Niti se mogu podeliti na glavnu nit i radilice (eng. *worker thread*). Svaka nit funkcioniše kao nezavisna prolazi kroz program i izvršava ga. Razlika između glavne i ostalih niti je u tome što glavna nit diktira početak i kraj izvršavanja, a pored toga i kontroliše ostale niti. Ovakvo ponašanje se postiže uz pomoć *OpenMP* direktiva. [8]

Kako se program izvršava paralelno putem niti, logično je zaključiti da veći broj niti automatski znači i brže izvršavanje programa. No ovo ne mora da bude uvek slučaj. Sam broj *OpenMP* niti treba da bude manji ili jednak ukupnom broju sistemskih niti kojim CPU raspolaže. Paralelizacija na većem broju niti nego što je hardverski moguće izaziva preemptivno izvršavanje, gde umesto da se dobije na brzini, zapravo se gubi na *context switching overhead-u*. [8]

OpenMP zasniva svoju implementaciju na paralelnim regionima, odnosno na *fork-join* modelu. Paralelni region je deo programa koji izvršava grupa niti. Nakon što se paralelni region završi, sve niti se ponovo priključuju glavnoj niti pomoću implicitne barijerne sinhronizacije i nastavak koda se izvršava sekvencijalno. *OpenMP* nudi i račvanje niti unutar paralelnog regiona, odnosno unutar *fork-a*. Kako je *OpenMP* baziran na modelu deljene memorije, nameće se zaključak da su i promenljive takođe deljene, odnosno da promeljivama sve niti mogu da pristupe. [8]

Za konfiguraciju sistema, *OpenMP* koristi sistemske promeljive, poput *OMP_NUM_THREADS, OMP_DYNAMIC, OMP_CANCELLATION, OMP_SCHEDULE* koje je moguće podesiti u okviru profila, u skripti koja pokreće aplikaciju ili putem komandne linije. Broju niti , odnosno vrednosti sistemske promeljive *OMP_NUM_THREADS* se pristupa pomoću funkcije *omp_get_num_threads()*.

```c
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        printf("Hello world!\n")č
    }
    return 0;
}
```

Izvorni kod 1: Primer jednostavnog *OpenMP* programa

Paralelno izvršavanje u C programskom jeziku se postiže uz pomoć `#pragma` direktiva. U listingu 1 prikazan je najjednostavniji *OpenMP* program. `#pragma omp` je početak svake *OpenMP* pragme, a nakon nje će uslediti ostale ključne reči, poput *parallel* i drugih. *Parallel* služi da izračva izvršavanje pred ulazak u blok, na onoliko niti koliko je specificirano. Ovo uključuje implicitno račvanje na početku i sinhronizaciju na kraju. *Private* služi da definiše niz promenljivih kao privatne za nit koja ih koristi. [8]

Kako *OpenMP* ima veoma labavu komunikaciju između niti, potencijalno postoji problem kada različite niti pristupaju istoj memoriji, te ona nakon toga završi u nedefinisanom stanju. Ovakvi ishodi su nepovoljni i potrebno ih je izbeći po svaku cenu. Zbog toga razvijeni su mehanizmi implicitne i eksplicitne sinhronizacije. Po pravilu je već svaki paralelni blok implicitno sinhronizovan, upravo zbog toga što sve niti moraju da sačekaju da se i ostale niti izvrše, pre nastavka. Sa druge strane, eksplicitna sinhronizacija se uvodi kroz direktive za kritični region `#pragma omp critical`, glavnu nit `#pragma omp master`, barijeru `#pragmaomp barrier` ili jednostruko izvršavanje `#pragma omp signle`. [8]

3.2 MPI

Message Passing Interface (MPI) je standard za razmenu poruka. *MPI* je namenjen upotrebi u paralelnom računarstvu. Bitno je istaći da *MPI* nije biblioteka. *MPI* predstavlja opis onoga što bi biblioteka trebala da implementira. Postoji veliki broj implementacija ovog standarda, a u ovom radu je za paralelizacij algoritma za računanje Mandelbrot seta korišćena *OpenMPI* implementacija.

Sam MPI opisuje načine razmene poruka između adresnih prostora različitih procesa. MPI se koristi u distribuiranim arhitekturama, arhitekturama sa deljenom memorijom, kao i hibridnim arhitekturama. Postoji nekoliko važnijih implementacija ovog standarda, poput gorenavedenog *OpenMPI, MVAPICH, IntelMPI*. U nastavku biće opisan OpenMPI, jer je on korišćen u implementaciji paralelnog računanja Mandelbrot seta.

3.2.1 OpenMPI

OpenMPI je *open-source* implementacija *Message Passing Interface-a*. [1] *OpenMPI* je protokol OSI nivoa 5. Kako ovu biblioteku održava konzorcijum istraživača, akademika i industrijskih partnera, *OpenMPI* predstavlja opšteprihvaćenu implementaciju *MPI* standarda. Ova biblioteka proširuje već postojeće jezike konstruktima za paralelizam i predstavlja veoma bitana alat u *High Performace Computing* svetu.

Kako je *OpenMPI* namenjen da radi na potencijalno udaljenim računarima, nije moguća komunikacija putem deljene memorije, na istom principu kao kod *OpenMP*. Shodno tome, potrebno je uvesti novi princip komunikacije. Sa time u vidu nastao je komunikator. Komunikator predstavlja kolekciju procesa, odnosno, nezavisno pokrenutih instanci programa, nešto poput radio frkvencija. [8] Minimalan broj komunikatora je jedan, odnosno, svaki *MPI* program mora posedovati makar jedan komunikator. Ovaj komunikator nije potrebno posebno kreirati, on već postoji pod imenom *MPI_COMM_WORLD*. *MPI_COMM_WORLD* predstavlja globalni kanal za komunikaciju, pa su samim time i svi procesi deo njega. Da bi bilo moguće komunicirati, potrebno je pristupiti samom komunikatoru, čiji je identifikator *MPI_Comm*.

Osnovna postavka *OpenMPI* programa je prikazana u listingu 2. Red (eng. *rank*) procesa je zapravo njegov identifikator unutar globalnog komunikatora. To je proizvoljan broj iz opsega od 0 do size-1.

Point-to-Point komunikacija predstavlja komunikaciju dva procesa između se-

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    // proizvoljan kod

    MPI_Finalize();

    return 0;
}
```

Izvorni kod 2: Primer osnovne *OpenMPI* strukture

be. Prvi proces šalje neku poruku, koju nakon toga drugi proces prihvata i obrađuje. *OpenMPI* realizuje ovaj vid komunikacije uz pomoć funkcija MPI_Send i MPI_Recv. Obe funkcije uzimaju parametre: pokazivač na poruku generičkog tipa koja se ili šalje ili prima, broj elemenata u poruci, tip poruke, red procesa kome se šalje ili od kojeg se prima, celobrojna vrednost rezervisana za proizvoljnu upotrebu i komunikator koji se koristi. [8]

Osim *Point-to-Point* komunikacije, *OpenMPI* podržava i kolektivnu komunikaciju (eng. *Collective communication*). Ovaj vid komunikacije se oslanja na funkcije MPI_Bcast, MPI_Scatter odnosno MPI_Gather i MPI_Allgather. U paralelnoj implementaciji Mandelbrot seta korišćene su funkcije za rasipanje na procese (eng. *scatter*) i prikupljanje (eng. *gather*). *MPI_Scatter* radi slično kao i *MPI_Bcast*, slučajno šalje sa jednog na više procesa. Osnovna razlika je u tome što proces koji šalje, pre slanja podeli podatke na onoliko podskupova koliko procesa ima, pa ih tek nakon toga šalje. Sa druge strane, *MPI_Gather* je obrnuta funkcija od *MPI_Scatter*. Više procesa šalje jednom procesu delove, a taj glavni proces ih onda skuplja i spaja u jednu celinu. To je moguće videti u listingu 3.

