SVEUČILIŠTE U SPLITU

FAKULTET ELEKTROTEHNIKE, STROJARSTVA I BRODOGRADNJE

Seminarski rad

Implementacija Gaussian filtera

Bruno Grbavac, Josip Sanader, Ivan Stojković

Split, lipanj 2023.

SADRŽAJ

UVOD

Digitalna obrada slika je ključni dio mnogih aplikacija, od medicinske dijagnostike do računalnog vida i umjetne inteligencije. Jedan od osnovnih postupaka u obradi slika je filtriranje, koje se koristi za uklanjanje šuma, poboljšanje kvalitete slike i izdvajanje bitnih značajki. Među raznim filterima koji se koriste, Gaussian filter je jedan od najčešće primijenjenih.

Gaussian filter je linearni filter koji primjenjuje konvoluciju između slike i Gaussian funkcije. Ovaj filter ima svojstvo izglađivanja slike, što znači da uklanja šum i smanjuje oštre prijelaze između piksela. Osim toga, Gaussian filter također može smanjiti visoku frekvencijsku komponentu slike, čime se postiže efekt blagog zamućivanja. U ovom seminarskom radu fokusirat ćemo se na implementaciju Gaussian filtra na različitim platformama, posebno na CPU-u, GPU-u i GPU-u s dijeljenom memorijom. Ovi uređaji imaju različite arhitekture i mogućnosti obrade, što će nam omogućiti detaljno razumijevanje prednosti i nedostataka svake platforme u kontekstu obrade slika.

Prvo ćemo opisati osnovni algoritam Gaussian filtriranja i matematičku formulaciju koja stoji iza njega. Zatim ćemo preći na implementaciju na CPU-u. Analizirat ćemo performanse, vremensku složenost i moguće optimizacije na CPU-u. Nakon toga, usredotočit ćemo se na implementaciju na GPU-u, koji ima masovno paralelne procesore pogodne za brzu obradu slika. Koristit ćemo programski jezik s podrškom za GPU programiranje, poput CUDA-e, te ćemo analizirati performanse i skalabilnost GPU implementacije. Konačno, proučit ćemo GPU s dijeljenom memorijom, koji kombinira prednosti CPU-a i GPU-a. Ova arhitektura omogućuje brzu komunikaciju između CPU-a i GPU-a, čime se povećava učinkovitost obrade slika. Istražit ćemo kako iskoristiti dijeljenu memoriju za optimizaciju Gaussian filtriranja i usporediti rezultate s prethodnim implementacijama. Svrha ovog seminarskog rada je stjecanje dubljeg razumijevanja Gaussian filtriranja i analiza različitih platformi za obradu slika.

GAUSSIAN FILTAR

U području elektronike i digitalne obrade signale, Gaussov filtar je filtar čiji je impulsni odziv jednak Gaussovoj funkciji, odnosno njenoj aproksimaciji, jer bi stvarni Gaussov odziv bio beskonačan. U pravilu se koristi da zamuti slike, ukloni šum i detalje te je u pravilu je to nisko-propusni filtar.

Glavna ideja iza Gaussian filtra je da svaki piksel u izlaznoj slici dobije vrijednost koja je težinski prosjek vrijednosti piksela u susjedstvu tog piksela. Težine se određuju prema Gaussovoj funkciji koja daje veću težinu pikselima bližima središnjem pikselu, dok pikselima udaljenim od središnjeg piksela dodjeljuje manju težinu.

Jedan od parametara koji se moraju odrediti prilikom primjene Gaussian filtra je standardna devijacija (sigma) Gaussove funkcije. Veći sigma rezultira širim prostornim rasponom i glađom izlaznom slikom, dok manji sigma rezultira oštrijim rubovima.

Postoji nekoliko metoda za primjenu Gaussian filtriranja na sliku. Jedan od najčešćih pristupa je konvolucija između Gaussove jezgre (kernela) i ulazne slike. Gaussova jezgra je dvodimenzionalna matrica koja predstavlja raspodjelu težina piksela u susjedstvu. Konvolucija se izvodi tako da se svaki piksel u ulaznoj slici pomnoži s pripadajućim težinskim koeficijentom iz Gaussove jezgre, a zatim se svi pomnoženi pikseli zbroje kako bi se dobio izlazni piksel.

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

1D

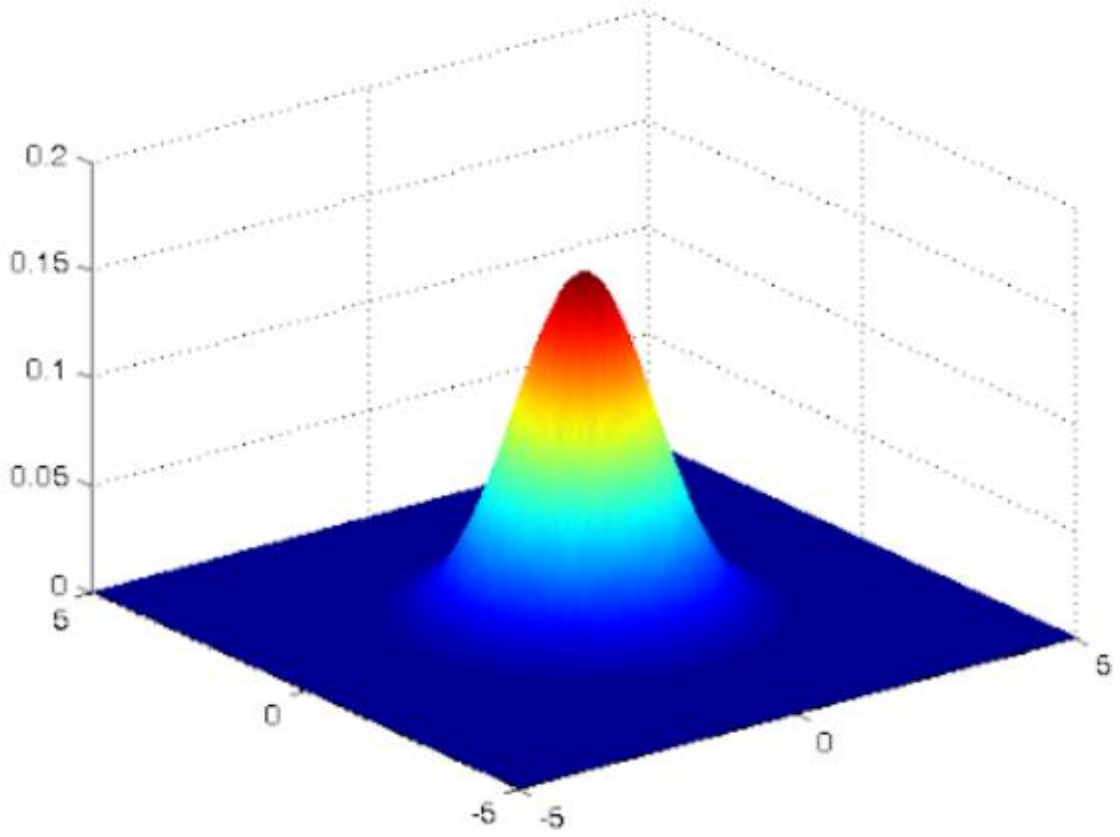
$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

