Univerzitet u Nišu

Elektronski fakultet u Nišu

2019. godina

**Seminarski rad iz predmeta**

**PROJEKTOVANJE RAČUNARSKOG HARDVERA**

**Detekcija ivica na slikama**

**uz pomoc MaxCompiler-a**

Student: Mentori:

Prof. dr Ivan Milentijević

Aleksandar Tasic Asistent Vladimir Simić

Sadržaj

[Tehnike za detekciju ivica na slikama 3](#_Toc17821204)

[Detekcija zasnovana na gradijentima 4](#_Toc17821205)

[Robert-ov operator 5](#_Toc17821206)

[Sobel-ov operator 5](#_Toc17821207)

[Prewitt-ov operator 6](#_Toc17821208)

[Canny-ev detektor ivice 6](#_Toc17821209)

[Detektori ivice drugog reda 6](#_Toc17821210)

[Marr - Hildrith-ov operator 7](#_Toc17821211)

[Data flow programiranje 8](#_Toc17821212)

[Data flow engine (DFE) 9](#_Toc17821213)

[Implementiranje kernela 10](#_Toc17821214)

[Projekat detekcije ivica na slikama u MaxCompiler-u 12](#_Toc17821215)

[Arhitektura kernela data flow programa 19](#_Toc17821216)

[Zaključak 20](#_Toc17821217)

[Literatura 21](#_Toc17821218)

# Tehnike za detekciju ivica na slikama

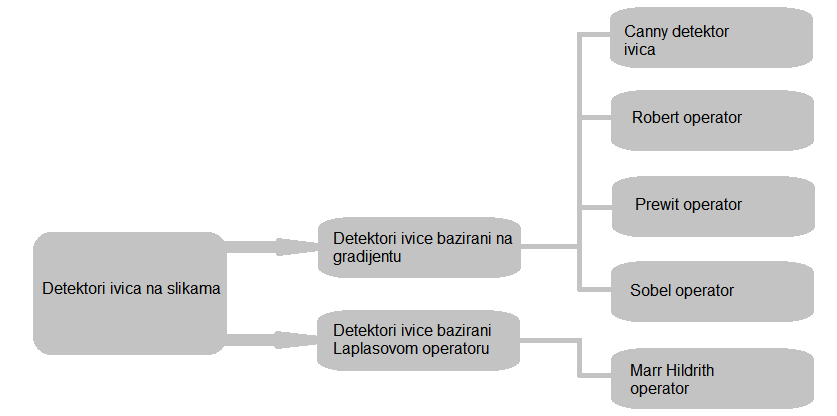
Ivica na slici je granica ili kontura na kojoj dolazi do značajnih promena u nekom od fizičkih aspekata slike, kao što su površinska refleksija, osvetljenje ili udaljenost vidljive površine od posmatrača. Promene u fizičkom aspektu slike manifestuju se u vidu promene intenziteta, boje i strukture slike.

Značaj fizičkih promena na jednoj slici zavisi od slučaja do slučaja. Promena intenziteta koja bi trebalo da bude klasifikovana kao ivica, u nekom drugom slučaju uopšte neće biti uzeta u obzir. U jednom sistemu za identifikaciju objekata, neke granice objekta mogu biti suvišne za identifikaciju i neke konture koje predstavljaju dodatne detalje u okviru objekta neće biti razmatrane kao ivice. Shodno tome, ne može se dati tačna definicija ivice, već ona zavisi od konteksta u kome će se ivica koristiti. Može se reći da detekcija ivice predstavlja tehniku segmentacije slike,tokom koje se prostorni domen u kome se slika definiše deli na značajne delove ili regione a detektovane ivice se obično javljaju na granici između dva različita regiona na slici.

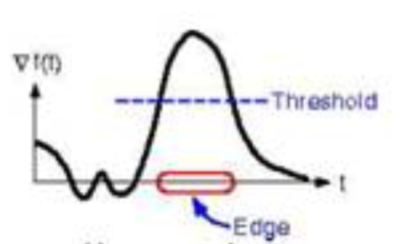
Detekcija ivica obezbeđuje informacije o poziciji ivica objekata i prisustvu diskontinuiteta. Diskontinuiteti su nagle promene u intezitetu piksela koje karakterizuju granice objekata u slici.

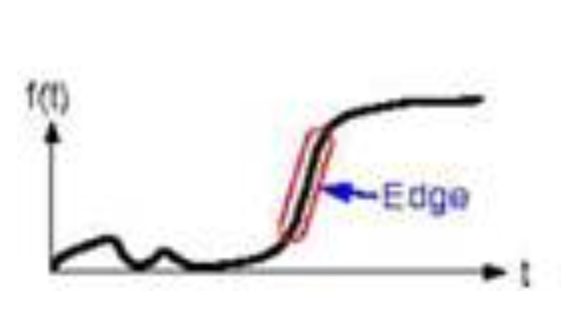
Klasične metode detekcije ivica podrazumevaju konvoluciju slike sa operatorom (2-D filter), koji je konstruisan da bude osetljiv na velike gradijente u slici, pri čemu se vraćaju vrednosti jednake nuli za uniformne delove slike.

Promenljive koje su uključene u izbor operatora za detekciju ivica uključuju i orijentaciju ivica, šum okoline i strukturu ivica. Geometrija operatora određuje karakterističan pravac u kome su najviše osetljivi na ivice. Operatori mogu da se optimizuju tako da se traže horizontalne, vertikalne ili dijagonalne ivice. U zašumljenim slikama je selekcija ivica otežana s obzirom na činjenicu da je sadržaj šuma i ivica visokofrekventan a takođe postoje problemi pogrešne detekcije ivica, nedostatak pravih ivica, lokalizacija ivica, dug period obrade, problemi vezani za šum, itd. Iz tog razloga je neophodno utvrditi razlike između različitih tehnika detekcije ivica i analizirati performanse ovih tehnika pod različitim uslovima. Iako postoji mnogo metoda detekcije ivica većina se može podeliti na gradijentne i Laplasove (Laplaceove). Gradijentne metode rade tako da traže maksimume i minimume u prvom izvodu slike, dok Laplace-ove metode traže nule u drugom izvodu slike.



## 1.Detekcija zasnovana na gradijentima

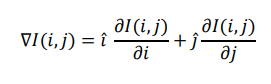
U ovom dokumentu ćemo se malo detaljnije pozabaviti raznim tehnikama za detekciju ivice kao što su Prewitt-ovi, Robert-ovi, Sobel-ovi, Marr Hildrith-ovi i Canny-evi operatori. 

