S V E U Č I L I Š T E U S P L I T U

FAKULTET ELEKTROTEHNIKE, STROJARSTVA I

BRODOGRADNJE

SEMINARSKI RAD

BOX FILTER

Luka Dorić, Martin Josipović i

Dario Stojanac

Split, srpanj 2025.

SADRŽAJ

1 UVOD

Digitalna obrada slike je područje koje se fokusira na manipulaciju slikama. Za razliku od analogne obrade slike, digitalna obrada slike uključuje pretvaranje slika u matricu vrijednosti piksela, što računalima omogućuje analizu, poboljšanje i transformaciju vizualnih podataka s pomoću matematičkih i algoritamskih tehnika. Primarni ciljevi digitalne obrade slike uključuju poboljšanje slike, smanjenje šuma, izdvajanje značajki, kompresiju slike i prepoznavanje objekata. Ove se tehnike koriste u raznim primjenama kao što su medicinske dijagnostike, daljinsko istraživanje, računalni vid, robotika, industrijske automatizacije, sigurnosni sustavi te kreativne i zabavne industrije. S brzim rastom tehnologije slika visoke rezolucije i videa, obrada slike u stvarnom vremenu postala je sve važnija. Obrada na centralnim procesorskim jedinicama (CPU) često nije dovoljno dobra i brza za zahtjevne zadatke u stvarnom vremenu, zbog čega se sve više koristi paralelna obrada na grafičkim procesorima (GPU).

U ovom seminarskom radu analizira se implementacija box filter algoritma za obradu slika na CPU i na GPU korištenjem CUDA tehnologije. Koristi se jedan od najjednostavnijih i najčešće korištenih filtera pod nazivom box filter. Box filter zaglađuje slike zamjenom svakog piksela prosjekom okolnih susjeda smanjujući šum i stvarajući efekt zamućenja. CPU koristi ugrađene funkcije OpenCV-a za sekvencijalnu obradu, dok GPU koristi CUDA s paralelnim izvršavanjem. CUDA omogućuje da iskoristimo paralelnu procesorsku snagu modernih NVIDIA GPU-ova omogućujući tisućama niti da istovremeno izvršavaju operacije filtriranja na različitim područjima slike. Analizirajući rezultate oba pristupa ističemo prednosti i mane svakog načina. Seminarski rad izveden je na računalu koje koristi procesor AMD Ryzen 7 5700x (8 jezgri, 16 threadova) te grafičku karticu NVIDIA RTX 3070Ti (8gb vRAM).

2 BOX FILTER

2.1 Općenito o Box Filteru

Box filter poznatiji pod nazivom kao filter srednje vrijednosti je linearni filter koji zaglađuje sliku zamjenom svakog piksela prosjekom okolnih susjeda unutar kvadratnog područja tzv. kernel (jezgra). smanjujući šum i stvarajući efekt zamućenja. Kernel je mala matrica brojeva najčešće 3x3, 5x5, 7x7 koja se koristi za filtriranje slike. Svaki element kernela se množi s odgovarajućim pikselom slike, a rezultati se zbrajaju da bi se dobila nova vrijednost piksela. U slučaju box filtera svi elementi kernela imaju istu vrijednost. U našem slučaju koristimo kernel 5x5 pa je svaki element 1/25. Za bilo koji kernel veličine (2r+1) x (2r+1) formula za izračun novog piksela izgleda ovako:

$$g(x,y) = \frac{1}{(2r+1)^2} \sum_{i=-r}^{r} \sum_{j=-r}^{r} f(x+i, y+j)$$

- g(x,y) je nova vrijednost piksela
- f(x+i,y+j): originalna vrijednost susjednih piksela
- ∑: zbrajanje svih piksela u okolini
- $\frac{1}{(2r+1)^2}$: dijeljenje s brojem piksela (računanje srednje vrijednosti)

Na (Slika 2-1) možemo vidjeti primjer primjene box filtera na sliku.