2D

Slika 1. Gaussova funkcija

- *x* - udaljenost od izvorišne točke u horizontalnom smjeru; *y* - udaljenost od izvorišne točke u vertikalnom smjeru; *σ* - standardna devijacija Gaussove distribucije

Kada se obrađuju slike treba se koristi dvodimenzionalna Gaussova funkcija, koja jednaka produktu dvije jednodimenzionalne funkcije (jedna za svaki smjer).



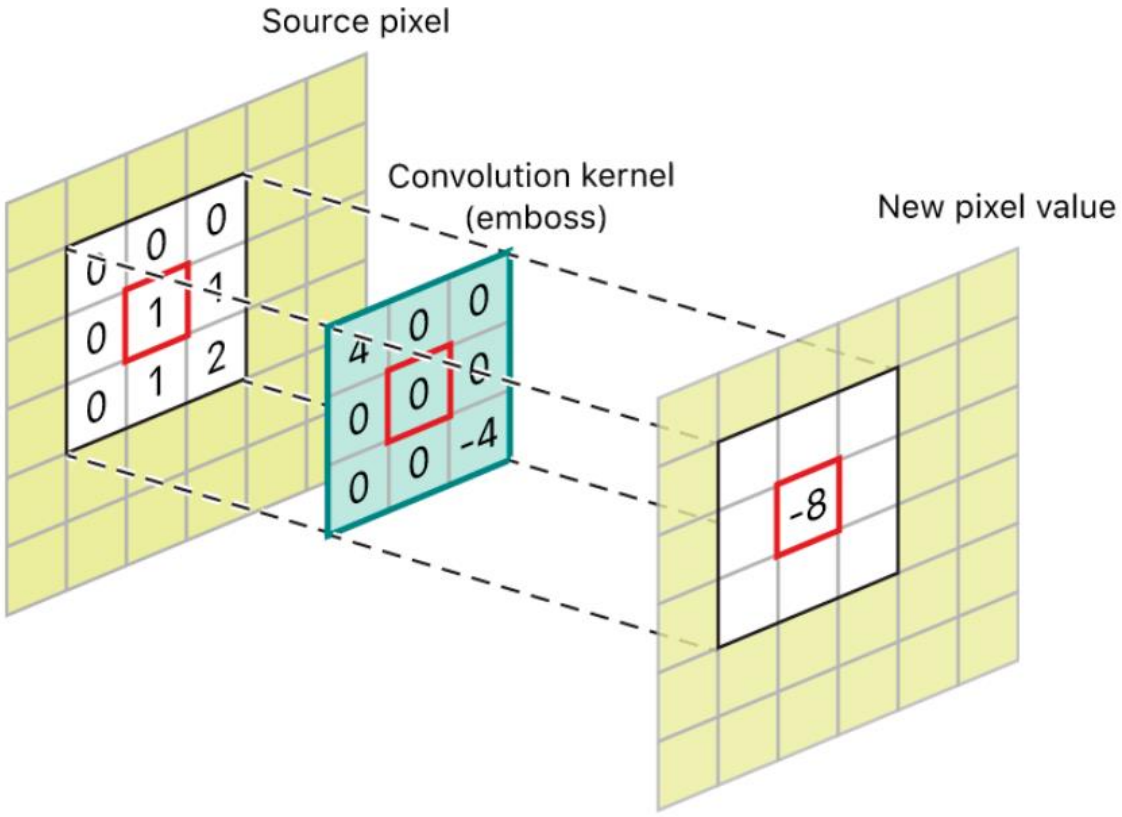
Slika 2. 2D Gaussova distribucija

Na slici 2. Prikazana je 2D Gaussova distribucija sa srednjom vrijednošću (0,0), i $\sigma = 1$.

Ako se zamisli da se ova distribucija superponira preko grupe piksela na nekoj slici, gledajući ovaj graf da se lako zaključiti da se radi o težinskom prosjeku vrijednosti piksela i visine krivulje u toj točci. Dakle, ovaj filter se primjenjuje tako da se slika konvoluira sa Gaussovom funkcijom. To znači, da se uzima Gaussova funkcija i generira se matrica dimenzija n x m. Koristeći matricu i visinu razdiobe na lokaciji određenog piksela izračunati ćemo novu vrijednost za zamućenu sliku.

Konvolucija je matematička operacija u kojoj se uzima mala matrica, koja se još naziva i kernel, i sa tim kernelom se prelazi preko matrice piksela slike. Na svakom pikselu se provodi

određena matematička operacija između kernela i vrijednosti piksela kako bi se odredila vrijednost izlaznog piksela u zamućenoj slici.