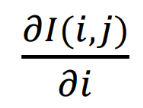


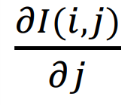
Na prethodnim slikama je predstavljen postupak detekcije ivice korišćenjem gradijenata.Prva slika predstavlja ivicu signala koji je jednodimenzionalan, dok druga predstavlja prvi izvod signala.

Ivica na slici znači naglu promenu intenziteta piksela, a upravo je to područje u kome prvi izvod ima maksimalnu vrednost. Slike u digitalnoj obradi nisu kontinuirane nego diskretne pa se ne može izračunati prava vrednost izvoda. Uprkos diskretnosti slike, postoji mnogo metoda za procenu vrednosti prvog izvoda. Nakon procene izvoda, kod gradijentnih metoda koristi se prag koji određuje koliko treba da bude iznos izvoda da bi tačku na tom mestu smatrali ivičnom.

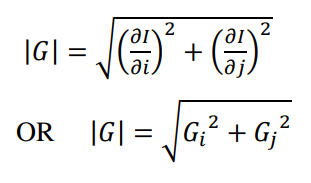
Ako I ( i, j ) predstavlja podatke vezane za ulaznu sliku, onda se gradijent slike dobija prema formuli:



 gradijent u i-tom smeru

 gradijent u j-tom smeru

Magnituda gradijenta se može izračunati po formuli :



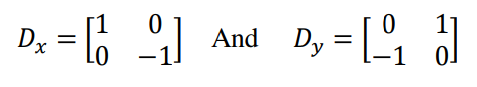
Postoji još jedna formula preko koje se može izračunati magnituda gradijenta, a to je :

C:\Users\Mihajlo\AppData\Local\Microsoft\Windows\INetCache\Content.Word\5.png

Druga formula za izračunavanje magnitude gradijenta daje jačinu ivice a smer gradijenta je uvek normalan na smer ivice.

## Robert-ov operator

Kod ovog operatora koji se bazira na gradijentu prvo se računa zbir kvadrata razlike dijagonalnih piksela kroz diskretnu diferencijaciju pa se onda računa gradijent slike. Ulazna slika se povezuje sa jediničnim matricama operatora i računaju se magnituda gradijenta i smerovi. Robert-ov operator koristi dve 2 x 2 matrice :



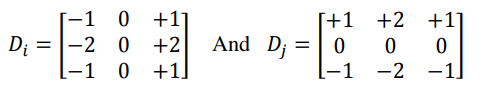
Negativna osobina ovog operatora je da je vrlo osetljiv na šum zbog matrice malih dimenzija.

## Sobel-ov operator

Ovaj operator je jedan od najčešće korišćenih detektora ivice. Ova metoda se zasniva na konvoluciji slike sa filtrom u horizontalnom i vertikalnom pravcu i stoga je relativno jednostavana za izračunavanje i primenu. Sobel filter za detekciju ivice ima prednost u tome što obezbeđuje diferencijaciju (koja omogućava detekciju ivica) i kao smanjenje šuma.

Za svaki piksel sa slike, Sobel operator daje odgovarajući vektor gradijenta. Povezuje ulazne podatke iz slike sa matricom i računa magnitudu gradijenta.Vrednosti -2 i 2 zapravo su koeficijenti koji najčesće imaju vrednosti od 0 do 5 tj. od -0 do -5.Ovi koeficijenti utiču na to koje ivice se mogu detektovati na slikama, tj. koliko te ivice moraju biti izražene da bi ih filter detektovao (Veće vrednosti koeficijenata detektuju i manje izražene ivice, ali i stvaraju veći šum).

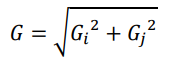
Sobel operator koristi dve 3 x 3 matrice :



Ove Sobelove maske kao rezultat daju gradijente u **i** i  **j** smerovima.

C:\Users\Mihajlo\AppData\Local\Microsoft\Windows\INetCache\Content.Word\13.png

Zato je jačina ivice ili magnituda gradijenta piksela data sa



Smer gradijenta je dat sa

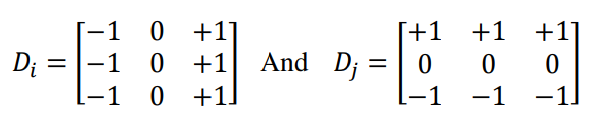
15

Gde su Gi i Gj su gradijenti u smerovima **i** i **j**, respektivno.

Kada bismo ovaj operator uporedili sa Robert-ovim operatorom, Sobel-ov je sporiji ali pošto ima veću matricu, manje je osetljiv na šum nego Robert-ov operator.

## Prewitt-ov operator

Funkcija Prewitt-ovog detektora ivice je gotovo ista kao i Sobel operatora, samo se koeficijenti kod Sobel-ovog detektora -1 i 1.



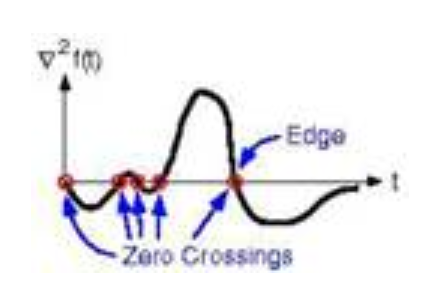
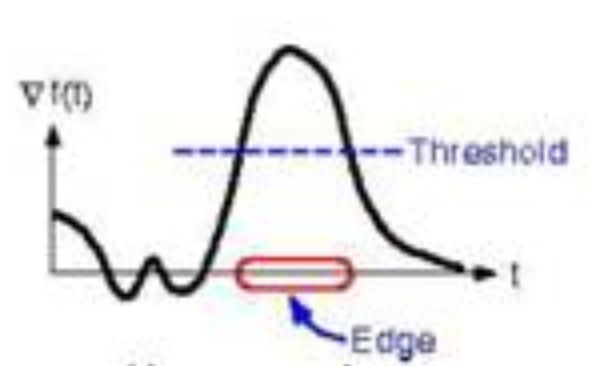
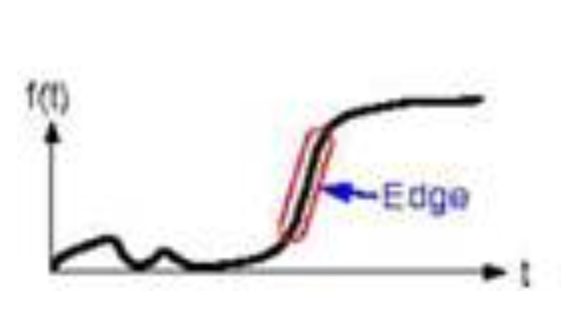
## Canny-ev detektor ivice

