SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 312

IZLUČIVANJE ISTAKNUTIH TOČAKA SLIKE U EKSTREMIMA KONVOLUCIJSKOG ODZIVA RAZLIKE GAUSSOVIH FUNKCIJA

Ozana Živković

Zagreb, lipanj 2011.

Zahvala mentoru doc. dr. sc. Siniši Šegviću na stručnoj pomoći i danim sugestijama pri izradi diplomskog rada.

Sadržaj

1. Uvod

Ljudsko oko prima zasebne slike koje mozak pronalaskom korespondentnih točaka povezuje u jedinstveni prikaz scene. Čovjek je u mogućnosti prepoznati objekte na slici i nakon što promjeni kut gledišta ili se nad slikom izvrše određene transformacije, kao što su rotacija, promjena rezolucije slike i slično. Po tom uzoru, u računalnom vidu postoji posebno područje koje ima u cilju ustanoviti korespondenciju između slika. Trenutno postoji veliki broj metoda čija je glavna namjena pronalazak takvih korespondentnih točaka u paru ili čak nizu slika. Pošto se nad slikama izvedu određene transformacije, sve metode ne daju zadovoljavajuće rezultate. Zbog toga je u ovom radu odabran algoritam SIFT koji slovi kao najstabilnija metoda za pronalaženje korespondencije. Rezultat algoritma SIFT su točke čija je glavna odlika invarijantnost na rotaciju, mjerilo, svjetlinu, šum, te djelomično na afine transformacije. Postupak pronalaženja karakterističnih značajki zahtjeva složena matematička izračunavanja koja povećavaju vrijeme izvođenja algoritma SIFT na procesoru opće namjene. To daje mogućnost ubrzanja izvođenja algoritma korištenjem potencijala grafičkog procesora. Naime, obrada slike predstavlja idealan zadatak za grafički procesor. Razlog tome je što se sastoji od velikog broja slikovnih elemenata, pa paralelna obrada može doći do izražaja. U ovom radu razmatrana je implementacija nekih dijelova algoritma SIFT na grafičkom procesoru.

Rad je strukturiran kako slijedi. U poglavlju 2. i 3. analiziraju se koncepti relevantni za navedeno područje. Teorijska osnova algoritma SIFT prikazana je u poglavlju 4. Osnovne značajke korištenog općenamjenskog okruženja CUDA za programiranje grafičkih procesora predstavljene su u poglavlju 5. Glavni dio rada je implementacija algoritma SIFT za izvođenje na procesoru opće namjene i implementacija u okruženju CUDA za izvođenje na grafičkom procesoru. U poglavlju 6. objašnjena je programska realizacija algoritma za procesor opće namjene. U poglavlju 7. napravljena je analiza rezultata, te su određeni kritični dijelovi algoritma SIFT. Programska realizacija kritičnih dijelova u okruženju CUDA prikazana je u poglavlju 8. Na kraju je provedena analiza rezultata i dobivenog ubrzanja, te su navedena buduća moguća poboljšanja u cilju postizanja još većeg ubrzanja.

2. Lokalne značajke

U računalnom vidu lokalne su značajke dijelovi slike koji se prema određenim svojstvima razlikuju od svih ostalih u svojoj neposrednoj okolini. Složenim matematičkim računanjima nastoji se određenu značajku izdvojiti iz okoline i učiniti je invarijantnom na tipične promjene kao što su promjene u svjetlini, skaliranju, rotaciji i slično. Da bi se pojedina točka slike proglasila značajkom, ona mora posjedovati neka određena svojstva, kao što su:

- **Ponovljivost**

  Značajka koja identificira objekt na slici trebala bi biti prepoznatljiva u većini slika tog objekta, pri čemu su slike snimljene sa različitih gledišta.
- **Prepoznatljivost**

  Značajka bi trebala sadržavati što više jednoznačnih informacija na temelju kojih bi se lakše detektirala [18].
- **Lokalnost**

  Značajka je lokalna, te se prilikom određivanja značajke vrlo često definira i uže susjedstvo [18].
- **Kvantiteta**

  Detektirani broj značajki treba biti dovoljan da bi se raspoznali i manji objekti slike [18].
- **Preciznost**

  Lokacija značajke treba biti to preciznije definirana, a to se najčešće postiže korištenjem različitih interpolacija [18].

Nabrojana svojstva većinom nisu u potpunosti prisutna kod pojedine vrste značajke. Ukoliko značajka posjeduje svojstvo lokalnosti imat će manje informacija, a samim time ne može istovremeno zadovoljavati kriterije svojstva prepoznatljivosti. Značajke se razlikuju po načinu, odnosno obliku reprezentacije. Primjerice značajke mogu biti obične točke, kutovi, rubovi, pa čak i regije. Koje će se značajke pronaći, te kojeg oblika, ovisi o primijenjenom detektoru. U nastavku je prikazana podjela značajki u dvije osnovne skupine i navedeni su detektori koji se najčešće koriste u takvoj podjeli.

2.1. Kutne značajke i detektori

Vrlo se često detektiraju u slici na spojevima ili teksturama, dok je glavna namjena pronalaženje pravih kutova 3D svijeta. Neki od detektora koji se koriste u pronalaženju ovakvih značajki su niže nabrojani:

- **Harrisov detektor**

  Predstavlja jedan od najefikasnijih pristupa s fiksnim mjerilom. Temelji se na autokorelacijskoj matrici koja opisuje distribuciju gradijenata u susjedstvu promatrane točke. Ideja je pronaći točke koje imaju maksimalnu prepoznatljivost unutar susjedstva unaprijed zadane veličine. Značajke pronađene ovim detektorom invarijantne su na rotaciju, translaciju i promjene svjetline [18].

- **SUSAN detektor** (engl. *Smallest Univalue Segment Assimilating Nucleus*, SUSAN)

  Za razliku od Harrisovog detektora ova metoda temelji se na morfološkom pristupu. Za svaki slikovni element definira se kružno susjedstvo fiksnog radijusa. Intenzitet slikovnog elementa koji je centralan u svom susjedstvu uzima se kao referentni. Preostali slikovni elementi u susjedstvu dijele se u dvije kategorije ovisno o tome da li je vrijednost njihovog intenziteta dovoljno ili nedovoljno slična referentnom. Rezultat su značajke koje su invarijantne na promjene u rotaciji i translaciji [18].

- **Afini Harris-Laplaceov detektor**

  Harris-Laplace temelji se na Harrisovom detektoru, ali se detekcija obavlja kroz više mjerila gdje se za selekciju karakterističnog mjerila koristi Laplaceov operator. Na taj se način uz invarijantnost na rotaciju i translaciju dobiva detektor inavrijantan na mjerilo. Harris-Afine detektor je prošireni Harris-Laplace detektor. Koristi se iterativni algoritam kojim se estimiraju eliptične afine regije koje omogućavaju pronalazak kutnih značajki invarijantnih na afine transformacije. [1, 18].

2.2. Regionalne značajke i detektori

Značajkama se odnose na dijelove slike koji se svojom svjetlinom ističu od okoline. Detektori ove skupine pronalaze regije na temelju intenziteta ili ekstrema čija je stabilnost prethodno ispitana.

- **Hesseov detektor**

  Koristi Hesseovu matricu drugog reda, gdje je svaki član matrice derivacija drugog reda Gaussovim filtrom zaglađene slike. Determinanta i trag ove matrice koriste se u raznoraznim filtrima. Najpoznatiji je Laplaceov filtar koji se koristi u detekciji regionalnih značajki. [1, 18].

- **Afine Hessian-Laplace detektor**

  Radi se o inačicama Hassian detektora gdje su detektirane značajke invarijatne na mjerilo i afinu transformaciju.

- **Detektor ispupčenih regija**

  Predstavlja novi princip pronalaska regionalnih značajki. Zapravo, ideja je pronaći ispupčene regije pomoću entropije vjerojatnosne distribucije intenziteta. Da bi se značajke precizno locirale koristi se dodatni kriterij vlastite različitosti (engl. *self-dissimilarity*) [1]. Broj značajki detektiran na ovaj način je vrlo mali, ali se ova metoda često koristi zbog njihove stabilnosti.

- **Detektor na temelju intenziteta** (engl. *Intensity based regions,* IBR)

  Detektor radi na principu da radijalno ispitujući intenzitet traži ekstreme kroz višestruka mjerila. Nakon što su pronađeni svi ekstremi promatranog lokalnog ekstrema, njihovim povezivanjem dobiva se regija. Detektirana regija je invarijantna na afine transformacije [1,18].

- **Detektor maksimalno stabilnih ekstrema** (engl. *Maximally Stable Extremal Regions*, MSER)

  Radi se o algoritmu povezanih komponenti nad slikom na kojoj je primijenjen određeni prag. Tako ovaj detektor pronalazi regije koje su maksimalno stabilne obzirom na primijenjen određeni prag. Pri tome nije riječ o lokalnom ili globalnom pragu, već se regije testiraju kroz čitav niz pragova. Pronađene regije su invarijantne na afine transformacije, a ova metoda je zapravo najbrža od svih iz tog skupa. Vrlo dobre rezultate daje na strukturiranim slikama [18].

3. Obrada slike linearnim filtrima

Predstavljaju jedne su od najvažnijih operacija u području obrade slike, a koriste se prilikom izgradnje algoritma SIFT. U cilju ubrzanja ključna je njihova implementacija u okruženju CUDA za izvođenje na grafičkom procesoru.

Osnovna je karakteristika ovih operacija da se izlazna vrijednost nekog slikovnog elementa uzima kao linearna kombinacija ulaznih slikovnih elementa u njegovoj okolini (susjedstvu). Operacije se izvode računanjem konvolucije slike s jezgrom filtra kojom je određeno ponašanje filtra, prema izrazu:

$$L[i,j] = I[i,j] * G[i,j] = \sum_m \sum_n I[i-m, j-n] * G[m,n] \tag{1.1}$$

Jezgra filtra je matrica koja sadrži težinske faktore kojima se množe vrijednosti slikovnih elementa prilikom računanja nove vrijednosti pojedinog elementa. Matrica se „prevlači“ preko elemenata promatrane slike. Svaki element slike množi se sa odgovarajućim vrijednostima matrice. Dobiveni umnošci se zbrajaju, a trenutna vrijednost promatranog slikovnog elementa odredišne slike zamjenjuje se krajnjim rezultatom.

Dvodimenzionalni konvolucijski filtar širine m i visine n za svaki izlazni slikovni element zahtijeva ukupno $m * n$ operacija množenja. Ako je korišteni filtar separabilan, postupak se uvelike ubrzava. Razlog je tome što su separabilni dvodimenzionalni filtri posebna vrsta filtara koji se mogu izraziti kao kompozicija dva jednodimenzionalna filtra.

Može se vidjeti da je konvolucija asocijativna operacija, što znači da vrijedi:

$$I * (R * S) = (I * R) * S \tag{1.2}$$

Isti će se rezultat dobiti ako konvoluciju nad slikom I obavimo dva puta, prvo retčanim vektorom R, a drugi put stupčanim vektorom S, koristeći prethodno dobiveni rezultat kao ulaz. Na taj način, nad slikom obavljaju se dvije jednodimenzionalne konvolucije, jedna nad recima slike, a druga nad stupcima slike. Ovim načinom, vrijeme obrade slike se znatno skraćuje. Zbog toga će se u ovom radu koristit svojstvo separabilnosti Gaussovog filtra.

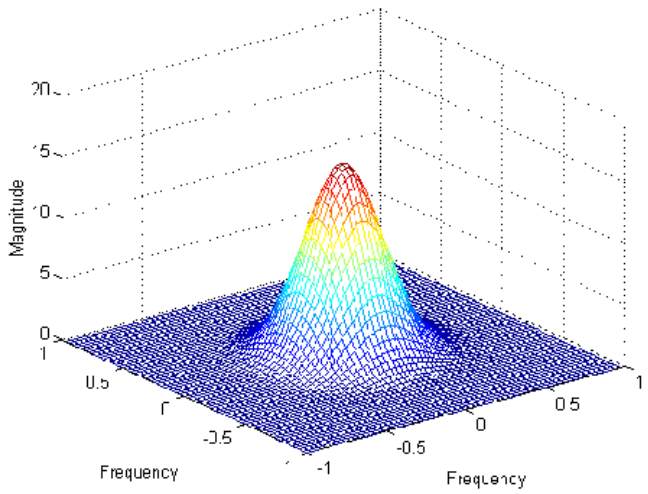
3.1. Zaglađivanje slike Gaussovim filtrom

Gaussov filtar je niskopropusni filtar koji se vrlo često koristi kod obrade slika. Glavno svojstvo Gaussovog filtra je smanjivanje šuma na slici. Šum se uglavnom sastoji od visokih frekvencija koje Gaussov filtar ne propušta, te zbog toga dolazi do uklanjanja šuma. Korištenjem Gaussovog filtra se smanjuje količina detalja, odnosno slika se zaglađuje. Gaussovo zaglađivanje se vrlo često primjenjuje nad slikama koje se dalje koriste u algoritmima računalnog vida. Tako se i prilikom pronalaženja lokalnih značajki na slikama koristi ova tehnika, te se na taj način poboljšava struktura slika različitih razina mjerila.