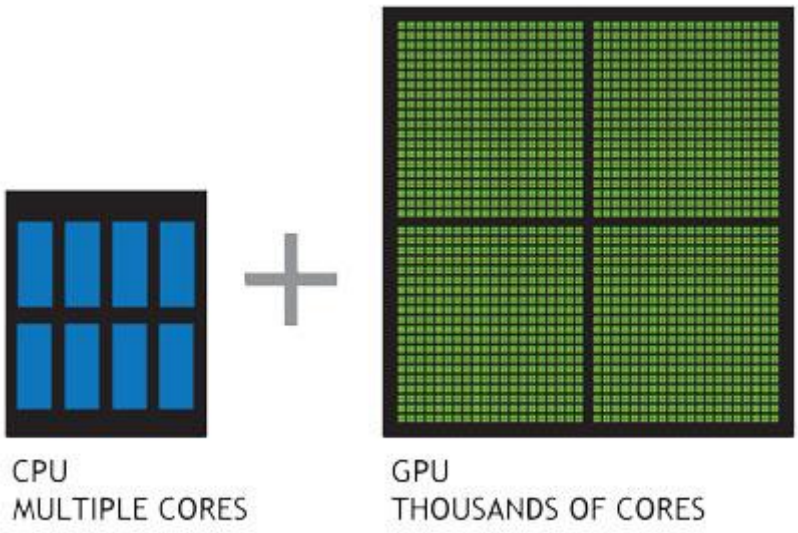
Slika 3. Primjer konvolucije

Mijenjajući vrijednosti kernela, mijenja se utjecaj na sliku te se tako postižu različiti efekti, zamućivanje, izoštravanje, detekcija rubova itd. Sigma je ta koja određuje širinu funkcije i nju korisnik unosi tijekom projektiranja filtra. Kernel se pomiče duž slike i tako stvara novu sliku čije su nove vrijednosti piksela izračunate na osnovu funkcije kernela.

CPU, GPU, CUDA

CUDA (Compute Unified Device Architecture) je paralelni programski model i platforma koju je razvila tvrtka NVIDIA kako bi omogućila programerima da iskoriste snagu grafičkih procesora (GPU) za izvođenje visoko paralelnih računskih zadataka. Ova tehnologija je revolucionarna jer omogućava ubrzanje obrade podataka upotrebom masivno paralelnih arhitektura GPU-a.

Tradicionalno, GPU-ovi su se koristili isključivo za grafičku obradu, ali NVIDIA je stvorila CUDA programski model koji omogućava programerima da izvršavaju opće namjenske računske zadatke na GPU-ima. CUDA omogućuje programerima da napišu programe koji koriste veliki broj CUDA jezgara na GPU-u kako bi izvršavali iste zadatke istovremeno, čime se postiže izvanredna paralelna obrada. Korištenje CUDA-e za paralelno izvršavanje računskih zadataka ima brojne prednosti. GPU-ovi imaju velik broj CUDA jezgara i visoku propusnost memorije, što omogućava efikasno izvršavanje matematičkih i drugih računskih operacija. Osim toga, CUDA pruža programerima fleksibilnost u korištenju različitih programskih jezika poput C, C++, Pythona i drugih, što olakšava integraciju postojećih aplikacija s GPU-om. U primjenama koje zahtijevaju intenzivnu obradu podataka, poput obrade slika, simulacija, dubokog učenja i znanstvenih istraživanja, CUDA se pokazala izuzetno korisnom. Omogućuje znatno ubrzanje izračuna i rješavanje problema koji bi inače zahtijevali puno više vremena na klasičnim procesorima.



Slika 4. CPU vs GPU

Centralna procesorska jedinica (CPU) i grafička procesorska jedinica (GPU) su ključni dijelovi računalnih sustava koji se razlikuju po arhitekturi, funkcionalnostima i primarnim namjenama.

CPU, kao središnji procesor računalnog sustava, ima nekoliko jezgara koja su relativno snažna i optimizirana za izvršavanje složenih instrukcija u sekvencijalnom redoslijedu. To ga čini pogodnim za razne zadatke koji zahtijevaju složene operacije i obradu podataka. CPU je dizajniran kao općeniti procesor koji obavlja razne zadatke, uključujući upravljanje operativnim sustavom, pokretanje aplikacija i izvršavanje algoritama.

S druge strane, GPU je dizajniran za visoko paralelnu obradu podataka. GPU ima veliki broj jezgara (stotine ili čak tisuće) organiziranih u grupe koje se nazivaju streaming multiprocesori (SM). Svako CUDA jezgro u GPU-u je manje snažno, ali može istovremeno izvršavati jednostavnije instrukcije paralelno. Ova paralelna arhitektura omogućava GPU-u da brzo i efikasno izvršava zadatke koji zahtijevaju istovremeno izvođenje velikog broja instrukcija na velikim skupovima podataka. GPU je posebno specijaliziran za obradu grafike, matematičke operacije i zadatke koji se mogu paralelizirati. Njegova arhitektura je optimizirana za brzu obradu vektora, matrica i drugih operacija potrebnih za grafiku, simulacije, duboko učenje i

druge računalno intenzivne zadatke. Također, GPU ima širu memorijsku sabirnicu koja pruža visoku propusnost podataka. To omogućuje brz pristup velikim skupovima podataka, što je posebno važno za zadatke kao što su obrada slika visoke rezolucije, renderiranje 3D grafike i simulacije.

Kombinacija CPU-a i GPU-a omogućava računalnim sustavima da iskoriste snagu oba procesora u skladu s njihovim specifičnim zahtjevima. CPU je ključan za opću obradu, upravljanje sustavom i izvršavanje složenih algoritama, dok GPU pruža brzu paralelnu obradu podataka i ubrzanje za grafičke i računalno intenzivne zadatke.

CUDA memorija je organizirana u hijerarhijsku strukturu kako bi podržala brzu i efikasnu obradu podataka na GPU-u. Struktura CUDA memorije uključuje tri glavne vrste memorije: globalnu memoriju, dijeljenu memoriju i lokalnu memoriju.