Canny-ev detektor ivice poseduje napredniji algoritam izveden iz prethodnog rada Marr-a i Hildreth-a. To je optimalna tehnika za detekciju ivice koja obezbeđuje dobru detekciju, jasne odgovore i dobru lokalizaciju. Ima široku primenu u tehnikama za obradu slike koje se sada koriste.Sastoji se od 5 koraka:

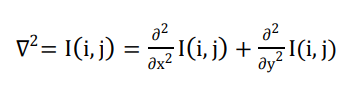
* Primeniti Gausov filter da ublaži šum na slici.
* Pronalaženje gradijenata na slici
* Primeniti nemaksimalnu supersiju da bi se oslobodili lažnih ivica
* Odrediti opseg za detekciju potencijalnih ivica
* Na osnovu histerezisa, odrediti ivice tako što će se eleminisati ivice koje su slabe ili nisu povezane sa sigurnim ivicama.

## Detektori ivice drugog reda

Detektor ivice drugog reda se bazira na primeni drugog izvoda.Na trećoj slici dat je drugi izvod signala.



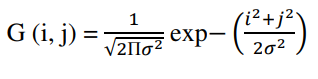
Kod Laplasovog operatora , piksel je označen kao ivica na poziciji gde je drugi izvod jednak 0. Laplasov operator **∇2** za 2D sliku I ( i , j ) se definiše sledećom formulom :



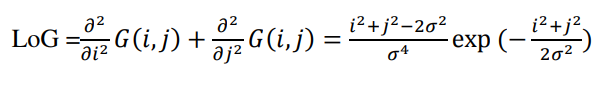
## Marr - Hildrith-ov operator

Marr – Hildrith-ov detektor ivice je bio vrlo popularan detektor ivice pre Canny-evog algoritma za detekciju ivice. To je operator baziran na gradijentu koji koristi Laplasov operator za uzimanje drugog izvoda slike. Koristi i Gausov i Laplasov operator, s tim što Gausov operator smanjuje šum a Laplasov operator detektuje oštre ivice.

Gausova funkcija je definisana formulom :



gde 19 označava standardnu devijaciju dok se LoG operator računa kao :



Marr - Hildrith-ov operator ipak ima 2 glavna ograničenja : Kao i svaki operator baziran na drugom izvodu, dolazi do detekcije lažnih ivica u situacijama kada slika ima previše šuma,a uzrok je u tome što ova metoda detektuje manje promene u intenzitetu piksela, u poređenju sa metodom zasnovanom na gradijentima.

# Data flow programiranje

U [računarskom programiranju](https://sr.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%B8%D1%80%D0%B0%D1%9A%D0%B5), data flow programiranje je [paradigma programiranja](https://sr.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%B8%D1%80%D0%B0%D1%9A%D0%B0) gde se modeli programa mogu predstaviti grafikom toka podataka, koji pokazuje tok podataka između operacija , primenjujući principe i arhitektrue data flow programiranja.

Današnji procesori, sledeći [Murov zakon](https://sr.wikipedia.org/wiki/%D0%9C%D1%83%D1%80%D0%BE%D0%B2_%D0%B7%D0%B0%D0%BA%D0%BE%D0%BD), trebalo bi da imaju frekvenciju radnog takta reda nekoliko desetina gigaherca, a time i veliku brzinu rada. Činjenica je da takvu brzinu današnji računari nemaju. Razlog se može naći u enormnoj količini energije koju bi procesori takvog kapaciteta proizvodili, i velika količina resursa morala bi se koristiti za obezbeđivanje adekvatnog sistema hlađenja za takve sisteme. Performanse današnjih procesora su ograničene i zbog intenzivne komunikacije procesora i memorije,što je veoma spor proces. Svaka instrukcija se mora dobaviti, dekodirati, izvršiti, što je vremenski zahtevno, tako da nezavisno od ubrzanja koje se nastoji ostvariti unapređenjem tehnologije ili primenom paralelizama u izvršenju programa, performanse današnjih računara su ograničene. Navedena ograničenja dovela su do razvoja drugačijeg pristupa u programiranju,odnosno do data flow programiranja. Ovaj drugačiji pristup omogućava izvršenje programa za jedan, dva ili čak tri reda veličine brže (zavisno od specifičnosti aplikacije) u odnosu na izvršenje programa na uobičajenim arhitekturama.

Paradigma data flow programiranja modeluje program kao usmereni graf toka podataka između operacija. Podaci su predstavljeni kao skup čvorova (takođe nazvanih blokovi) sa ulaznim „i/ili” portovima u njima. Čvorovi su povezani usmerenim granama koje definišu protok podataka kroz njih. Paradigma protoka podataka ima široku primenu, podržava i izračunavanja nad velikom količinom podataka . Koncepti protoka podataka su razvijeni kao rešenje za ograničenja i uska grla, koja se po pitanju performansi javljaju kod klasnične arhitekture. Računari koji koriste protok podataka prema proceni stručnjaka iz ove oblasti, predstavljaju tehnologiju koja će sve više biti u upotrebi. Razlog leži u činjenici da je ova arhitektura veoma podesna za paralelno izvršavanje programa i veoma dobro se primenjuje na probleme širokih namena.Kod ove arhitekture ključni element je DFE(data flow engine) koji služi za obradu velike količine podataka.

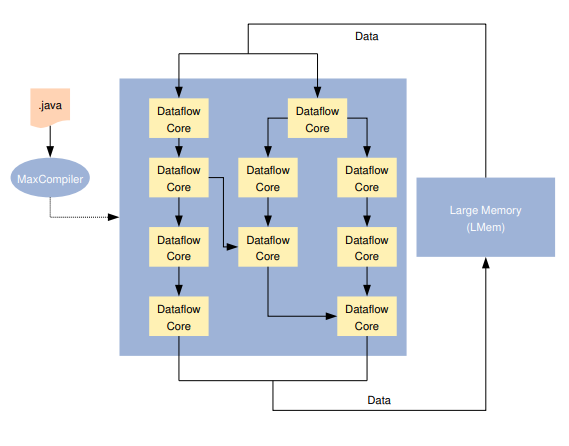
Iako su u osnovi zamišljeni kao akceleratori programa koji su prethodno bili sekvencijalne prirode, danas postoje algoritmi koji su pisani isključivo za izvršavanje na DFE (data flow engine) , koji služi za obradu velike količine podataka kod data flow arhitekture. Jedina poteškoća u radu sa ovim ahitekturama može biti u kreiranju visoko paralelizovanih programa koji bi se mogli iskoristiti pune performanse ovakvih arhitektura.