Slika 2-1 Primjena box filtera [1]

Sada ćemo opisati zašto i kako se koristi box filter. Kao što smo prethodno opisali i rekli smanjuje rubove odnosno zaglađuje oštre prijelaze na slici. Koristan je kod računalnog vida gdje služi za predobradu, odnosno prije detekcije rubova, kontura ili objekata, slike se često prvo zaglade kako bi se smanjio utjecaj šuma.

Neke od tehničkih specifikacija box filtera uključuju linearnost što olakšava optimizaciju i implementaciju. Separabilnost gdje se 2D filtriranje može razdvojiti na dvije 1D operacije, prvo horizontalno pa vertikalno čime smanjujemo složenost izračuna. U frekvencijskoj domeni box filter se ponaša kao niskopropusni filter uklanjajući visoke frekvencije. Na slici visoke frekvencije predstavljaju oštre rubove objekata, detalje i šumove te box filter to blokira i propušta niske frekvencije koje predstavljaju glatke promjene te površine bez naglih prijelaza.

Sve ove specifikacije i značajke box filtera mu daju određene prednosti i mane. Jednostavan je za implementaciju te pogodan za brzu izvedbu pogotovo korištenjem CUDA. Koristan je u pripremi slika za daljnju analizu, uklanjanje šumova prije detekcije rubova te efekt „blur“ u grafici i video efektima. Nedostaci su što stvara tzv. „ringing effect“ oko rubova, ne čuva rubove već ih razmazuje te nije toliko precizan u odnosu na neke druge na primjer Gaussian blur.

2.2 Matematička osnova box filtera

Box filtriranje u osnovi koristi 2D konvoluciju, što znači da kernel prelazi preko slike i računa nove vrijednosti svakog piksela prema lokalnom susjedstvu.

$$g(x,y) = (f * h)(x,y) = \sum_{i=-r}^{r} \sum_{j=-r}^{r} h(i,j) f(x+i, y+j)$$

- f(x,y) originalna slika
- g(x,y) filtrirana slika
- h(i,j) vrijednosti kernela (u box filteru sve jednake)

Kao što smo naveli 2D konvolucija se može podijeliti u dvije 1D konvolucije, primjenjujući prvo horizontalnu pa vertikalnu. To nam je korisno kod korištenja kliznog prozora. Umjesto da svaki put ponovno zbrajamo svih 25 piksela u našem slučaju koristi se klizni prozor:

- Oduzima se vrijednost piksela koji „izlazi“ iz prozora
- Dodaje se vrijednost piksela koji „ulazi“

Primjer za horizontalnu liniju:

$$S(x) = \sum_{i=-2}^{2} f(x+i)$$

$$S(x+1) = S(x) - f(x-2) + f(x+3)$$

Ova metoda smanjuje broj operacija i ubrzava izvođenje što je posebno važno kod GPU implementacije.

Primjer izračuna box filtera korištenjem matrice i kernela 3x3 prikazan je na (Slika 2-2).

Image Matrix
[100, 120, 130]
[80, 90, 100]
[60, 70, 80]

Kernel Matrix	
$\frac{1}{/9}$	$\frac{1}{/9}$
$\frac{1}{/9}$	$\frac{1}{/9}$
$\frac{1}{/9}$	$\frac{1}{/9}$

Slika 2-2 Primjer izračuna box filter algoritma

Za svaki piksel, filtrirana vrijednost se računa kao:

$$Novi\ piksel = \sum_{i=-1}^{1} \sum_{j=-1}^{1} f(x+i, y+j) * K(i,j)$$

U našem slučaju je to ovako:

$$g(x,y) = \frac{1}{9}(100 + 120 + 130 + 80 + 90 + 100 + 60 + 70 + 80)$$

Zbroj svih piksela je:

$$100 + 120 + 130 + 80 + 90 + 100 + 60 + 70 + 80 = 830$$

Rezultat:

$$g(x,y) = \frac{830}{9} \approx 92.22$$

Dakle, središnji piksel koji je bio 90 sada postaje 92.22.