```
    MPI_Scatter(result, part_width, MPI_CHAR, partial_result, part_width,
    MPI_CHAR, 0, MPI_COMM_WORLD);

    // proizvoljan kod

    MPI_Gather(partial_result, part_width, MPI_CHAR, result, part_width,
    MPI_CHAR, 0, MPI_COMM_WORLD);
```

Izvorni kod 3: Primer upotrebe *MPI_Scatter* i *MPI_Gather* funkcija

3.3 CUDA

CUDA predstavlja model programiranja kao i platformu za paralelno računarstvo na akceleratorskim arhitekturama. Ovaj model je integrisan u skup alata koje proizvodi *Nvidia*, što uključuje osnovnu biblioteku, kao i neke reimplementacije već postojećih standarda za *HPC* u *CUDA* obliku. [7] *CUDA* radi na *Nvidia* grafičkim karticama, a na sledećem linku.

Pogodnosti koje *CUDA* pruža su nebrojene. *CUDA* inkorporira u sebe tri glavna nivoa apstrakcije: hijerarhiju grupa niti, hijerarhiju deljenih memorija i barijernu sinhronizaciju. Uz pomoć navedenih hijerarhija, *CUDA* deli proglem koji se rešava tako da je moguće razdvojiti grube paralelne zadatke i sitne parelelizme podataka i niti, koji se moraju izvršavati unutar *Streaming Multiprocessor-a*. *Streaming Multiprocessor*, odnosno kraće *SM* predstavlja osnovnu gradivnu jedinicu uređaja koji podržava *CUDA*. [7]

Veoma bitan pojam u *CUDA* programiranje je *kernel*. Kako je za *CUDA* okruženje nativan jezik C++, u njemu se među osnovnim funkcijama nalazi *kernel*. Kada se neka funkcija proglasi za *kernel*, to znači da je cilj da se ta funkcija izvršava paralelno, uz pomoć više niti. Kako se *CUDA* pokreće uz pomoć više niti, svaka instanca funkcije prilikom izvršavanja poseduje informaciju kojoj niti pripada. Ovo je omogućeno pomoću ugrađene promeljive *threadIdx*, a ona je dostupna u samom telu *kernel-a*. Vrednost *threadIdx* je zapravo vektor koji se sastoji iz *X, Y* i *Z* komponenti, zbog mapiranaj na 2D, odnosno 3D. Ukoliko se prilikom invokacije specificira 1,N, to bi značilo da je željena dimenzija 1D, gde se gleda samo *X* komponenta.[7] Za višedimenzionalnu vrednost, potrebno je da broj niti po bloku poseduje više dimenzija, a za ovo se koristi ugrađeni *CUDA* tip: *dim3*. Trenutno postoji ograničenje od 1024 niti po bloku, na modernim GPU uređajima. Pravilo je da ne sme postojati komunikacija između blokova i da oni moraju biti jednaki.

Pored *threadIdx* koji određuje koja nit trenutno izvršava kernel, dostupni su i podaci o tome u kom se bloku trenutno nalazi proces uz pomoć ugrađene *blockIdx* promeljive, kao i informacija o dimenziji trenutnog bloka putem ugrađene *blockDim* promenljive. Takođe, postoji i *gridDim* promenljiva koja nosi informacije koliko postoji ukupno blokova i kako su raspoređeni. [7]

Prilikom *CUDA* programiranja potrebno je daleko ozbiljnije voditi računa o memoriji, pa samim time se memorija deli na memoriju *host* računara, odnosno radnu memoriju na računaru na kom se pokreće *CUDA* program i na memoriju GPU-a, odnosno *device* memoriju. Nakon toga, na GPU-u se memorija deli na *on-chip* i *off-chip* memoriju. [7] *On-chip* memorija je sastavni deo samog grafičkog procesora, veoma mala, ali i veoma brza, poput keš memorije, sa razlikom što je moguće upravljati potpuno direktno. Sa druge strane, *Off-chip* memorija je memorija GPU uređaja. Za razliku od *on-chip* memorije, pristup ovoj memoriji je veoma spor i sa velikim stepenom kašnjenja. [7] Da bi *CUDA* program bio dobar, potrebno je obratiti pažnju na kritične delove. Imajući to u vidu, cilj ne minimizovati broj pristupa *off-chip* memoriji i minimizovati broj transfera između *host-a* i *device-a*.

```
1  static int compute_point( double x, double y, int max )
2  {
3          double complex z = 0;
4          double complex alpha = x + I*y;
5
6          int iter = 0;
7
8          while( cabs(z)<4 && iter < max ) {
9                  z = cpow(z,2) + alpha;
10                 iter++;
11         }
12
13         return iter;
14 }
```

Izvorni kod 4: Primer implementacije sekvencijalnog *escape time* algoritma. Link do github repozitorijuma

4 Implementacija

U nastavku poglavlja biće opisana tri principa implementacije paralelizacije računanja fraktala. Biće prikazane implementacija sekvencijalnog algoritma, implementacije paralelnih algoritama sa deljenom i distribuiranom memorijom, kao i paralelna implementacija na GPU-u. Kao što je već navedeno, za paralelizaciju algoritma za izračunavanje Mandelbrot seta sa deljenom memorijom je korišćen *OpenMP*, dok je za distribuiranu memoriju korišćen *OpenMPI*. Kao alat za paralelizaciju na GPU je korišćen *CUDA toolkit*.

4.1 Implementacija sekvencijalnog algoritma za računanje fraktala u `C` kodu

U listinzima 4 i 5 prikazane su sekvencijalne implementacije računanja Mandelbrot seta i iscrtavanje dobijene slike. Za iscrtavanje slike korišćena je biblioteka *stb_image*. Dokumentacija je dostupna na sledećem linku. Kao parametri se zadaju početne koordinate *ymin, xmin, ymax, xmax*, veličina slike, kao i maksimalan broj iteracija. Veličina slike je zadata preko parametara *width* i *height*.

```
void compute_image( double xmin, double xmax, double ymin, double ymax,
            int maxiter, int width, int height, int num_chanels, int rgb){
    int i,j;
    unsigned char buffer[width*height*num_chanels];
    char name[60];
    for(i=0;i<height;i++) {
        for(j=0;j<width;j++) {
            double x = xmin + i*(xmax-xmin)/width;
            double y = ymin + j*(ymax-ymin)/height;
            int iter = compute_point(x,y,maxiter);

            if(rgb==1){
                buffer[3*(j*height+i)] = (unsigned char)
                                        ((int)(iter* sin(iter/maxiter))%255);
                buffer[3*(j*height+i)+1] = (unsigned char) ((iter*iter)%255);
                buffer[3*(j*height+i)+2] = (unsigned char)(iter%256);
            }else{
                int gray = 255 * iter / maxiter;
                buffer[j*height+i] = (char) gray;
            }
        }
    }
    if(rgb==1){
        sprintf(name,"fractal-images/seq/rgb_img%dx%d_%d.jpg", width,
                height, maxiter);
    }else{
        sprintf(name,"fractal-images/seq/bw_img%dx%d_%d.jpg", width,
                height, maxiter);
    }
    stbi_write_jpg(name, width, height, num_chanels, buffer,  100 );
}
```

Izvorni kod 5: Primer sekvencijalnog algoritma za iscrtavanje slike Mandelbrot seta. Link do github repozitorijuma

4.2 Implementacija paralelnog algoritma sa deljenom memorijom za računanje fraktala u `C` kodu

Za implementaciju ove paralelizacije korišćen je *OpenMP*. U listingu 6 data je implementacija paralelnog algoritma sa deljenom memorijom. Korišćene su `#pragma omp parallel shared(result,maxiter) private(i,iter)` i `#pragma omp for shedule(runtime)` direktive.

`Shared` direktiva se koristi za označavanje zajedničkih resursa. Odnosno, zahvaljujući njoj, promenljive `result` i `maxiter` postaju deljene promenljive. Sa druge strane, direktiva `private`, služi da označi promenljive `i` i `iter` kao privatne. Zahvaljujući `for`, *OpenMP* zna da je u pitanju for koji se paralelizuje, dok `scheduled(runtime)` određuje način prekljućivanja niti, koji je postavljen u *OMP_SCHEDULE* sistemskoj promenljivoj.

4.3 Implementacija paralelnog algoritma sa distribuiranom memorijom za računanje fraktala u `C` kodu

Za implementaciju ove paralelizacije korišćen je *OpenMPI*. U listingu 7 data je inicijalizacija *OpenMPI* programa. Na početku se zadaju veličina i rang komunikatora. Nakon toga se distribuirano pokreće funkcija *run*. *MPI_Finalize()* označava kraj *OpenMPI* sekcije, dok *MPI_Barrier(MPI_COMM_WORLD)* služi za sinhronizaciju distribuiranih procesa.

U listinzima 8 i 9 data je implementacija paralelnog algoritma sa distribuiranom memorijom. Niz koji treba da predstavlja rezultat je podeljen na više podnizova, gde se svaki računa posebno u svom procesu. Nulti proces je odabran jer ga je lako pronaći, tako da se sa nultim procesom inicijalizuje rezultat i pokreće tajmer. Nakon toga se uz pomoć *MPI_Scatter* računanje deli na više procesa. Broj procesa određen je veličinom komunikatora. Svaki proces u sebi poseduje dodatnu paralelizaciju, uz pomoć *OpenMP* koja dodatno ubrzava računanje. Nakon što se svi procesi izvrše, uz pomoć *MPI_Gather* funkcije, rezultati se prikupljaju i sinhronizuju u jednu celinu koja predstavlja rezultat.

4.4 Implementacija paralelnog algoritma na GPU-u uz pomoć `CUDA`

Za implementaciju ove paralelizacije korišćen je *CUDA toolkit*. U listingu 12 dat je primer inicijalizacije *CUDA* programa. Funkcija *cudaMalloc* zauzima memoriju na *device-u*. Veoma je bitno obratiti pažnju da se ne pokuša zauzeti više memorije

