Univerzitet u Novom Sadu

Fakultet tehničkih nauka

Dokumentacija projektnog zadatka

Student: Živanović Marija, SV 19/2021

Predmet: Paralelno programiranje

Broj projektnog zadatka: 1

Tema projektnog zadatka: Analiza dve implementacije serijskog i paralelnog algoritma za detekciju ivica ulazne slike u programskom jeziku C++

Datum: 07.06.2023.

Sadržaj

1. Analiza problema

Tema ovog projekta jeste detekcija ivica slika zadatih u BMP formatu, pomoću operacija dvodimenzionalne diskretne konvolucije.

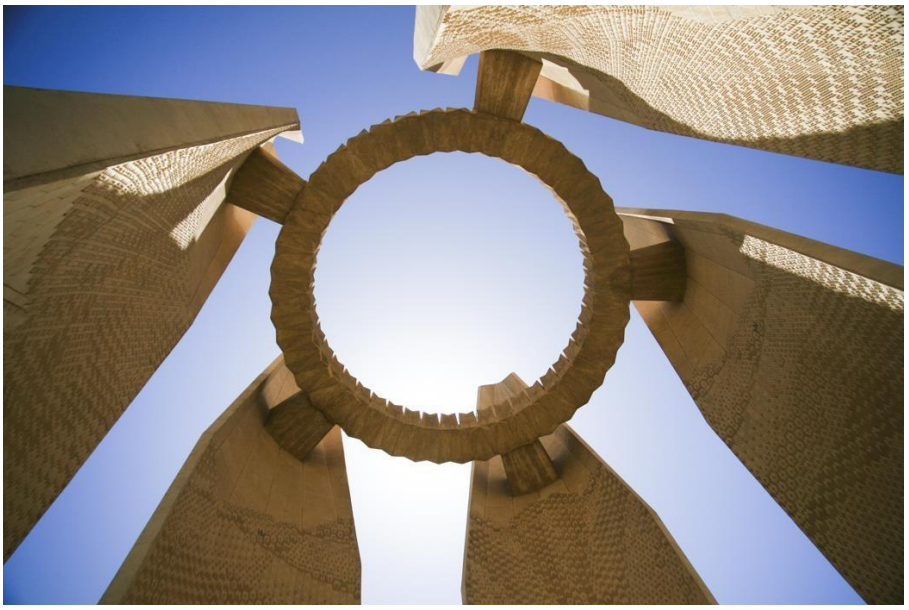
U kontekstu slika, dvodimenzionalna diskretna konvolucija se primenjuje na matricu piksela slike (ulazna slika) i jezgro ili filter. Jezgro je manja matrica koja se kreće po ulaznoj slici, a na svakom koraku se elementi jezgra množe s odgovarajućim elementima piksela ulazne slike. Rezultat množenja se zatim sabira kako bi se dobio jedan rezultujući piksel. Ovaj proces se ponavlja za svaki piksel na slici. Ova metoda omogućuje razne manipulacije slikom među kojima je i problem detekcije ivica.

Ivice predstavljaju područja slike sa velikim razlikama u intenzitetu tačaka i označavaju granice objekata. To se može iskoristiti za prepoznavanje objekata, detekciju položaja objekta u slici i detekciju orijentacije objekta.

Za rešavanje ovog problema trebalo je primeniti dva algoritma za detekciju ivica: algoritam za detekciju ivica slike pomoću Prewitt operatora i algoritam za detekciju ivica slike posmatranjem okoline piksela.

Oba algoritma obrađena u ovom projektu se svode na množenje odgovarajućih elemenata dveju matrica i sumiranje rezultata. Prva matrica je jezgro digitalnog filtra, uglavnom je dimenzija 3x3, 5x5, 7x7, a može biti i veća. Druga matrica (X) je podmatrica slike koja sadrži podatke o vrednostima boja pojedinačnih piksela ulazne slike, i istih je dimenzija kao filter. Rezultat operacija izvršenih nad ovim matricama se smešta u treću matricu (Y).

2. Koncept rešenja



Slika 1 - Ulazna slika

Slika 2 - Ulazna slika

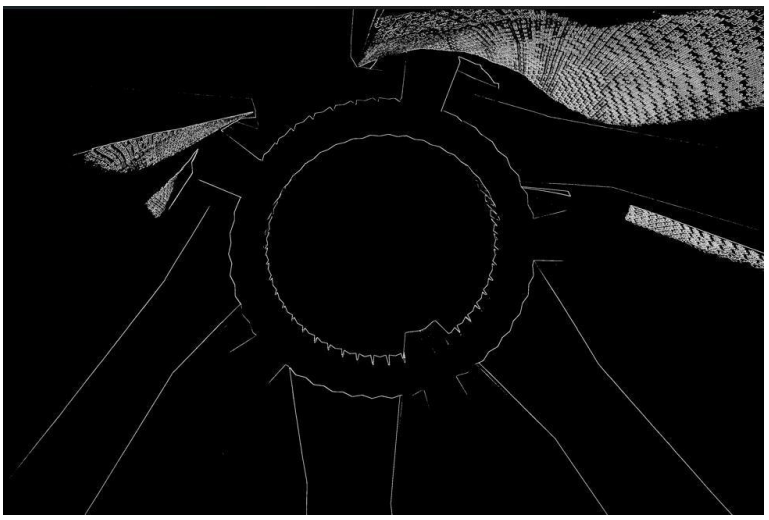
2.1. Algoritam za detekciju ivica slike pomoću Prewitt operatora

Kod Prewitt algoritma podmatrica ulazne slike se množi sa filterima za vertikalne, odnosno horizontalne ivice, a krajnji rezultat se dobija kao zbir apsolutnih vrednosti ove dve komponente.

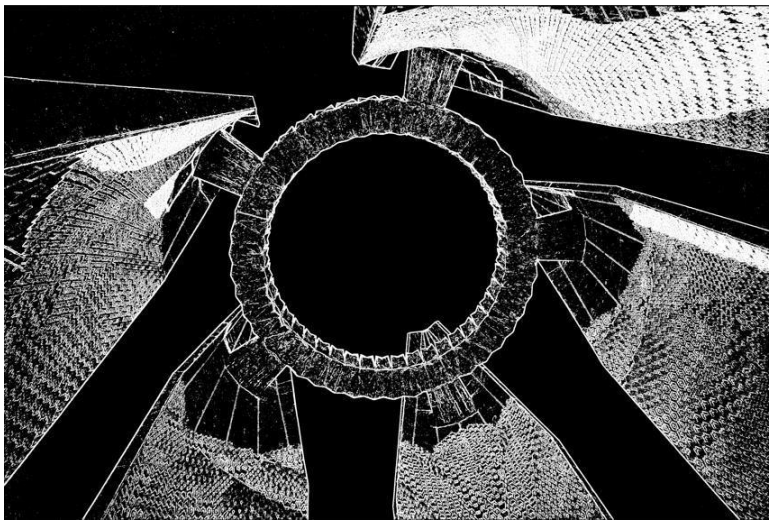
$$Y[x,y] = \sum_{n=0}^{2} \sum_{m=0}^{2} X[x-1+m, y-1+n] * F[m,n]$$

$$Gx = \begin{bmatrix} -1 & 0 & +1 \\ -1 & 0 & +1 \\ -1 & 0 & +1 \end{bmatrix} * A \qquad Gy = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ +1 & +1 & +1 \end{bmatrix} * A$$