3 CUDA TEHNOLOGIJA

CUDA je platforma za paralelno računanje i programski model koji je kreirala tvrtka NVIDIA. S više od 20 milijuna preuzimanja do danas, CUDA pomaže programerima da ubrzaju svoje aplikacije iskorištavanjem snage GPU akceleratora. CUDA omogućuje korištenje grafičke kartice (GPU) za opće programske zadatke, čime se otvara mogućnost izvođenja znatno bržih izračuna u usporedbi s tradicionalnim CPU-based pristupom.

U mnogim područjima poput obrade slike, znanstvenih simulacija, umjetne inteligencije i mnogih drugih, CUDA je postala nezaobilazna tehnologija. U kontekstu box filtra: CPU obrađuje piksel po piksel redom, dok GPU može istovremeno obraditi tisuće piksela s pomoću CUDA threadova, čime se postiže višestruko ubrzanje.

CUDA omogućava programeru da napiše program u jeziku poput C/C++ s CUDA ekstenzijama. Osnovna ideja je da se određeni dijelovi programa, tzv. kerneli, izvršavaju paralelno na GPU-u. [2]

Ključni pojmovi:

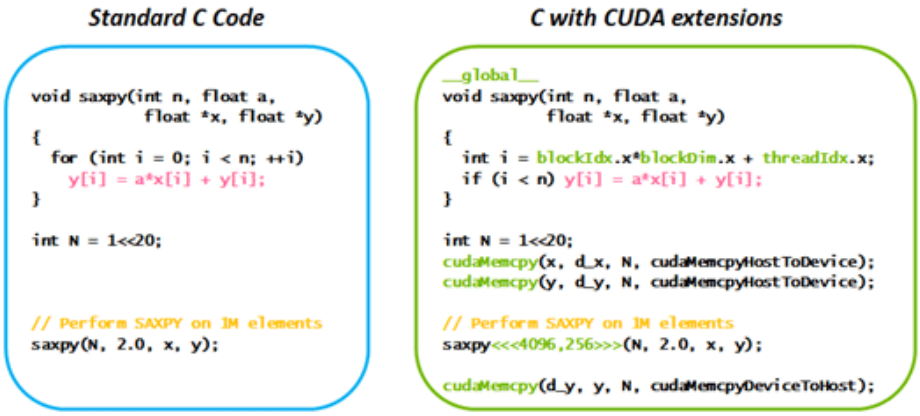
- **Kernel** – funkcija koja se izvršava na GPU-u; svaka nit (thread) izvršava istu funkciju ali nad različitim podacima.
- **Thread** – pojedinačna jedinica izvršavanja na GPU-u.
- **Block** – skup threadova; svaki block može imati npr. 256 ili 1024 threadova.
- **Grid** – organizacija više blokova koji zajedno čine cijeli radni zadatak.
- **Shared memory** – brza memorija unutar jednog bloka koju dijele threadovi.
- **Global memory** – sporija, ali veća memorija dostupna svim threadovima.

Prednosti CUDA pristupa:

1. **Veliko ubrzanje obrade**
    - U zadacima kao što je box filter nad velikim slikama, CUDA omogućuje višestruko ubrzanje (10x, 50x, pa i više).

2. **Pristup paralelizmu na niskoj razini**
    - Programer može upravljati raspodjelom memorije, rasporedom threadova, i optimizirati performanse.
3. **Fleksibilnost**
    - CUDA omogućuje obradu ne samo slika, nego i drugih oblika podataka – znanstveni izračuni, simulacije, itd.

Jednostavan primjer u nastavku pokazuje kako se standardni C program može ubrzati pomoću CUDA-e.



Slika 3-1 Primjer kako se kod u C-u može ubrzati korištenjem CUDA [2]

4 IMPLEMENTACIJA NA CPU

Implementacija na CPU koristi OpenCV za primjenu box filtera. OpenCV nudi optimizirane funkcije za obradu slika. U ovom slučaju, koristi se ugrađena funkcija cv::boxFilter za izvođenje operacije box filtera na CPU-u.

```
cv::Mat applyBoxFilterCPU(const cv::Mat& image, int kernelSize) {
    std::cout << "Primjena Box filtera na CPU-u..." << std::endl;
    cv::Mat outputImage;
    cv::boxFilter(image, outputImage, -1, cv::Size(kernelSize, kernelSize));
    return outputImage;
}
```