U softverskoj aplikaciji, izvorni kod programa se pretvara u listu instrukcija za određeni procesor ,koja se zatim učitava u memoriju. Instrukcije se kreću kroz procesor i povremeno čitaju ili upisuju podatke u memoriju i iz nje. Moderni procesori sadrže mnogo nivoa keširanja, prosleđivanja i predviđanja da bi se poboljšala efikasnost ove paradigme, međutim, programski model je sam po sebi sekvencijalan i performanse zavise od latencije pristupa memoriji i radnog takta procesora.

U data flow programu, opisujemo operacije i tok podataka za određeni algoritam . U Dataflow Engine-u (DFE), podaci se prenose iz memorije u procesni čip gde se podaci prosleđuju direktno iz jedne aritmetičke jedinice („jezgra podataka“) u drugu dok se obrada ne završi. Jednom kada je izvršenje programa završeno, data flow engine sa može se rekonfigurisati za novu aplikaciju za manje od sekunde.

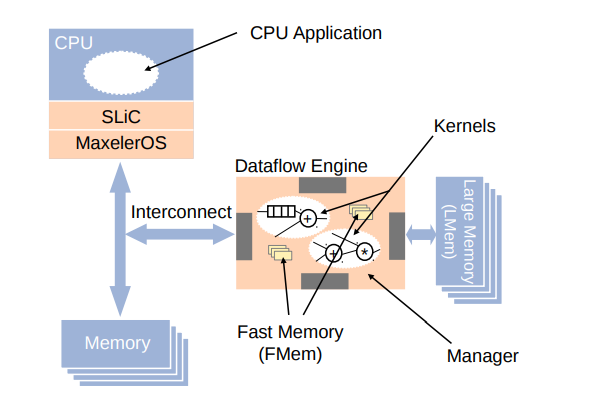
Svako data-flow jezgro izračunava samo jednu vrstu aritmetičkih operacija (na primer, sabiranje ili množenje) i zato je jednostavno da se hiljade tih jezgara smeste u jedan data flow engine. U DFE obradi , svako data flow jezgro izvršava operacije u jednom trenutku nad odgovarajućim podacima, za razliku od control flow jezgara (jezgra za upravljanje tokom podataka) gde se operacije izvode u različitim trenucima na istoj funkcionalnoj jedinici.

Sama dataflow struktura omogućava izračunavanje, tako da nema potrebe za instrukcijama; instrukcije su zamenjene aritmetičkim jedinicama (jezgrima) povezanim na način da obavljaju odgovarajuće operacije.Takođe, nema potrebe za dekodiranjem instrukcija, keširanjem instrukcija i upravljanjem tokom instrukcija, već svi resursi mogu preusmeriti na obavljanje izračunavanja. Data flow engine upravlja operacijama koje se izvršavaju tokom dataflow programa , dok CPU na kome se izvršava Linux upravlja IO operacijama.Na sledećoj slici data je šema data flow programa prilikom izvršenja.



## Data flow engine (DFE)

Na narednoj slici prikazana je arhitektura sistema sačinjenog od DFE-a,sa lokalnim memorijama i CPU-om. Svaki DFE može implementirati više kernela, koji izvode izračunavanja dok se podaci kreću između CPU-a, DFE-a i njihovih memorija.Pored kernela, DFE sadrži i deo koji je zadužen za menadžment tj za upravljanje protokom podataka unutar DFE-a.DFE ima dva tipa memorija:FMem (brze memorije) sa kapacitetom od nekoliko megabajta podataka sa brzinom prenosa podataka reda veličine terabajta/sekundi i LMem (velike memorije) koje mogu pamtiti više gigabajta podataka, ali su znatno sporije.Brze memorije su te koje omogućavaju dobre performanse pri izvršenju kompleksnih data-flow programa.



Implementiranje kernela

Elementi koji čine kernel su:

FFF.png

Čvorovi koji izvode aritmetičke i logičke operacije (g., +,\*,<,&) kao i konverziju između podataka različitih tipova.

Predstavlja vrednosti koje su konstantne ili postavljene od strane CPU aplikacije tokom izvršenja programa.

Omogućavaju pristup starim i budućim elementima iz toka podataka.

Multiplekser , koristi se kada je potrebno izabrati tok podataka.

Čvorovi za usmeravanje i praćenje toka

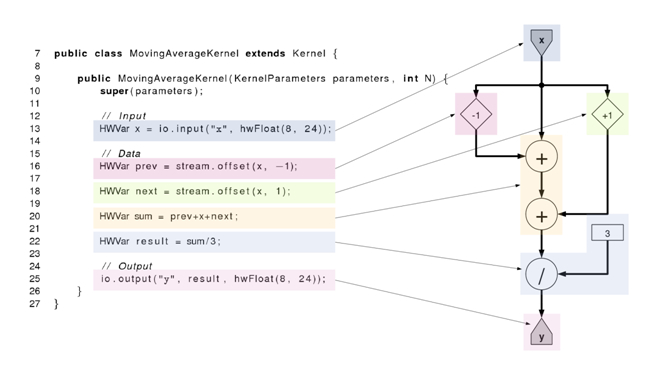
I/O čvorovi koji povezuju tokove podataka između Kernela i Mnager-a

Na jednostavnom primeru vidimo osnovnu strukturu i elemente programa napisanog u MaxCompiler-u.Na osnovu ulazne sekvence od N podataka, izlaz se dobija po formulama:

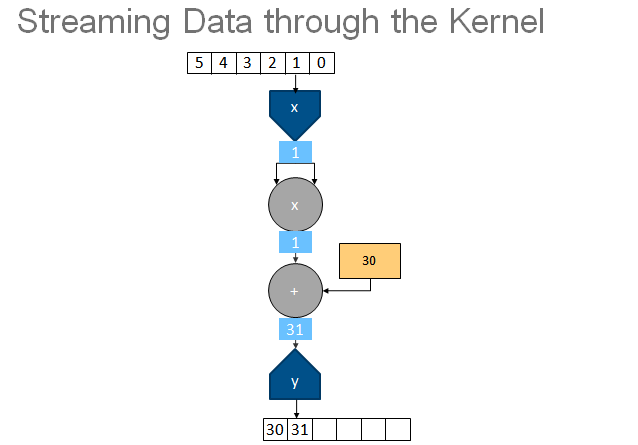
yi= (xi + xi+1)/2 ,ako je i = 0

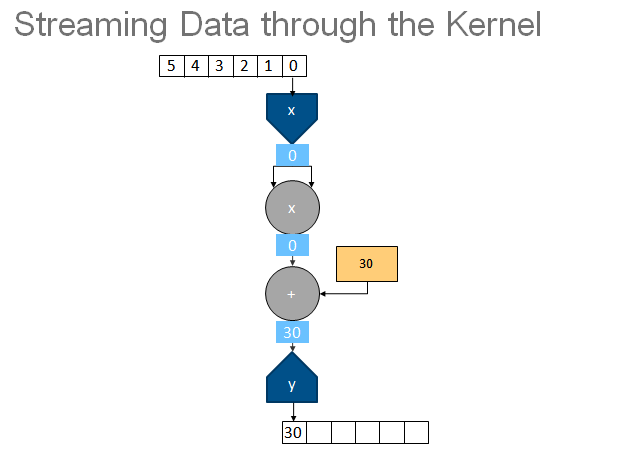
yi=(xi−1 + xi)/2 ,ako je i = N−1

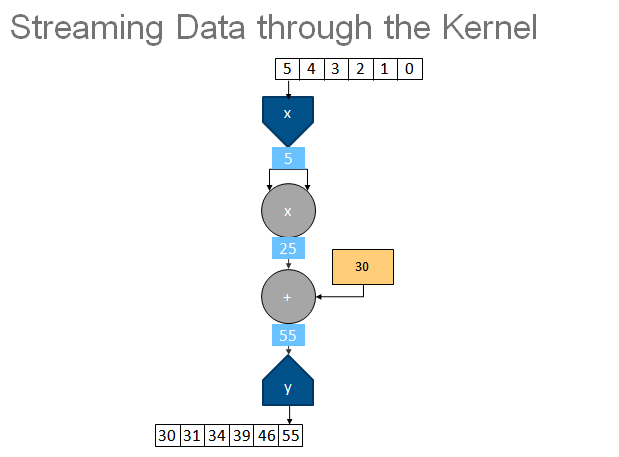
yi=(xi−1 + xi + xi+1)/3, u ostalim slučajevima

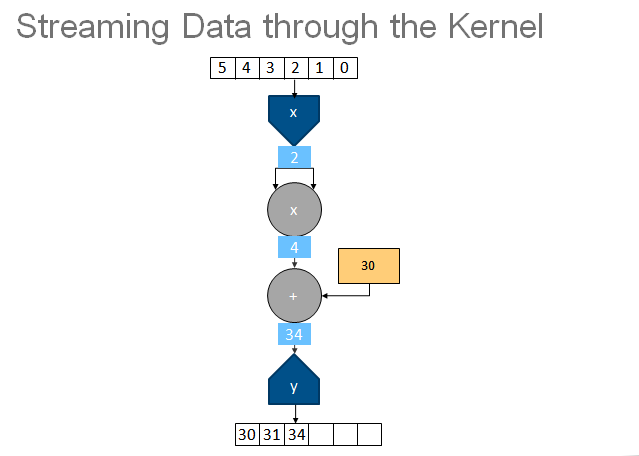


Na narednim slikama prikazan je protok podataka kroz kernel prilikom zadatih izračunavanja.









# Projekat detekcije ivica na slikama u MaxCompiler-u

U nastavku ću predstaviti i detaljno objasniti kod svog projekta u MaxCompiler-u, gde sam , koristeći principe data-flow programiranja , detektovao ivice na slikama uz pomoć Sobel filtera.

package movingaveragesimple;

import com.maxeler.maxcompiler.v2.kernelcompiler.Kernel;

import com.maxeler.maxcompiler.v2.kernelcompiler.KernelParameters;

import com.maxeler.maxcompiler.v2.kernelcompiler.stdlib.KernelMath;

import com.maxeler.maxcompiler.v2.kernelcompiler.types.base.DFEVar;

import java.lang.\*;

class MovingAverageSimpleKernel extends Kernel {

MovingAverageSimpleKernel(KernelParameters parameters) {

super(parameters);

int width = 920;

DFEVar inImage = io.input("x", dfeInt(32));

Klasa MovingAverageSimpleKernel služi da definise ono što će se izvršavati u okviru Kernela. Promenjiva width predstavlja dužinu slike koje na kojoj se detektuju ivice.

DFEVar inImage služi za pamćenje ulaznih vrednosti.Ulazne vrednosti su nam 32-bitni podaci , pošto je slika predstavljena preko niza koji za svaki piksel na slici u 32- bitnoj promenjivoj pamti 3 vrednosti: integer vrednosti za crvenu, plavu i zelenu boju u opsegu od 0 do 255.

Potrebno je iz te 32-bitne promenjive izdvojiti vrednosti za crvenu, plavu i zelenu boju, a to se radi pomoću operatora za shiftovanje.Takođe, pošto je potrebno dobiti osmobitne vrednosti za svaku od promenjivih, potrebno je i izvršiti operaciju AND sa brojem koji je u binarnom zapisu predstavljen kao 1111 1111.Na taj način će u promenjivama G i B biti zapamćeno samo poslednjih 8 bitova.

DFEVar window[][] = new DFEVar[3][3];

for (int x = -1; x <= 1; x++)

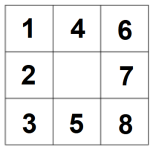
for (int y = -1; y <= 1; y++)

window[x + 1][y + 1] = stream.offset(inImage, y \* width + x);

Pre toga, oko svakog učitanog piksela potrebno je da izdvojimo okolne elemente, tj. matricu dimenzija 3 \* 3. Zbog toga definišemo promenjivu window[][] koja pamti matricu sa elementima koji okružuju piksel koji računamo.

Učitavanje okolnih elemenata vršimo preko stream.offset(inImage, y \* width + x); pri čemu nam je potrebna promenjiva width koja predstavlja širinu slike , kako bi se izračunao pomeraj u nizu piksela ulazne slike.

Redosled obilaska matrice koji ću koristiti prilikom implementiranja maske Sobel filtera je na narednoj slici.