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2} \tag{1.3}$$

Jezgra Gaussovog filtra popunjava se vrijednostima koje se računaju na temelju izraza 1.3. Gdje je:

- σ standardna devijacija Gaussove funkcije
- x je udaljenost od centralnog slikovnog elementa po osi x
- y je udaljenost od centralnog slikovnog elementa po osi y



Slika 1. 2D Gaussova funkcija

4. Opis algoritma SIFT (engl. Scale Invariant Feature Transform)

Prvotni cilj ovog rada je pronaći stabilne značajke koje će se dalje koristiti u ustanovljavanju korespondencije u slikama. Za rješavanje problema određivanja korespondentnih značajki u slikama odabran je algoritam SIFT (engl. *Scale Invariant Feature Transform*). Autor algoritma David G. Lowe je 1999. godine predložio njegovu prvu verziju. Algoritam je patentiran u Američkom uredu za patente 2000., a kasnije je poboljšan i ponovno izdan 2004. godine. Temelji se na ekstrakciji značajki koje su invarijantne na rotaciju i mjerilo. Relativno je otporan na afine transformacije, varijacije u osvjetljenju i zaklonjenost objekata. Razlog zašto je odabran ovaj algoritam je njegova robusnost. Odnosno, mogućnost detekcije karakterističnih značajki u slikama koje ostaju očuvane i nakon prethodno spomenutih transformacija. Dobivene značajke imaju poznatu lokaciju, mjerilo, orijentaciju i opisni vektor, te se mogu dalje koristiti za ustanovljavanje korespondencije u slikama. Algoritam karakterizira mala vjerojatnost pogrešnog uparivanja korespondentnih značajki i uparivanje s velikim bazama podataka lokalnih značajki [9].

Osnovni postupak ovog algoritma može se podijeliti u nekoliko koraka. Na početku algoritma koristeći razlike Gaussovim filtrom zaglađenih slika kao aproksimaciju Laplaceovog rubnog operatora pronalaze se ekstremi invarijantni na mjerilo. Na taj način dobiva se veliki skup ekstrema iz kojeg je potrebno ukloniti one kandidate koji su nestabilni u pogledu niskog kontrasta, a osjetljivi su na šum i one kandidate koji su slabo lokalizirani uz rub. Nakon dobivanja skupa karakterističnih značajki slijedi dodjeljivanje jedne ili više orijentacija svakoj značajki iz skupa. Orijentacija se određuje računanjem gradijenata u okolini promatrane značajke, a osigurava invarijatnost na rotaciju. Na kraju algoritma svakoj značajki dodjeljuje se opisni vektor koji osigurava djelomičnu invarijatnost na varijacije u osvjetljenju, te promjene položaja kamere. Spomenuti postupak formira četiri osnovna koraka algoritma SIFT [9]. U nastavku rada slijedi detaljniji opis svakog koraka.

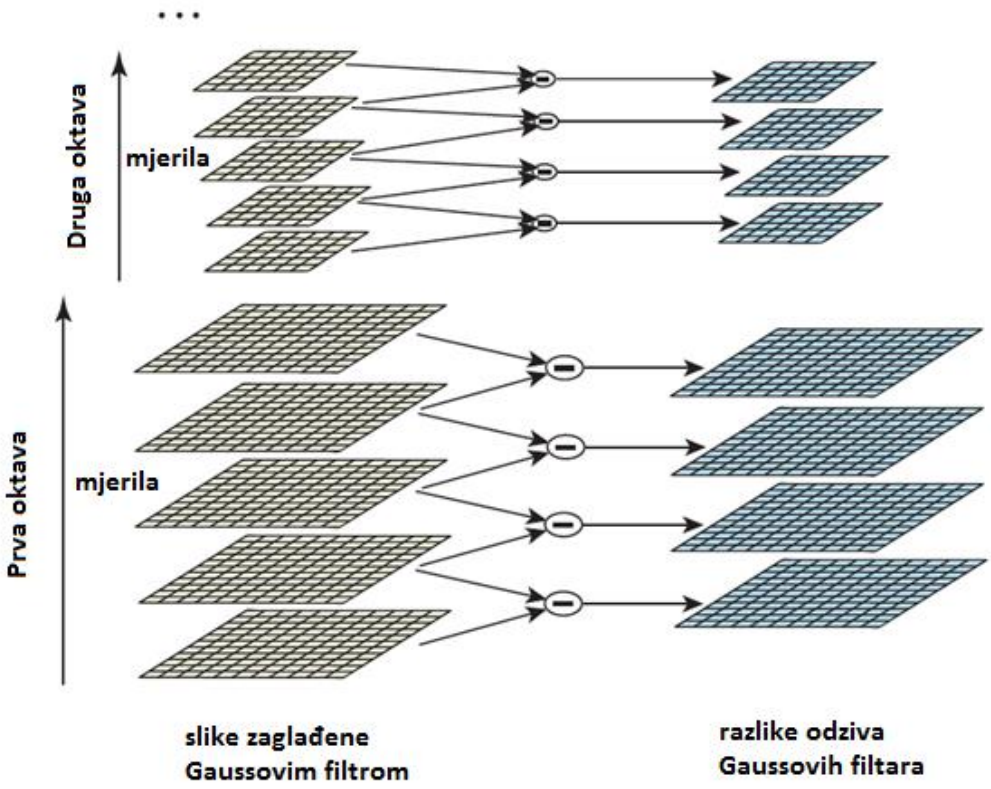
4.1. Izgradnja prostora mjerila

U prvom koraku određuju se slike u kojima će se tražiti potencijalni kandidati značajki. Na samom početku ulazna slika se zaglađuje Gaussovim filtrom sa standardnom devijacijom iznosa 0.5. Time se uklanjaju neželjeni učinci uslijed diskretizacije (engl. *aliasing*). Idući korak je povećanje rezolucije ulazne slike koristeći linearnu interpolaciju. D. Lowe predlaže da je dovoljno ulaznu sliku povećati samo jednom za faktor 2. Povećana slika se dalje zaglađuje Gaussovim filtrom jer ovaj filtar ne uvodi nove lažne strukture u rezultantnu sliku.

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \tag{1.4}$$

Kao što se vidi iz izraza 1.4 zaglađena slika L rezultat je konvolucije slike I i Gaussovog operatora G definiranog izrazom 1.3. Što je veći σ u danom izrazu, to je veće zaglađivanje. U idućem koraku σ se poveća za faktor k, a kao rezultat dobije se zaglađena slika L za iduću razinu. U prostoru mjerila slike susjednih razina razlikuju se za faktor k. Sve slike jednake veličine po svim razinama mjerila čine oktavu. Nakon što se završi sa zaglađivanjem slike unutar prve oktave za sva mjerila, rezolucija slike se smanjuje za faktor 2. Smanjivanje slike izvodi se na način da se uzme svaki drugi slikovni element unutar retka i stupca slike. Smanjena slika predstavlja osnovnu sliku iduće oktave koja se dalje zaglađuje Gassovim filtrom za sva mjerila. Opisani se postupak ponavlja za svaku oktavu. Broj oktava i razina mjerila ovisi o veličini ulazne slike, a D. Lowe sugerira kako je 5 mjerila po 4 oktave dovoljno za uspješno detektiranje značajki [9].

Temeljem zaglađenih slika L koje čine Gaussovu piramidu (vidi sliku 2. lijevo) generira se novi skup slika tzv. razlike Gaussiana (engl*. Difference of Gaussians, DoG*) prikazanih na slici 2. desno.



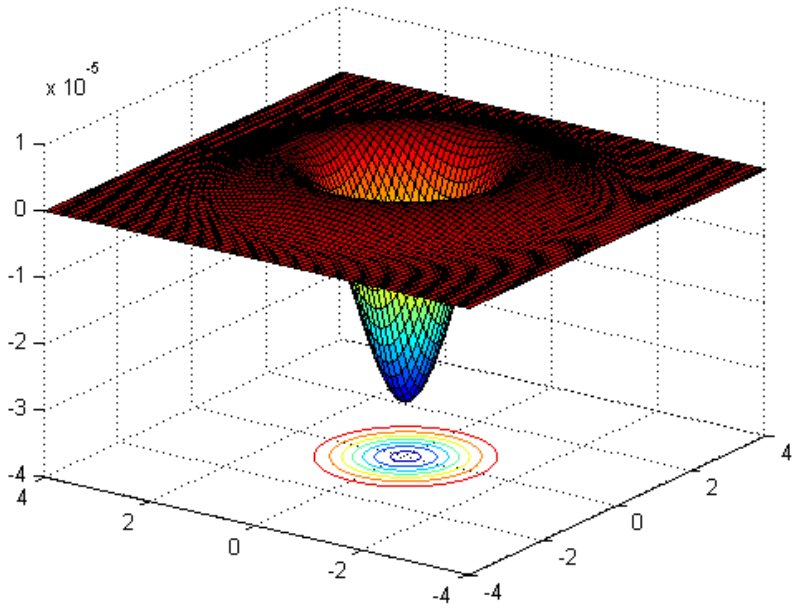
Slika 2. Gaussova piramida (lijevo) i *DoG* piramida (desno) [9]

DoG slike D se računaju oduzimanjem susjednih slika unutar svake oktave Gaussove piramide. Njihov broj jednak je u svim oktavama, a za jedan manji od broja razina mjerila.

$$D(x, y, \sigma) = \big(G(x, y, k\sigma) - G(x, y, \sigma)\big) * I(x, y) \tag{1.5}$$

$$D(x, y, \sigma) = L(x.\, y, k\sigma) - L(x, y, \sigma) \tag{1.6}$$

Za otkivanje točaka maksimuma/minimuma u prostoru mjerila, koje su ujedno i najstabilniji kandidati značajki, nakon zaglađivanja Gaussovim filtrom primjenjuje se Laplaceov rubni operator (engl. *Laplacian of Gaussian, LoG*). Trodimenzionalni prikaz *LoG*-a dan je na slici 3.



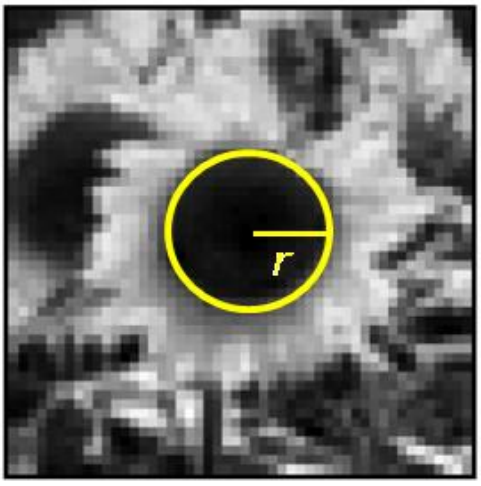
Slika 3. 3D prikaz *LoG*-a u Matlabu

LoG daje ekstremni odziv u regiji prilagođene veličine. Najveći odziv za krug prikazan na slici 4. a) postiže se kad je odziv *LoG*-a poravnat sa krugom (vidi sl. 4 b). *LoG* je definiran kao:

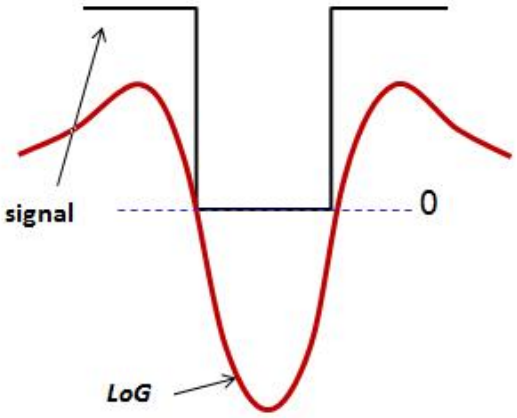
$$\nabla^2 G = \frac{\partial^2 G}{\partial x^2} + \frac{\partial^2 G}{\partial y^2} = (x^2 + y^2 - 2\sigma^2) e^{\frac{-(x^2+y^2)}{2\sigma^2}} \tag{1.7}$$

Zbog toga se maksimalan odziv *LoG*-a za binarni krug radijusa *r* postiže za mjerilo:

$$\sigma = \frac{r}{\sqrt{2}} \tag{1.8}$$

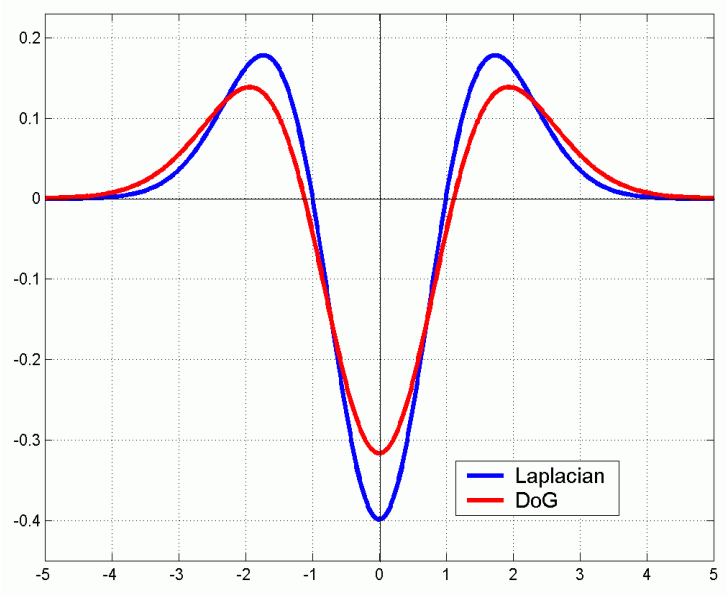


Slika 4. a) Originalna slika



Slika 4. b) Maksimalan odziv LoG-a

Iz izraza 1.7 vidi se da *LoG* zahtjeva proračun drugih parcijalnih derivacija što bi moglo nepogodno djelovati na brzinu cjelokupnog algoritma SIFT. Zbog toga se koriste *DoG* slike koje su dobra aproksimacija *LoG*-a. Na taj se način računanje parcijalnih derivacija svodi na jednu operaciju oduzimanja dviju zaglađenih slika.