Slika 4-1 applyBoxFilterCPU funkcija

Argument funckije cv::Mat Image označava ulaznu sliku na koju se primjenjuje filter. Drugi argument je kernelSize koji predstavlja dimenziju kernela u ovom slučaju će biti 5x5. OutputImage označava izlaznu sliku gdje će se spremiti rezultat. U funkciji boxFilter -1 označava da izlazna slika ima istu dubinu kao i ulazna. Primjenom boxFilter-a funkcija vraća obrađenu sliku.

```
std::vector<float> createBoxKernel(int kernelSize) {
    std::vector<float> kernel(kernelSize * kernelSize);
    float kernelValue = 1.0f / (kernelSize * kernelSize);

    for (int i = 0; i < kernelSize * kernelSize; ++i) {
        kernel[i] = kernelValue;
    }
    return kernel;
}
```

Slika 4-2 createBoxKernel funkcija

Svrha funkcije createBoxKernel je generirati standardni box kernel gdje su svi elementi jednaki 1/ (kernelSize x kernelSize). Za primjer kernela 5x5 svaka vrijednost će biti 1/25.

```
void processImage(const cv::Mat& image, const std::string& imageName, int kernelSize) {
    int rows = image.rows;
    int cols = image.cols;
    int channels = image.channels();
    size_t imageSizeInBytes = rows * cols * channels;

    std::cout << "\n--- " << imageName << " CPU Box Filter ---\n";
    std::cout << "Dimenzije: " << rows << "x" << cols << " kanala: " << channels << std::endl;

    auto startCpu = std::chrono::high_resolution_clock::now();
    cv::Mat cpuResult = applyBoxFilterCPU(image, kernelSize);
    auto endCpu = std::chrono::high_resolution_clock::now();
    auto cpuTime = std::chrono::duration_cast<std::chrono::microseconds>(endCpu - startCpu).count();
    std::cout << "CPU vrijeme: " << cpuTime / 1000.0 << " ms\n";

    std::string cpuOutputPath = "../results/box_cpu_" + imageName + ".jpg";
    cv::imwrite(cpuOutputPath, cpuResult);
```

Slika 4-3 processImage funkcija

Ova funkcija koordinira cijeli proces obrade, uključujući mjerenje vremena i spremanje rezultata za CPU dio.

Dohvaćanje informacija o slici se vrši kroz prve tri linije koda funkcije gdje dobijamo podatke o visini, širini i broju kanala ulazne slike. size_t imageSizeInBytes izračunava ukupnu veličinu slike u bajtovima. Varijable startCpu, endCpu koriste biblioteku std::chrono za precizno mjerenje početka i kraja izvršavanja CPU operacije. Kod linije koja sadržava cv::Mat cpuResult se poziva prethodno opisana funkcija applyBoxFilterCPU koja obavlja filtriranje na CPU-u. Ispisujemo CPU vrijeme te se sa cpuOutputPath definira putanja za spremanje rezultirajuće CPU obrađene slike. Zadnja linija koda koristi imwrite funkciju za spremanje cpuResult slike na disk.

```cpp
int main() {
    std::string imagePath = "../assets/drone_shot.jpg";
    std::cout << "\nUcitavanje slike: " << imagePath << std::endl;
    cv::Mat image = cv::imread(imagePath, cv::IMREAD_COLOR);

    if (image.empty()) {
        std::cerr << "Greska: Slika se ne moze ucitati.\n";
        return EXIT_FAILURE;
    }

    // Kernel size - 5x5
    int kernelSize = 5;

    // Process color version
    std::cout << "\n========== OBRADA SLIKE U BOJI ==========\n";
    processImage(image, "color", kernelSize);

    // Process grayscale version
    std::cout << "\n========== OBRADA SLIKE U SIVIM TONOVIMA ==========\n";
    cv::Mat grayImage;
    cv::cvtColor(image, grayImage, cv::COLOR_BGR2GRAY);
    processImage(grayImage, "grayscale", kernelSize);

    return 0;
}
```

Slika 4-4 Glavna funkcija main

Varijabla imagePath definira putanju do ulazne slike. cv::Mat image = cv::imread(imagePath, cv::IMREAD_COLOR) je OpenCV funkcija koja služi za učitavanje slike s diska u cv::Mat objekt. cv::IMREAD_COLOR osigurava da se slika učitava sa 3 kanala.