```
void compute_image_opt( double xmin, double xmax, double ymin, double ymax,
    int maxiter, int width, int height, char* result, int threads, int num_chanels,
    int rgb){
    int i, iter;
    char name[60];
        double xstep = (xmax-xmin) / (width-1);
        double ystep = (ymax-ymin) / (height-1);

    #pragma omp parallel shared(result, maxiter) private(i,iter) num_threads(thred
    #pragma omp for schedule(runtime)
    for (i = 0; i < width*height; i++) {

                double x = xmin + (i%width)*xstep;
                double y = ymin + (i/height)*ystep;

                iter = compute_point(x,y,maxiter);
                if(rgb==1){
                    result[num_chanels*i ] = (unsigned char)
                                            ((int)(iter* sin(iter/maxiter))%255);
                    result[num_chanels*i +1] = (unsigned char)(iter%256);
                    result[num_chanels*i +2] = (unsigned char) ((iter*iter)%255);
                }
                else{
                    int gray = 255 * iter / maxiter;
                    result[i] = (unsigned char) gray;
                }
    }
    if(rgb==1){
        sprintf(name,"fractal-images/shared/rgb_img%dx%d_%d.jpg", width,
                height, maxiter);
    }else{
        sprintf(name,"fractal-images/shared/bw_img%dx%d_%d.jpg", width,
                height, maxiter);
    }
    stbi_write_jpg(name, width, height, num_chanels, buffer,  100 );
}
```

Izvorni kod 6: Primer implementacije paralelnog algoritma sa deljenom memorijom. Link do github repozitorijuma

```
1     int size, rank;
2     MPI_Init(&argc, &argv);
3     MPI_Comm_size(MPI_COMM_WORLD, &size);
4     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
5
6         run(maxiter, rank, size, width, height, xmin, xmax, ymin, ymax,
7             out_file,num_chanels, rgb);
8
9     MPI_Barrier(MPI_COMM_WORLD);
10    MPI_Finalize();
```

Izvorni kod 7: Primer inicijalizacije *OpenMPI* programa. Link do github repozitorijuma

na *device-u* nego što je zapravo dostupno. U listingu 13 dat je prikaz dela koda koji pokreće kernel i prebacuje rezultat iz memorije *device-a* na memoriju *host-a*. Kernel je ona funkcija koja se izvršava paralelno, a u ovom slučaju to je funkcija `compute_image_kernel<<<width, height>>>(xmin, xmax, ymin, ymax, max_iter, width,height, result);`

U funkciji *compute_image_kernel* izračunava se slika. Niz blokova je jednodimenzionalni niz u kome se svaka nit bavi svojim delom slike zahvaljujući promenljivoj *threadIdx*. [5] Kako se u promenljivoj *blockDim.x* nalazi broj niti po bloku, u *blockIdx.x* indeks trenutnog bloka, trenutna nit koja pokreće program se može jedinstveno identifikovati uz pomoć formule `blockDim.x * blockIdx.x + threadIdx.x`.

Konačno, u listingu se nalazi prikaz algoritma za generisanje sledećeg kompleksnog broja u Mandelbrot setu.

```
void run(int maxiter, int current_processor, int processors_amount, int width,
        int height, double xmin, double xmax, double ymin, double ymax)
{
        char* result = (char *) malloc(width*height);
    int part_width =  (width*height) / processors_amount;
    int start = current_processor * part_width;
    char* partial_result = (char *) malloc(part_width);

        clock_t begin;
        clock_t end;
    if (current_processor == 0 ) {
        result = (char *) malloc(width*height);
                printf("Timer started\n");
                begin = clock();
    }
    MPI_Scatter(result, part_width, MPI_CHAR, partial_result, part_width,
                MPI_CHAR, 0, MPI_COMM_WORLD);
        printf("Distributing work on node %d out of %d\n",
                current_processor, processors_amount);
        compute_image(xmin,xmax,ymin,ymax,maxiter,width, height, start,
                    start+part_width, partial_result, current_processor,
                    processors_amount);
        MPI_Gather(partial_result, part_width, MPI_CHAR, result, part_width,
                    MPI_CHAR, 0, MPI_COMM_WORLD);

          if (current_processor == 0) {
            end = clock();
                double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

                printf("time took for execution of distributed algorithm: %f\n",
                        time_spent);
                if(rgb==1){
                    sprintf(name,"fractal-images/distributed/rgb_img%dx%d_%d.jpg",
                    width, height, maxiter);
                }else{
                    sprintf(name,"fractal-images/distributed/bw_img%dx%d_%d.jpg",
                    width, height, maxiter);
                }
                stbi_write_jpg(name, width, height, num_chanels, result, 300);
        free(result);
    }
    free(partial_result);

}
```


Izvorni kod 8: Primer implementacije paralelnog algoritma sa deljenom memorijom. Link do github repozitorijuma

```
void compute_image( double xmin, double xmax, double ymin, double ymax,
        int maxiter, int width, int height, int start, int end, char* result,
        int rank, int size, int rbg){

        printf("Working with rank %d and size %d\n\n", rank, size);

        int i, iter;
        double xstep = (xmax-xmin) / (width-1);
        double ystep = (ymax-ymin) / (height-1);

    #pragma omp parallel for shared(result, maxiter, start, end) private(i,iter)
    for (i = start; i < end; i++) {

                double x = xmin + (i%width)*xstep;
                double y = ymin + (i/height)*ystep;

                iter = compute_point(x,y,maxiter);

                if(rgb==1){
                    result[3*(i-start) ] = (unsigned char)
                                            ((int)(iter* sin(iter/maxiter))%255);
                    result[3*(i-start) +1] = (unsigned char)(iter%256);
                    result[3*(i-start) +2] = (unsigned char) ((iter*iter)%255);
                }
                else{
                    int gray = 255 * iter / maxiter;
                    result[i-start] = (unsigned char) gray;
                }
    }

}
```

Izvorni kod 9: Primer implementacije izračunavanja vrednosti piksela u slici. Link do github repozitorijuma

```
__device__ int compute_point( double x, double y, int max )
{
        double zr = 0;
    double zi = 0;
    double zrsqr = 0;
    double zisqr = 0;

    int iter;

    for (iter = 0; iter < max; iter++){
                zi = zr * zi;
                zi += zi;
                zi += y;
                zr = (zrsqr - zisqr) + x;
                zrsqr = zr * zr;
                zisqr = zi * zi;

                if (zrsqr + zisqr >= 4.0) break;
    }

    return iter;
}
```

Izvorni kod 10: Primer implementacije *escape time* alogritma u *CUDA*. Link do github repozitorijuma

```
__global__ void compute_image_kernel(double xmin, double xmax, double ymin,
double ymax, int maxiter, int width, int height, char* result,int num_chanels){
    int pix_per_thread = width * height / (gridDim.x * blockDim.x);
    int tId = blockDim.x * blockIdx.x + threadIdx.x;
    int offset = pix_per_thread * tId;
    int iter;
    double xstep = (xmax-xmin) / (width-1);
    double ystep = (ymax-ymin) / (height-1);
    for (int i = offset; i < offset + pix_per_thread; i++){
        int iw = i%width;
        int ih = i/height;
        double x = xmin + iw*xstep;
        double y = ymin + ih*ystep;
        iter = compute_point(x, y, maxiter);
        if(num_chanels>1){
            result[num_chanels*(ih*width+iw)]=(char)
                                                ((int)(iter * iter/maxiter)%255);
            result[num_chanels*(ih*width+iw) + 1]=(char)(iter%256);
            result[num_chanels*(ih*width+iw) + 2]=(char) ((iter*iter)%255);
        }else{
            int gray = 255 * iter / maxiter;
            result[ih * width + iw]  = (char)gray;
        }
    }
    if (gridDim.x * blockDim.x * pix_per_thread < width * height &&
        tId < (width * height) - (blockDim.x * gridDim.x)){

        int i = blockDim.x * gridDim.x * pix_per_thread + tId;
        int iw = i%width;
        int ih = i/height;
        double x = xmin + iw*xstep;
        double y = ymin + ih*ystep;
        iter = compute_point(x, y, maxiter);
        if(num_chanels>1){
            result[num_chanels*(ih*width+iw)]=(char)
                                                ((int)(iter * iter/maxiter)%255);
            result[num_chanels*(ih*width+iw) + 1]=(char)(iter%256);
            result[num_chanels*(ih*width+iw) + 2]=(char) ((iter*iter)%255);
        }else{
            int gray = 255 * iter / maxiter;
            result[ih * width + iw]  = (char)gray;
        }
    }
}
```


Izvorni kod 11: Primer *CUDA* kernela. Link do github repozitorijuma