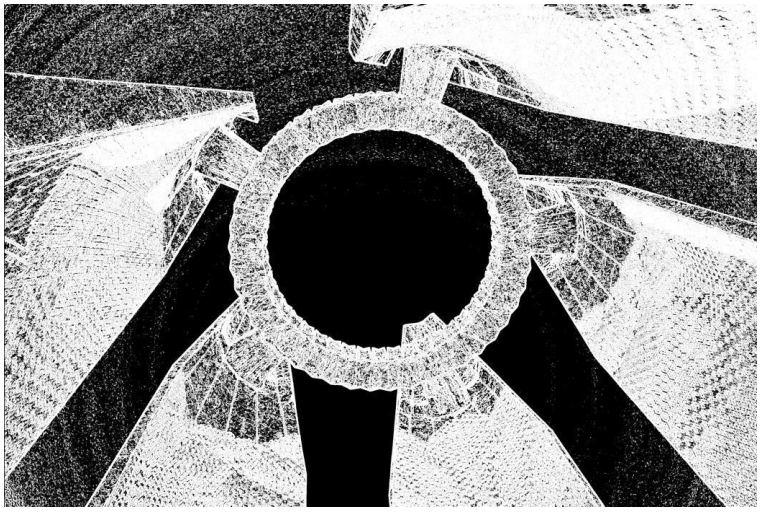
$$|G| = |Gx| + |Gy|$$



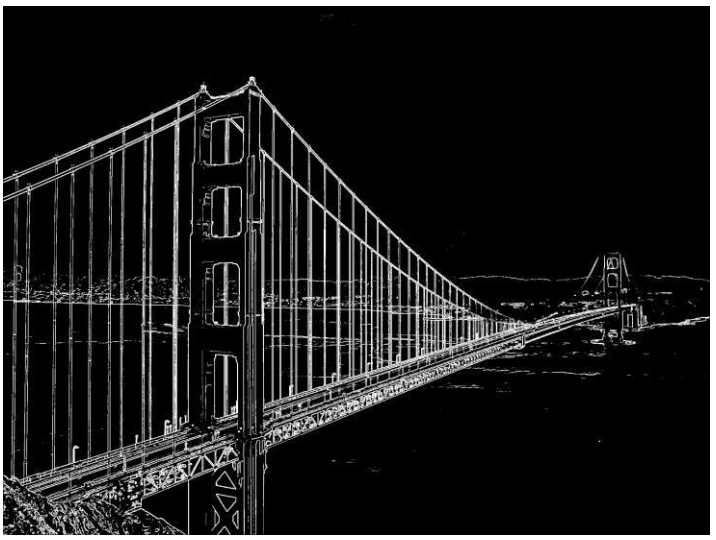
Slika 3 – Izlazna slika detekcije ivica pomoću Prewitt operatora sa filterom 3x3



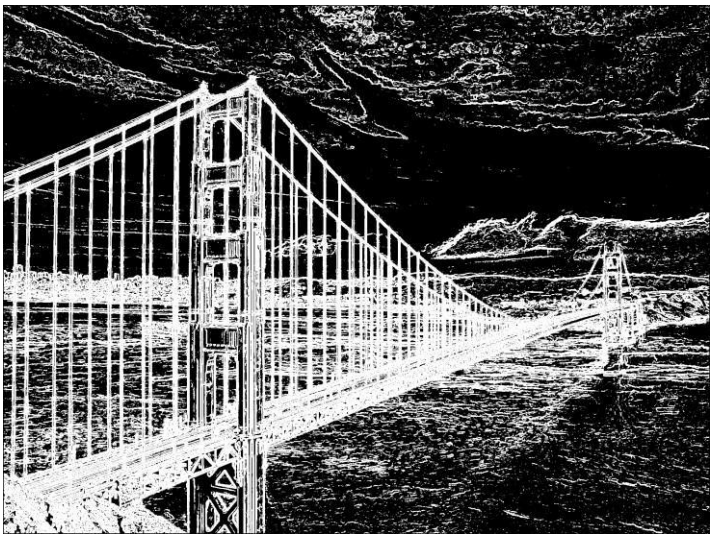
Slika 4 – Izlazna slika detekcije ivica pomoću Prewitt operatora sa filterom 5x5



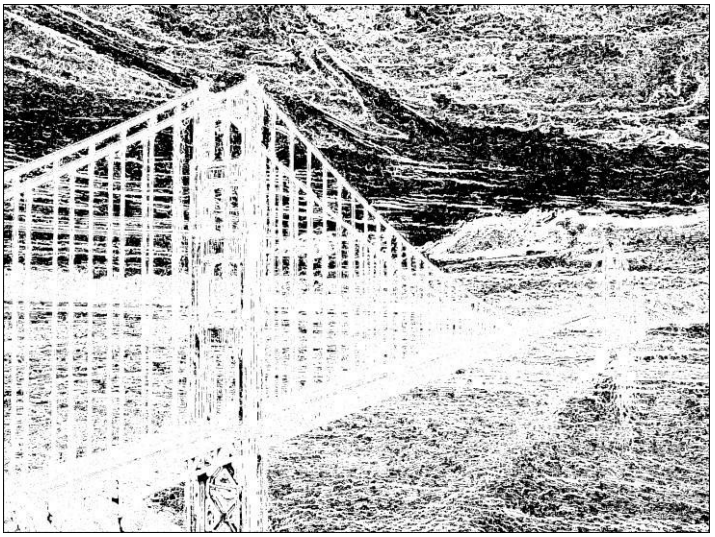
Slika 5 – Izlazna slika detekcije ivica pomoću Prewitt operatora sa filterom 7x7



Slika 6 – Izlazna slika detekcije ivica pomoću Prewitt operatora sa filterom 3x3



Slika 7 - Izlazna slika detekcije ivica pomoću Prewitt operatora sa filterom 5x5



Slika 8 - Izlazna slika detekcije ivica pomoću Prewitt operatora sa filterom 7x7

2.2. Algoritam za detekciju ivica slike posmatranjem okoline piksela

Kod drugog algoritma se posmatra crno-bela slika, ukoliko se detektuju ivice na slici u boji potrebno je vrednosti njenih piksela postaviti na 0 i 255, odnosno ulaznu sliku u boji konvertovati u crno-belu. Zatim se gleda okolina svakog piksela i na osnovu broja crnih, odnosno belih piksela u toj okolini se dobijaju dva rezultata. I to na sledeći način:

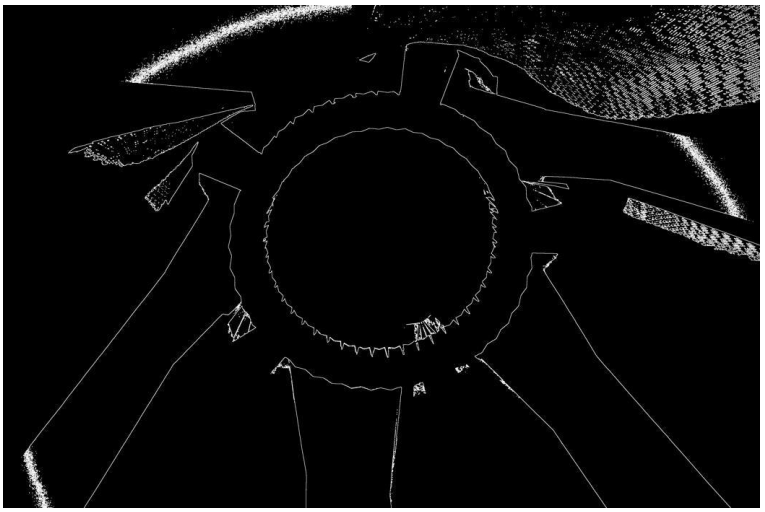
P(i,j) = 1, ako u okolini tačke postoji tačka sa vrednošću 1

P(i,j) = 0, ako u okolini tačke ne postoji tačka sa vrednošću

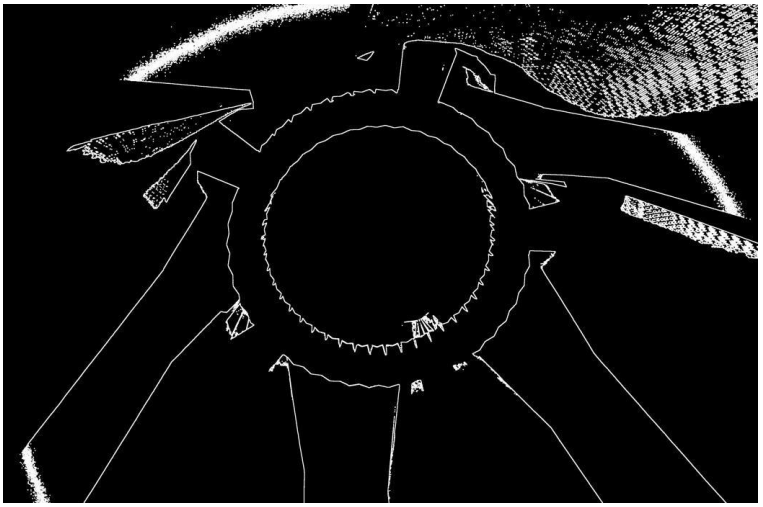
1 O(i,j) = 0, ako u okolini tačke postoji tačka sa vrednošću 0

O(i,j) = 1, ako u okolini tačke ne postoji tačka sa vrednošću 0

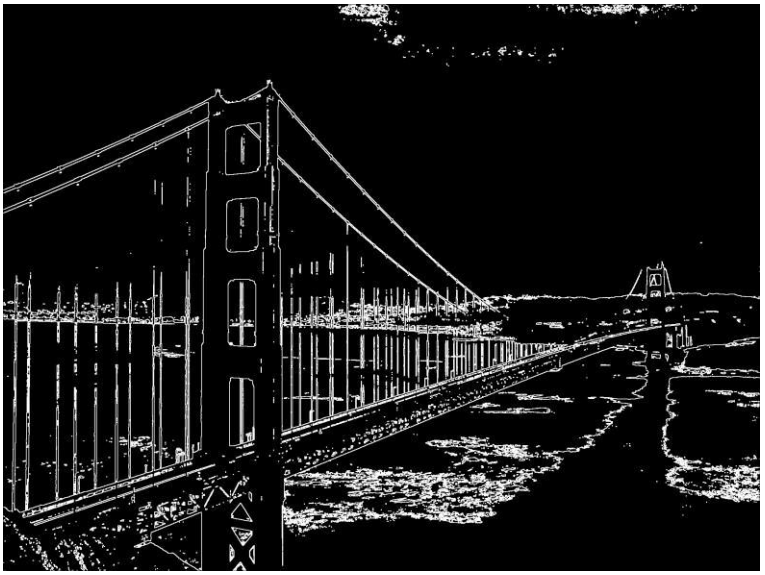
Krajnji rezultat posmatranog piksela jeste 0 ukoliko su O i P jednaki, odnosno 255 ako su O i P različiti.