Nakon toga imamo definiranu varijablu kernelSize koja postavlja veličinu kernela na 5x5 za box filter. To znači da se u izračunu novog piksela uzima prosjek 25 okolnih piksela. Zatima pozivamo processImage funkciju koja se poziva sa originalnom slikom te će CPU izvršiti funkciju applyBoxFilterCPU na ovoj slici.

Na kraju deklariramo cv::Mat grayImage za grayscale verziju slike. Pomoću OpenCV funkcije cv::cvtColor konvertiramo originalnu sliku u boji u sliku u sivim tonovima. Slika u sivim tonovima ima samo jedan kanal. Na kraju pozivamo processImage funkciju još jednom ali sada šaljemo sliku u sivim tonovima.

CPU implementacija se oslanja na serijsku, piksel po piksel obradu koju efikasno obavlja optimizirana OpenCV biblioteka.

5 IMPLEMENTACIJA NA GPU – CUDA

5.1 Priprema Okruženja

Za potrebe ove CUDA implementacije korištena je NVIDIA GeForce RTX 3070 Ti grafička kartica s 8 GB VRAM-a. Na početku koda uključujemo zaglavlje #include <cuda_runtime.h>, koje omogućuje pristup CUDA Runtime API-ju. Ono omogućuje korištenje osnovnih funkcija za upravljanje memorijom, pokretanje kernel funkcija, upotrebu optimiziranih matematičkih funkcija te jednostavnu komunikaciju između CPU i GPU dijela aplikacije.

```
// Kernel size - 5x5
int kernelSize = 5;

// Process color version
std::cout << "\n========== OBRADA SLIKE U BOJI ==========\n";
processImage(image, "color", kernelSize);

// Process grayscale version
std::cout << "\n========== OBRADA SLIKE U SIVIM TONOVIMA ==========\n";
cv::Mat grayImage;
cv::cvtColor(image, grayImage, cv::COLOR_BGR2GRAY);
processImage(grayImage, "grayscale", kernelSize);
```

Slika 5-1 poziv funkcije processImage()

Nakon učitavanja slike te određivanje broja elemenata kernela (Slika 5-1) pozivamo funkciju processImage() za sliku u boji i sivim tonovima (grayscale).

```
void processImage(const cv::Mat& image, const std::string& imageName, int kernelSize) {
    int rows = image.rows;
    int cols = image.cols;
    int channels = image.channels();
    size_t imageSizeInBytes = rows * cols * channels;
```

Slika 5-2 funkcija processImage() CUDA 1

Funkcija processImage() (Slika 5-2) kao argumente prima referencu na ulaznu sliku(const cv::Mat& image), koja omogućuje učinkoviti pristup bez kopiranja, varijablu imageName koja označava je li slika grayscale ili color te veličinu kernela.

U početnoj fazi funkcija dohvaća broj redaka i stupaca slike te broj kanala koji će za grayscale biti 1, a za color slike 3. Na temelju tih podatak se računa veličina slike u bajtovima, što je ključno za pravilnu alokaciju memorije na GPU-u.

```
std::cout << "\n--- " << imageName << " CUDA Box Filter ---\n";
unsigned char* d_input = nullptr;
unsigned char* d_output = nullptr;
float* d_kernel = nullptr;

CUDA_CHECK(cudaMalloc(&d_input, imageSizeInBytes));
CUDA_CHECK(cudaMalloc(&d_output, imageSizeInBytes));

CUDA_CHECK(cudaMemcpy(d_input, image.data, imageSizeInBytes, cudaMemcpyHostToDevice));

std::vector<float> h_kernel = createBoxKernel(kernelSize);
size_t kernelBytes = h_kernel.size() * sizeof(float);

CUDA_CHECK(cudaMalloc(&d_kernel, kernelBytes));
CUDA_CHECK(cudaMemcpy(d_kernel, h_kernel.data(), kernelBytes, cudaMemcpyHostToDevice));

dim3 threads(16, 16);
dim3 blocks((cols + 15) / 16, (rows + 15) / 16);

cudaEvent_t startEvent, endEvent;
CUDA_CHECK(cudaEventCreate(&startEvent));
CUDA_CHECK(cudaEventCreate(&endEvent));
CUDA_CHECK(cudaEventRecord(startEvent, 0));
```