Globalna memorija:

- Globalna memorija je najveća memorija dostupna na GPU-u.
- Koristi se za dugotrajno pohranjivanje podataka i ima globalni doseg.
- Podaci u globalnoj memoriji mogu se čitati i pisati iz svih CUDA jezgara.
- Pristup globalnoj memoriji ima veću latenciju u odnosu na dijeljenu i lokalnu memoriju.
- Globalna memorija se koristi za prijenos podataka između CPU-a i GPU-a te za dugotrajno pohranjivanje izlaznih rezultata.

Dijeljena memorija:

- Dijeljena memorija je brza i lokalna memorija koja je fizički dijeljena unutar svakog streaming multiprocesora (SM) na GPU-u.
- Dijeljena memorija se koristi za privremeno pohranjivanje podataka koji su zajednički svim nitima unutar bloka niti.

- Omogućuje brzu razmjenu podataka između CUDA jezgara unutar istog SM-a.
- Dijeljena memorija je relativno ograničenog kapaciteta (obično nekoliko desetaka do nekoliko stotina kilobajta).
- Pristup dijeljenoj memoriji je vrlo brz i ima nižu latenciju u odnosu na globalnu memoriju.

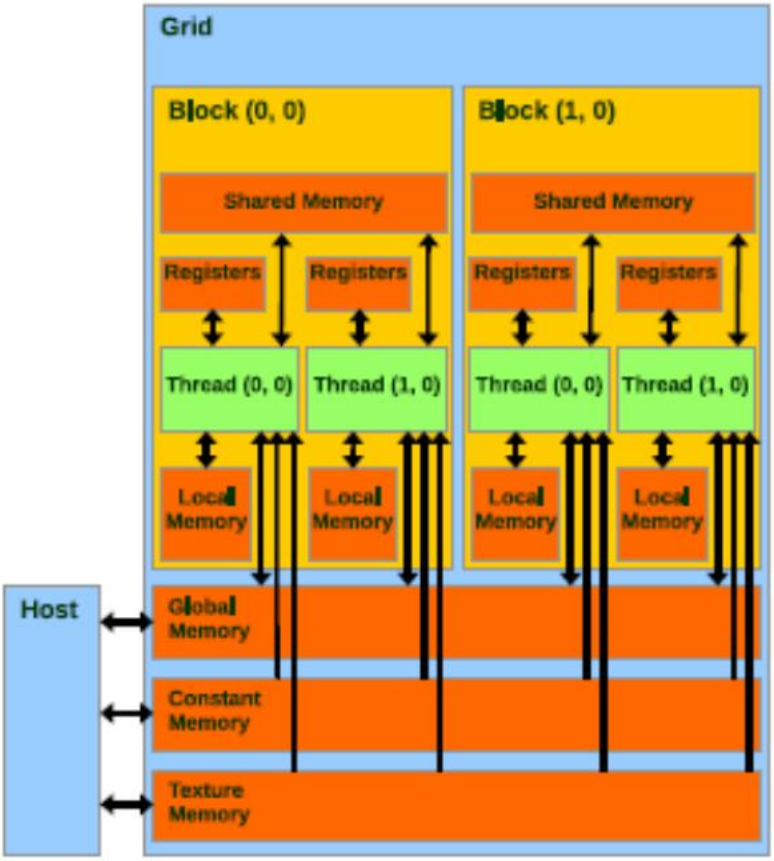
Lokalna memorija:

- Lokalna memorija je memorija koja je specifična za svaku nit unutar CUDA jezgara.
- Koristi se za privremeno pohranjivanje lokalnih varijabli, privremenih rezultata i privremenog spremanja podataka tijekom izvođenja niti.
- Pristup lokalnoj memoriji ima veću latenciju od dijeljene i globalne memorije.
- Lokalna memorija se automatski dodjeljuje svakoj niti i koristi se za podršku lokalnih izračuna i privremenih podataka.

Osim ovih glavnih vrsta memorije, postoje i druge vrste memorije u CUDA programiranju, kao što su konstantna memorija (za čitanje samo podataka) i teksturna memorija (za optimizirani pristup podacima).

CUDA arhitektura se sastoji od tri osnovna dijela, koja pružaju podršku pri ostvarivanju punog potencijala paralelnog programiranja na grafičkoj kartici. Arhitektura se dijeli na rešetke, blokove i niti s hijerarhijskom strukturom. Kako jedna rešetka ima više blokova, a jedan blok više niti, paralelizam koji se postiže kroz navedenu arhitekturu je inherentan. Rešetka je grupa niti koje izvode isti kernel. Niti rešetke nisu sinkronizirane. Svaki poziv CUDA-i od strane CPU-a se radi kroz jednu rešetku. Pokretanje rešetke na CPU je sinkrona operacija, ali više rešetki se može istovremeno izvoditi. Na sustavima s više GPU-ova, rešetke se ne mogu dijeliti između više GPU-ova. Rešetke su sastavljene od blokova. Svaki blok je logička jedinica koja se sastoji od niza koordiniranih niti te određenom količinom dijeljene memorije. Kao što rešetke ne mogu biti dijeljene između GPU-ova, blokovi ne mogu biti dijeljeni između multiprocesora. Svi blokovi u rešetci izvode isti program. Ugrađena varijabla blockIdx se

koristi za identifikaciju trenutnog bloka. BlockIdx može biti jednodimenzionalan ili najviše trodimenzionalan, ovisno o dimenzionalnosti rešetke. Blokovi su sastavljeni od niti. Niti se izvode na pojedinačnim jezgrama te nisu ograničene na pojedinu jezgru, za razliku od rešetki i blokova. Svaka nit ima svoj threadIdx, koji može biti jednodimenzionalan ili višedimenzionalan, ovisno o dimenzionalnosti bloka, a threadIdx je relativan u odnosu na niti bloka u kojem se nalazi. Niti imaju određenu količinu registar memorije, a jedan blok ima 512, dok novije kartice podržavaju do 1024 niti u bloku. CUDA arhitektura prikazana je na slici 5.