DFEVar R\_1 = window[0][0] >> 16; //shift 16 bita u desno da bi dobili vrednost R

DFEVar G\_1 = (window[0][0] >> 8) & 0xFF; // shift 8 bita i and sa 1111 1111 da bi dobili G

DFEVar B\_1 = window[0][0] & 0xFF; //and 1111 1111 da bi dobili vrednost B\*/

DFEVar r1 = R\_1 ;

DFEVar g1 = G\_1 ;

DFEVar b1 = B\_1 ;

////////////////////////////////////////

R\_1 = window[1][0] >> 16;

G\_1 = (window[1][0] >> 8) & 0xFF;

B\_1 = window[1][0] & 0xFF;

DFEVar r24 = R\_1 ;

DFEVar g24 = G\_1 ;

DFEVar b24 = B\_1 ;

//////////////////////////////////////

R\_1 = window[2][0] >> 16;

G\_1 = (window[2][0] >> 8) & 0xFF;

B\_1 = window[2][0] & 0xFF;

DFEVar r3 = R\_1 ;

DFEVar g3 = G\_1 ;

DFEVar b3 = B\_1 ;

///////////////////////////////////

R\_1 = window[0][2] >> 16;

G\_1 = (window[0][2] >> 8) & 0xFF;

B\_1 = window[0][2] & 0xFF;

DFEVar r6 = R\_1 ;

DFEVar g6 = G\_1 ;

DFEVar b6 = B\_1 ;

//////////////////////////////////

R\_1 = window[1][2] >> 16;

G\_1 = (window[1][2] >> 8) & 0xFF;

B\_1 = window[1][2] & 0xFF;

DFEVar r75 = R\_1 ;

DFEVar g75 = G\_1 ;

DFEVar b75 = B\_1 ;

///////////////////////////////////

R\_1 = window[2][2] >> 16;

G\_1 = (window[2][2] >> 8) & 0xFF;

B\_1 = window[2][2] & 0xFF;

DFEVar r8 = R\_1 ;

DFEVar g8 = G\_1 ;

DFEVar b8 = B\_1 ;

U prethodnom kodu izdvojili smo vrednosti boja za svaki piksel koji nam je potreban za dalje izračunavanje.Radi korišćenja manjeg broja promenjivih, za piksele oznacene brojevima 2 i 4, kao i za piksele označene sa 7 i 5 koristićemo promenjive sa oznakama 24 i 75.

DFEVar result\_R = (r6+2\*r75+r8)-(r1+2\*r24+r3);

DFEVar result\_G = (g6+2\*g75+g8)-(g1+2\*g24+g3);

DFEVar result\_B = (b6+2\*b75+b8)-(b1+2\*b24+b3);

r24 = window[0][1] >> 16;///r4

g24 = (window[0][1] >> 8) & 0xFF;

b24 = window[0][1] & 0xFF;

r75 = window[2][1] >> 16;///r5

g75 = (window[2][1] >> 8) & 0xFF;

b75 = window[2][1] & 0xFF;

DFEVar result\_RR = (r1+2\*r24+r6)-(r3+2\*r75+r8);

DFEVar result\_GG = (g1+2\*g24+g6)-(g3+2\*g75+g8);

DFEVar result\_BB = (b1+2\*b24+b6)-(b3+2\*b75+b8);

DFEVar result\_Red= KernelMath.sqrt((result\_R\*result\_R+(result\_RR\*result\_RR)));

DFEVar result\_Green= KernelMath.sqrt((result\_G\*result\_G)+(result\_GG\*result\_GG));

DFEVar result\_Blue= KernelMath.sqrt(result\_B\*result\_B+result\_BB\*result\_BB);

result\_R = ((result\_Red & 0xFF ) << 16 ) | ((result\_Green & 0xFF) << 8)| (result\_Blue & 0xFF);

io.output("y", result\_R, result\_R.getType());

}

}

Sobel filter za konvoluciju sa ulaznom slikom koristi kernel tj. matricu dimenzija 3\*3.Kernel se sastoji se od piksela koji okružuju središnji piksel koji je na kraju potrebno izračunati.Vrednosti okolnih piksela određuju rezultujuću vrednost za središnji piksel.

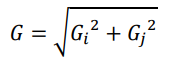
Kod Sobel filtera, detekcija ivica vrši se u dva smera. Zato je potrebno izračunati dve vrednosti koje predstavljaju gradijente po x i po y osi.Gradijent po x osi računa se na sledeći način

rezultatX =(r6+2\*r7+r8)-(r1+2\*r2+r3)

Pri čemu r6 , r7 i r8 predstavljaju vrednosti boje za elemente u matrici 3\*3 koji su u poslednjoj koloni, dok r1, r2 i r3 predstavljaju vrednosti za elemente u matrici koji su u prvoj koloni.

Analogno, vrednosti za gradijent po y osi dobijamo po formuli rezultatY=(r1+2\*r4+r6)-(r3+2\*r5+r8), gde su r1, r4 i r6 vrednosti u prvoj vrsti, a r3, r5 i r8 vrednosti u poslednjoj vrsti matrice.

Jačina ivice računa se po Pitagorinoj teoremi, tako da se rezultujući gradijent računa kao



Gde Gi i Gj predstavljaju gradijente po x i y osi.

Kada smo dobili rezultujuće gradijente za svaku boju od koje se sastoje pikseli, potrebno je da tu vrednost vratimo, tj. da je prikažemo u krajnjoj slici koja će prikazivati ivice. Za to nam je ponovo potrebna operacija shiftovanja, pomoću koje ćemo sada od 3 osmobitne promenljive, dobiti 32-bitnu promenjivu za izlaznu sliku.

#include <math.h>

#include <stdio.h>

#include <stdlib.h>

#include <MaxSLiCInterface.h>

#include <stdint.h>

void loadImage(char \*filename, int \*\*dest, int \*width, int \*height,int grayscale) {

char buffer[200];

FILE \*file = fopen(filename, "r");

if (file == NULL) {

printf("Error opening file %s.", filename);

exit(1);

}

fgets(buffer, 200, file); //

fscanf(file, "%d %d\n", width, height);

fgets(buffer, 200, file); //

\*dest = malloc((\*width) \* (\*height) \* sizeof(int));

int pixelsRead = 0;

int pixel = 0;

int currComponent = 0;

for (int i = 0; i < ((\*width) \* (\*height) \* 3); i++) {

int v;

int got = fscanf(file, "%d", &v);

if (got == 0) {

printf("Unexpected end of file after reading %d color values", i);

exit(1);

}

switch (currComponent) {

case 0:

pixel = v;

currComponent++;

break;

case 1:

pixel = (pixel << 8) | (v & 0xFF);

currComponent++;

break;

case 2:

pixel = (pixel << 8) | (v & 0xFF);

currComponent = 0;

if (grayscale)

pixel = ((pixel >> 16) & 0xff) \* 3 / 10 + ((pixel >> 8) & 0xff)\* 59 / 100 + ((pixel) & 0xff) \* 11 / 100;

(\*dest)[pixelsRead] = pixel;

pixelsRead++;

break;

}

}

fclose(file);

}

Funkcija za učitavanje slike formata .ppm. Radi se o formatu slika bez kompresije, gde je svaki piksel prikazan sa 3 vrednosti u opsegu od 0 do 255, za svaku od boja (crvenu , zelenu i plavu ). Na početku fajla u kom je slika , nalaze se i tri podatka, height i width, tj. visina i širina slike, kao i grayscale, tj. podatak o tome da li je slika crno-bela.