Slika 9 – Izlazna slika detekcije ivica posmatranjem okoline sa filterom 3x3



Slika 10 – Izlazna slika detekcije ivica posmatranjem okoline sa filterom 5x5



Slika 11 – Izlazna slika detekcije ivica posmatranjem okoline sa filterom 3x3

Slika 12 – Izlazna slika detekcije ivica posmatranjem okoline sa filterom 5x5

2.3. Paralelizacija oba algoritma

Pri paralelizaciji serijskog algoritma korišćena je apstrakcija zadataka, obezbeđena od strane Intel-ove Thread Building Blocks biblioteke.

U početku se formira jedan zadatak kojem se prosleđuje cela matrica ulazne slike. Ako je veličina matrice veća od unapred zadate vrednosti, ovaj zadatak formira nova četiri zadatka kojima se prosleđuju odgovarajuće podmatrice polazne matrice.

3. Programsko rešenje

Pre nego što se pozove bilo koje rešenje detekcije ivica, pozove se funkcija generateFilter, koja u zavisnosti od prosleđenog broja (int n) generiše filter. Radi pojednostavljenja koda filteri 3x3, 5x5, 7x7, … su vrednosti udaljenosti piksela od posmatranog piksela po x, odnosno y osi.

```
-1  0  1        -1  -1  -1          -2  -1  0  1   2        -2  -2  -2  -2  -2
-1  0  1         0   0   0          -2  -1  0  1   2        -1  -1  -1  -1  -1
-1  0  1         1   1   1          -2  -1  0  1   2         0   0   0   0   0

filterHor      filterVer            -2 -1  0  1 2            1   1 1   1     1
                                    -2 -1  0  1   2          2   2 2 2       2
                                    filterHor5x5            filterVer5x5
```

Da bi se testirala sva 4 rešenja pozivamo datu funkciju run_test_nr, u kojoj se vrši merenje vremena potrebnog da se svako od 4 rešenja izvrši. Vreme se meri pomoću TBB-ovog tick_count-a.

3.1. Algoritam za detekciju ivica slike pomoću Prewitt operatora

Serijska implementacija

Kod serijskog rešenja ovog algoritma vrši se prolazak kroz svaki piksel, osim ivičnih, redom. Parametri koji se prosleđuju ovoj funkciji su bafer ulazne i bafer izlazne slike, širina i visina slike. Ulazni bafer (inBuffer) sadrži sliku čiji pikseli se obrađuju dok izlazni bafer (outBuffer) sadržati rezultujuću sliku sa primenjenim filterom.

```
void filter_serial_prewitt(int* inBuffer, int* outBuffer, int width, int height)  //TODO obrisati
{

    for (int i = FILTER_SIZE / 2; i < width - FILTER_SIZE / 2; i++) {
        for (int j = FILTER_SIZE / 2; j < height - FILTER_SIZE / 2; j++) {
            int Gx = 0;
            int Gy = 0;
            int G = 0;
            for (int k = 0; k < FILTER_SIZE * FILTER_SIZE; k++) {
                Gx += inBuffer[(j + filterHor[k]) * width + (i + filterVer[k])] * filterHor[k];         //horizontalna komponentna
                Gy += inBuffer[(j + filterHor[k]) * width + (i + filterVer[k])] * filterVer[k];         //vertikalna komponenta
            }

            G = abs(Gx) + abs(Gy);
            outBuffer[j * width + i] = (G < THRESHOLD) ? 0 : 255;                    //0 (crna boja), 255 (bela)
        }
    }

}
```

Slika 13 – Serijsko rešenje algoritma

Kao što je već spomenuto, algoritam prolazi kroz svaki piksel slike, osim ivičnih piksela, i primenjuje filter Prewit. Za svaki piksel, izračunava se horizontalna komponentna (Gx) kao i vertikalna komponenta (Gy). Zatim se izračunava ukupni zbir (G) kao suma apsolutnih vrednosti komponenti Gx i Gy.

Na kraju, rezultat se binarizuje na osnovu zadate granice (THRESHOLD). Ako je ukupni zbir manji od zadate granice, piksel koji se obrađuje će biti postavljen na 0 (crna boja), u suprotnom će biti postavljen na 255 (bela boja).

Da bi se izbegla obrada ivičnih piksela (koji ne bi imali dovoljan broj piksela oko sebe da se primeni cela maska filtera), početak i kraj unutrašnje petlje ograničeni su sa FILTER_SIZE / 2 i width - FILTER_SIZE / 2, odnosno FILTER_SIZE / 2 i height - FILTER_SIZE / 2.
Na taj način se osigurava da se filter primenjuje samo na piksele unutar slike, odnosno samo na piksele koji imaju dovoljan broj susednih piksela u svim smerovima.

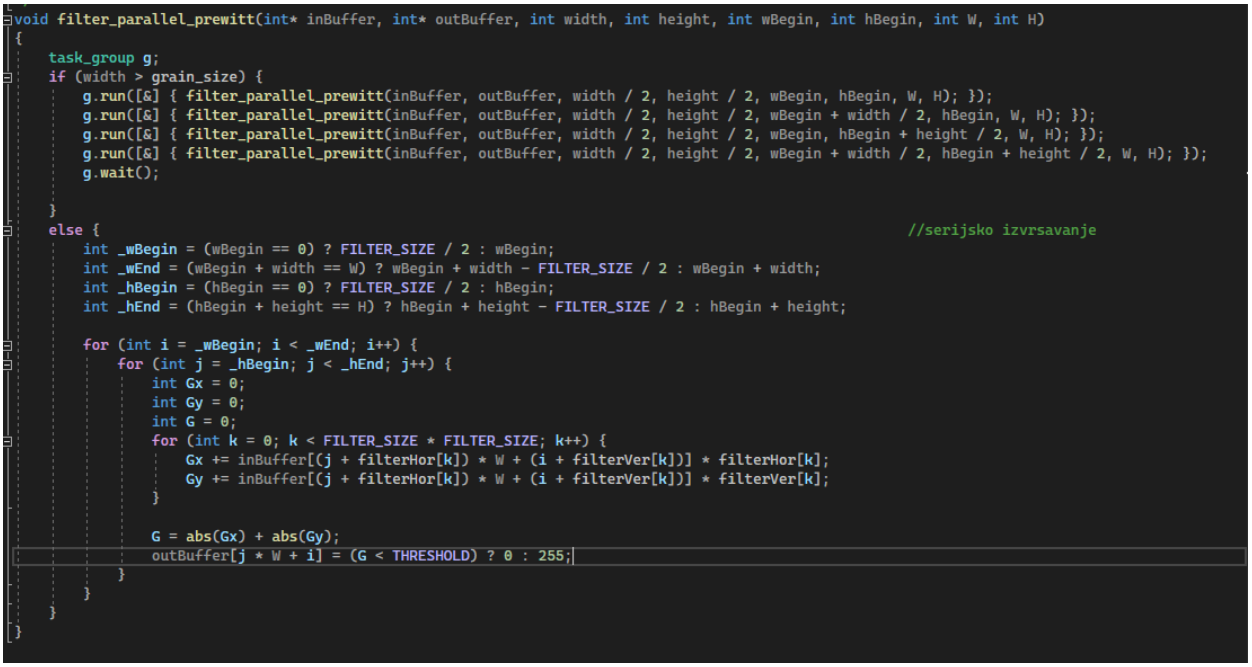
Paralelna implemetacija Prewitt algoritma

Paralelni algoritam omogućava da se obrada slike ubrza tako što se više podregiona obrađuje paralelno, koristeći više procesorskih jezgara ili niti.
Pri paralelizaciji serijskog algoritma korišćena je apstrakcija zadataka, obezbeđena od strane Intel-ove Thread Building Blocks biblioteke.