Slika 5. CUDA arhitektura

CPU REALIZACIJA

```cpp
#include <iostream>
#include <vector>
#include <cmath>
#include <fstream>
#include <string>
#include <chrono>
#include <opencv2/opencv.hpp>

using namespace std;
using namespace cv;
using namespace chrono;

// fja za spremanje RGB slike u JPG
bool saveImage(const string& filename, const vector<vector<Vec3b>>& image) {
    int width = static_cast<int>(image[0].size());
    int height = static_cast<int>(image.size());

    Mat outputImage(height, width, CV_8UC3);
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            outputImage.at<Vec3b>(i, j) = image[i][j];
        }
    }

    return imwrite(filename, outputImage);
}

// funkcija koja obavlja cjelokupno Gaussovo filtriranje
void applyGaussianBlur(vector<vector<Vec3b>>& image, int size, double sigma) {
    int height = static_cast<int>(image.size());
    int width = static_cast<int>(image[0].size());
    vector<vector<Vec3b>> blurredImage(height, vector<Vec3b>(width));

    // stvara Gaussian kernel koristeci zadane parametre i Gaussovu funkciju
    vector<vector<double>> kernel(size, vector<double>(size));
    double sum = 0.0;
    int halfSize = size / 2;
    for (int i = -halfSize; i <= halfSize; i++) {
        for (int j = -halfSize; j <= halfSize; j++) {
            double exponent = -(i * i + j * j) / (2.0 * sigma * sigma);
            double value = (1.0 / (2.0 * 3.14159 * sigma * sigma)) * exp(exponent);
            kernel[i + halfSize][j + halfSize] = value;
            sum += value;
        }
    }

    // normalizacija kernela
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            kernel[i][j] /= sum;
        }
    }
```

```
    // primjena Gaussian filtera na slici
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            double sumR = 0.0;
            double sumG = 0.0;
            double sumB = 0.0;

            // konvolucija matrice kernela (filtra) i matrice slike
            for (int k = -halfSize; k <= halfSize; k++) {
                for (int l = -halfSize; l <= halfSize; l++) {
                    int rowIndex = min(max(i + k, 0), height - 1);
                    int colIndex = min(max(j + l, 0), width - 1);

                    double weight = kernel[k + halfSize][l + halfSize];
                    sumR += weight * image[rowIndex][colIndex][0];
                    sumG += weight * image[rowIndex][colIndex][1];
                    sumB += weight * image[rowIndex][colIndex][2];
                }
            }

            blurredImage[i][j][0] = static_cast<uint8_t>(sumR);
            blurredImage[i][j][1] = static_cast<uint8_t>(sumG);
            blurredImage[i][j][2] = static_cast<uint8_t>(sumB);
        }
    }

    image = blurredImage;
}
int main() {
    auto start = high_resolution_clock::now();

    // ucitavanje ulazne slike
    Mat cvImage = imread("C:\\Users\Ivan\\Desktop\\nar i forenzika\\jezero.jpg");
    if (cvImage.empty()) {
        cout << "Failed to load the image." << endl;
        return -1;
    }

    // funkcija vezana uz koristenje OpenCV librarya - pretvorba u Vec3b vektor iz
OpenCV slike
    vector<vector<Vec3b>> image(cvImage.rows, vector<Vec3b>(cvImage.cols));
    for (int i = 0; i < cvImage.rows; i++) {
        for (int j = 0; j < cvImage.cols; j++) {
            image[i][j] = cvImage.at<Vec3b>(i, j);
        }
    }

    // parametri kernela i primjena filtra na sliku
    int size = 5;
    double sigma = 5.0;
    applyGaussianBlur(image, size, sigma);

    // funkcija vezana uz koristenje OpenCV librarya - pretvorba nazad u OpenCV
sliku iz Vec3b vektora
    Mat outputImage(cvImage.rows, cvImage.cols, CV_8UC3);
    for (int i = 0; i < cvImage.rows; i++) {
        for (int j = 0; j < cvImage.cols; j++) {
            outputImage.at<Vec3b>(i, j) = image[i][j];
```

```
        }
    }

    // sprema izlaznu sliku
    if (!imwrite("C:\\Users\Ivan\\Desktop\\nar i forenzika\\filtered_jezero.jpg",
outputImage)) {
        cout << "Failed to save the image." << endl;
        return -1;
    }

    auto end = high_resolution_clock::now();

    // Izračunavanje ukupnog vremena izvršavanja u milisekundama
    duration<double, milli> duration = end - start;
    double executionTime = duration.count();

    cout << "Vrijeme izvršavanja: " << executionTime << " ms" << endl;

    return 0;
}
```

U ovom kodu korištena je OpenCV bibioteka za učtavanje i konverziju JPG slike u program kao i za spremanje rezultante slike. Funkcija **saveImage** sprema RGB sliku u JPG format. Ona prima naziv datoteke i 2D vektor koji predstavlja sliku kao ulazni podatak. Funkcija pretvara vektor u OpenCV objekt tipa **Mat** i koristi funkciju **imwrite** za spremanje slike.

Funkcija **applyGaussianBlur** vrši Gaussovo zamućivanje nad ulaznom slikom. Prima 2D vektor koji predstavlja sliku, veličinu Gaussova kernela i vrijednost sigme kao ulazne podatke. Funkcija stvara Gaussov kernel koristeći zadane parametre i primjenjuje konvolucijsku operaciju za zamućivanje slike. Vrši se normalizacija koeficijenata tako da se osigura da zbroj koeficijenata ne prelazi 1, te se na taj način osigurava da se ne promijeni ukupna svjetlina slike.