Slika 5. Aproksimacija *LoG*-a *DoG*-om

Kako bi se pokazalo da je *DoG* zaista bliska aproksimacija *LoG* operatora (vidi sl. 5) koji je normaliziran po mjerilu može se iskoristiti difuzna toplinska jednadžba koja je parametrizirana po σ :

$$\frac{\partial \mathrm{G}}{\partial \sigma} = \sigma \nabla^2 G \tag{1.9}$$

Pošto se $\frac{\partial \mathrm{G}}{\partial \sigma}$ može aproksimirati pomoću diferencija, dobiva se slijedeća jednadžba:

$$\sigma \nabla^2 G = \frac{\partial G}{\partial \sigma} \approx \frac{G(x,y,k\sigma) - G(x,y,\sigma)}{k\sigma - \sigma} \tag{1.10}$$

Iz čega slijedi:

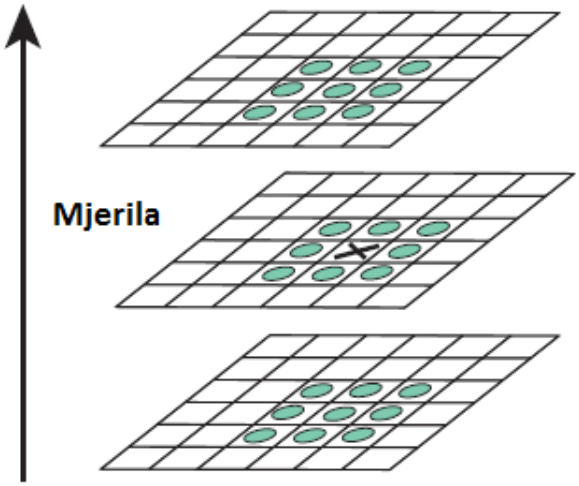
$$G(x,y,k\sigma) - G(x,y,\sigma) \approx (k-1)\sigma^2 \nabla^2 G \tag{1.11}$$

Faktor $(k-1)$ u izrazu 1.11 konstantan je u svim mjerilima, te će stoga točke minimuma/maksimuma ostati na istoj lokaciji. Pogreška aproksimacije će ići u nulu kako

se k približava jedinici. Istraživanje je pokazalo da čak i veće razlike između mjerila (npr. za $k = \sqrt{2}$) ne utječu na stabilnost detekcije ekstrema, kao ni na njihovu lokaciju [9].

4.2. Detekcija i lokalizacija značajki

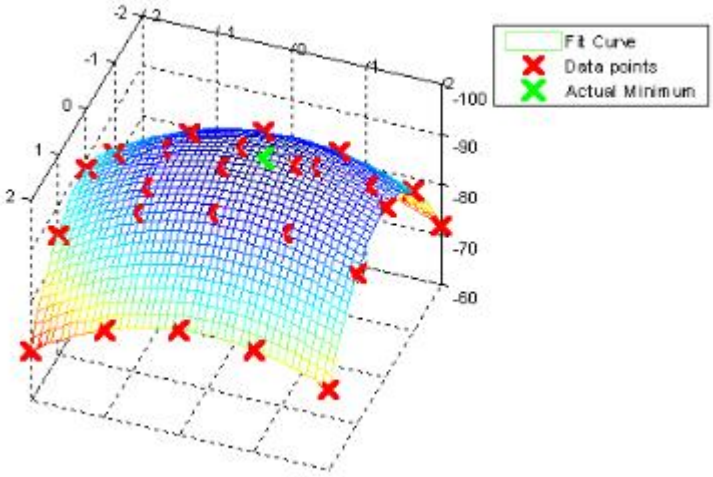
Da bi se kao krajnji rezultat ovog koraka dobile stabilne značajke, potrebno je prvo pronaći kandidate značajki. Kandidati se traže na temelju slika *DoG* iz prethodnog koraka.



Slika 6. Traženje ekstrema u *DoG* slici [9]

Za sva mjerila po svim oktavama, izuzevši rubna mjerila, vrijednost svakog slikovnog elementa slike *DoG* uspoređuje se sa vrijednostima susjednih elemenata. Provjerava se da li je slikovni element slike *DoG* manji ili veći od svih elemenata u regiji 3x3 unutar trenutnog mjerila, te mjerila iznad i ispod (vidi sl. 6). Ukoliko je pojedini slikovni element manji ili veći od svih 26 susjeda, tada on dobiva status kandidata značajke.

Rezultat je velik broj kandidata koji su aproksimacija minimuma/maksimuma. Razlog je tome što se minimumi/maksimumi gotovo nikad ne nalaze točno na poziciji slikovnog elementa. Ekstremi se nalaze između slikovnih elemenata [19]. Na slici 7. može se vidjeti jedan takav primjer. Crveni križići označavaju pozicije slikovnih elemenata, dok je pronađeni lokalni ekstrem označen zeleno.



Slika 7. Minimum lociran između slikovnih elemenata [19]

Međutim, kako se ne može pristupiti podacima između slikovnih elemenata, potrebno je matematički odrediti tu lokaciju. Lokacija se određuje Taylorovom ekspanzijom slike u okolini kandidata značajke [9].

$$D(x) = D + \frac{\partial D^T}{\partial x}x + \frac{1}{2}x^T\frac{\partial^2 D}{\partial x^2}x \tag{1.12}$$

Gdje se D i njezine derivacije procjenjuju na temelju razlike susjeda kandidata značajke, a $x = (x, y, \sigma)^T$ je odmak od te značajke. Lokacija ekstrema $\hat{x}$ određuje se deriviranjem funkcije 1.12 obzirom na x i izjednačavanjem s nulom, što daje:

$$\hat{x} = -\frac{\partial^2 D^{-1}}{\partial x^2}\frac{\partial D}{\partial x} \tag{1.13}$$

Dobiva se lokacija značajke u smislu (x, y, σ), čime se povećava mogućnost podudarnosti i stabilnosti algoritma SIFT.

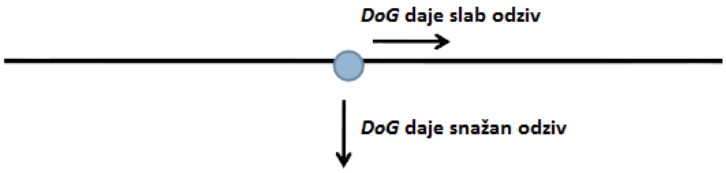
Ako je odmak $\hat{x}$ veći od 0.5 u bilo kojoj dimenziji, onda postoji neki drugi kandidat kojemu je promatrani ekstrem bliži. U tom slučaju lokacija ekstrema se mijenja i ponovno se radi interpolacija korištenjem Taylorovog reda drugog stupnja (vidi izraz 1.12). Konačna vrijednost odmaka $\hat{x}$ dodaje se lokaciji ekstrema kako bi se dobila interpolirana procjena lokacije kandidata značajke.

Ubacivanjem izraza 1.13 u 1.12 dobiva se:

$$D(\hat{x}) = D + \frac{1}{2}\frac{\partial D^T}{\partial x}\hat{x} \tag{1.14}$$

Vrijednost funkcije 1.14 koristi se za odbacivanje ekstrema niskog kontrasta. Odbacuju se svi ekstremi kojima je vrijednost $|D(\hat{x})|$ manja od 0.03. Na taj su način iz skupa kandidata značajki maknuti svi nestabilni kandidati podložni šumu.

Kako bi se dobile samo stabilne značajke nije dovoljno odbacivanje ekstrema niskog kontrasta. *DoG* funkcija daje snažan odziv uz rub, a mali u ostalim smjerovima, kao što je prikazano na slici 8.



Slika 8. Odziv *DoG* funkcije

Postoji veliki broj kandidata značajki koji su i dalje nestabilni obzirom na male količine šuma. Za odbacivanje takvih kandidata iz preostalog skupa kandidata značajki koristi se Hesseova matrica dimenzija 2x2.

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{xy} & D_{yy} \end{bmatrix} \tag{1.15}$$

Derivacije D_{xx}, D_{xy} i D_{yy} procjenjuju se na temelju razlike susjednih slikovnih elemenata kandidata značajke (objašnjeno u odjeljku 6.2). Računanje svojstvenih vrijednosti matrice *H* može se izbjeći računanjem njihovog omjera. Ukoliko se svojstvene vrijednosti označe sa α i β, dobivaju se slijedeće jednadžbe (α je veća od njih):

$$Tr(H) = D_{xx} + D_{yy} = \alpha + \beta \tag{1.16}$$

$$Det(H) = D_{xx}D_{yy} - {D_{xy}}^2 = \alpha\beta \tag{1.17}$$

Ako je determinanta matice *H* negativna promatrani se kandidat odbacuje. U suprotnom se koristi usporedba omjera s određenim pragom r. Odnosno $\alpha = r\beta$, odakle slijedi:

$$\frac{Tr(H)^2}{Det(H)} = \frac{(\alpha+\beta)^2}{\alpha\beta} = \frac{(r\beta+\beta)^2}{r\beta^2} = \frac{(r+1)^2}{r} \tag{1.18}$$

Izraz 1.18 dokazuje kako nisu potrebne pojedinačne svojstvene vrijednosti, već samo njihov omjer. Omjer će biti najmanji u slučaju kad su svojstvene vrijednosti iste, a povećavat će se sa r. Na kraju preostaje provjeriti da li je izraz 1.18 istinit za $r = 10$.

$$\frac{Tr(H)^2}{Det(H)} < \frac{(r+1)^2}{r} \tag{1.19}$$

U skupu ostaju samo stabilni kandidati i dobivaju status značajke. Svakoj značajki pridjeljuje se lokacija, odnosno x i y koordinate, oktava i mjerilo na kojem je pronađena, te vrijednost σ koja se računa prema formuli:

$$\sigma = \sigma_0 * 2^{o+\frac{s}{S}} \tag{1.20}$$

Indeks *o* označava oktavu, *s* mjerilo unutar oktave, a *S* broj intervala na koji je podijeljena svaka oktava. U pravilu *S*+3 predstavlja ukupan broj razina mjerila unutar oktave [9].

4.3. Dodjela orijentacije

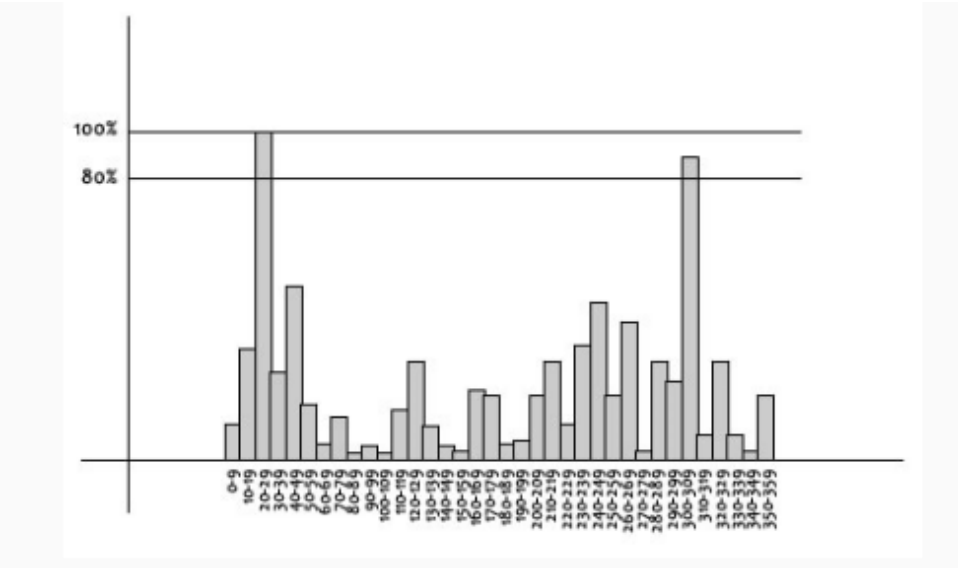
U ovom koraku dodjeljuje se orijentacija svim značajkama koje su prošle selekciju iz prethodnog koraka. Orijentacija se dodjeljuje s obzirom na smjerove gradijenata tog dijela slike. Prvo se odabire Gaussovom funkcijom zaglađena slika L sa mjerilom koje odgovara promatranoj značajki. Na temelju vrijednosti susjednih slikovnih elemenata promatrane značajke, iz odgovarajuće slike L, dobiva se amplituda i kut gradijenta.

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2} \tag{1.21}$$

$$\theta(x,y) = \tan^{-1}\left(\frac{L(x,y+1)-L(x,y-1)}{L(x+1,y)-L(x-1,y)}\right) \quad (1.22)$$

Amplituda, $m(x,y)$ i kut gradijenta, $\theta(x,y)$ računaju se prema izrazima 1.21 i 1.22 za svaki slikovni element u okolini značajke. Okolina, odnosno regija ovisi o mjerilu promatrane značajke, točnije o parametru $1.5 * \sigma$. Zatim slijedi formiranje orijentacijskog histograma. Histogram se sastoji od 36 smjerova, pri čemu svaki smjer pokriva 10 stupnjeva, što daje ukupan raspon od 360 stupnjeva. Svaki izračun θ unutar regije značajke ponderiran vlastitim doprinosom dodaje se odgovarajućem smjeru u histogramu. Doprinos se računa na temelju Gaussove funkcije sa σ koja je 1.5 puta veća od mjerila promatrane značajke [9].

Nakon što je završeno računanje histograma, slijedi traženje smjera s najvećom vrijednosti u histogramu, tzv. vrh histograma. Vrh s pridruženim smjerom odgovara dominantnoj orijentaciji u histogramu. No, sam vrh nije dovoljan da bi se odredila orijentacija promatrane značajke. Zato se u izračun orijentacije uzimaju i svi lokalni vrhovi, odnosno smjerovi čije su vrijednosti unutar 80% najveće vrijednosti (vidi sl. 9). Ukoliko postoji više takvih vrhova podjednake amplitude, promatranoj značajki pridjeljuje se više orijentacija. Stvaraju se nove značajke alternativnog smjera sa istom lokacijom i mjerilom. Prema istraživanju, to se događa samo u 15% slučajeva i značajno pridonosi stabilnosti podudaranja [9, 21].