Algoritam paralelizovanog koda prima ulazni bafer (inBuffer) koji sadrži sliku čiji se pikseli obrađuju, izlazni bafer (outBuffer) koji će sadržati rezultujuću sliku sa primenjenim filterom, širinu slike (width), visinu slike (height), a dodatno prosleđujemo i indekse matrice po širini (x- osi) wBegin i visini (y-osi) tj tačku hBegin za podregion slike koji se trenutno obrađuje, odnosno oblast zaduženja zadatka, kao i ukupnu širinu slike (W) i ukupnu visinu slike (H).
Algoritam koristi paralelnu rekurziju kako bi podelio sliku na manje podregione i paralelno primenio filter Prewitt na svaki podregion. Ova paralelizacija se vrši na osnovu granice grain_size, gde se ako je širina slike veća od grain_size, slika deli na četiri manja podregiona i za svaki podregion se rekurzivno poziva funkcija filter_parallel_prewitt koristeći task_group za upravljanje paralelnim izvršavanjem.

Na kraju pozivamo funkciju g.wait() koja predstavlja teorijsku naredbu „sync“ kako bi se sačekalo da se svi rekurzivni pozivi vrate pre povratka same roditeljske funkcije.

Ovaj proces se ponavlja sve dok je širina prosleđene slike veća od zadate vrednosti. Tek tada se izvrši serijsko rešenje nad delom slike, a rezultat se upiše na odgovarajući deo izlaznog bafera. Pre samog izvršenja serijskog dela, potrebne su male korekcije koje se tiču graničnih piksela delova. Ako granični piksel dela nije granični piksel početne slike, onda treba da se obradi, u suprotnom treba da se preskoči.



```
void filter_parallel_prewitt(int* inBuffer, int* outBuffer, int width, int height, int wBegin, int hBegin, int W, int H)
{
    task_group g;
    if (width > grain_size) {
        g.run([&] { filter_parallel_prewitt(inBuffer, outBuffer, width / 2, height / 2, wBegin, hBegin, W, H); });
        g.run([&] { filter_parallel_prewitt(inBuffer, outBuffer, width / 2, height / 2, wBegin + width / 2, hBegin, W, H); });
        g.run([&] { filter_parallel_prewitt(inBuffer, outBuffer, width / 2, height / 2, wBegin, hBegin + height / 2, W, H); });
        g.run([&] { filter_parallel_prewitt(inBuffer, outBuffer, width / 2, height / 2, wBegin + width / 2, hBegin + height / 2, W, H); });
        g.wait();

    }
    else {                                                                          //serijsko izvrsavanje
        int _wBegin = (wBegin == 0) ? FILTER_SIZE / 2 : wBegin;
        int _wEnd = (wBegin + width == W) ? wBegin + width - FILTER_SIZE / 2 : wBegin + width;
        int _hBegin = (hBegin == 0) ? FILTER_SIZE / 2 : hBegin;
        int _hEnd = (hBegin + height == H) ? hBegin + height - FILTER_SIZE / 2 : hBegin + height;

        for (int i = _wBegin; i < _wEnd; i++) {
            for (int j = _hBegin; j < _hEnd; j++) {
                int Gx = 0;
                int Gy = 0;
                int G = 0;
                for (int k = 0; k < FILTER_SIZE * FILTER_SIZE; k++) {
                    Gx += inBuffer[(j + filterHor[k]) * W + (i + filterVer[k])] * filterHor[k];
                    Gy += inBuffer[(j + filterHor[k]) * W + (i + filterVer[k])] * filterVer[k];
                }

                G = abs(Gx) + abs(Gy);
                outBuffer[j * W + i] = (G < THRESHOLD) ? 0 : 255;
            }
        }
    }
}
```

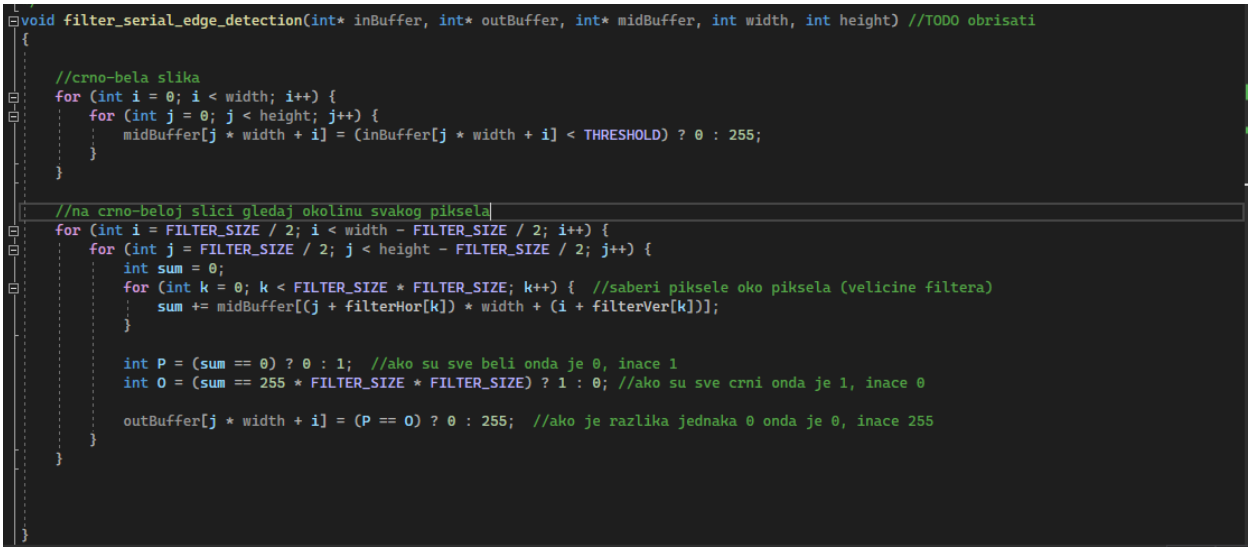
Slika 17 – Paralelno rešenje algoritma

3.2. Algoritam za detekciju ivica slike posmatranjem okoline piksela

Za razliku od algoritma sa Prewitt-om, u ovom algoritmu je neophodan još jedan bafer, gde će biti smeštena crno-bela slika, kroz čije piksele će se posle prolaziti. Prilikom posmatranja okoline, nisam množila matrice, nego sam sabirala piksele i na osnovu rezultata postavljala vrednost posmatranog piksela na 0 ili 255.

Prva petlja for prolazi kroz sve piksele slike dimenzija width i height. Za svaki piksel, vrednost piksela se proverava da li je manja od THRESHOLD. Ako jeste, srednji bafer (midBuffer) na toj poziciji se postavlja na 0 (crna boja), inače se postavlja na 255 (bela boja). Ovaj korak konvertuje ulaznu sliku u crno-belu sliku, gde se vrednosti piksela svedu na crnu ili belu boju na osnovu definisanog praga. Druga petlja for prolazi kroz piksele crno-bele slike, ali preskače ivične piksele za koje nije moguće primeniti potpuni filter. Unutar ove petlje, vrši se obrada svakog piksela i njegove okoline koristeći filter veličine FILTER_SIZE.

- Spoljašnja petlja `for (int i = FILTER_SIZE / 2; i < width - FILTER_SIZE / 2; i++)` iterira kroz piksele slike po horizontalnoj osi, izuzev ivica koje su manje od polovine veličine filtera.
- Unutrašnja petlja `for (int j = FILTER_SIZE / 2; j < height - FILTER_SIZE / 2; j++)` iterira kroz piksele slike po vertikalnoj osi, izuzev ivica koje su manje od polovine veličine filtera.
- Unutar petlji se formira suma piksela oko trenutnog piksela (veličina filtera je `FILTER_SIZE`). Petlja `for (int k = 0; k < FILTER_SIZE * FILTER_SIZE; k++)` iterira kroz sve elemente filtera.
- Suma piksela se koristi za određivanje vrednosti promenljivih `P` i `O`. Promenljiva `P` je postavljena na 0 ako je suma piksela jednaka 0 (sve beli pikseli oko trenutnog piksela), inače je postavljena na 1. Promenljiva `O` je postavljena na 1 ako je suma piksela jednaka maksimalnoj vrednosti (sve crni pikseli oko trenutnog piksela), inače je postavljena na 0.
- Na osnovu vrednosti promenljivih `P` i `O` određuje se vrednost piksela u rezultujućem izlaznom bufferu `outBuffer`. Ako je razlika između `P` i `O` jednaka 0, onda je piksel postavljen na 0 (beli), inače je postavljen na 255 (crni).



```
void filter_serial_edge_detection(int* inBuffer, int* outBuffer, int* midBuffer, int width, int height) //TODO obrisati
{

    //crno-bela slika
    for (int i = 0; i < width; i++) {
        for (int j = 0; j < height; j++) {
            midBuffer[j * width + i] = (inBuffer[j * width + i] < THRESHOLD) ? 0 : 255;
        }
    }

    //na crno-beloj slici gledaj okolinu svakog piksela
    for (int i = FILTER_SIZE / 2; i < width - FILTER_SIZE / 2; i++) {
        for (int j = FILTER_SIZE / 2; j < height - FILTER_SIZE / 2; j++) {
            int sum = 0;
            for (int k = 0; k < FILTER_SIZE * FILTER_SIZE; k++) {  //saberi piksele oko piksela (velicine filtera)
                sum += midBuffer[(j + filterHor[k]) * width + (i + filterVer[k])];
            }

            int P = (sum == 0) ? 0 : 1;  //ako su sve beli onda je 0, inace 1
            int O = (sum == 255 * FILTER_SIZE * FILTER_SIZE) ? 1 : 0; //ako su sve crni onda je 1, inace 0

            outBuffer[j * width + i] = (P == O) ? 0 : 255;  //ako je razlika jednaka 0 onda je 0, inace 255
        }
    }


}
```