Unutar **main** funkcije bilježi vrijeme se koristeći **high_resolution_clock** kako bi se moglo izračunati ukupno vrijeme izvršavanja programa, ulazna slika se učitava pomoću funkcije **imread** iz OpenCV biblioteke. Ako učitavanje slike ne uspije, ispisuje se poruka o grešci i program se prekida, učitana slika se pretvara iz objekta **Mat** u 2D vektor **Vec3b** pomoću petlji kako bi se omogućila daljnja obrada slike, postavljaju se parametri za Gaussovo zamućivanje: veličina kernela i vrijednost sigme, poziva se funkcija **applyGaussianBlur** kako bi se primijenilo Gaussovo zamućivanje na sliku. Zamagljena slika se ponovno pretvara iz 2D vektora **Vec3b** u objekt **Mat**. Izlazna slika se sprema koristeći funkciju **imwrite**. Ako spremanje slike ne uspije,

ispisuje se poruka o grešci i program se prekida. Vrijeme izvršavanja se ponovno bilježi koristeći **high_resolution_clock**. Izračunava se ukupno vrijeme izvršavanja tako da se od završnog vremena oduzme početno vrijeme. Vrijeme izračunato u sekundama se pretvara u milisekunde te se ispisuje se vrijeme izvršavanja programa na konzolu.

GPU REALIZACIJA

```
#include <iostream>
#include <vector>
#include <cmath>
#include <fstream>
#include <string>
#include <chrono>
#include <opencv2/opencv.hpp>
#include <cuda_runtime.h>
#include "device_launch_parameters.h"
#include <opencv2/core.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/highgui.hpp>

using namespace std;
using namespace cv;
using namespace chrono;

__global__ void gaussianBlur(const unsigned char* inputImage, unsigned char*
outputImage, int width, int height, float sigma) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < width && y < height) {
        int index = (y * width + x) * 3;

        int halfSize = 2;
        float sumR = 0.0f;
        float sumG = 0.0f;
        float sumB = 0.0f;
        float sumWeight = 0.0f;


        for (int dy = -halfSize; dy <= halfSize; dy++) {
            for (int dx = -halfSize; dx <= halfSize; dx++) {
                int nx = x + dx;
                int ny = y + dy;

                if (nx >= 0 && nx < width && ny >= 0 && ny < height) {
                    int nIndex = (ny * width + nx) * 3;
```

```
                    unsigned char r = inputImage[nIndex];
                    unsigned char g = inputImage[nIndex + 1];
                    unsigned char b = inputImage[nIndex + 2];

                    float weight = exp(-(dx * dx + dy * dy) / (5.0f * sigma *
sigma));


                    sumR += weight * r;
                    sumG += weight * g;
                    sumB += weight * b;
                    sumWeight += weight;
                }
            }
        }

        // Normalizacija
        unsigned char resultR = static_cast<unsigned char>(sumR / sumWeight);
        unsigned char resultG = static_cast<unsigned char>(sumG / sumWeight);
        unsigned char resultB = static_cast<unsigned char>(sumB / sumWeight);

        outputImage[index] = resultR;
        outputImage[index + 1] = resultG;
        outputImage[index + 2] = resultB;
    }
}
int main() {
    auto start = high_resolution_clock::now();

    Mat inputImage = imread("C:\\Users\\Ivan\\Desktop\\nariforenzika\\jezero.jpg",
IMREAD_COLOR);
    if (inputImage.empty()) {
        std::cerr << "Failed to open image file." << std::endl;
        return -1;
    }


    int width = inputImage.cols;
    int height = inputImage.rows;
    int numPixels = width * height;


    unsigned char* inputImageHost = inputImage.data;
    unsigned char* outputImageHost = new unsigned char[numPixels * 3];


    unsigned char* inputImageDevice;
    unsigned char* outputImageDevice;
    cudaMalloc((void**)&inputImageDevice, numPixels * 3 * sizeof(unsigned char));
    cudaMalloc((void**)&outputImageDevice, numPixels * 3 * sizeof(unsigned char));


    cudaMemcpy(inputImageDevice, inputImageHost, numPixels * 3 * sizeof(unsigned
char), cudaMemcpyHostToDevice);

    dim3 blockSize(16, 16);
```

```
    dim3 gridSize((width + blockSize.x - 1) / blockSize.x, (height + blockSize.y -
1) / blockSize.y);


    gaussianBlur << <gridSize, blockSize >> > (inputImageDevice, outputImageDevice,
width, height, 5.0f);

    cudaMemcpy(outputImageHost, outputImageDevice, numPixels * 3 * sizeof(unsigned
char), cudaMemcpyDeviceToHost);

    Mat outputImage(height, width, CV_8UC3, outputImageHost);
    imwrite("C:\\Users\\Ivan\\Desktop\\nariforenzika\\filtered2_jezero.jpg",
outputImage);


    cudaFree(inputImageDevice);
    cudaFree(outputImageDevice);


    delete[] outputImageHost;

    auto end = high_resolution_clock::now();

    duration<double, milli> duration = end - start;
    double executionTime = duration.count();

    cout << "Vrijeme izvršavanja: " << executionTime << " ms" << endl;

    return 0;
}
```

Nakon uključivanja biblioteka definira se CUDA kernel funkcija **gaussianBlur** koja se ovaj put izvršava na GPU. Funkcija na ulaznu sliku primjenjuje Gaussov filter. Parametri funkcije su:

- **inputImage**: Pokazivač na ulaznu sliku
- **outputImage**: Pokazivač na izlaznu sliku
- **width**, **height**: Širina i visina slike
- **sigma**: Parametar koji određuje veličinu zamućenja (standardna devijacija Gaussove funkcije)

U kernel funkciji, svaki CUDA thread obrađuje jedan piksel slike. Threadovi se grupiraju u blokove i mreže kako bi se iskoristili više CUDA jezgri na GPU-u za paralelno izvršavanje. Oznaka **__global__** koristi se za definiranje kernel funkcije koja se izvršava na GPU-u.