```
    cudaError_t err = cudaSuccess;

    char *result = NULL;
    err = cudaMalloc(&result, width*height*num_of_chanels*sizeof(char));
    checkErr(err, "Failed to allocate result memory on gpu");

    run(xmin, xmax, ymin, ymax, width, height, max_iter, result,rgb);

    cudaFree(result);
```

Izvorni kod 12: Primer inicijalizacije *CUDA* programa. Link do github repozitorijuma

5 Komparativna analiza dobijenih rezultata

U nastavku će biti data komparativna analiza vremena izvršavanja nakon različitih paralelnih implementacija. Logično je zaključiti da je sekvencijalno izvršavanje najsporije, a kako je reč o obradi slike, *CUDA* prednjači u brzini izvršavanja.

5.1 Analiza dobijenih rezultata u zavisnosti od rezolucije

Kao što je moguće videti iz tabele 1, kada se različite implementacije pokreću pri maksimalnom broju iteracija od 5000, dolazi do zaključka da sva paralelna rešenja nude znatno ubrzanje sekvencijalnog algoritma. Ograničenje u ovoj analizi predstavlja memorija koju je moguće koristiti za izvršavnaje *CUDA* programa, pa stoga je analiza ograničena na rezoluciju 1024x1024. Iz tabele 1 vidi se da na manjim rezolucijama, *OpenMP* i *OpenMPI* imaju slične rezultate, čak je i *OpenMP* brži za nijansu. Ali kako se pređe preko rezolucije od 512x512, *OpenMPI* počinje da dominira, da bi pri maksimalnoj rezoluciji korišćenoj u analizi bio čak za četvrtinu brži od *OpenMP-a*. *CUDA* implementacija je daleko najbolje rešenje ukoliko GPU poseduje dovoljno memorije.

```
__host__ static void run(double xmin, double xmax, double ymin, double ymax,
                int width, int height, int max_iter, char* result, int rgb)
{
    int num_of_chanels;

    if(rgb==1){
        num_of_chanels = 3;
    }else{
        num_of_chanels=1;
    }
    dim3 numBlocks(width*num_of_chanels,height);
    cudaError_t err = cudaSuccess;
    compute_image_kernel<<<width*num_of_chanels, height>>>(xmin, xmax, ymin, ymax,
                                max_iter, width, height, result);
    checkErr(err, "Failed to run Kernel");
    void *data = malloc(height * width * num_of_chanels * sizeof(char));
    err = cudaMemcpy(data, result, width * height * num_of_chanels * sizeof(char),
                    cudaMemcpyDeviceToHost);
    checkErr(err, "Failed to copy result back");

    char name[60];
    if(rgb==0){
            sprintf(name,"fractal-images/cuda/bw_img%dx%d_%d.jpg", width, height,
                    max_iter);
    }else{
            sprintf(name,"fractal-images/cuda/rgb_img%dx%d_%d.jpg", width, height,
                    max_iter);
    }
    stbi_write_jpg(name, width, height, num_of_chanels, data, 200);