Slika 18 – Serijsko rešenje algoritma

Paralelna implementacija

1. Prvo se proverava da li je širina slike (width) veća od granice podele (grain_size). Ako jeste, to znači da je slika dovoljno velika da se može paralelno obraditi. U ovom slučaju, funkcija filter_parallel_edge_detection se rekurzivno poziva za četiri podregiona slike, podeljena na pola u širini i visini. Svaki od tih podregiona se obrađuje u zasebnom paralelnom zadatku. Nakon pokretanja svih zadataka, koristi se g.wait() da se sačeka završetak izvršavanja svih paralelnih zadataka.
2. U suprotnom, ako širina nije veća od grain_size, prelazi se na serijsku obradu.

Ukratko, ovaj kod paralelno detektuje ivice na crno-beloj slici tako što rekurzivno deli sliku na manje podregione koji se obrađuju paralelno, dok se za manje slike koristi serijska obrada. Rezultat detekcije ivica se smešta u izlazni bafer.

```
void filter_parallel_edge_detection(int* inBuffer, int* outBuffer, int* midBuffer, int width, int height, int wBegin, int hBegin, int W, int H)
{

    task_group g;
    if (width > grain_size) {
        g.run([&] { filter_parallel_edge_detection(inBuffer, outBuffer, midBuffer, width / 2, height / 2, wBegin, hBegin, W, H); });
        g.run([&] { filter_parallel_edge_detection(inBuffer, outBuffer, midBuffer, width / 2, height / 2, wBegin + width / 2, hBegin, W, H); });
        g.run([&] { filter_parallel_edge_detection(inBuffer, outBuffer, midBuffer, width / 2, height / 2, wBegin, hBegin + height / 2, W, H); });
        g.run([&] { filter_parallel_edge_detection(inBuffer, outBuffer, midBuffer, width / 2, height / 2, wBegin + width / 2, hBegin + height /
        g.wait();

    }
    else {
        int _wBegin = (wBegin == 0) ? FILTER_SIZE / 2 : wBegin - FILTER_SIZE / 2;
        int _wEnd = (wBegin + width == W) ? wBegin + width - FILTER_SIZE / 2 : wBegin + width + FILTER_SIZE / 2;
        int _hBegin = (hBegin == 0) ? FILTER_SIZE / 2 : hBegin - FILTER_SIZE / 2;
        int _hEnd = (hBegin + height == H) ? hBegin + height - FILTER_SIZE / 2 : hBegin + height + FILTER_SIZE / 2;

        //crno-bela slika
        for (int i = wBegin; i < wBegin + width; i++) {
            for (int j = hBegin; j < hBegin + height; j++) {
                midBuffer[j * W + i] = (inBuffer[j * W + i] < THRESHOLD) ? 0 : 255;
            }
        }


        for (int i = _wBegin; i < _wEnd; i++) {
            for (int j = _hBegin; j < _hEnd; j++) {
                int sum = 0;
                for (int k = 0; k < FILTER_SIZE * FILTER_SIZE; k++) {
                    sum += midBuffer[(j + filterHor[k]) * W + (i + filterVer[k])];
                }

                int P = (sum == 0) ? 0 : 1;
                int O = (sum == 255 * FILTER_SIZE * FILTER_SIZE) ? 1 : 0;

                outBuffer[j * W + i] = (P == O) ? 0 : 255;
            }
```

Slika 19 – Paralelno rešenje algoritma

4. Ispitivanje

Sva merenja prilikom analize rešenja izračunata su na računaru sa komponentama:

- Procesor - Intel Core i5-6300U CPU @ 2.40GHz

  (Cores 2, Threads 4)
- Memorija - 16.00 GB
- Grafička kartica - Intel HD Graphics 520 (HP)

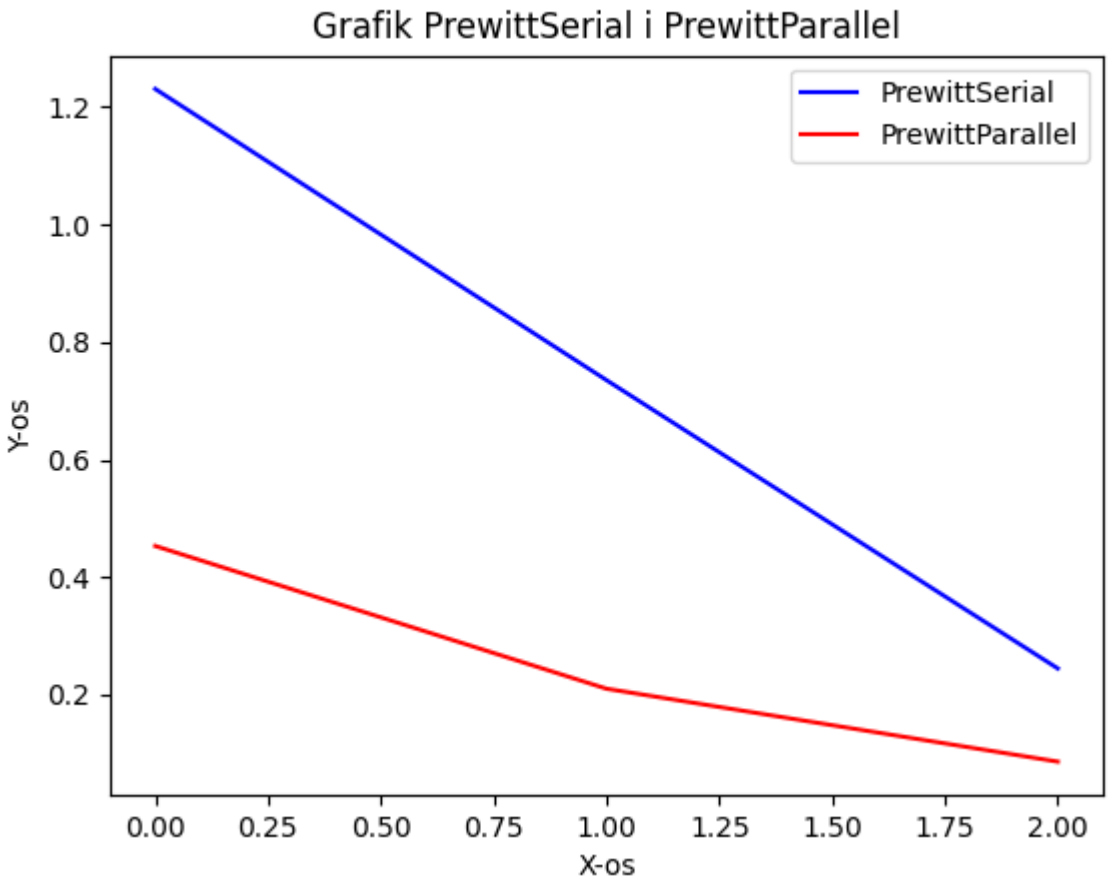
4.1. Grafički prikaz tabela

U nastavku rada su prikazani grafici analize performansi za prethodne slike.