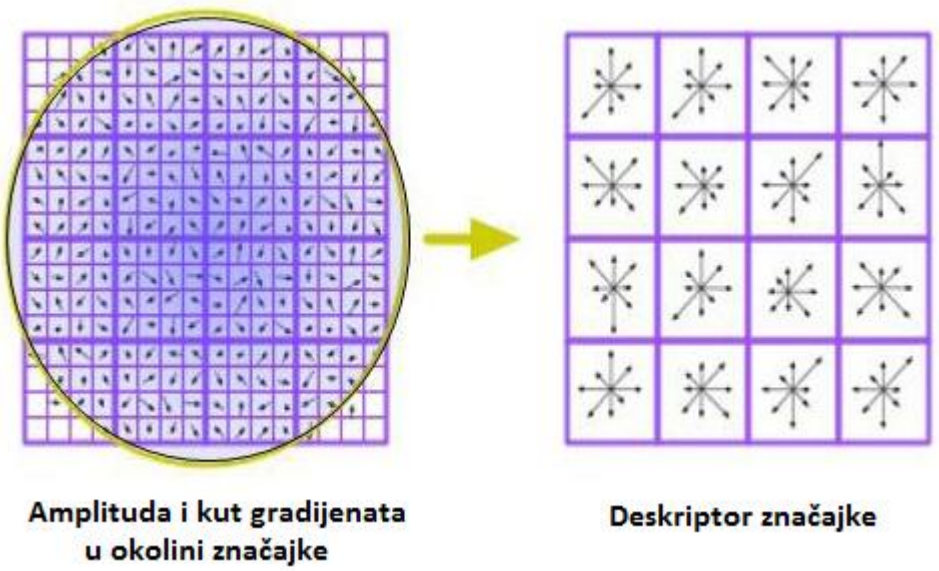
Slika 9. Orijentacijski histogram [19]

Na kraju se konstruira parabola kroz tri točke histograma. Za središnju točku uzima se vrh histograma, dok ostale dvije odgovaraju najbližim susjedima lijevo i desno od vrha. Na taj način vrši se kvadratna interpolacija, te se dobiva finija raspodjela orijentacije.

4.4. Izgradnja deskriptora

Prijašnji koraci pronalaze i pridjeljuju značajki lokaciju, mjerilo i orijentaciju. Postiže se invarijantnost obzirom na mjerilo i rotaciju. U ovom koraku svakoj značajki dodjeljuje se opisni vektor i time se postiže invarijantnost na razlike u boji, osvjetljenju i točki gledišta. Kratak opis ovog koraka dan je u nastavku, a temelji se na osnovi modela kojeg su 1997. godine Edelman, Intrator i Poggio predstavili kao jedan od boljih pristupa [9]. Zasniva se na pokušaju oponašanja ljudskog vida i daje znatno bolje rezultate pri promjeni kuta gledanja u odnosu na alternativne metode.

Svakoj značajki dodjeljuje se opisni vektor, odnosno deskriptor koji karakterizira okolinu te značajke. Za kreiranje deskriptora potrebno je izračunati amplitudu i kut gradijenta za svaki element unutar okoline promatrane značajke. Okolina, odnosno regija kao u prethodnom koraku i dalje ovisi o mjerilu. Ilustracija gradijenata upotrebom strelica može se vidjeti na slici 10. lijevo.



Slika 10. Kreiranje deskriptora značajke [1]

Amplituda svakog elementa okoline množi se sa težinskim faktorom Gaussove funkcije, gdje je σ jednaka polovini širine prozora deskriptora. Cilj je izbjeći iznenadne promjene u deskriptoru koje su uzrokovane manjim promjenama u poziciji prozora te dati manji značaj gradijentima koji su udaljeniji od središta deskriptora. Regija značajke dijeli se u podregije dimenzija 4x4 koje predstavljaju skup orijentacijskih histograma. Svaki histogram iz spomenutog skupa sastoji se od osam potencijalnih smjerova. Dužina strelice pridružene pojedinom smjeru odgovara ukupnom zbroju amplituda u blizini tog smjera unutar promatrane podregije (vidi sl. 10 desno). Deskriptor pojedine značajke sadrži 4x4x8 = 128 podataka za tu značajku. Dobiveni opisni vektor, odnosno deskriptor potrebno je normalizirati. Normalizacija podrazumijeva dovođenje vektora na jediničnu duljinu i time umanjuje utjecaj promjena u kontrastu. Linearna promjena kontrasta podrazumijeva množenje vrijednosti slikovnih elemenata određenom konstantom. Iz čega slijedi da je ukupan gradijent množen istom konstantnom, a ona se poništava prilikom normalizacije [9]. Također, ni promjene u svjetlini slike neće utjecati na vrijednost gradijenata, a time ni na rezultat algoritma SIFT. Razlog je tome računanje gradijenata kao razlike susjednih elemenata, te dolazi do poništavanja konstante dodane slikovnim elementima [9]. S druge strane, moguć je utjecaj u slučajevima gdje svjetlost udara o površine koje drugačije reagiraju na nju. Ovdje se misli na transmisije, refleksije i apsorpcije, te isto tako na općenit slučaj trodimenzionalnosti gdje svjetlost jednostavno udara na različite dijelove objekta drugačijim intenzitetom. Ovakve varijacije uglavnom utječu na amplitudu gradijenta, a mnogo rjeđe na samu orijentaciju, tj. kut gradijenta. U ovom koraku velike vrijednosti amplitude gradijenta ograničavaju na određenu (eksperimentalno utvrđenu) vrijednost i ponovo renormaliziraju na jedinične vrijednosti [9, 11]. To znači da veličina samih gradijenata nije toliko važna, te da je veći naglasak na distribuciji orijentacije. Kao konačan rezultat ovog koraka dobiven je deskriptor, čijim se uspoređivanjem pronalaze korespondentne točke među slikama [9, 11].

4.5. Primjene algoritma SIFT

Algoritam SIFT se zbog svoje robusnosti koristi na različitim područjima računalnog vida. Neka područja su:

- **Stvaranje panoramskih snimaka**

  Značajke pronađene algoritmom SIFT omogućuju automatizaciju stvaranja panoramskih snimaka.

- **Lokalizacija robota**

  Robot pomoću dvije kamere izgrađuje 3D model nepoznate okoline u kojoj se nalazi i omogućava proračun svoje trenutačne pozicije.

- **Proširena stvarnost** (engl. *augmented reality*)

  Generiranje imaginarnih objekata zabavnog ili informativnog sadržaja unutar živih scena.

- **Interakcija čovjeka i računala**

  Odnosi se na prepoznavanje određenih ljudskih akcija od strane računala.

- **Filmska industrija**

  Većinom se odnosi na scene u koje se dodaju efekti računalne grafike.

- **Medicina**

  Najznačajnija upotreba je prilikom analize ljudskog mozga pomoću MRI slika.

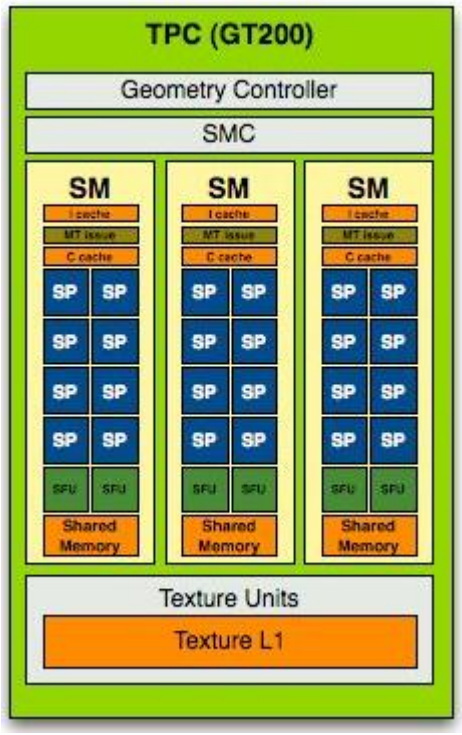
5. Programsko okruženje CUDA

Prije nekoliko godina tvrtka NVIDIA predstavlja novu arhitekturu za paralelno programiranje pod imenom CUDA (engl. *Compute Unified Device Architecture*). Radi se o općenamijenskoj paralelnoj arhitekturi s novim paralelnim programskim modelom i skupom instrukcija, koja daleko pojednostavljuje paralelno programiranje na grafičkim procesorima (engl. *Graphics Proccesing Unit*, GPU). Time je omogućeno učinkovitije rješavanje mnogih složenijih računalnih problema na grafičkim procesorima nego što to omogućavaju procesori opće namjene (engl. *Central Processing Unit*, CPU). Danas CUDU koriste znanstvenici i istraživači na raznim područjima, kao što je: obrada slike i videa, istraživanja u biologiji i kemiji, dinamičkim simulacijama fluida, CT rekonstrukcije slike, seizmičke analize i mnoge druge [24].

5.1. Organizacija sklopovlja

Budući da su grafički procesori dizajnirani za drugačije tipove aplikacija u odnosu na procesore opće namjene, njihova se arhitektura razvijala u drugačijem smjeru. Grafički procesori sastoje se od nekoliko razina koje su organizirane hijerarhijski. Na prvoj razini nalaze se protočni multiprocesori (engl. *streaming multiprocessors*), odnosno računske jedinice. Svaki multiprocesor sastoji se od nekoliko protočnih procesora (engl. *stream processors*), koristi se još i naziv *jezgra*. Multiprocesori su organizirani prema arhitekturi SIMD (engl. *Single Instruction Multiple Data*). Temeljem toga slijedi bitna razlika između procesora opće namjene i grafičkih procesora. Jezgre procesora opće namjene od samog početka dizajnirane su da slijedno izvode niz instrukcija maksimalnom brzinom. Za razliku od toga, grafički procesor omogućava paralelnu obradu velike količine podataka. Jedna instrukcija obavlja se nad više podatkovnih elemenata, tj. nakon svakog procesorskog takta, svaki procesor multiprocesora obavlja istu instrukciju, ali nad različitim podacima. Valja primijetiti kako različiti multiprocesori mogu paralelno izvoditi različite naredbe u istom periodu glavnog takta grafičkog procesora, dakle oni su međusobno nezavisni. Procesori unutar određenog multiprocesora paralelno izvode istu naredbu.

Potrebno je napomenuti da arhitektura samih grafičkih procesora ovisi o njihovoj seriji. Na slici 11. prikazana je arhitektura grafičkog procesora novije generacije, GT200 [5,6].



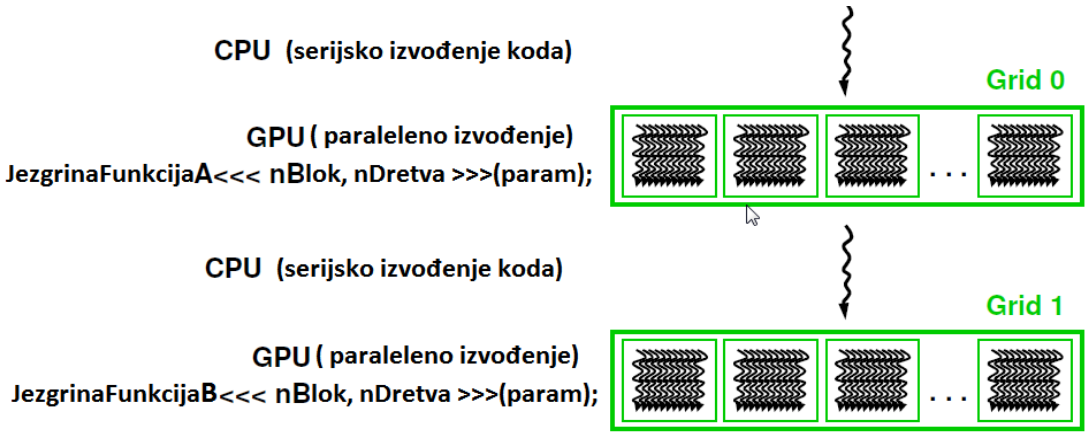
Slika 11. Arhitektura grafičkog procesora GT200 tvrtke NVIDIA

Ovaj grafički procesor sastoji se od trideset protočnih multiprocesora, a unutar svakog multiprocesora nalazi se osam protočnih procesora. Tri takva multiprocesora mogu se vidjeti na slici. Protočni procesori imaju mogućnost izvođenja operacija nad cjelobrojnim podacima jednostruke (32 bita) preciznosti (engl. *single-precision integer*) i operacija nad podacima s pomičnim zarezom jednostruke (32 bita) preciznosti (engl. *single-precision foating point*). Pored osam protočnih procesora, unutar svakog multiprocesora, nalaze se: zajednička pričuvna memorija za podatke i instrukcije, logička kontrolna jedinica, 16kB zajedničke memorije, dvije jedinice za specijalne funkcije (engl. *special function units,* SFU) i instrukcijska jedinica (engl. *multithreaded instruction unit*). Specijalne jedinice, SFU koriste se za obavljanje posebnih funkcija, kao što su operacije dvostruke preciznosti (64 bita) pomičnog zareza (engl. *double-precision foating-point*) i transcendentalne operacije, tj. računanje sinusa, kosinusa i slično. Instrukcijska jedinica ima posebnu namjenu, njezin

zadatak je raspoređivanje instrukcija svim protočnim procesorima i specijalnim jedinicama za složene funkcije [17].

5.2. Programski model

Program koji se izvodi u okruženju CUDA sastoji se od dijelova koji se odvojeno izvode na procesoru opće namjene i grafičkom procesoru. Dio programa s vrlo malo podatkovnog paralelizma ili onaj dio gdje ga uopće nema, implementira se standardnim jezikom C. Prevodi se i izvodi se na procesoru opće namjene kao i svaki drugi običan program. S druge strane, dio koda koji posjeduje veliku količinu podatkovnog paralelizma implementira se jezikom C i njegovim CUDA proširenjima za označavanje paralelnih funkcija i pridružene podatkovne strukture. Funkcija za takvu paralelnu obradu podataka na grafičkom procesoru u daljem tekstu naziva se funkcija jezgre [5]. Izvođenje programa započinje na procesoru opće namjene, a pozivom funkcija jezgre izvođenje se nastavlja na grafičkom procesoru (vidi sl. 12.).