}
```

Izvorni kod 13: Primer dela *CUDA* koda koji pokreće kernel i iscrtava sliku. Link do github repozitorijuma

rezolucija	sekvencijalno	OpenMP	OpenMPI	CUDA
128x128	2.772106	0.373146	0.449109	0.102358
256x256	10.802276	1.629261	1.900149	0.131312
512x512	42.910819	6.238660	4.684450	0.244554
1024x1024	169.526222	27.375434	20.825298	0.600570

Tabela 1: Rezultati dobijeni nakon pokretanja algoritama sa 5000 iteracija i različitim rezolucijama crno bele slike

5.2 Analiza dobijenih rezultata u zavisnosti od maksimalnog broja iteracija

Kao što je moguće videti iz tabele 2, kada se sa različitim brojem iteracija pokreće algoritam, pri rezoluciji 1024x1024px, dolazi se do zaključka da sva paralelna rešenja nude znatna ubrzanja pri računanju. Najbolje rešenje je ponovo implementirano u *CUDA*. Isto kao i u prethodnom slučaju, *OpenMP* ima blagu prednost u odnosu na *OpenMPI* do broja maksimalnog broja iteracija od 100. Nakon toga, prednost ima *OpenMPI* i to se najbolje može videti pri maksimalnom broju iteracija od 25000, kada je implementirano rešenje u *OpenMPI* za trećinu bolje.

iteracije	sekvencijalno	OpenMP	OpenMPI	CUDA
10	0.916775	0.186156	0.342896	0.135440
20	1.447436	0.271119	0.430123	0.130254
50	2.608455	0.447061	0.525623	0.138004