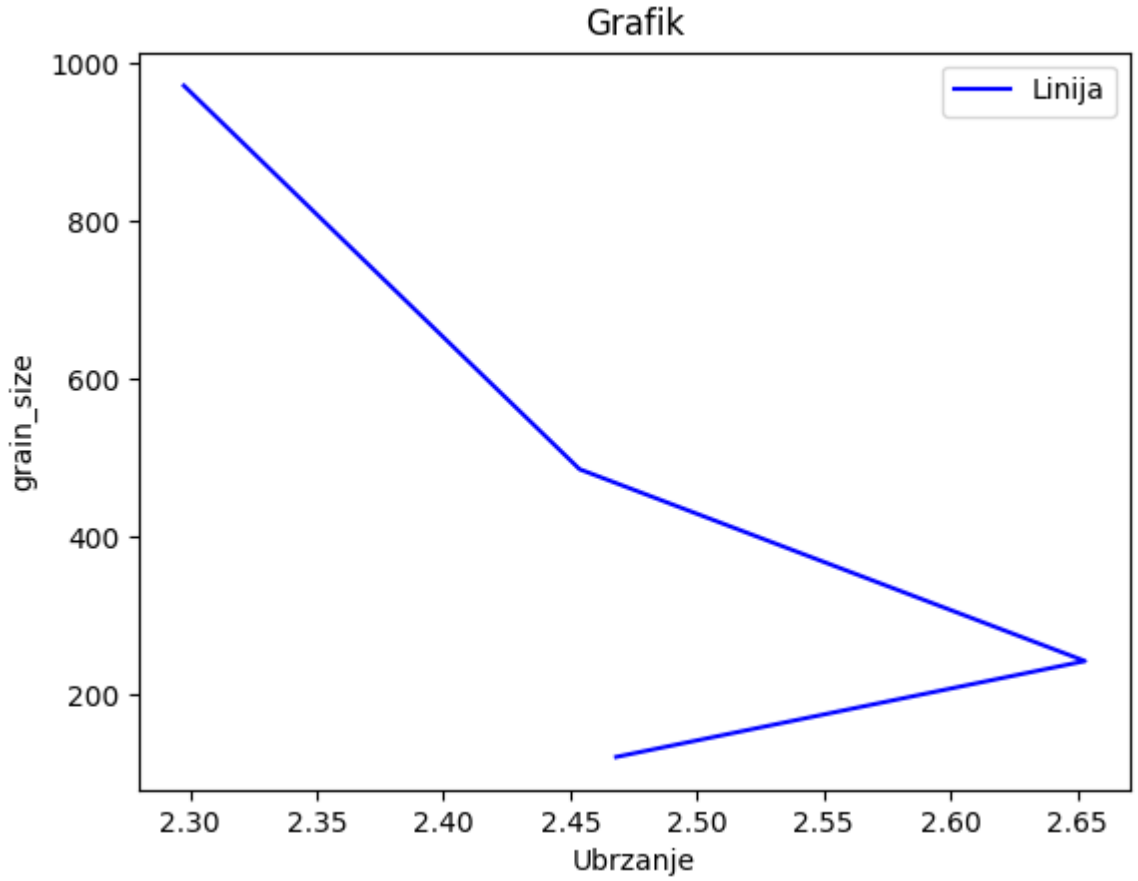
Svi grafici su crtani u Pythonu pomoću Matplotlib biblioteke.

Algoritam za detekciju ivica slike pomoću Prewitt operatora

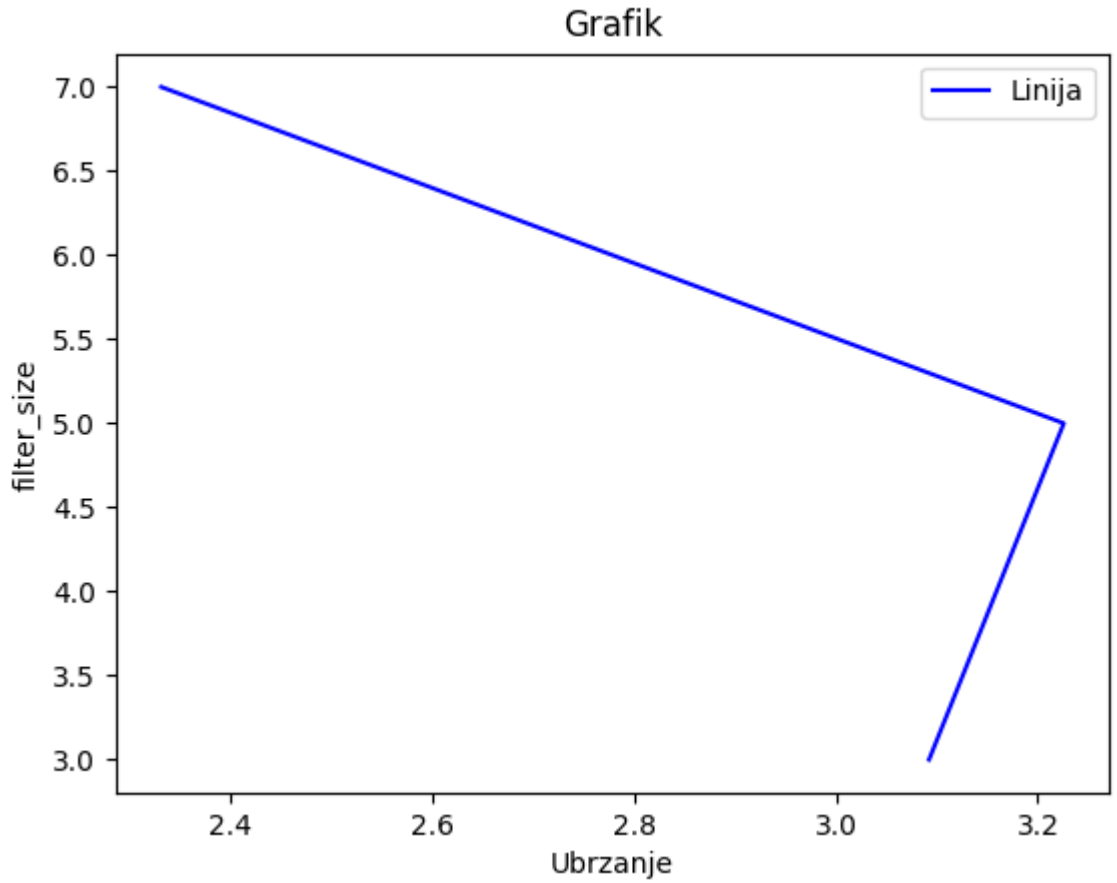
I) Na prvom grafiku ćemo prikazati odnos brzine izvršavanja serijskog i paralelnog algoritma detekcije ivica Prewitt operatorom u zavisnosti od dimenzija filtera (3, 5, 7), a granica podele (grain_size) je postavljena na width / 4. Ovo ćemo uraditi za tri različite slike od one sa najvećim dimenzijama do one sa najmanjim: “color.bmp” (dimenzije 3888 x 2592), “brigde.bmp” (dimenzije 2048x1536) i “cave.bmp” (dimenzije 1800x1200).



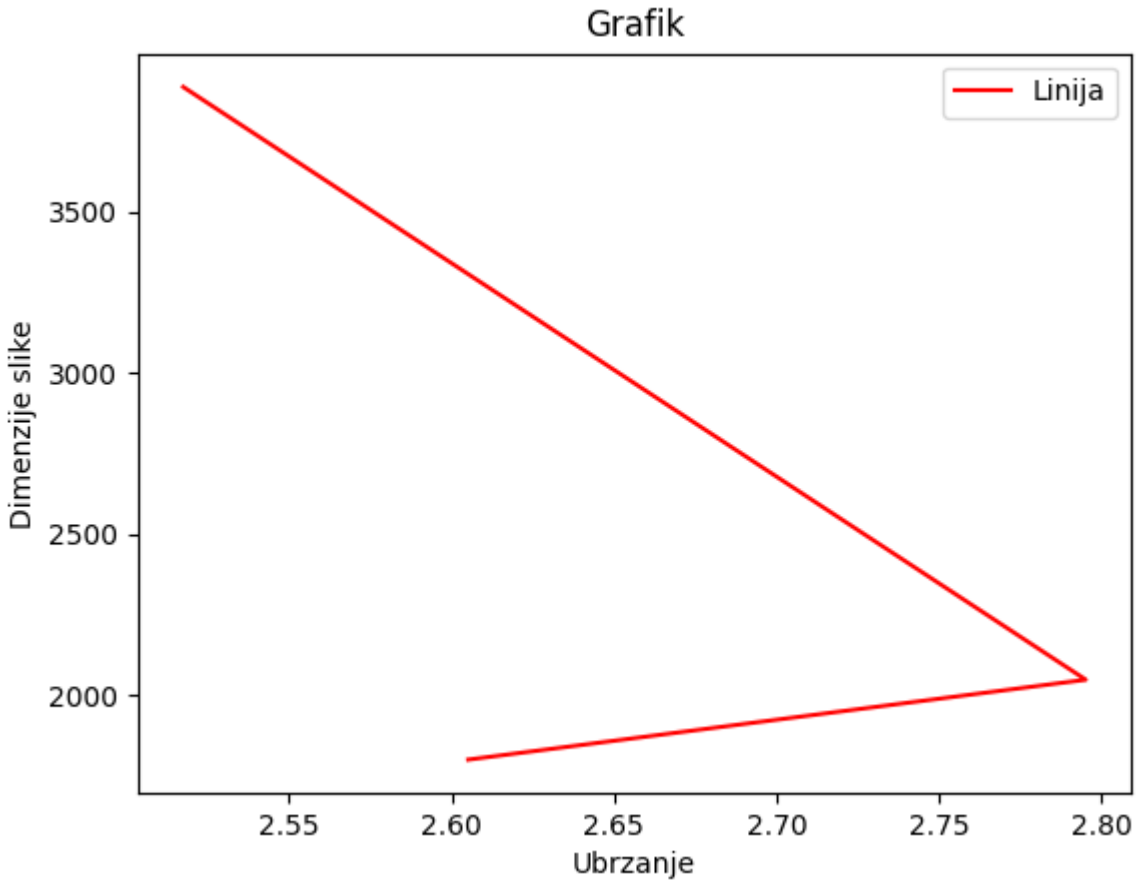
II) Na sledećem grafiku je prikazan odnos ubrzanja u odnosu na različite vrednosti za grain_size, s tim da je korišćen filter dimenzija 5x5 za svaku različitu vrednost grain_size-a (dimenzije ulazne slike su 3888 x 2592).