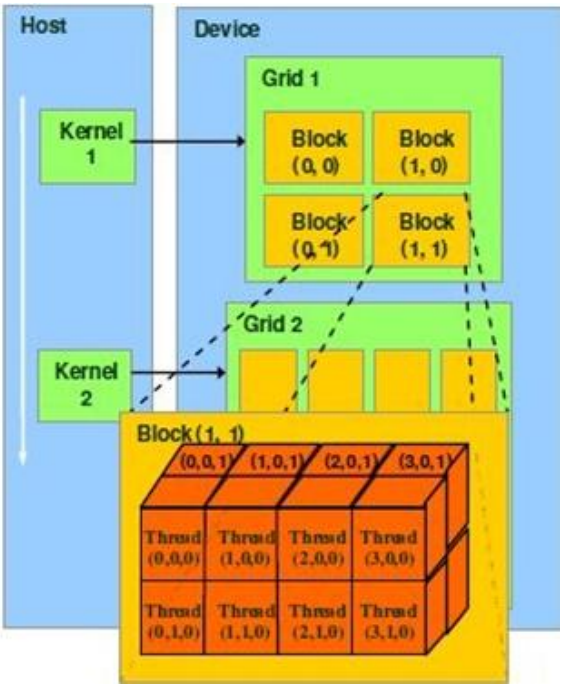
Slika 12. Izvođenje programa na arhitekturi CUDA

Kako bi se iskoristila velika količina podatkovnog paralelizma, prilikom poziva funkcije jezgre generira se ogroman broj dretvi. Funkcija jezgre predstavlja dio koda koji će paralelno izvoditi sve dretve. Generirane dretve sačinjavaju mrežu dretvi (engl. *grid*). Više o njihovoj organizaciji može se pročitati u odjeljku 5.3. Kad sve dretve generirane prilikom poziva funkcije jezgre završe, izvođenje se nastavlja na procesoru opće namjene. Programer je dužan prethodno zauzeti potrebnu količinu memorije GPU da bi se pozivom

funkcije jezgre omogućilo izvođenja na grafičkom procesoru. Nakon toga, u zauzetu memoriju GPU prebacuju se svi podaci iz memorije *hosta* nužni za uspješno izvođenje funkcija jezgre. Ovdje se *hostom* smatra cjelina unutar koje se nalazi CPU. Nakon završetka izvođenja, potrebno je rezultat prebaciti natrag u memoriju *hosta*, te osloboditi prethodno zauzetu memoriju GPU. Mnogo detaljnije o memoriji može se pročitati u odjeljku 5.4.

5.3. Organizacija dretvi

Kao što je već ranije spomenuto, pozivom funkcije jezgre generira se mreža dretvi. Sve dretve unutar mreže izvodit će isti odsječak koda. Dretve dohvaćaju različite podatke iz globalne memorije, nad kojima paralelno izvode istu instrukciju. Dretve unutar mreže podijeljene su u blokove. Blokovi mogu biti jednodimenzionalni, dvodimenzionalni ili trodimenzionalni. Broj blokova u svakoj dimenziji i broj dretvi unutar jednog bloka određeni su ugrađenim varijablama koje je programer dužan navesti kod poziva funkcija jezgre. Njihova glavna namjena je omogućiti da svaki blok i svaka dretva unutar tog bloka posjeduje jedinstvene koordinate pomoću kojih se pojedina dretva razlikuje od svih ostalih. Osim njihovog međusobnog raspoznavanja, varijable se koriste i za određivanje pozicije podatka koji će pojedina dretva obrađivati. Na slici 13. prikazana je organizacija dretvi unutar četiri bloka. Kako svi blokovi sadrže jednak broj dretvi organiziranih na isti način, dovoljno je prikazati dretve jednog bloka. Prikazani blok organiziran je kao trodimenzionalno 4×2×2 polje dretvi. U jednom bloku nalazi se 16 dretvi, što daje ukupno 64 dretve unutar mreže. U stvarnosti je taj ukupan broj dretvi daleko veći.



Slika 13. Organizacija dretvi [6]

Potrebno je osigurati da sve dretve unutar bloka završe s izvođenjem jedne faze prije nego što nastave s izvođenjem iduće faze. To se osigurava posebnom funkcijom za sinkronizaciju. Po pozivu te funkcije dretva zaustavlja svoje izvođenje i čeka da preostale dretve unutar bloka dođu do tog mjesta. Sinkronizacija dretvi između različitih blokova unutar mreže nije moguća. Mehanizam izvođenja instrukcija grupira dretve jednog bloka u male skupine, tzv. osnove (engl. wraps). Broj dretvi unutar jedne osnove kod današnjih grafičkih procesora jednak je 32. Osnove se slijedno dodjeljuju multiprocesorima, gdje se sve dretve te osnove istovremeno izvode na protočnim procesorima dodijeljenog multiprocesora. Prilikom raspodjele dretvi protočnim procesorima, protočni procesori će izvesti najmanje jednu instrukciju, nakon čega se mogu dodijeliti drugoj osnovi. Sve dretve iste osnove istovremeno izvode istu operaciju. Ukoliko dretve jedne osnove pristupaju podacima globalne memorije, za što je potrebno 400-600 ciklusa da bi adresirani podatak bio raspoloživ, za to se vrijeme izvodi se druga osnova. Nakon što resursi postanu raspoloživi, nastavit će se sa izvođenjem prve osnove. Time će se izbjeći memorijska latencija, a za to se brine mehanizam zasnovan na prioritetima. Maksimalna učinkovitost postiže se i paralelnim izvođenjem blokova. Ako grafički procesor ne posjeduje dovoljno resursa (multiprocesora), blokovi se mogu izvoditi jedan po jedan, izvođenje se ne mora odvijati istom brzinom. Svojstvo izvođenja iste aplikacije različitim brzinama naziva se

transparentna skalabilnost i omogućava programerima lakše programiranje, te povećava iskoristivost aplikacija [17].

5.4. Organizacija memorije

Grafički procesori sadrže nekoliko tipova memorije koje su hijerarhijski organizirane. Neke od njih su već ranije spomenute, a ovdje je dan njihov detaljniji opis.

Globalna memorija naziva se još i VRAM (engl. *Video Random Access Memory*), te predstavlja najveću količinu memorije koja je dostupna svim protočnim multiprocesorima. Količina globalne memorije varira od nekoliko stotina megabajta do reda veličine gigabajta, koliko nalazimo na najjačim grafičkim karticama (GTX295). Iznimka su integrirane grafičke kartice koje dijele memoriju sa središnjim procesorom. Pruža veliku propusnost (do 100GB/s), ali operacije pristupanja toj memoriji posjeduju veliku latenciju (nekoliko stotina ciklusa). Kako je vrijeme pristupanja ovoj memoriji vrlo sporo, protočni procesori dohvaćaju podatke u vlastite registre (privatna memorija) ili u dijeljenu memoriju i izvršavaju zadane operacije nad podacima. Po završetku obrade podataka, rezultati se ponovo upisuju u globalnu memoriju. Potrebno je spomenuti da je kod grafičkih kartica s podrškom većom ili jednakom *Compute Capability* 1.2 globalna memorija podijeljena u segmente od 32, 64 i 128 bajta koji su poravnati na adresama istih višekratnika. Prilikom dohvaćanja podataka u globalnu memoriju prvi podatak se sprema na adresu koja je višekratnik željenog segmenta. Programer treba voditi računa o tome jer broj memorijskih transakcija ovisi o broju pristupanja dretvi iste osnove različitim segmentima. Točnije, koliko dretvi iste osnove pristupa različitim segmentima toliko će biti memorijskih transakcija, bez obzira što se radi o slijednim adresama [7,17].

Dijeljena memorija, koristi se još i naziv zajednička memorija. Radi se o izuzetno brzoj memoriji malog kapaciteta. Kod većine današnjih grafičkih procesora ona iznosi oko 16kB, ali uglavnom to ovisi o seriji GPU. Nalazi se na razini multiprocesora, a svaki multiprocesor posjeduje svoju dijeljenu memoriju kojoj mogu pristupati svi procesori tog multiprocesora. Pristupanje dijeljenoj memoriji drugog multiprocesora nije moguće. Protočni procesori mogu čitati i pisati podatke kao i kod globalne memorije. Pristupanje

ovoj memoriji je oko 150 puta brže u odnosu na globalnu memoriju. Zbog toga ona ima ulogu priručne memorije (engl. *cache*) procesora opće namjene. U nju se dohvaćaju svi podaci koji se često koriste i tako se ubrzava izvođenje programa. Razlika u odnosu na procesore opće namjene je da se programer brine o dohvaćanju podataka u dijeljenu memoriju, dok je to kod procesora opće namjene riješeno na sklopovskoj razini. To zahtjeva od programera veliko iskustvo i dobro poznavanje organizacije memorije jer u suprotnom neće doći do iskorištavanja brzine koju pruža dijeljena memorija.

Lokalna memorija je mala količina memorije na razini protočnog procesora. Svaki protočan procesor sadrži vlastitu lokalnu memoriju kojoj može pristupati. Pristupanje lokalnoj memoriji drugog protočnog procesora, bez obzira da li je izvan ili unutar multiprocesora, nije moguće. Kao i kod globalne memorije, pristupanje je ovoj memoriji izrazito sporo.

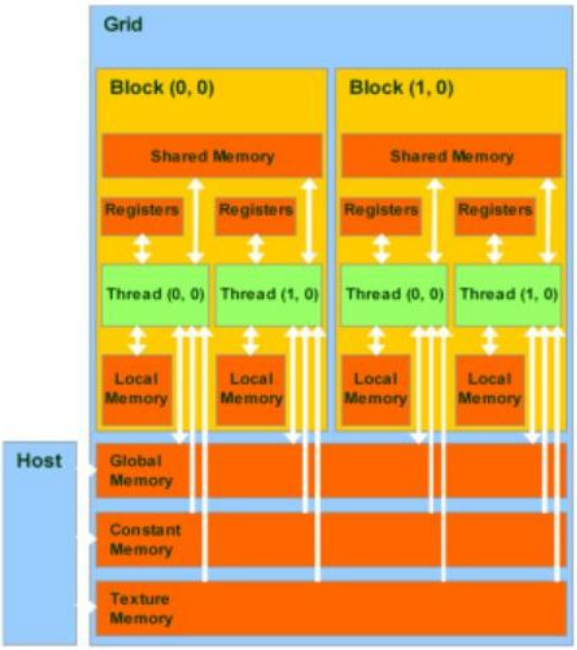
Privatna memorija naziv je za 32-bitne registre pridružene svakom protočnom procesoru. Jednako kao i kod lokalne memorije svaki protočan procesor može pristupati jedino vlastitom registru. Razlika je u brzini, pristupanje ovoj memoriji daleko je brže od pristupanja lokalnoj memoriji. Ovaj tip memorije najčešće se koristi za pohranu privatnih varijabli.

Memorija konstante je mala količina memorije veličine oko stotinu kilobajta. Podatke iz ove memorije mogu čitati svi multiprocesori, ali nemaju mogućnost ponovnog pisanja. Njezina uloga je slična priručnoj memoriji procesora opće namjene, a koristi se za pohranjivanje konstantnih vrijednosti. Pristupanje ovoj memoriji brže je od pristupanja globalnoj memoriji, no još je uvijek sporije od pristupanja dijeljenoj memoriji.

Memorija teksture definirana je na razini multiprocesora s ograničenim pristupom. To znači da svi multiprocesori mogu samo čitati podatke iz memorije teksture, dok njihovo mijenjanje ili upisivanje novih nije moguće. Memorija teksture se i dalje vrlo često koristi jer omogućava niz prednosti u odnosu na globalnu memoriju. Radi se o priručnoj memoriji iz koje se podaci daleko brže čitaju za razliku od globalne memorije. Nadalje, memorija teksture optimizirana je za 2D ili 3D prostorni lokalitet. Najbolje performanse postići će dretve iste osnove koje pristupaju adresama memorije teksture u neposrednoj blizini [7,24]. Također, 8-bitni ili 16-bitni cjelobrojni podaci mogu se opcionalno konvertirati u

32-bitne podatke s pomičnim zarezom u rasponu [0.0 , 1.0] ili [-1.0 , 1.0]. Razlog tome je što u mnogim slučajevima grafički procesor brže izvodi operacije u aritmetici sa pomičnim zarezom nego operacije s cijelim brojevima.

Svi prethodno opisani tipovi memorije GPU prikazani su na slici 14., kao i mogućnosti pristupanja od strane protočnog procesora i/ili multiprocesora [7].



Slika 14. Prikaz memorijskog sklopovlja GPU-a

Potrebno je napomenuti da lokalna, globalna, memorija za konstante i memorija teksture fizički čine jednu memoriju (VRAM), ali s različitim mogućnostima pristupa.

6. Programsko rješenje algoritma SIFT za CPU

Praktičan dio ovog rada jest implementacija algoritma za pronalaženje značajki koje se koriste za uspostavljanje korespondencije u slikama. Kao što je opisano u odjeljku 4.3, značajke su rezultat trećeg koraka algoritma SIFT. Upravo zbog toga praktičan dio obuhvaća samo implementaciju prva tri koraka, bez određivanja deskriptora. Izgradnja programskog rješenja vođena je postojećom implementacijom A. Vedaldija. Algoritam je u potpunosti implementiran u programskom jeziku C++ unutar programskog okruženja Visual Studio 2008 tvrtke Microsoft.