100	4.556283	0.691048	0.740468	0.139495
500	18.800238	2.556585	2.114096	0.202363
1000	36.004139	5.342406	3.982649	0.245656
5000	173.567248	25.337006	18.228549	0.618199
10000	343.800713	49.603081	41.930541	1.087092
25000	853.083032	121.834386	85.864986	2.495231

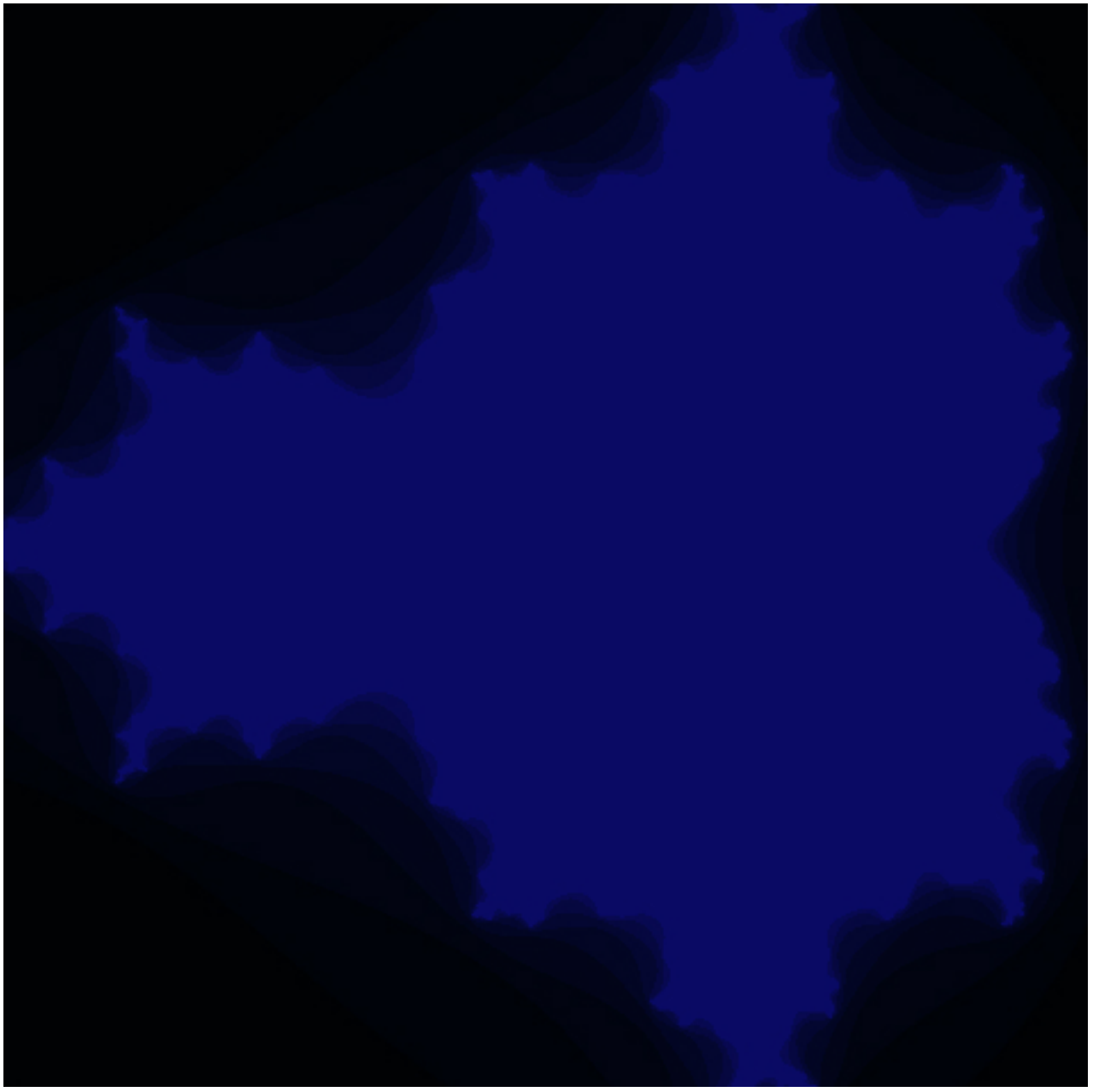
Tabela 2: Rezultati pri rezoluciji 1024x1024 dobijeni nakon pokretanja algoritama sa različitim brojem iteracija

6 Zaključak

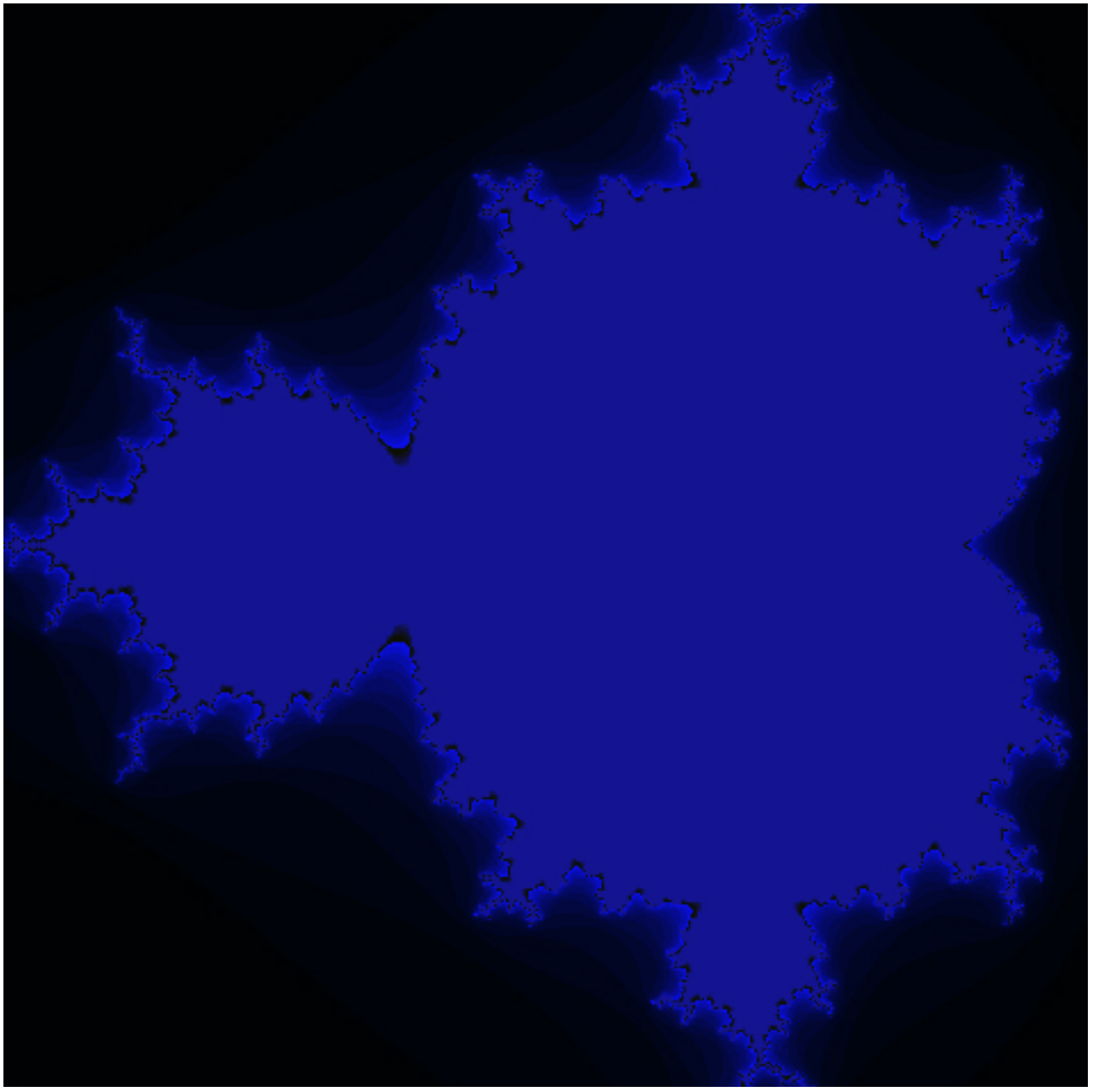
U radu je obrađena paralelizacija algoritma za generisanje Mandelbrot seta. Za paralelizaciju su korišćeni `C` programski jezik i *OpenMP*, *OpenMPI* i *CUDA*. Svim paralelizacijama se dobija znatno ubrzanje. *CUDA* nudi najbolja vremena izvršavanja, ali sa time dolazi i ograničenje u pogledu veličine slike. Naime, kao što je navedeno u poglavlju 5, na *NVIDIA GeForce GTX 1650* grafičkom procesoru, crno bele slike Mandelbrot seta je moguće generisati u maksimalnoj rezoluciji od 1024x1024, dok kada je reč o *RGB* slikama maksimalna rezolucija je 482x482. Kako *CUDA* rešenje predstavlja najbolju implementaciju, ukoliko bi račun bio izvršavan na boljem GPU-u, *CUDA* implementacija bi predstavljala veoma moćan alat sa kojim bi se veoma zanimljive slike mogle generisati.