III) Sledeći grafik prikazuje odnos brzine izvršavanja serijskog i paralelnog algoritma detekcije ivica Prewitt operatorom u zavisnosti od dimenzija filtera (3, 5, 7), a korišćen je grain_size=width/8 koji je eksperimentalnom metodom i na osnovu prethodnih grafika izabran kao optimalan.

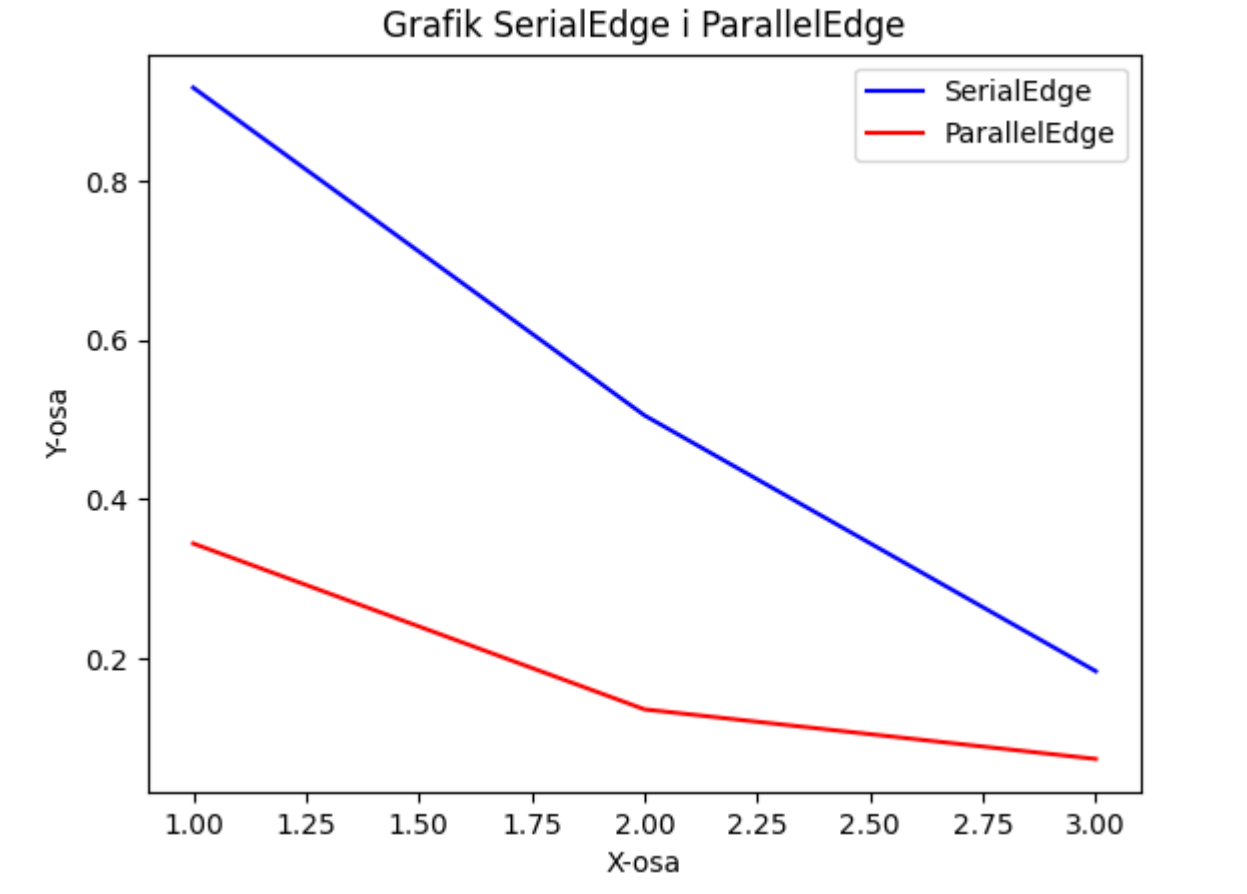


Sa optimalnim rešenjima izabranim za dimenziju filtera (filter_size=5) i za granicu podele (grain_size=width/8) na osnovu eksperimentalnog postupka prikazanog i na prethodnim graficima crtamo novi grafik na kom menjamo samo dimenzije ulazne slike.

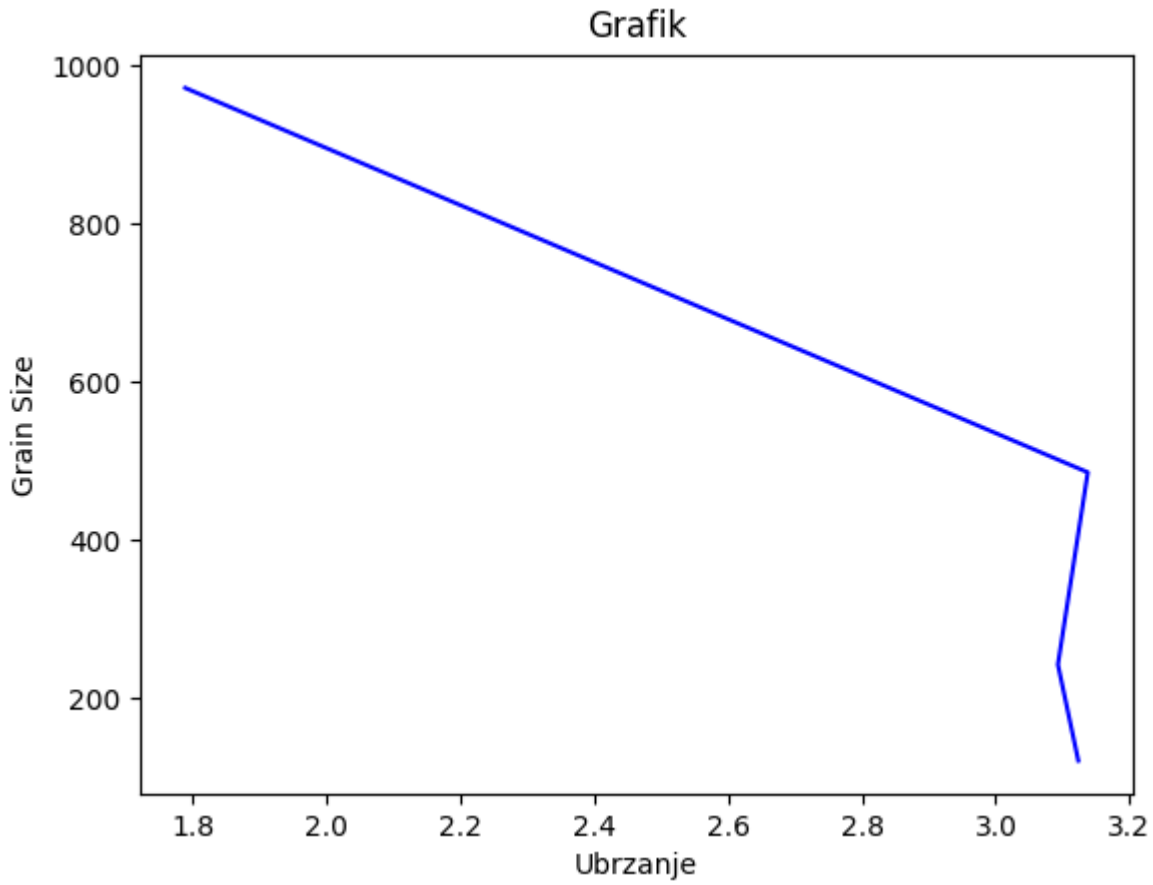

Grafik
Linija
3500
3000
2500
2000
Dimenzije slike
2.55
2.60
2.65
2.70
2.75
2.80
Ubrzanje