Slika 5-3 funkcija processImage() CUDA 2

Kod (Slika 5-3) postavlja CUDA okruženje i priprema podatke za obradu na GPU-u. Stvaraju se 3 pokazivača za GPU označena s d_. Pomoću cudaMalloc se alocira memorija za ulaznu i izlaznu sliku. Funkcija cudMemcpy služi za kopiranje ulazne slike s hosta (CPU) na device (GPU).

Koristeći prethodno opisani createBoxKernel, kreiramo kernel na CPU. Nakon alokacije potrebne memorije, kernel se kopira na GPU-u. Nakon toga, postavlja se konfiguracija za izvršavanje kernela gdje svaki blok sadrži 16x16 threads-a. Dimenzije grida se računaju tako da se pokrije cijela slika, čak i ako dimenzija slike nije djeljiva sa 16.

Kreira se cudaEvent koji služi za precizno mjerenje vremena izvršavanja.

5.2 Korištenje Box Filtera

```
__global__ void boxFilterKernel(const unsigned char* inputImage,
    unsigned char* outputImage,
    int rows, int cols, int channels,
    const float* kernel, int kernelSize) {

    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int offset = kernelSize / 2;

    if (row < rows && col < cols) {
        // Process each channel
        for (int ch = 0; ch < channels; ++ch) {
            float pixelSum = 0.0f;

            for (int ky = 0; ky < kernelSize; ++ky) {
                for (int kx = 0; kx < kernelSize; ++kx) {
                    int imgY = row + (ky - offset);
                    int imgX = col + (kx - offset);

                    // Handle border by clamping
                    imgY = max(0, min(rows - 1, imgY));
                    imgX = max(0, min(cols - 1, imgX));

                    pixelSum += static_cast<float>(
                        inputImage[(imgY * cols + imgX) * channels + ch]) *
                        kernel[ky * kernelSize + kx];
                }
            }

            unsigned char finalPixelValue = static_cast<unsigned char>(
                fminf(255.0f, fmaxf(0.0f, pixelSum)));

            outputImage[(row * cols + col) * channels + ch] = finalPixelValue;
        }
    }
}
```

Slika 5-4 funkcija boxFilterKernel()

Funkcija boxFilterKernel() (Slika 5-4) predstavlja srž GPU implementacije box filtera. Pokreće se kao 2D mreža thread-ova, pri čemu svaki thread obrađuje jedan piksel slike. Argumenti funkcije uključuju pokazivače na ulaznu i izlaznu sliku na GPU-u, dimenzije slike, broj kanala, kernel s filtrirajućim vrijednostima te veličinu kernela.

Svaki thread izračunava svoju poziciju na slici te prolazi kroz susjedstvo piksela unutar zadanog kernela. Za rubne dijelove slike koristi se clamping kako bi se izbjeglo izlazak iz granica slike. U svakom kanalu se neovisno računa konvolucija tako da se vrijednosti piksela množe s

pripadajućim težinama kernela, a zatim zbrajaju. Nakon toga se rezultat ograničava na raspon 0–255 i zapisuje u odgovarajuću poziciju izlazne slike.

Ovakav pristup omogućuje paralelnu obradu svakog piksela i maksimalno iskorištavanje GPU arhitekture. Nakon izvršavanja, mjeri se vrijeme obrade, rezultati se kopiraju s GPU-a natrag na CPU, te se obrađena slika sprema na disk.