U dodatku su prikazane slike generisane paralelnim algoritmom implementiranim u *CUDA*.

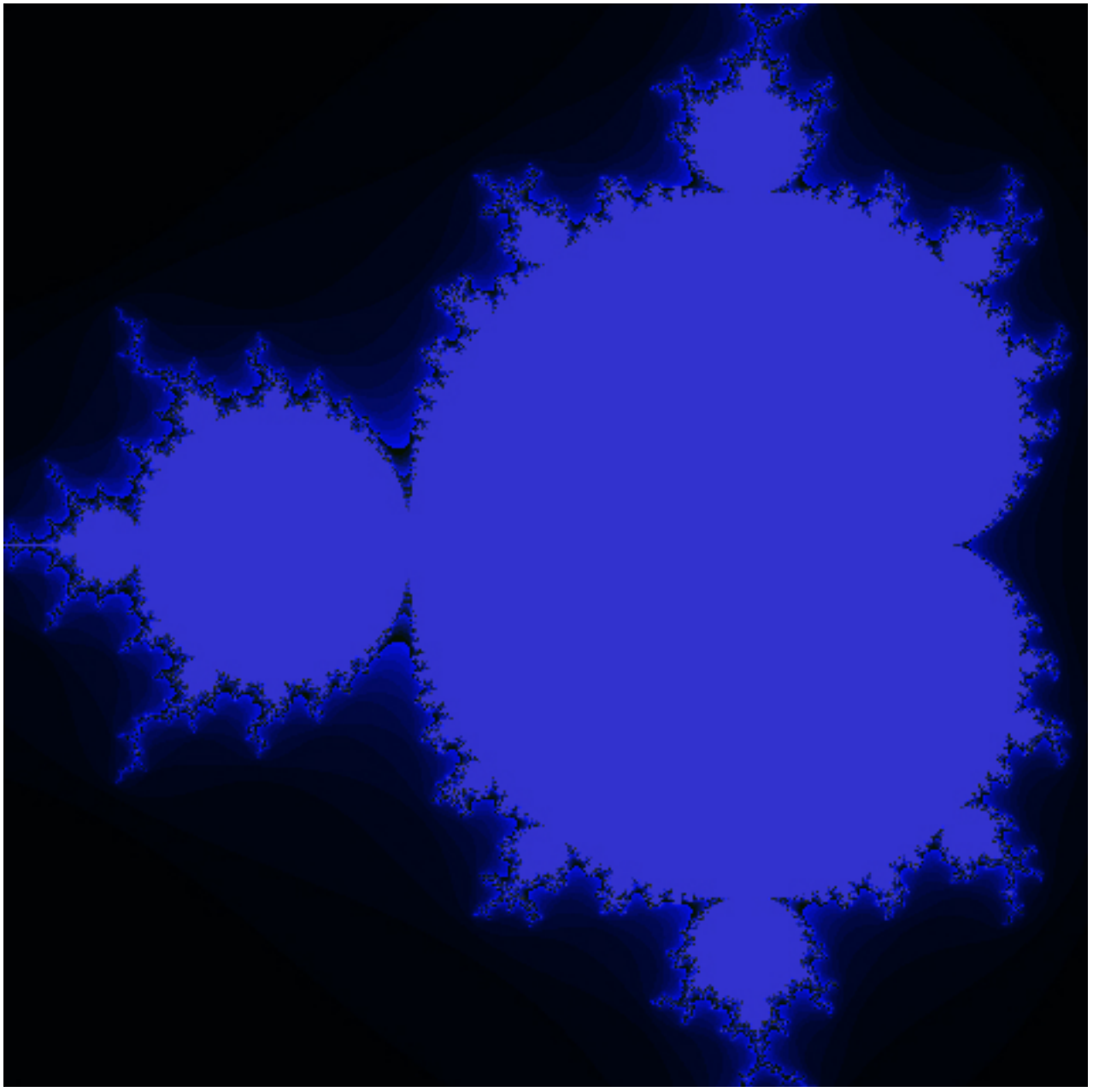
A Dodatak - Generisani RGB Mandelbrot setovi



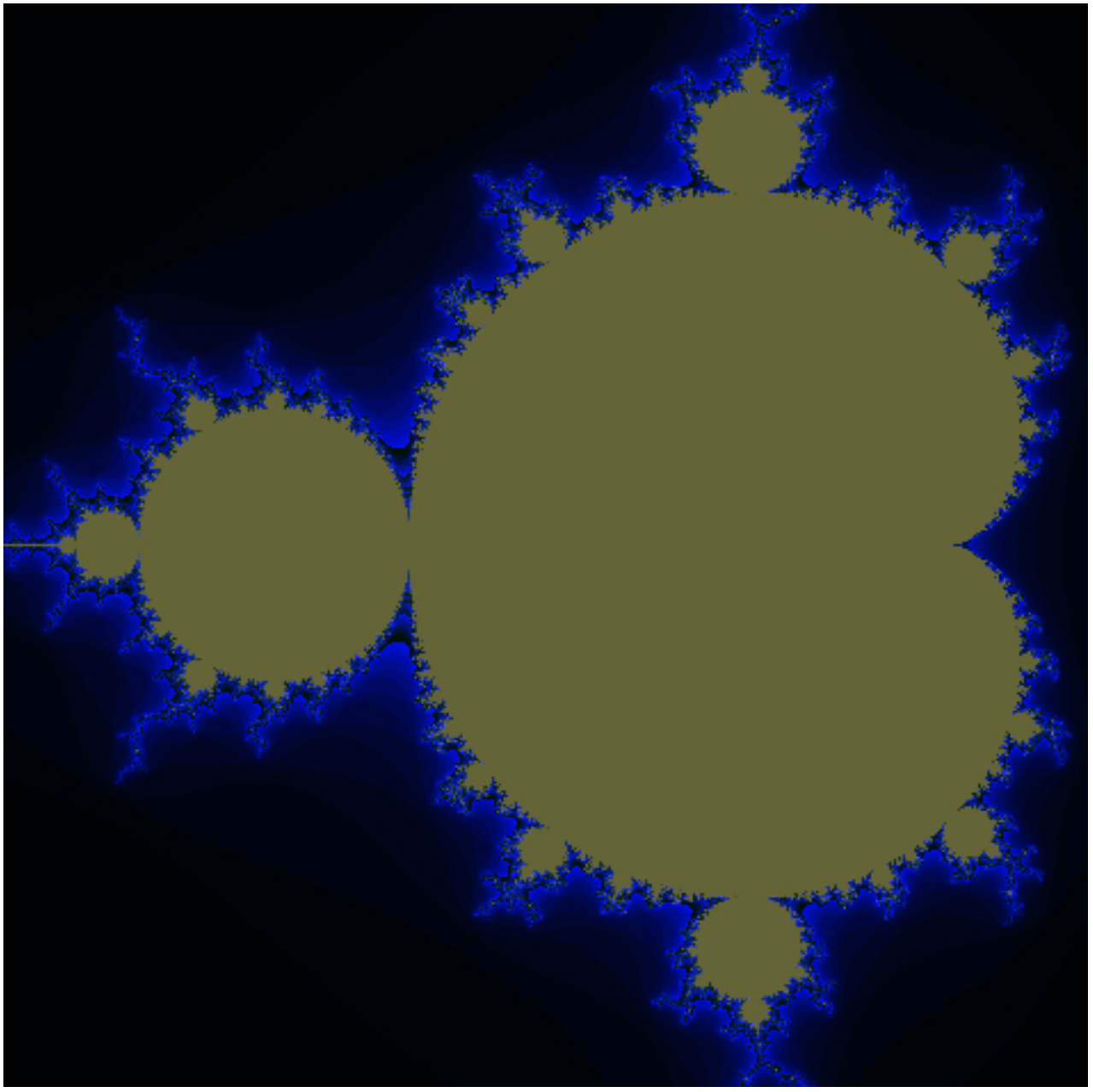
Slika 5: Generisan Madnelbrot set sa *CUDA* implementacijom pri rezoluciji 475x475 i 10 iteracija



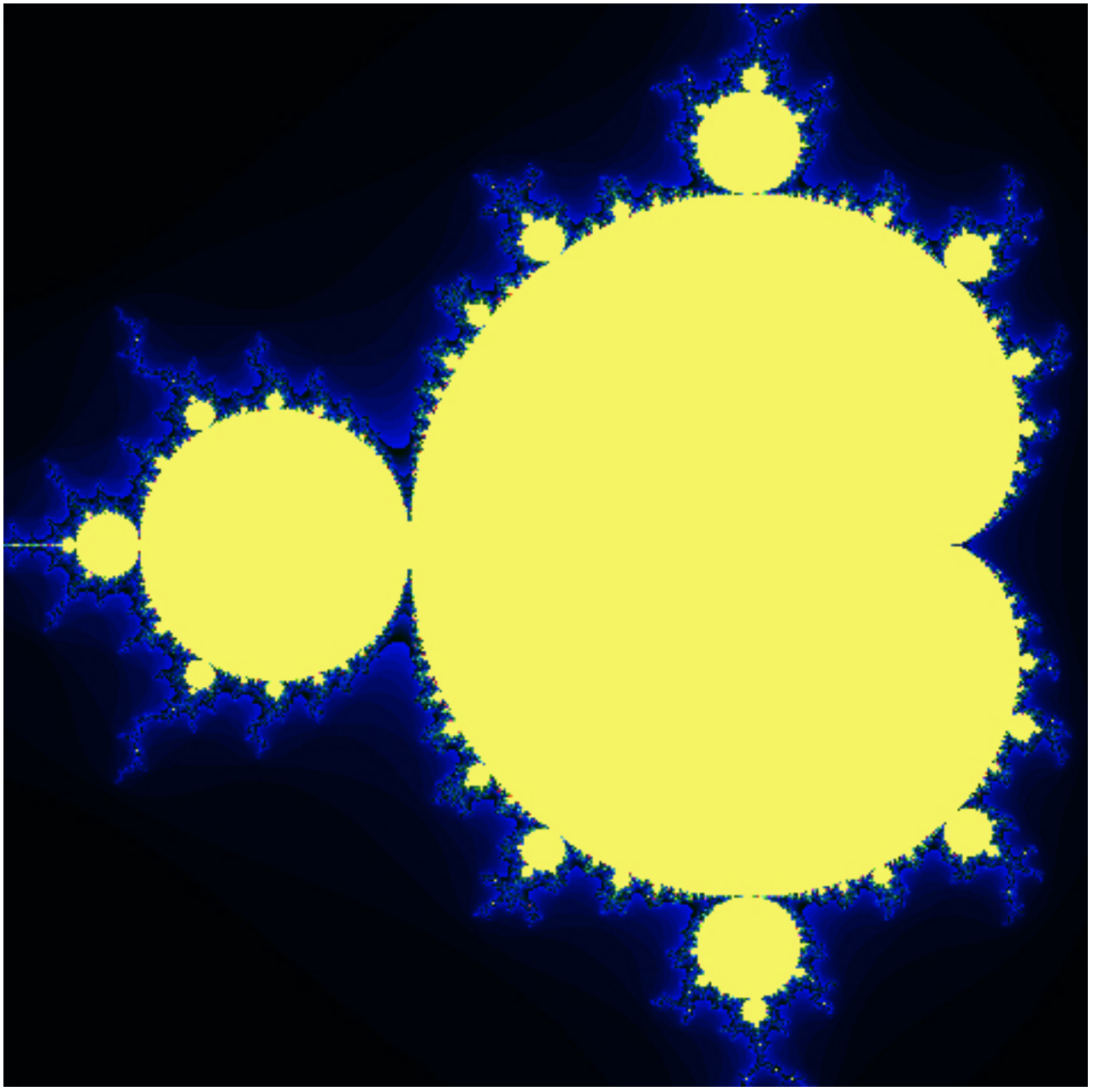
Slika 6: Generisan Madnelbrot set sa *CUDA* implementacijom pri rezoluciji 475x475 i 20 iteracija



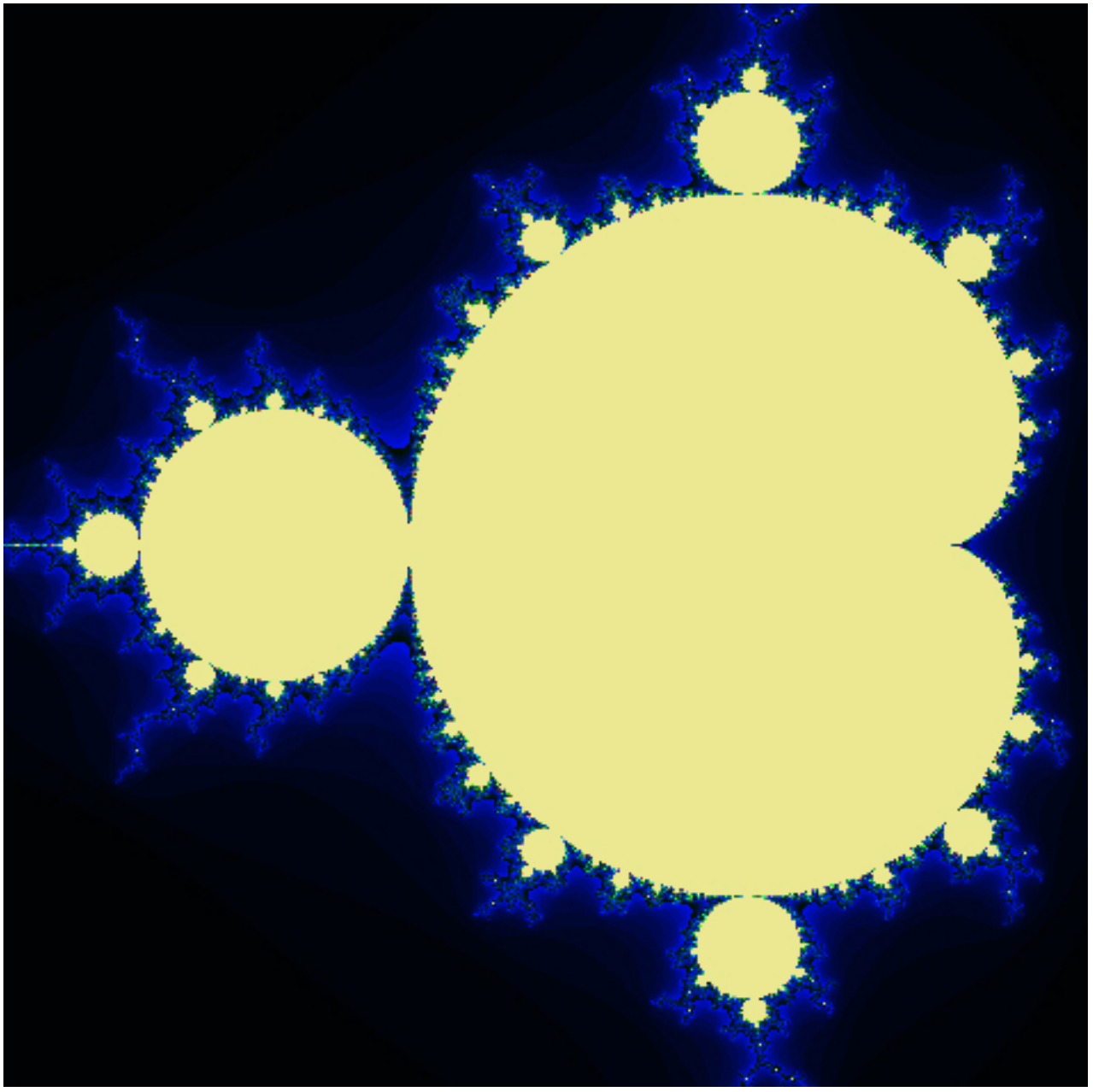
Slika 7: Generisan Madnelbrot set sa *CUDA* implementacijom pri rezoluciji 475x475 i 50 iteracija



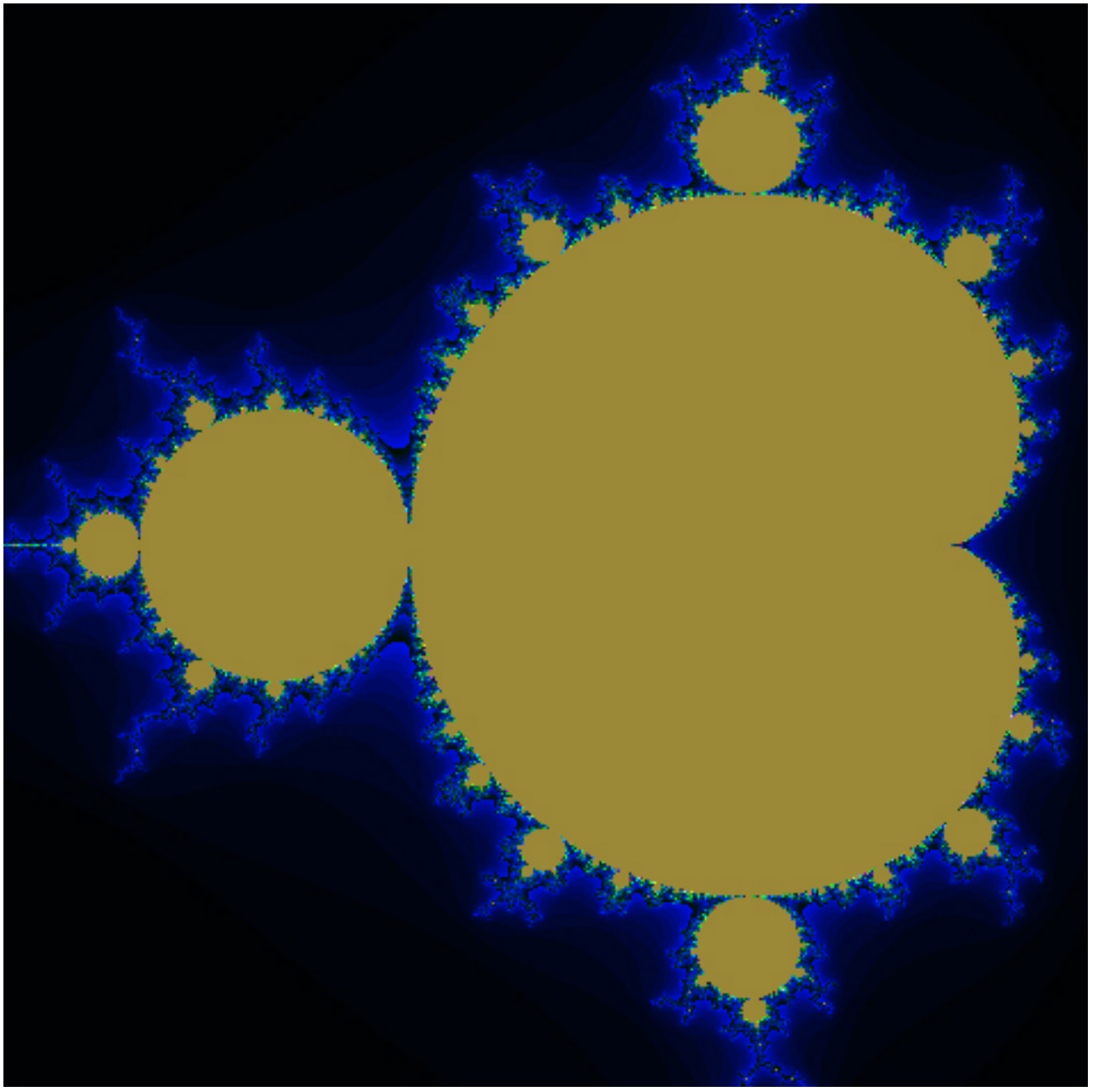
Slika 8: Generisan Madnelbrot set sa *CUDA* implementacijom pri rezoluciji 475x475 i 100 iteracija



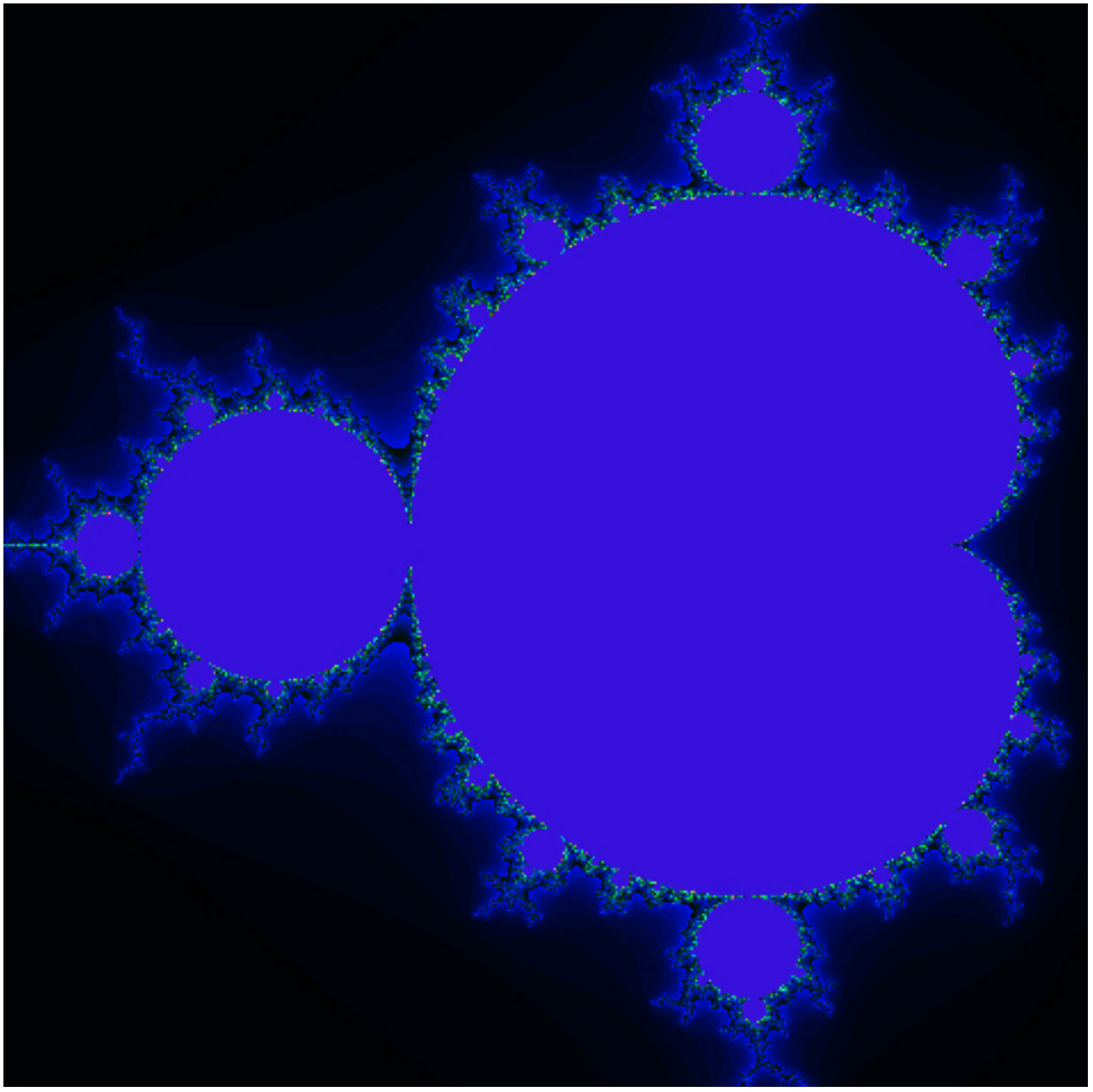
Slika 9: Generisan Madnelbrot set sa *CUDA* implementacijom pri rezoluciji 475x475 i 500 iteracija



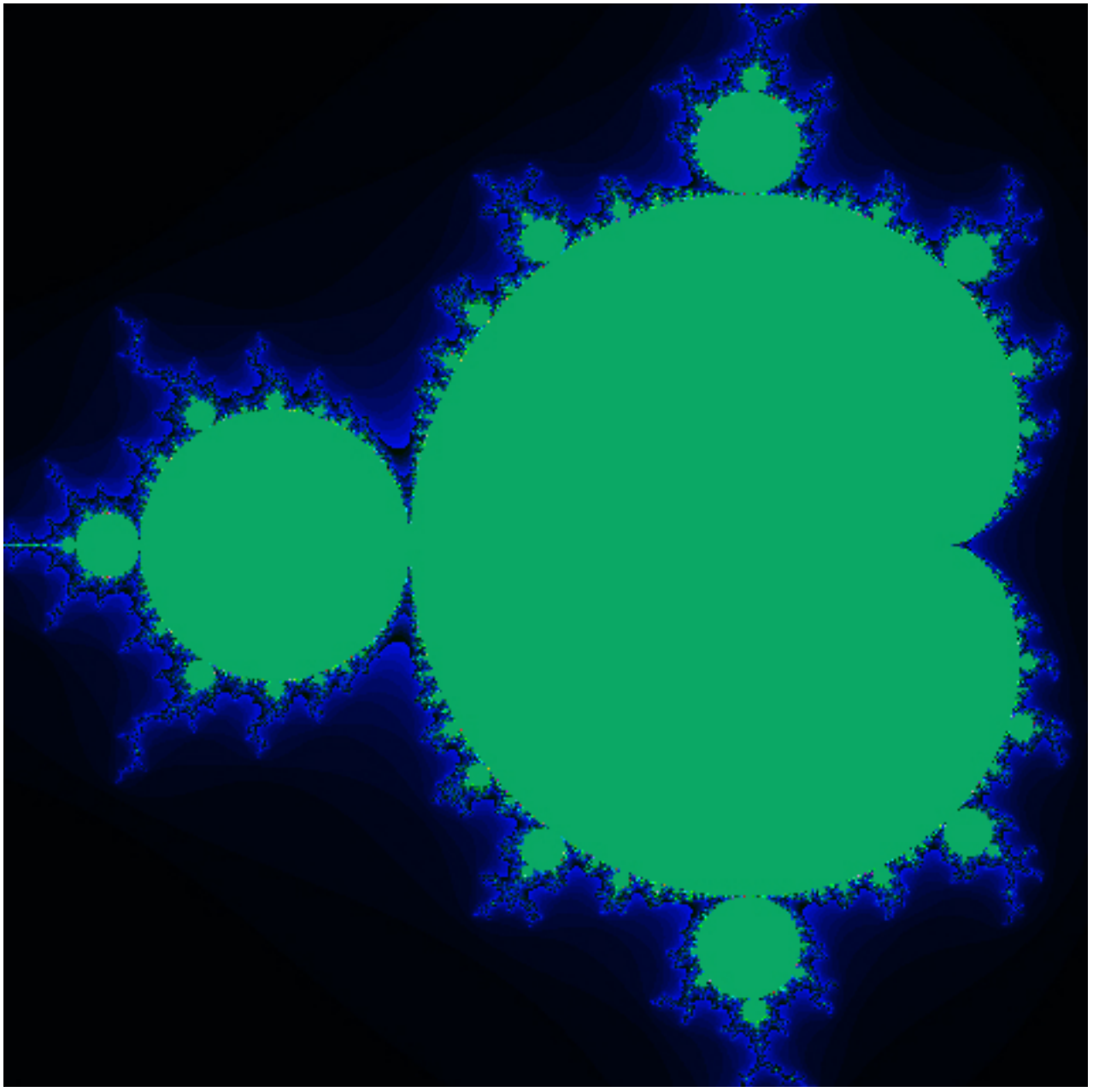
Slika 10: Generisan Madnelbrot set sa *CUDA* implementacijom pri rezoluciji 475x475 i 1000 iteracija



Slika 11: Generisan Madnelbrot set sa *CUDA* implementacijom pri rezoluciji 475x475 i 5000 iteracija



Slika 12: Generisan Madnelbrot set sa *CUDA* implementacijom pri rezoluciji 475x475 i 10000 iteracija