Algoritam za detekciju ivica slike posmatranjem okoline piksela

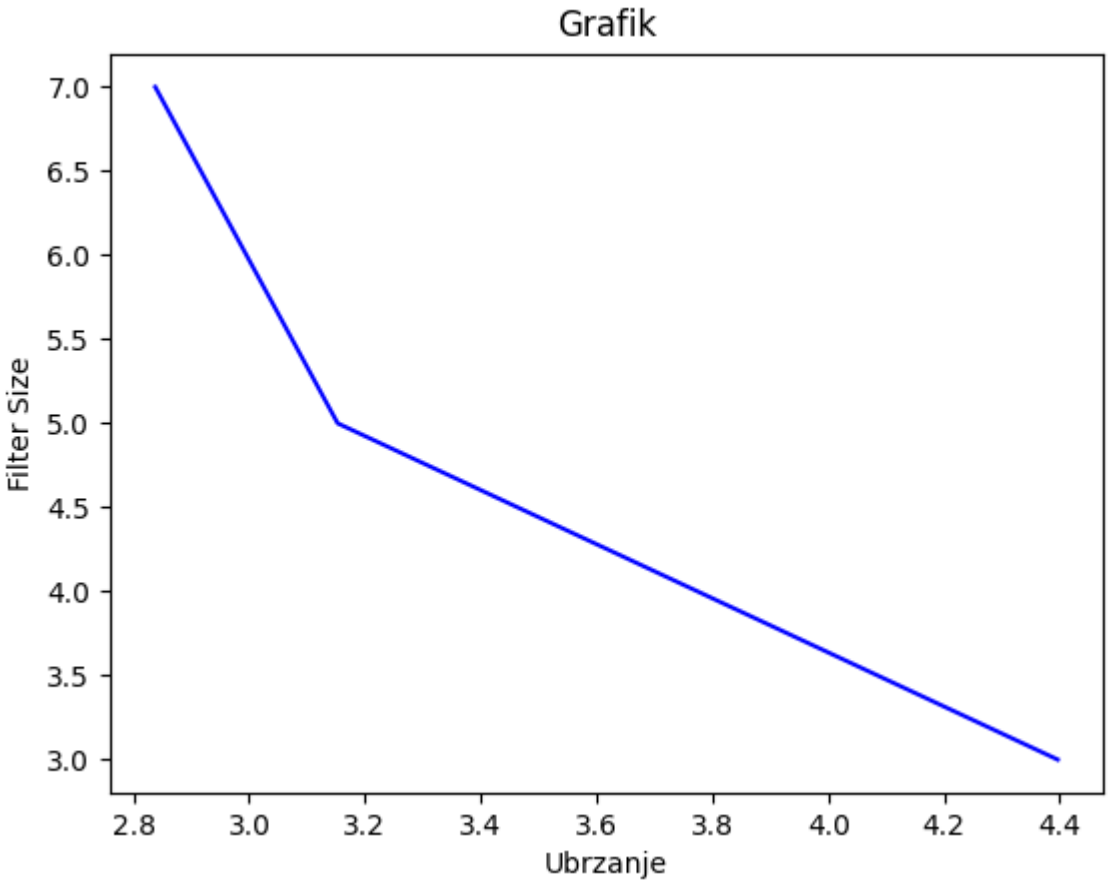
I) Na prvom grafiku ćemo prikazati odnos brzine izvršavanja serijskog i paralelnog algoritma detekcije ivica posmatranjem okoline piksela u zavisnosti od dimenzija filtera (3, 5, 7), a granica podele (grain_size) je postavljena na width / 4 gde je width širina ulazne slike. Ovo ćemo uraditi za tri različite slike od one sa najvećim dimenzijama do one sa najmanjim: “color.bmp” (dimenzije 3888 x 2592), “brigde.bmp” (dimenzije 2048x1536) i “cave.bmp” (dimenzije 1800x1200).



II) Na sledećem grafiku je prikazan odnos ubrzanja u odnosu na različite vrednosti za grain_size , s tim da je korišćen filter dimenzija 5x5 za svaku različitu vrednost grain_size-a (dimenzije ulazne slike su 3888 x 2592).



III) Sledeći grafik prikazuje odnos brzine izvršavanja serijskog i paralelnog algoritma detekcije ivica posmatranjem okoline piksela u zavisnosti od dimenzija filtera (3, 5, 7), a korišćen je grain_size=width/8 koji je eksperimentalnom metodom i na osnovu prethodnih grafika izabran kao optimalan.


Grafik
Filter Size
7.0
6.5
6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.8
3.0
3.2
3.4
3.6
3.8
4.0
4.2
4.4
Ubrzanje

4.2. Tabele sa vremenom izvršavanja

Vrednosti su u sekundama.

Primer 1 – color.bmp (Slika 1)

Dimenzije: 3888 x 2592

Prewitt

Serijski 3x3 filter	Paralelno 3x3 filter	Serijski 5x5 filter	Paralelno 5x5 filter	grain_size
2.35229	1.04238	0.917472	0.344307	width /4
2.68054	1.09244	3.04308	0.943353	width/8
2.70616	1.02	1.82894	0.397345	width/16

Sa posmatranjem okoline piksela

Serijski 3x3 filter	Paralelno 3x3 filter	Serijski 5x5 filter	Paralelno 5x5 filter	grain_size
1.59038	0.591935	1. 615564	0. 528073	width /4
1.61577	0.514987	1.51772	0.481393	width/8
1.49702	0.484021	0.527539	0.497253	width/16

Primer 2 – bridge.bmp (Slika 2)

Dimenzije: 2048x1536

Prewitt

Serijski 3x3 filter	Paralelno 3x3 filter	Serijski 5x5 filter	Paralelno 5x5 filter	grain_size
3.499893	1.0655617	3.93011	1.68611	width /4
3.427922	1.0557836	3.611043	1.12982	width/8
3.491798	1.0607767	3.672019	1.129967	width/16

Sa posmatranjem okoline piksela

Serijski 3x3 filter	Paralelno 3x3 filter	Serijski 5x5 filter	Paralelno 5x5 filter	grain_size
2.291644	0.8338534	2.553729	0.810027	width /4
2.310156	0.8311014	2.54037	0.808243	width/8
2.309491	0.8304447	2.550153	0.7401391	width/16

5. Analiza rezultata

Kao što se iz tabela i grafova vidi, paralelizacijom oba algoritma smo dobili ubrzanje. Vreme izvršenja paralelnih algoritama je bilo od 3 do 5 puta brže, čak i preko zavisno od parametara i ulaza.

Vreme izvršavanja našeg paralelnog algoritma u određenoj meri zavisi od tri stvari:

- dimenzija ulazne slike
- vrednosti na kojoj prekidamo deljenje matrice
- veličine filtera.

Kada je reč o paralelizaciji Prewitt operatora imamo grafik koji nam prikazuje odnos brzine izvršavanja serijskog i paralelnog algoritma za različite dimenzije. Primećujemo veoma značajno ubrzanje gde se slika dimenzija 3888x2592 filterom dimenzija 3x3 serijski obrađuje za 1.23s dok paralelno za 0.45s što je značajno ubrzanje. Ubrzanje kod slike manjih dimenzija je prema grafiku istog reda veličine kao i kod veće slike, ali zbog svojih manjih dimenzija i serijski algoritam se izvršava relativno brzo, s toga se paralelizacija vidi više na slikama većih dimenzija.

Povećanjem filtera se takođe usporava izvršenje, međutim rezultat je mnogo precizniji, detektovano je više ivica što se utvrđuje posmatranjem izlaznih slika.

Napomenula sam već da je za paralelizaciju pomoću zadataka veoma bitna vrednost na osnovu koje prekidamo deljenje matrice i ostatak izvršvamo serijski, jer ako izaberemo suviše veliku vrednost zadaci će biti i dalje veliki i nećemo dobiti traženo ubrzanje. Sa druge strane treba paziti da nam ta vrednost ne bude previše mala, jer u tom slučaju kreiranje zadatak će biti skuplje od same serijske obrade što takođe želimo da izbegnemo. Vrednost grain_size-a je birana eksperimentalnom metodom, tako što je prvo izabrana veća vrednost i zatim smanjivana dok nismo dobili optimalno rešenje. Najbolji performansi su utvrđeni za grain_size=width/8.

Naravno, sva ova diskusija dosta zavisi i od samih performansi računara na kom se izvršava algoritam, odnosno procesora, propusnog opsega itd.

Paralelizacijom druge metode za detekciju ivica smo takođe dobili zadovoljavajuće ubrzanje, gde se za sliku dimenzija 3888x2592 sa grain_size-om

od 486 serijski kod izvršava za 0.92s, a paralelni za 0.34s.

Krajnji zaključak je da bez paralelizacije naši projekti u velikom broju slučajeva ne bi bili sposobni da zadovolje određene performanse zbog čega je ovo izuzetno važan koncept.