Unutar kernel funkcije, svaki CUDA thread će obraditi jedan piksel slike. Svaki thread izračunava svoje koordinate **x** i **y** na temelju blokova i niti. Nakon što su dobivene koordinate piksela,

provjerava se je li thread unutar granica slike. Nakon definicije kernel funkcije, slijedi glavna funkcija **main().** Mjeri se vrijeme izvršavanja programa pomoću **high_resolution_clock**. Učitava se ulazna slika iz datoteke koristeći imread funkciju iz OpenCV biblioteke. Ako se slika ne može otvoriti, ispisuje se pogreška i program završava. Dobivaju se širina, visina i broj piksela slike. Alocira se memorija na hostu za ulaznu i izlaznu sliku. Alocira se memorija na uređaju (GPU) za ulaznu i izlaznu sliku pomoću cudaMalloc. Ova funkcija alocira memoriju na uređaju (GPU) za nizove koji će se koristiti za pohranu ulazne i izlazne slike. Poziva se sa sljedećim argumentima: cudaMalloc((void**)&inputImageDevice, numPixels * 3 * sizeof(unsigned char)) i cudaMalloc((void**)&outputImageDevice, numPixels * 3 * sizeof(unsigned char)). Prvi argument je adresa pokazivača na koji će se pohraniti dobiveni uređajski pokazivač, a drugi argument je veličina memorije koju treba alocirati. Kopira se ulazna sliku s hosta na uređaj pomoću cudaMemcpy. Ova funkcija kopira podatke između glavne memorije (CPU) i uređaja (GPU). U kodu se koristi dvaput: cudaMemcpy(inputImageDevice, inputImageHost, numPixels * 3 * sizeof(unsigned char), cudaMemcpyHostToDevice) i cudaMemcpy(outputImageHost, outputImageDevice, numPixels * 3 * sizeof(unsigned char), cudaMemcpyDeviceToHost). Prvi poziv kopira ulaznu sliku s CPU-a na GPU, dok 15 drugi poziv kopira izlaznu sliku s GPU-a na CPU. Argumenti funkcije uključuju izvor kopiranja, odredište kopiranja, veličinu kopiranih podataka i smjer kopiranja. Definiraju se veličine blokova i mreža za izvršavanje kernel funkcije. Pokreće se kernel funkcija gaussianBlur na uređaju pomoću CUDA sintakse << <gridSize, blockSize >> >. Kopira se izlazna slika s uređaja na host pomoću cudaMemcpy. Stvara se objekt outputImage iz izlazne slike na hostu koristeći konstruktor klase Mat iz OpenCV biblioteke. Sprema se izlazna slika na disk koristeći imwrite. Oslobađa se memorija na uređaju pomoću cudaFree. Oslobađa se memorija na hostu pomoću delete[].

Izračunava se vrijeme izvršavanja programa i ispisuje se.

REALIZACIJA NA GPU KORISTEĆI DIJELJENU MEMORIJU

```
#include <iostream>
#include <vector>
#include <cmath>
#include <fstream>
#include <string>
#include <chrono>
#include <opencv2/opencv.hpp>
#include <cuda_runtime.h>
#include "device_launch_parameters.h"
#include <opencv2/core.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/highgui.hpp>


using namespace std;
using namespace cv;
using namespace chrono;

#define BLOCK_SIZE 32

__global__ void gaussianBlur(const unsigned char* inputImage, unsigned char*
outputImage, int width, int height, float sigma) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;

    // Shared memory for inputImage
    __shared__ unsigned char sharedImage[BLOCK_SIZE + 2][BLOCK_SIZE + 2][3];

    // Populate shared memory
    int sharedX = threadIdx.x + 2;
    int sharedY = threadIdx.y + 2;

    if (x < width && y < height) {
        int index = (y * width + x) * 3;
        sharedImage[sharedY][sharedX][0] = inputImage[index];
        sharedImage[sharedY][sharedX][1] = inputImage[index + 1];
        sharedImage[sharedY][sharedX][2] = inputImage[index + 2];

        // Border regions of shared memory
        if (threadIdx.x == 0) {
            // Left border
            sharedImage[sharedY][0][0] = inputImage[(y * width + max(x - 1, 0)) *
3];
            sharedImage[sharedY][0][1] = inputImage[(y * width + max(x - 1, 0)) * 3
+ 1];
            sharedImage[sharedY][0][2] = inputImage[(y * width + max(x - 1, 0)) * 3
+ 2];
        }
        else if (threadIdx.x == blockDim.x - 1) {
```

```
            // Right border
            sharedImage[sharedY][BLOCK_SIZE + 1][0] = inputImage[(y * width + min(x
+ 1, width - 1)) * 3];
            sharedImage[sharedY][BLOCK_SIZE + 1][1] = inputImage[(y * width + min(x
+ 1, width - 1)) * 3 + 1];
            sharedImage[sharedY][BLOCK_SIZE + 1][2] = inputImage[(y * width + min(x
+ 1, width - 1)) * 3 + 2];
        }

        if (threadIdx.y == 0) {
            // Top border
            sharedImage[0][sharedX][0] = inputImage[(max(y - 1, 0) * width + x) *
3];
            sharedImage[0][sharedX][1] = inputImage[(max(y - 1, 0) * width + x) * 3
+ 1];
            sharedImage[0][sharedX][2] = inputImage[(max(y - 1, 0) * width + x) * 3
+ 2];
        }
        else if (threadIdx.y == blockDim.y - 1) {
            // Bottom border
            sharedImage[BLOCK_SIZE + 1][sharedX][0] = inputImage[(min(y + 1, height
- 1) * width + x) * 3];
            sharedImage[BLOCK_SIZE + 1][sharedX][1] = inputImage[(min(y + 1, height
- 1) * width + x) * 3];
            sharedImage[BLOCK_SIZE + 1][sharedX][2] = inputImage[(min(y + 1, height
- 1) * width + x) * 3 + 2];
        }
    }

    // Synchronize threads to ensure all shared memory is populated
    __syncthreads();

    if (x < width && y < height) {
        int index = (y * width + x) * 3;

        int halfSize = 2;
        float sumR = 0.0f;
        float sumG = 0.0f;
        float sumB = 0.0f;
        float sumWeight = 0.0f;

        for (int dy = -halfSize; dy <= halfSize; dy++) {
            for (int dx = -halfSize; dx <= halfSize; dx++) {
                int nx = sharedX + dx;
                int ny = sharedY + dy;

                unsigned char r = sharedImage[ny][nx][0];
                unsigned char g = sharedImage[ny][nx][1];
                unsigned char b = sharedImage[ny][nx][2];

                float weight = exp(-(dx * dx + dy * dy) / (2.0f * sigma * sigma));

                sumR += weight * r;
                sumG += weight * g;
                sumB += weight * b;
                sumWeight += weight;
            }
        }
```

```
        // Synchronize threads to ensure all shared memory is populated
        __syncthreads();

        // Normalizacija
        unsigned char resultR = static_cast<unsigned char>(sumR / sumWeight);
        unsigned char resultG = static_cast<unsigned char>(sumG / sumWeight);
        unsigned char resultB = static_cast<unsigned char>(sumB / sumWeight);

        outputImage[index] = resultR;
        outputImage[index + 1] = resultG;
        outputImage[index + 2] = resultB;
    }
}

int main() {
    auto start = high_resolution_clock::now();

    Mat inputImage = imread("C:\\Users\\Ivan\\Desktop\\nariforenzika\\jezero.jpg", ,
IMREAD_COLOR);
    if (inputImage.empty()) {
        std::cerr << "Failed to open image file." << std::endl;
        return -1;
    }

    int width = inputImage.cols;
    int height = inputImage.rows;
    int numPixels = width * height;

    unsigned char* inputImageHost = inputImage.data;
    unsigned char* outputImageHost = new unsigned char[numPixels * 3];

    unsigned char* inputImageDevice;
    unsigned char* outputImageDevice;
    cudaMalloc((void**)&inputImageDevice, numPixels * 3 * sizeof(unsigned char));
    cudaMalloc((void**)&outputImageDevice, numPixels * 3 * sizeof(unsigned char));

    cudaMemcpy(inputImageDevice, inputImageHost, numPixels * 3 * sizeof(unsigned
char), cudaMemcpyHostToDevice);

    dim3 blockSize(BLOCK_SIZE, BLOCK_SIZE);
    dim3 gridSize((width + blockSize.x - 1) / blockSize.x, (height + blockSize.y -
1) / blockSize.y);

    gaussianBlur << <gridSize, blockSize >> > (inputImageDevice, outputImageDevice,
width, height, 5.0f);

    cudaMemcpy(outputImageHost, outputImageDevice, numPixels * 3 * sizeof(unsigned
char), cudaMemcpyDeviceToHost);

    Mat outputImage(height, width, CV_8UC3, outputImageHost);
    imwrite(("C:\\Users\\Ivan\\Desktop\\nariforenzika\\filtered2_jezero.jpg",
outputImage);

    cudaFree(inputImageDevice);
    cudaFree(outputImageDevice);

    delete[] outputImageHost;
```

```
    auto end = high_resolution_clock::now();
    duration<double, milli> duration = end - start;
    double executionTime = duration.count();

    cout << "Vrijeme izvršavanja: " << executionTime << " ms" << endl;

    return 0;
}
```