Slika 13: Generisan Madnelbrot set sa *CUDA* implementacijom pri rezoluciji 475x475 i 25000 iteracija

Bibliografija

[1] Open mpi: Open source high performance computing.

[2] Kenneth Falconer. *Fractal geometry: mathematical foundations and applications*. John Wiley & Sons, 2004.

[3] Tilmann Gneiting, Hana Ševčíková, and Donald B Percival. Estimators of fractal dimension: Assessing the roughness of time series and spatial data. *Statistical Science*, pages 247–277, 2012.

[4] Shuai Liu, Zheng Pan, Weina Fu, and Xiaochun Cheng. Fractal generation method based on asymptote family of generalized mandelbrot set and its application. *J. Nonlinear Sci. Appl*, pages 1148–1161, 2016.

[5] Jeffrey Lyman. Optimizing *CUDA* for *GPU* architecture: Mandelbrot set.

[6] Savina Banasl Neetu and RK Bansal. Design and analysis of fractal antennas based on koch and sierpinski fractal geometries. *International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering*, 2(6):2110–2116, 2013.

[7] Prof. Veljko Petrović. Applied computer sciences: Cuda.

[8] Prof. Veljko Petrović. Applied computer sciences: Technologies of parallel programming.

[9] Weihua Sun and Shutang Liu. Consensus of julia sets. *Fractal and Fractional*, 6(1), 2022.