6.1. Implementacija Gaussove i DoG piramide

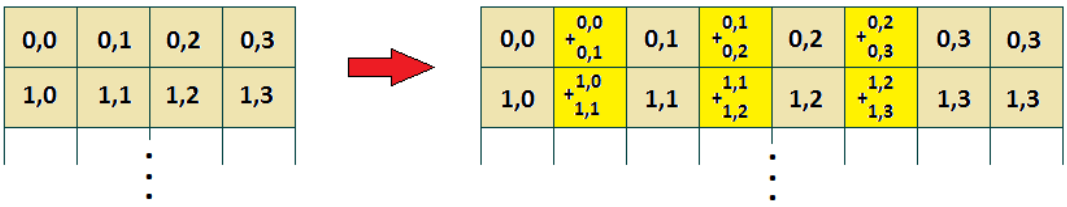
Neovisno o broju oktava, samo slike prve oktave biti će dvostruko veće od originalne slike predane programu prilikom pokretanja. Povećavanjem rezolucije slike za faktor 2 povećava se broj stabilnih značajki skoro četiri puta [9]. Svaka oktava prostora mjerila dijeli se na određeni broj intervala, definiran sa *levels*. U pravilu broj mjerila za svaku oktavu jednak je *levels+3*. Razlog tome je slučaj kad se *levels* postavi na 1, tada se generiraju točno četiri mjerila po oktavi što je dovoljno da se ispuni uvjet ispitivanja *26-susjedstva* implementiranog u idućem koraku.

Izgradnja prostora mjerila implementirana je unutar metode `buildPyramid()` razreda `MySift`. Na samom početku, originalna slika se zaglađuje Gaussovim filtrom sa standardnom devijacijom *sigmaN* iznosa 0.5. Na taj način sprječava se *aliasing*. Zaglađena slika predstavlja ulaznu sliku na temelju koje će se graditi prostor mjerila. U izvornoj implementaciji A. Vedaldija ne zaglađuju se originalne slike, već se pretpostavlja da je na ulaznu sliku prethodno primijenjen *antialias* filtar [20].

Zaglađivanje, odnosno konvolucija slike Gaussovim filtrom implementirana je unutar statičke metode `gaussSmooth` razreda `DoG` i pozivat će se prilikom svake slijedeće konvolucije. Polumjer Gaussovog filtra određuje se na temelju standardne devijacije (u nastavku rada koristit će se naziv *sigmaG*) predane ovoj metodi. Filtar se dalje puni

vrijednostima koje se računaju prema Gaussovoj funkciji, te se normalizira. Nakon što je određen filtar, izvodi se konvolucija. Prvo se izvodi konvolucija po recima, a zatim po stupcima gdje se prethodan rezultat koristi kao ulaz. Bilo da se radi jednodimenzionalna konvolucija po recima ili stupcima potrebno je posebnu pažnju posvetiti slučajevima kad se za centrali slikovni element odabiru elementi ruba. Tu se misli na slikovne elemente u dužini polumjera filtra sa svake strane slike. Elemente ruba nije moguće zanemariti prilikom izračunavanja konvolucije jer bi tada postojao nagli prijelaz na tim dijelovima slike i postoji vjerojatnost da bi se u idućem koraku algoritma SIFT detektirali ekstremi. Drugi način bi bio postaviti te elemente na nulu, ali to nije učinkovito u slučajevima sa velikom polumjerom filtra. Zbog toga prilikom računanja konvolucije i rubni elementi uzimaju se kao centralni. Kod onih elemenata za koje ne postoje susjedni elementi, umjesto susjeda uzima se prva vrijednost uz rub, koja se dalje množi sa odgovarajućom vrijednošću filtra [20].

Nakon konvolucije slijedi povećavanje rezolucije ulazne slike za faktor 2. Udvostručavanje rezolucije slike je implementirano metodom `upSample` razreda `Image`, a zasniva se na linearnoj interpolaciji. Rezolucija se povećava u dva prolaza. Kao što je prikazano na slici 15., u prvom prolazu udvostručuje se broj slikovnih elemenata retka. Na poziciju svakog drugog elementa dodaje se novi slikovni element čija je vrijednost jednaka zbroju susjeda. Isti postupak provodi se nad stupcima slike, a ukupan broj slikovnih elementa je povećan četiri puta.



Slika 15. Povećanje širine slike

Povećana slika koristit će se za izgradnju prve oktave. Općenito, za svako mjerilo pojedine oktave slika se konvoluira drugačijem Gaussovim filtrom. Kao što je ranije spomenuto, filtar se generira na temelju *sigmaG*. Potrebno je odrediti $sigmaG_{o,s}$ za svako mjerilo pojedine oktave (indeks *o* označuje oktavu, a *s* mjerilo). U nastavku su prikazani parametri i izrazi koji se pritom koriste [20].

$$sigma0 = 1.6\,, \quad sigmaK = 2^{\frac{1}{LEVELS}} \tag{1.23}$$

$$sigmaG_{o,s} = \sqrt{{sigma_{o,s}}^2 - {sigma_{o,s-1}}^2} \tag{1.24}$$

Sigma0 predstavlja baznu standardnu devijaciju, a *sigmaK* je faktor za koji će se razlikovati iznos *sigmaG* između susjednih mjerila iste oktave. Vrijednost S*igmaG*$_{o,s,}$ koja se koristi za zaglađivanje pojedine razine mjerila računa se po formuli 1.24. Gdje *sigma*$_{o,s}$ predstavlja iznos za konvoluiranje trenutne slike, a *sigma*$_{o,s-1}$ jednaka je iznosu kojim je konvoluirana slika prethodnog mjerila. Prema izrazu 1.24 vrijednost *sigmaG*$_{0,0}$ za prvu oktavu prve razine mjerila iznosi (indeks 0,0 jer brojanje oktava i mjerila kreće od nula):

$$sigmaG_{0,0} = \sqrt{sigma0^2 - 1} = \sqrt{1.6^2 - 1} \tag{1.25}$$

Ovdje je *sigma*$_{0,0}$ jednaka osnovnoj vrijednosti zaglađivanja s*igma0*, dok je *sigma*$_{0,-1}$ jednak jedinici. Razlog tome je što se na samom početku originalna slika konvoluirana sa *sigmaN* iznosa 0.5. Udvostručavanjem rezolucije slike udvostručuje se i *sigmaN*, te sada iznosi 1.0. Nakon što je određena slika prve razine mjerila za prvu oktavu, kreće se sa određivanjem slike druge razine, po formuli 1.24 dobiva se:

$$\begin{aligned} sigmaG_{0,1} &= \sqrt{(sigma_{0,0} * sigmaK)^2 - {sigma_{0,0}}^2} \\ &= \sqrt{(sigma0 * sigmaK)^2 - sigma0^2} \\ &= sigma0\sqrt{sigmaK^2 - 1} \end{aligned} \tag{1.26}$$

Iz izraza 1.26 vidi se da je vrijednost *sigma*$_{0,1}$ jednaka umnošku faktora *sigmaK* i vrijednosti kojim je konvoluirana slika prve razine mjerila, *sigma*$_{0,0}$. Za svaku slijedeću razinu *s,* vrijednost *sigma*$_{o,s}$ biti će *sigmaK* puta veća od vrijednosti prethodne razine *sigma*$_{o,s-1}$. Iznos *sigmaG*$_{0,2}$ za generiranje Gaussovog filtra treće razine mjerila je:

$$\begin{aligned} sigmaG_{0,2} &= \sqrt{(sigma_{0,1} * sigmaK)^2 - {sigma_{0,1}}^2} \\ &= \sqrt{sigma0^2 * sigmaK^4 - sigma0^2 * sigmaK^2} \\ &= \left(sigma0\sqrt{sigmaK^2 - 1}\right) * sigmaK \end{aligned} \tag{1.27}$$

Temeljem izraza 1.26 i 1.27 vidi se da je produkt $sigma0\sqrt{sigmaK^2 - 1}$ prisutan u svim izračunima. Zbog toga se izvlači van, te se računanje *sigmaG* svodi na množenje tog produkta sa faktorom *sigmaK* na potenciju koja odgovara trenutnom mjerilu (vidi 1.28 i 1.29).

$$dsigma0 = sigma0\sqrt{sigmaK^2 - 1} \tag{1.28}$$

$$sigmaG_{o,s} = dsigma0 * sigmaK^{s-1}, \text{ gdje } s = 1, \ldots., LEVELS + 2 \tag{1.29}$$

Poslije određivanja svih razina mjerila prve oktave, kreće izgradnja iduće oktave u nizu. Osnovna slika koja se koristi za izgradnju iduće oktave u nizu, nalazi se na razini koja odgovara broju intervala definiranom sa *levels* iz prethodne oktave. Dohvaćanje željene slike implementirano je metodom `loadData` razreda `Image`. Nakon što je dohvaćena slika, potrebno je smanjiti njezinu rezoluciju. Smanjivanje rezolucije implementirano je metodom `downSample`. Radi se o vrlo jednostavnoj implementaciji u kojoj se prolazi po svim slikovnim elementima, pri čemu se izbacuje svaki drugi element unutar retka i stupca. Smanjena slika predstavlja osnovnu sliku koja će se dalje zaglađivati Gaussovim filtrom za svako mjerilo. Postupak izgradnje idućih oktava sa svim pridruženim mjerilima identičan je prethodno opisanom postupku za prvu oktavu. Na slici 16. prikazan je primjer Gaussove piramide koja se sastoji od četiri oktave po pet razina mjerila.

Slika 16. Primjer Gaussove piramide

Nakon što je izgrađena Gaussova piramida, kreće se sa generiranjem *DoG* piramide. *DoG* piramida gradi se na način da se prolazi po svim oktavama i oduzimaju se vrijednosti slikovnih elementa na istim pozicijama mjerila ispod od mjerila iznad. Tako se od druge razine mjerila Gaussove piramide oduzima prva, od treće druga itd. Opisano oduzimanje slikovnih elemenata susjednih mjerila implementirano je u statičkoj metodi `differenceGaussian` razreda `DoG`.

U izvornom kodu niže, prikazan je opisani postupak izgradnje prostora mjerila. Gdje su `gaussList` i `DoGList` dvodimenzionalna polja slika, a dimenzije pojedine slike postavljaju se pozivom metode `setDimension`.

```
void MySift::buildPyramid(){
   float sigma1, sigmaG;
   float dsigma0=float(this->sigma0*sqrt((this->sigmaK*this->sigmaK)-1.0f));

   gaussList[0][0]=*(this->original);
```

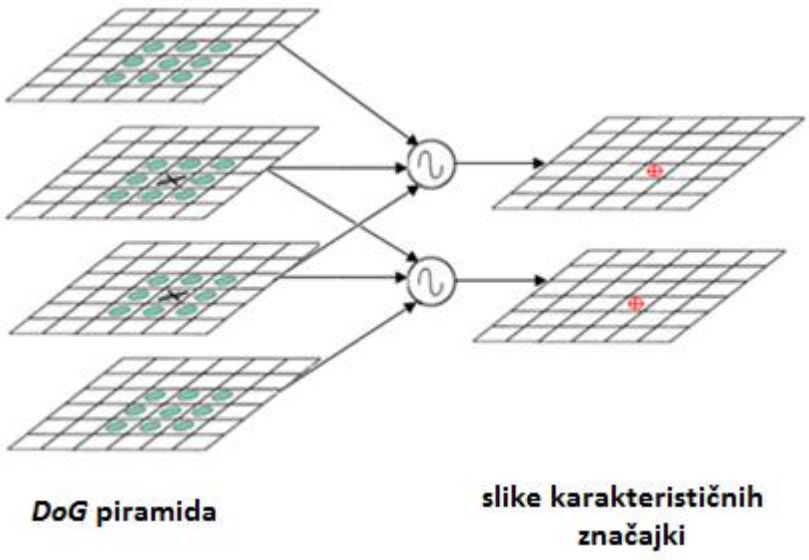
```
   gaussList[0][0].upSample(1);
   gaussList[0][0].upSample(2);

   sigma1 = float(this->sigmaN / pow(2.0, OMIN)) ;
   if( this->sigma0 > sigma1 ) {

      sigmaG = sqrt ( this->sigma0*this->sigma0 - sigma1*sigma1) ;

      DoG::gaussSmooth(gaussList[0][0],gaussList[0][0],
                               sigmaG,this->filter, this->leg);
    }

   for(int o = 0 ; o < OCTAVE; ++o) {
      if( o > 0 ) {
         gaussList[o][0].loadData(gaussList[o-1][LEVELS].dataImage,
                 gaussList[o-1][LEVELS].height, gaussList[o-1][LEVELS].width);
         gaussList[o][0].downSample();
       }

       for(int s = 1; s < LEVELS+3 ; ++s) {
         sigmaG = dsigma0 * pow(this->sigmaK, s-1) ;

         gaussList[o][s].setDimension(gaussList[o][s-1].height,
                         gaussList[o][s-1].width);
         DoG::gaussSmooth(gaussList[o][s-1], gaussList[o][s],sigmaG,
                         this->filter, this->leg);

        DoGList[o][s-1].setDimension(gaussList[o][s-1].height,
                          gaussList[o][s-1].width);
         DoG::differenceGaussian(gaussList[o][s],
                         gaussList[o][s-1], DoGList[o][s-1]);

       }
  }

}
```

6.2. Implementacija detekcije i lokalizacije značajki

Implementacija drugog koraka algoritma SIFT obuhvaća pronalaženje kandidata značajki, određivanje njihove lokacije s obzirom na x, y koordinate i mjerilo, te

odbacivanje nestabilnih kandidata. Sve prethodno nabrojano implementirano je metodom `detectLocalExtrema` razreda `MySift`.