Ako je slika crno-bela, vrši se njena konverzija na osnovu formule date u kodu.

void writeImage(char \*filename, int \*data, int width, int height, int grayscale) {

FILE \*file = fopen(filename, "w");

fprintf(file, "P3\n");

fprintf(file, "%d %d\n", width, height);

fprintf(file, "255\n");

for (int i = 0; i < width \* height; i++) {

if (grayscale) {

if (data[i] > 255)

printf("Out of range %d at pixel %d\n.", data[i], i);

if (data[i] < 0)

printf("Data < 0. %d at pixel %d\n.", data[i], i);

for (int j = 0; j < 3; j++)

fprintf(file, "%d\n", (int) data[i]);

} else {

fprintf(file, "%d\n", ((int) data[i]) >> 16);

fprintf(file, "%d\n", (((int) data[i]) >> 8) & 0xff);

fprintf(file, "%d\n", (((int) data[i])) & 0xff);

}

}

fclose(file);

}

Funkcija writeImage služi za pamćenje slike.Ako je slika crno-bela, onda se svaki piksel opisuje jednim brojem od 0 do 255, dok ako se radi o slici u boji, po 8 bitova rezervisano je za svaku od 3 boje.

int main(void)

{

printf("Ucitava se slika.\n");

int32\_t \*inImage;

int width = 0, height = 0;

loadImage("slikaferrari1.ppm", &inImage, &width, &height, 0);

int dataSize = width \* height \* sizeof(int);

// Alocira bufer za output sliku

int32\_t \*outImage = malloc(dataSize);

printf("Izvrsava se Kernel.\n");

MovingAverageSimple(width \* height, inImage ,width\*height\*sizeof(int), outImage, width\*height\*sizeof(int));

printf("Sacuvana je slika .\n");

writeImage("imagess.ppm", outImage, width, height, 0);

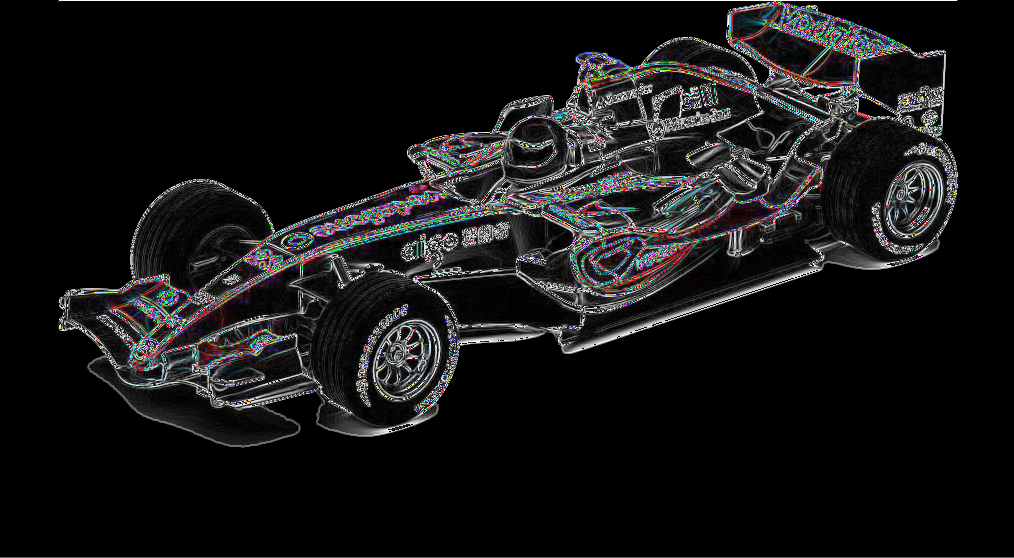
return 0;

}

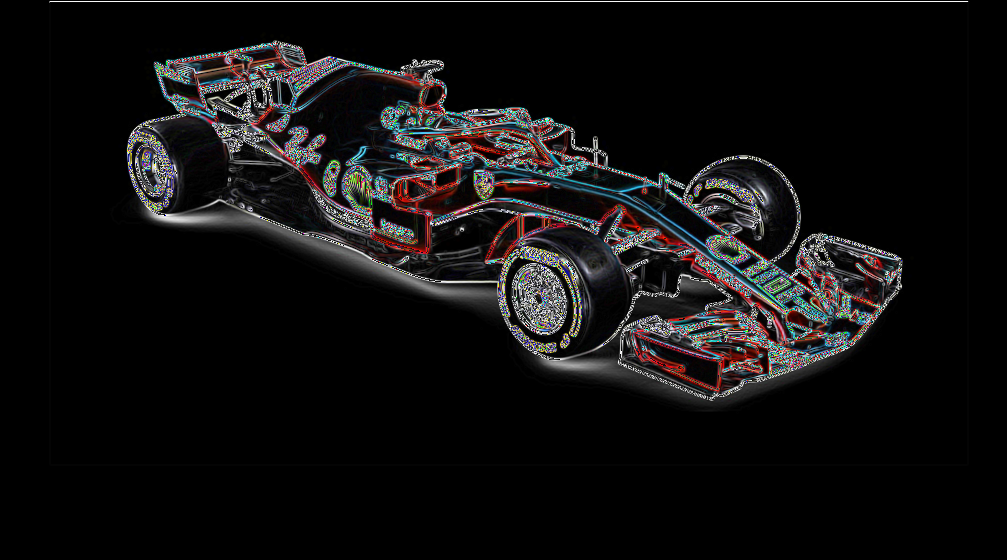
Glavna funkcija koja učitava sliku, zatim poziva izvršenje kernela kroz funkciju MovingAverageSimple(width \* height, inImage ,width\*height\*sizeof(int), outImage, width\*height\*sizeof(int)) i na kraju pamti rezultujuću sliku koja prikazuje detektovane ivice.