5.3 Čišćenje Memorije

```
CUDA_CHECK(cudaFree(d_input));
CUDA_CHECK(cudaFree(d_output));
CUDA_CHECK(cudaFree(d_kernel));
CUDA_CHECK(cudaEventDestroy(startEvent));
CUDA_CHECK(cudaEventDestroy(endEvent));
```

Slika 5-5 Čišćenje memorije

Pomoću cudaFree() (Slika 5-5) oslobađa se memorija koja je prethodno alocirana za ulaznu sliku (d_input), izlaznu sliku (d_output) i kernel (d_kernel). Također, cudaEventDestroy() uklanja CUDA event-ove korištene za mjerenje vremena.

Ova vrsta čišćenja je važna kako bi se spriječili memory leakovi, oslobodili resursi za druge programe i osigurala stabilnost i pouzdanost aplikacije.

6 REZULTATI I ANALIZA

6.1 Testiranje Box Filtera

Slika 6-1 Testiranje Box Filtera 1

Slika 6-2 Testiranje Box Filtera 2

Slika 6-1 i Slika 6-2 prikazuju 2 slike na kojima je testiran Box Filter implementiran s pomoću CUDA-e u boji. Kvaliteta slika se u ovom dokumentu ne može vidjeti, za punu kvalitetu postoji dokumentacija [3].

6.2 Rezultati Mjerenja

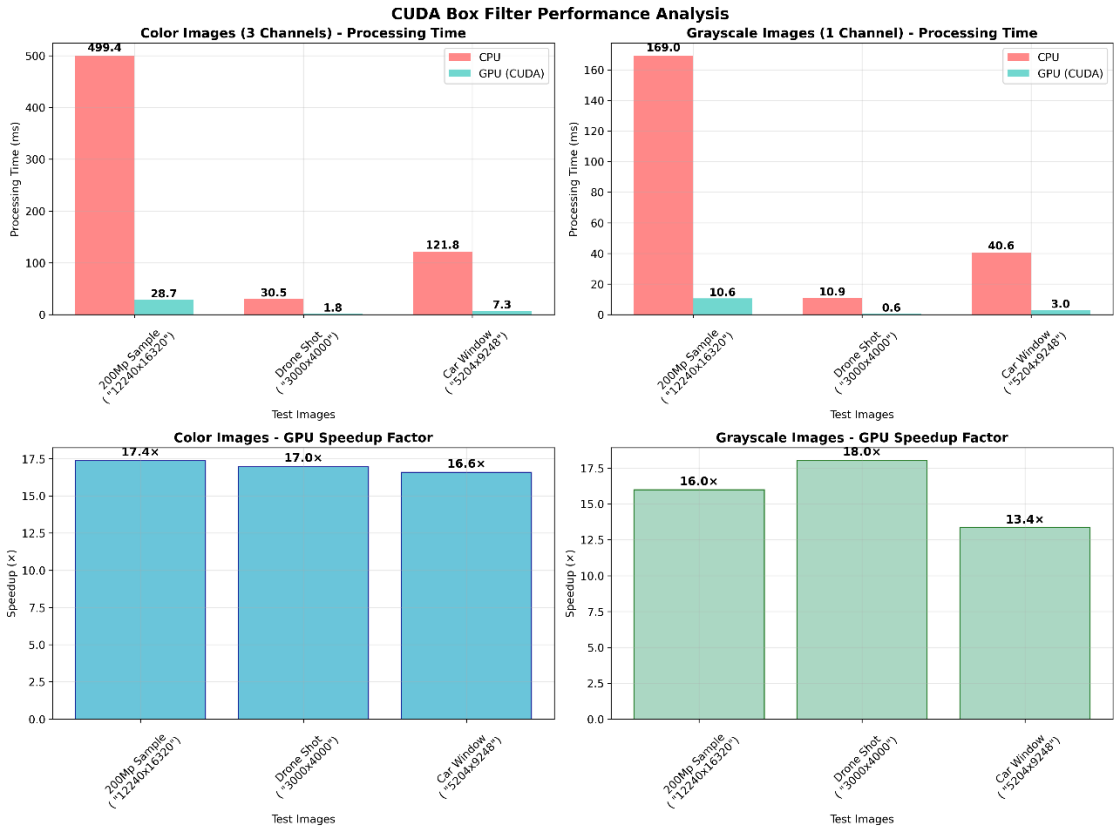
Color Images (3 Channels)