Na samom početku implementacije ovog koraka pronalaze se minimumi/maksimumi na slikama *DoG* generiranim u prethodnom koraku. Ekstremi se pronalaze uspoređivanjem slikovnih elemenata triju mjerila *DoG* piramide. Kao što je prikazano na slici 17. lijevo, za svaki element provjerava se da li je veći ili manji od svojih najbližih susjeda. Osim uspoređivanja unutar slike istog mjerila, vrijednost odabranog slikovnog elementa uspoređuje sa susjedima na istim pozicijama mjerila iznad i mjerila ispod. Tim postupkom generiraju se nove slike karakterističnih značajki, čiji je broj manji za dva od broja mjerila unutar oktave *DoG* piramide. Na primjer, ukoliko se Gaussova piramida sastoji od pet mjerila po oktavi, dobiva se *DoG* piramida koja se sastoji od četiri mjerila po oktavi, a na temelju *DoG* piramide generiraju se dvije slike karakterističnih značajki za svaku oktavu. Generirane slike značajki nalaze se na pozicijama unutrašnjih razina mjerila *DoG* piramide, a u implementaciji spremaju se u intervalu [0, *levels*]. Sadrže skup detektiranih značajki označenih na odgovarajućim pozicijama u slici. (vidi sl. 17. desno).



Slika 17. Detekcija ekstrema

Međutim da bi se ekstremi proglasili značajkama i dodali u slike karakterističnih značajki potrebno ih je precizno locirati i ispitati njihovu stabilnost. Prvo se određuje lokacija detektiranog ekstrema prema izrazima 1.12 i 1.13. Kako je slika diskretna,

gradijent se aproksimira razlikom vrijednosti susjednih slikovnih elemenata u slici *DoG* (u nastavku za *DoG* koristi se oznaka *D*). U izvornom kodu za izračun gradijenta implementirane su iduće jednadžbe:

$$D_x = \frac{D(x+1,y,s)-D(x-1,y,s)}{2} \tag{1.30}$$

$$D_y = \frac{D(x,y+1,s)-D(x,y-1,s)}{2} \tag{1.31}$$

$$D_s = \frac{D(x,y,s+1)-D(x,y,s-1)}{2} \tag{1.32}$$

Gdje su D_x, D_y i D_s početne vrijednosti odmaka u sve tri dimenzije (x, y i mjerilo) od promatranog kandidata značajke. Potrebno je izračunati vrijednosti ostalih gradijenata da bi se odredila konačna vrijednost pomaka, koje će se koristiti i prilikom određivanja nestabilnog kandidata značajke. Implementirane su sljedeće jednadžbe:

$$D_{xx} = D(x+1,y,s) + D(x-1,y,s) - 2 * D(x,y,s) \tag{1.33}$$

$$D_{yy} = D(x,y+1,s) + D(x,y-1,s) - 2 * D(x,y,s) \tag{1.34}$$

$$D_{ss} = D(x,y,s+1) + D(x,y,s-1) - 2 * D(x,y,s) \tag{1.35}$$

$$D_{xy} = \frac{D(x+1,y+1,s)+D(x-1,y-1,s)-D(x+1,y-1,s)-D(x-1,y+1,s)}{4} \tag{1.36}$$

$$D_{xs} = \frac{D(x+1,y,s+1)+D(x-1,y,s-1)-D(x+1,y,s-1)-D(x-1,y,s+1)}{4} \tag{1.37}$$

$$D_{ys} = \frac{D(x,y+1,s+1)+D(x,y-1,s-1)-D(x,y+1,s-1)-D(x,y-1,s+1)}{4} \tag{1.38}$$

Preostaje rješavanje 3x3 linearnog sustava:

$$\begin{bmatrix} D_{xx} & D_{xy} & D_{xs} \mid -D_x \\ D_{xy} & D_{yy} & D_{ys} \mid -D_y \\ D_{xs} & D_{ys} & D_{ss} \mid -D_s \end{bmatrix} \tag{1.39}$$

Sustav jednadžbi rješava se po principu Gaussove eliminacije. Elementarnim transformacijama ekvivalentna matrica se svodi na gornju trokutastu matricu [20, 21].

$$\begin{bmatrix} 1 & xy & xs & | & d_x \\ 0 & 1 & ys & | & d_y \\ 0 & 0 & 1 & | & d_s \end{bmatrix} \tag{1.40}$$

Ukupna vrijednost pomaka u svakoj dimenziji je tada:

$$d_s = d_s \tag{1.41}$$

$$d_y -= ys * d_s \tag{1.42}$$

$$d_x -= (xy * d_y + xs * d_s) \tag{1.43}$$

Opisani postupak implementiran je metodom `gaussElimination` razreda `Matrix`. Poslije određivanja vrijednosti ukupnog pomaka d_x i d_y, povjerava se da li su one veće od 0.6 ili manje od -0.6. Koordinate x i y povećavaju se za jedan ako su dobivene vrijednosti veće od 0.6, a u slučaju da su manje od -0.06 onda se koordinate smanjuju za jedan. Na taj način dobivena je nova lokacija, za koju je potrebno ponovno izračunati vrijednosti gradijenata i riješiti Gaussovu eliminaciju. Ako se u nizu dobivaju vrijednosti d_x i d_y veće ili manje od +/-0.6, maksimalno se izvodi pet iteracija računanja nove lokacije. U suprotnom kreće se sa testiranjem stabilnosti promatranog kandidata značajke. Na temelju početnog pomaka D_x, D_y i zadnje izračunatog pomaka d_x, d_y prema izrazu 1.14 dobiva se nova vrijednost koja će se koristi za odbacivanje kandidata niskog kontrasta. U izvornom kodu je to jednadžba:

```
float contrast= t + 0.5 * (Dx * dx + Dy * dy + Ds * ds);
```

Gdje je *t* vrijednost detektiranog ekstrema, odnosno kandidata značajke. Da bi se promatrani kandidat proglasio značajkom, potrebno je odrediti odziv *DoG* funkcije uz rub. Za to se koriste izračunate vrijednosti gradijenata iz posljednje iteracije. Temeljem gradijenata određuje se omjer svojstvenih vrijednosti Hesseove matrice, prema izrazima 1.16 i 1.17. U izvornom kodu implementirane jednadžbe glase:

```
float TrHm = Dxx + Dyy;
float DetH = Dxx*Dyy - Dxy*Dxy;
float edge = ((TrHm*TrHm)/DetH);
```

Ako je izračunata vrijednost `contrast` manja od zadane vrijednosti praga 0.03 ili je za vrijednost `edge` istini izraz 1.44, gdje je *r* jednak 10, promatrani kandidat se odbacuje.

$$edge \geq \frac{(r+1)^2}{r} \tag{1.44}$$

U suprotnom kandidat se proglašava značajkom i dodaje u vektor značajki. Pri tome se svakoj značajki pridružuju x, y koordinate, oktava i mjerilo na kojoj je pronađena, te vrijednost *sigma* koja se koristila prilikom zaglađivanja Gaussovim filtrom. Vrijednost *sigma* računa se po formuli 1.45.

$$sigma = sigma0 * sigmaK * 2^{o+\frac{s+ds}{LEVELS}} \tag{1.45}$$

Radi se o interpoliranoj vrijednosti, na način da se mjerilu *s* na kojem je pronađena značajka pribroji pomak obzirom na mjerilo, tj. ds iz posljednje iteracije. Vrijednost *sigma* pridružena pojedinoj značajki u literaturi naziva se još mjerilo značajke [10,21]. Pronađena značajka dodaje se u sliku karakterističnih značajki.

6.3. Implementacija dodjele orijentacije

Ovaj korak je zapravo i posljednji korak implementacije u kojem se svim detektiranim značajkama iz prethodnog koraka dodjeljuje orijentacija. Dodjela orijentacije zasniva se na računanju amplitude i kuta gradijenta za sve elemente u okolini promatrane značajke iz određene slike Gaussove piramide. To bi značilo da se za svaku značajku mora ponovno dohvaćati određena slika iz Gaussove piramide, a za pojedine elemente te slike u okolini značajke iznova se računaju amplituda i kut gradijenta. Zbog toga, kako se u prosjeku po slici nalazi dosta veliki broj značajki, najjednostavnije je izračunati amplitudu i kut gradijenta za sve elemente određene slike Gaussove piramide. Obrađuju se sve slike Gaussove piramide na pozicijama koje odgovaraju slikama *DoG* piramide u kojima su detektirane značajke. Amplituda se računa prema izrazu 1.21, a kut gradijenta prema 1.22. Niže prikazan je izvorni dio koda u kojem se obavlja opisani postupak izračunavanja amplitude i kuta za sve elemente slike. U kodu su `magnitude` i `orientation`

dvodimenzionalna polja slika, gdje `magnitude` služi za spremanje izračunate vrijednosti amplitude, a `orientation` za spremanje kuta gradijenta. Kut gradijenta izračunava se ugrađenom funkcijom `atan2`, koja vraća vrijednosti iz intervala [-π, π]. Vrijednost funkcije `atan2` predaje se funkciji `fast_mod_2pi` koja osigurava da konačna vrijednost kuta gradijenta bude u intervalu $[0, 2\pi]$.

```
for(int o=0; o<OCTAVE; o++){
   for(int s=1; s<LEVELS+1; s++){

      magnitude[o][s-1].setDimension(gaussList[o][s-1].height,
                                gaussList[o][s-1].width);
      orientation[o][s-1].setDimension(gaussList[o][s-1].height,
                                gaussList[o][s-1].width);

      for(y=1; y<gaussList[o][s].height-1; y++){
         for(x=1; x<gaussList[o][s].width-1; x++){

            float x2=gaussList[o][s].GetPixel(y,x+1);
            float x1=gaussList[o][s].GetPixel(y,x-1);
            float y2=gaussList[o][s].GetPixel(y+1,x);
            float y1=gaussList[o][s].GetPixel(y-1,x);
            float Dx=float(0.5*(x2-x1));
            float Dy=float(0.5*(y2-y1));

            float m=sqrt((Dx*Dx)+(Dy*Dy));
            magnitude[o][s-1].SetPixel(y,x,m);

            float th= fast_mod_2pi(atan2(Dy,Dx));
            orientation[o][s-1].SetPixel(y,x,th);

         }
      }
   }
}
```

Nakon što su izračunate potrebne vrijednosti amplitude i kuta gradijenta, kreće sa dodjelom orijentacije pojedinoj značajki. Redom se prolazi po slikama karakterističnih značajki. Detektiranjem pojedine točke u slici karakterističnih značajki, dohvaća se tražena točka iz vektora značajki. Uspoređivanjem x i y koordinate dodatno se provjerava da li je

zaista riječ o traženoj značajki. Nakon što je utvrđena vjerodostojnost značajke, slijedi dohvaćanje vrijednosti *sigma* pridružene toj značajki. Vrijednost *sigma* za svaku značajku izračunata je i objašnjena u prethodnom koraku implementacije. Temeljem te vrijednosti određuje se okolina značajke. Određuje se nova vrijednost *sigmaW* koja je jednaka umnošku vrijednosti *sigma* i faktora 1.5. Dobivena vrijednost *sigmaW* koristi se za računanje polumjera prozora kojim se obuhvaćaju svi potrebni elementi u susjedstvu značajke.

Iz definirane okoline značajke, redom se dohvaćaju prethodno izračunate vrijednosti amplitude i kuta gradijenta, te se izgrađuje orijentacijski histogram za tu značajku. Histogram je prikazan jednodimenzionalnim poljem od 36 elemenata. Svaki element histograma pokriva 10 stupnjeva, što daje ukupan raspon od 360 stupnjeva. Za svaki element iz okoline značajke dohvaća se vrijednost kuta gradijenta, te se određuje kojem elementu histograma pripada. Određeni element histograma povećava se za iznos amplitude tog elementa ponderirane vlastitim doprinosom. Tako će elementi koji se nalaze bliže promatranoj značajki imat veći utjecaj u izgradnji njezinog histograma. Doprinos se računa kao eksponent Gaussove funkcije prikazane izrazom 1.3, gdje x i y koordinate predstavljaju udaljenost promatranog elementa od značajke. Histogram je izgrađen u potpunosti nakon što se na prethodno opisani način pribroje vrijednosti svih elemenata iz okoline promatrane značajke.

Ovako izgrađen histogram sastoji se od diskretnih vrijednosti, A. Vedaldi savjetuje da se prije određivanja maksimalne vrijednosti, histogram dodatno zagladi računanjem prosjeka. Za svaku se vrijednost histograma dohvaćaju i vrijednosti elementa lijevo i desno, te se računa prosječna vrijednost ta tri elementa [20]. Dobiveni rezultat predstavlja novu vrijednost promatranog elementa u histogramu. Tek nakon što je završeno zaglađivanje histograma, traži se najveća vrijednost, odnosno vrh histograma. Zatim se prolazi kroz sve elemente histograma, te se za svaki element ispituje da li se nalazi unutar 80% najveće vrijednosti i da li je veći od susjednih elementa lijevo i desno. Element koji ispunjava navedene kriterije naziva se lokalni vrh [20]. U slučaju da je promatrani element zapravo lokalni vrh, izvodi se kvadratna interpolacija. Rezultat interpolacije je orijentacija promatrane značajke, te se dodaje u vektor orijentacija. Svaki lokalni vrh dat će jednu orijentaciju. Poslije detekcije svih lokalnih vrhova, značajki se

pridjeljuje generirani vektor orijentacija. Broj orijentacija svake značajke ovisi o broju detektiranih lokalnih vrhova. U osnovnoj implementaciji A. Vedaldija pronalaze se maksimalno četiri orijentacije [21]. Odnosno, interpoliraju se samo prva četiri detektirana lokalna vrha. No, D. Lowe savjetuje pronalaženje svih mogućih orijentacija jer to značajno pridonosi uspješnosti traženja korespondentnih točaka [9].

6.4. Pokretanje programa

Konačno rješenje ove verzije ugrađeno je u ljusku cvsh2 razvijenu na ZEMRISU od strane doc. dr. sc. Siniše Šegvića. Ljuska cvsh2 koristi se za iscrtavanje značajki detektiranih algoritmom SIFT.

Prilikom pokretanja programa predaje se ulazna datoteka koja sadrži niz slikovnih okvira u kojima će se detektirati značajke. Potrebno je napomenuti da slike ne trebaju biti istih dimenzija. Program će raditi ukoliko je iduća slika u nizu manja ili veća od prethodno obrađene. Kao izlaz programa dobivaju se dvije slike sa označenim detektiranim značajkama. Prva slika predstavlja referentnu sliku, a druga rezultat trenutno obrađene slike. Time korisnik ima mogućnost uspoređivati detektirane značajke nad različitim slikama.