U kernel funkciji **gaussianBlur**, dio koda je izmijenjen kako bi se koristila zajednička memorija. Dijeljenja memorija se deklarira pomoću ključne riječi **__shared__.** __shared__ unsigned char sharedImage[BLOCK_SIZE + 2][BLOCK_SIZE + 2][3];→Ovdje se stvara trodimenzionalni niz **sharedImage** u zajedničkoj memoriji. Veličina niza je **BLOCK_SIZE + 2** u oba smjera, dodajući 2 dodatna elementa s obje strane kako bi se obuhvatile granice bloka threadova. Zatim slijedi dio koda koji kopira piksele ulazne slike u zajedničku memoriju. Svaki thread kopira jedan piksel u odgovarajuću poziciju u zajedničkoj memoriji. Također, granice bloka threadova kopiraju piksele iz susjedstva kako bi se osiguralo da svi potrebni pikseli budu dostupni. Nakon toga slijedi dio koda koji kopira piksele izvan granica bloka threadova u zajedničku memoriju kako bi se omogućilo pristupanje tim pikselima tijekom izračuna zamućenja. Kasnije se izračunavaju normalizirane vrijednosti komponenti boje za izlazni piksel. Rezultati su pohranjeni u izlaznoj slici. Ostatak koda u funkciji **main** ostaje gotovo nepromijenjen, samo su dodane biblioteke i makro definicije koje su potrebne za CUDA i OpenCV funkcionalnosti. Također, učitavanje i spremanje slike koristi se OpenCV funkcije **imread** i **imwrite**.

ZAKLJUČAK

Ulazna slika: 1500x1500

Izlazna slika:

Vremena izvođenja:

CPU- 4099 ms

GPU - 844 ms

GPU-shared memory - 855 ms

Iz dobivenih rezultata može se zaključiti da je vrijeme izvođenja na GPU višestruko kraće nego li kod izvedbe na CPU bez paralelizacije. Iako smo očekivali da će u slučaju korištenja dijeljenje memorije izvođenje biti još brže to nije slučaj. Razlozi za to su mnogobrojni, može biti do neefikasnog iskorištavanje dijeljene memorije, neoptimalna raspodjela threadova, kompleksnost algoritma, mali broj podataka jer ako je veličina podataka koju obrađuje svaki thread mala u odnosu na veličinu dijeljene memorije, korist od dijeljene memorije može biti minimalna ili čak negativna. U takvim slučajevima, pristupanje zajedničkoj memoriji može biti sporije od pristupanja globalnoj memoriji, što dovodi do povećanja vremena izvršavanja. Pri obradi jako malih slika, CPU ima prednost jer on ne troši vrijeme za alociranje prostora i inicijalizaciju niti. Pri obradi većih slika, prikazuje se prednost GPU-a te se postižu i značajna ubrzanja u odnosu na CPU. Daljnjim povećavanjem dimenzija slika, korištenjem kompliciranijih filtriranja i korištenjem još efikasnijih pristupa upravljanjem memorijom pri obradi na GPU, omjer vremena izvršavanja bi trebao biti i veći.