Na narednim slikama dati su primeri ulaznih i izlaznih slika, tj. slika na kojima su detektovane ivice pomoću Sobel filtera.



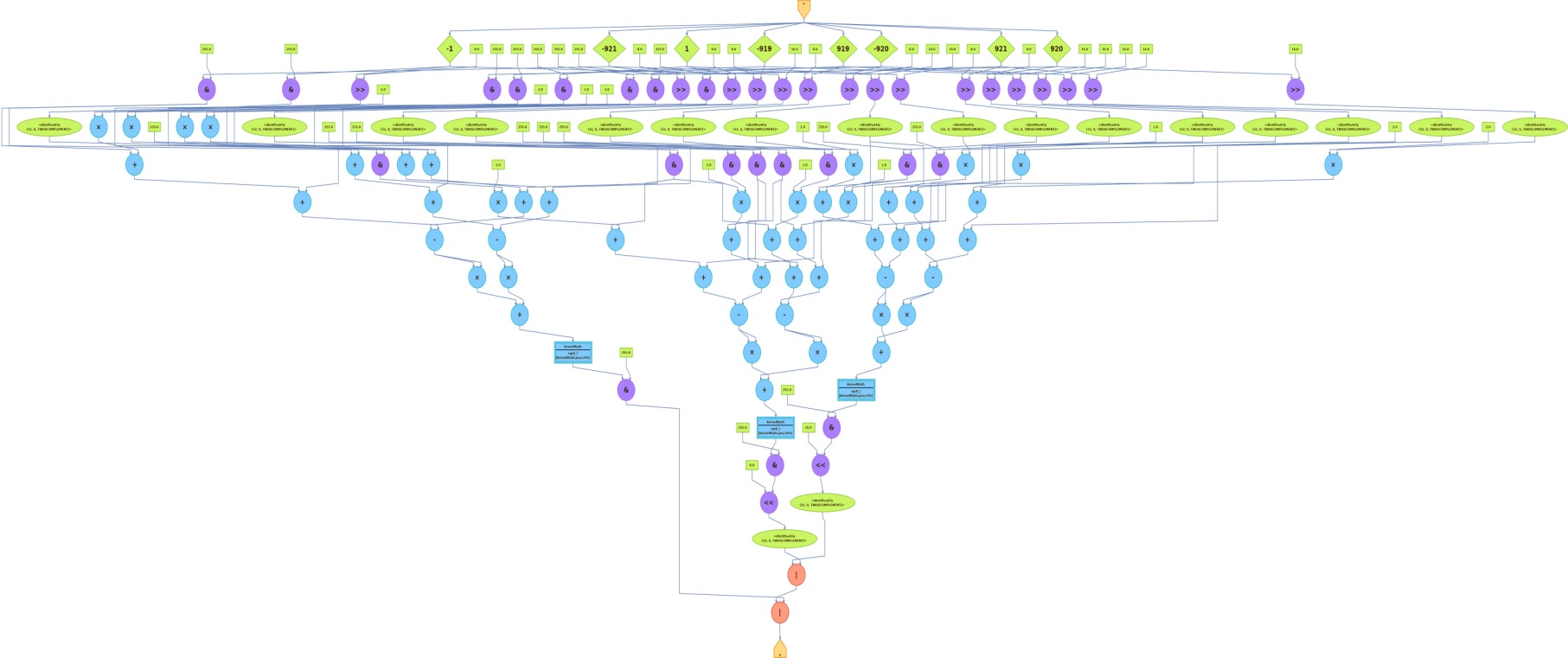






Arhitektura kernela data flow programa

Arhitektura kernela data je na narednoj slici i u prilogu uz ovaj seminarski rad .Na ovoj slici vidimo koji su elementi upotrebljeni u okviru kernela.



# Zaključak

Detekciju ivica na slikama je moguće izvršiti uz pomoć data flow programiranja ali i koristeći druge klasične načine programiranja. Pošto su ulazni podaci pikseli slike, učitavanje se vrši uz pomoć toka podataka (i pomoću offset funkcije koja na osnovu pomeraja učitava podatke iz toka podataka), a na osnovu slike na kojoj je data arhitektura kernela data flow programa, vidimo i da je paralelizacija moguća kada se na osnovu učitanih piksela računaju gradijenti po X i Y osi.Data flow programiranje predstavlja dobro rešenje u slučajevima kada je moguće paralelizovati izračunavanja u okviru programa i kada se učitavanje podataka vrši iz toka podataka u kom su podaci u istom formatu, tako da je detekciju ivica na slikama moguće efikasno realizovati koristeći koncept data flow programiranja.

# Literatura

Multiscale Dataﬂow Programming Version 2014.1, Maxeler Technologies. (zvanično uputstvo za okruženje u kome je projekat urađen)

<https://rti.etf.bg.ac.rs/rti/ir4vls/vezbe/> Materijal sa predavanja i vezbi iz predmeta Računarski VLSI sistemi, Elektrotehnički fakultet – Beograd ,Katedra za Računarsku tehniku i informatiku

<https://www.researchgate.net/publication/297736749_Comparison_of_Edge_Detection_Techniques>, Comparison of Edge Detection Techniques,Radhika Chandwadkar, Saurabh Dhole, VaibhavGadewar,Deepika Raut, Prof. S. A. Tiwaskar,Computer Engineering Department, VishwakarmaInstitute of Information Technology,Pune

<http://portal.sinteza.singidunum.ac.rs/Media/files/2015/123-128.pdf> POREĐENJE PERFORMANSI RAZLIČITIH METODA DETEKCIJE IVICE U DIGITALNOJ SLICI, Hana Stefanović\*1 , Svetlana Štrbac-Savić\*1 , Dejan Milić\*2 1\* Visoka škola elektrotehnike i računarstva strukovnih studija u Beogradu, Vojvode Stepe 283, Beograd, Srbija 2\* Elektronski fakultet Univerziteta u Nišu, Katedra za Telekomunikacije, Aleksandra Medvedeva 14, Niš, Srbija

<http://www.cse.usf.edu/~r1k/MachineVisionBook/MachineVision.files/MachineVision_Chapter5.pdf>

<https://www.maxeler.com/media/documents/MaxelerWhitePaperProgramming.pdf>, Programming MPC Systems White Paper ,Maxeler Technologies

<https://github.com/Altron01/Sobel-operator-Edge-Detection/blob/master/sobelOperator.py>