Image	Dimensions	CPU Time (ms)	GPU Time (ms)	Speedup
200MP_sample.jpg	12240×16320	499.45	28.74	**17.38×**
car_window.jpg	5204×9248	121.83	7.35	**16.58×**
drone_shot.jpg	3000×4000	30.54	1.80	**16.97×**

Grayscale Images (1 Channel)

Image	Dimensions	CPU Time (ms)	GPU Time (ms)	Speedup
200MP_sample.jpg	12240×16320	168.99	10.57	**15.99×**
car_window.jpg	5204×9248	40.63	3.04	**13.36×**
drone_shot.jpg	3000×4000	10.93	0.61	**18.03×**

Slika 6-3 Tablica usporebe brzine CPU i GPU implementacije



Slika 6-4 Grafički prikaz usporedbe

6.3 Analiza

Obje implementacije, CPU i GPU, uspješno su izvršile zadatak primjene box filtera na više slika, kako u boji tako i u grayscale formatu. Međutim, rezultati mjerenja (Slika 6-3) jasno pokazuju značajnu prednost GPU implementacije zahvaljujući paralelizmu koji omogućuje istovremenu obradu velikog broja piksela. U prosjeku je GPU bio oko 16 puta brži od CPU-a, pri čemu su najveće slike ostvarile i najveće ubrzanje. Primjerice, kod slike 200MP_sample.jpg u boji, GPU je bio 17.38 puta brži, dok je kod iste slike u grayscale formatu ostvareno 16 puta ubrzanje. Uočena je i blago veća razlika u vremenu obrade kod slika u boji, što je očekivano s obzirom na tri kanala koje GPU može paralelno obraditi. Također, performanse GPU-a sve više dolaze do izražaja kako raste dimenzija slike. To potvrđuje i slučaj s najvećom slikom (12240×16320 piksela), koja je ostvarila najveće ubrzanje jer GPU može optimalno iskoristiti velik broj thread-ova za obradu svakog piksela.

Ovo je očekivano jer box filter koristi jednostavne, repetitivne operacije koje svaki GPU thread može efikasno i neovisno izvesti. Stoga je ova implementacija izvrstan primjer kako paralelizam na GPU-u može višestruko ubrzati obradu slike u odnosu na serijsku CPU obradu.

7 ZAKLJUČAK

U ovom radu uspješno je prikazana implementacija box filtera na CPU-u i GPU-u, s jasnim naglaskom na prednosti paralelne obrade slike s pomoću CUDA tehnologije. Rezultati su potvrdili da GPU implementacija višestruko nadmašuje CPU izvedbu, posebno kod velikih slika. Prosječno ubrzanje od oko 16 puta jasno pokazuje koliko se računalna snaga GPU-a može iskoristiti za jednostavne, ali frekventne operacije poput filtriranja slike. Veće slike, zbog većeg broja piksela, bolje iskorištavaju paralelizam i pokazuju najveće performanse. Uočena razlika u vremenu obrade između grayscale i color slika dodatno potvrđuje učinkovitost CUDA pristupa kod složenijih ulaza. Ova implementacija pokazuje koliko je GPU pogodniji za zadatke koji zahtijevaju obradu velikih količina podataka u stvarnom vremenu. Box filter je odličan primjer jednostavnog algoritma koji, kada se paralelizira, može dramatično smanjiti vrijeme izvođenja.

LITERATURA

[1] Shashika Dilhani (2021). Digital Image Processing Filters, s Interneta: https://medium.com/@shashikadilhani97/digital-image-processing-filters-832ec6d18a73, zadnji pristup: 25.06.2025.

[2] Fred Oh (2012). What Is CUDA?, s Interneta: https://blogs.nvidia.com/blog/what-is-cuda-2/, zadnji pristup: 25.06.2025.

[3] Dorić, Josipović i Stojanac, Github: https://github.com/mjospovich/box-filter-cuda/tree/main

PRILOZI

Kazalo slika