Argumenti komandne linije kojom se definiraju parametri ljuske su:

```
-a=ozana_sift  -i="p n" -sf="D:\tony1/"
```

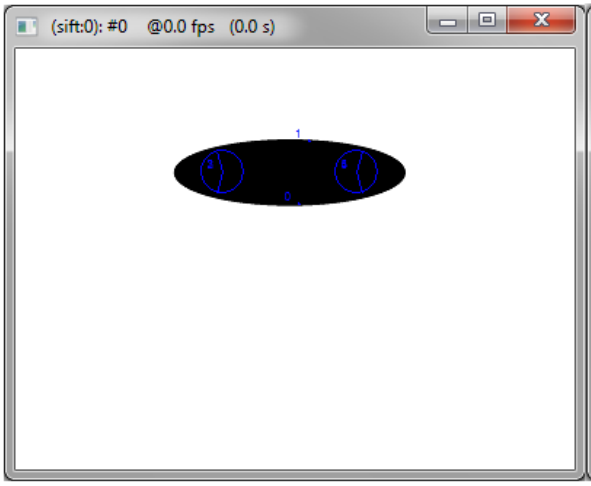
Parametar „-a“ označava da će se unutar ljuske pokrenuti algoritam upravo opisan u ovom radu. Dalje, „-i“ određuje naredbu koja će se izvršiti prilikom pokretanja programa, te „-sf“ označava putanju do datoteke koja, kao što je već ranije spomenuto, sadrži niz slikovnih okvira.

7. Analiza rezultata algoritma SIFT za CPU

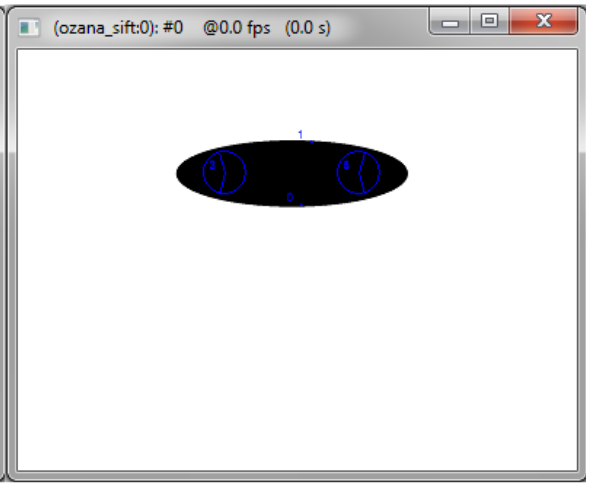
Razvijena verzija implementacije za izvođenje na procesoru opće namjene (u nastavku koristi se naziv verzija CPU), isprobat će se na parovima slika, te će se evaluirati performanse u odnosu na osnovnu, postojeću implementaciju A. Vedaldija. Izvesti će se profiliranje algoritma SIFT i time ustvrditi koji dijelovi troše najviše vremena.

7.1. Prikaz rezultata

Prvo će se usporediti dobiveni rezultati verzije CPU sa rezultatima koje daje implementacija A. Vedaldija. Svi nužni parametri u obje implementacije postavljeni su na iste vrijednosti koje predlaže D. Lowe. Prag za odbacivanje ekstrema niskog kontrasta postavljen je na 0.03, a prag za odbacivanje kandidata značajki slabo lokaliziranih uz rub na 10. D. Lowe u svom radu ne navodi na koji način se izračunava ukupan broj oktava, ali kaže da slike posljednje oktave u nizu ne bi trebale biti manje od 30px [9]. Zato je za ulazne slike dimenzija 384x288 određeno da se izvođenjem algoritma SIFT generiraju četiri oktave sa pet razina mjerila (*levels* je 2).



Slika 18. a) Rezultat implementacije A. Vedaldija



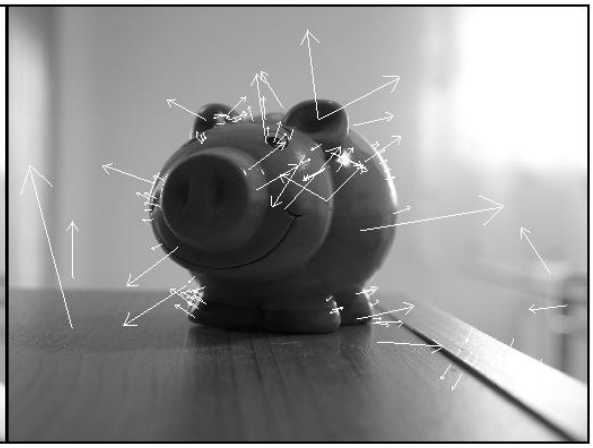
Slika 18. b) Rezultat vlastite implementacije

Izvođenjem je detektirano ukupno 52 kandidata značajki. Detektiranjem nestabilnih kandidata odbačeno je ukupno 47 kandidata, tako da status značajki imaju samo 4

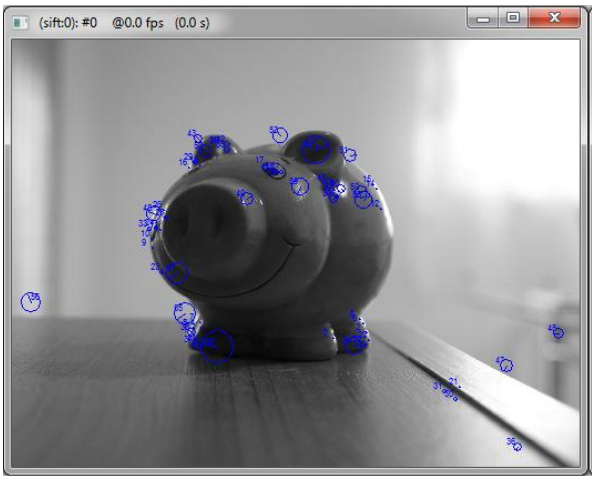
ekstrema prikazana na slici 18. Kako se ulazna slika sastoji samo od jedne crne elipse na bijeloj pozadini, algoritam bi trebao pronaći dvije značajke uz svaki kraj elipse. Pri tome lokacije značajki trebaju biti u blizini točaka žarišta. Takvi rezultati su dobiveni i prikazani na slici 18. Radijus kružnice je jednak mjerilu promatrane značajke, a orijentacija je prikazana kao kut danog radijusa. Za usporedbu se koriste rezultati trećeg koraka algoritma SIFT implementacije A. Vedaldija, bez implementacije deskriptora.

Ovdje je prikazana usporedba tri verzije algoritma SIFT. Radi se o implementacijama autora A. Vedaldija, D. Lowea i verzije CPU opisane u ovom radu.

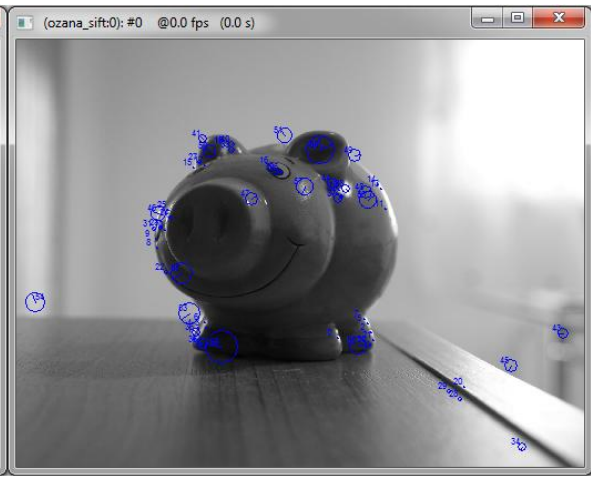
Slika 19 a) Originalna slika



Slika 19 b) Rezultat implementacije D. Lowea



Slika 19. c) Rezultat implementacije A. Vedaldija



Slika 19. d) Rezultati implementacije verzije CPU

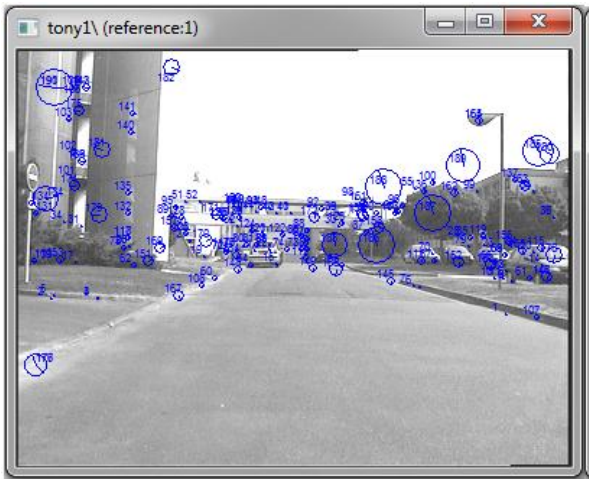
Testiranje je provedeno na sivoj slici dimenzija 512x384. Prostor mjerila sastoji se od četiri oktave, a svaka oktava sadrži ukupno pet razina mjerila. Detektirani broj značajki u verziji CPU i A. Vedaldija je 61, dok su u verziji D. Lowea detektirane 93 značajke.

Implementacija A. Vedaldija i verzija CPU detektiraju sve značajke na istim mjestima u slici. Osim jednakih vrijednosti x i y koordinata daju identične rezultate za mjerilo promatrane značajke. Minorne razlike su u orijentaciji, koja se kod nekih značajki razlikuje samo za par stupnjeva. U cilju određivanja odstupanja orijentacije izračunata je mjera varijabilnosti, tj. standardna devijacija koja iznosi 0.005. Potrebno je napomenuti da su rezultati na slikama c) i d) zapravo rezultati trećeg koraka algoritma SIFT. Slika b) prikazuje rezultate nakon izgradnje deskriptora, te strelice na slikama odgovaraju vektorima deskriptora. Osim što implementacija D. Lowea detektira veći broj značajki, postoje minorne razlike u x, y koordinatama, mjerilu i orijentaciji u odnosu na značajke detektirane verzijama A. Vedaldija i CPU. Za sve značajke koje se pronalaze u sve tri verzije određeno je odstupanje po svim parametrima. Kako verzije CPU i A. Vedaldija daju gotovo identične rezultate u tablici 1. prikazano je odstupanje u odnosu na implementaciju D. Lowea.

Tablica 1. Kvantifikacija odstupanja implementacije D. Lowea i verzije CPU

	X koordinata	Y koordinata	MJERILO	ORIJENTACIJA
STANDARDNA DEVIJACIJA	0.111	0.174	0.135	0.073
VARIJANCA	0.012	0.030	0.018	0.005
SREDNJA VRIJEDNOST ODSTUPANJA	0.006	0.007	0.053	0.074

Niže su prikazani rezultati izvođenja verzije CPU na parovima prometnih slika.



Slika 20. a) Referentna slika

Slika 20. b) Pedeseta slika niza

Slika 20. c) Stota slika niza

Slika 20. d) Dvjestota slika niza

Na slici 20. a) prikazana je referentna početna slika, b) pedeseta slika u nizu, c) stota slika, a d) odgovara dvjestotoj slici u nizu. Zapravo, b), c) i d) prikazuju slike na kojima je kamera više približena objektu i pod različitim kutom. Broj detektiranih značajki referentne slike jednak je 165, a broj značajki slike b) nešto je veći i iznosi 173. Na slici c) detektirano je 161, a na slici d) 169 značajki. Na slikama b), c) i d) detektira se većina značajki koje se nazale na referentnoj slici, a približavanjem kamere i dalje su u kadru. Pri tome se misli na položaj značajke u odnosu na pojedini objekt na slici. Parovi ovih slika dokazuju invarijantnost algoritma SIFT obzirom na rotaciju i mjerilo.

7.2. Vrijeme izvođenja

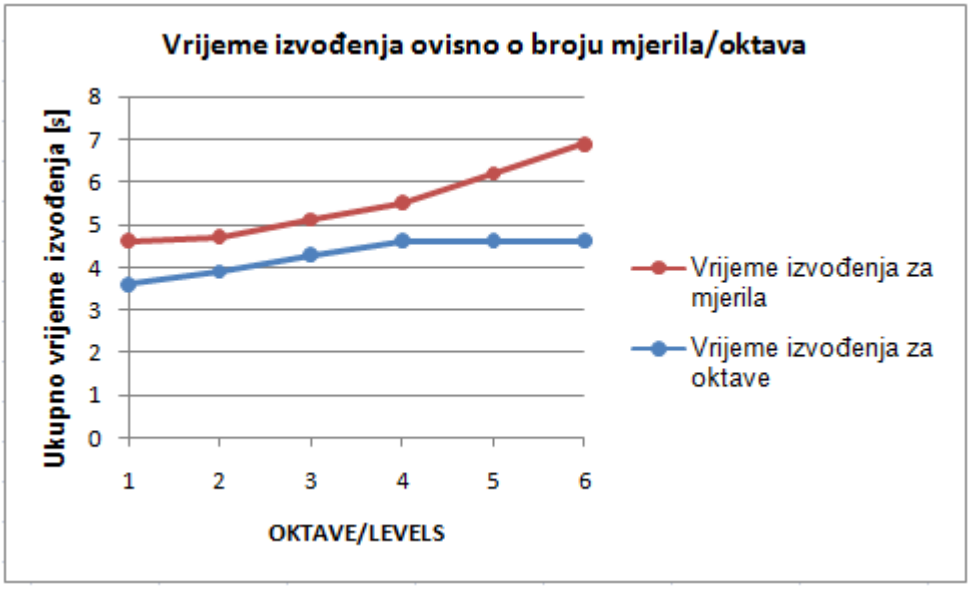
Potrebno je odrediti na što se troši najviše vremena prilikom izvođenja. Da bi se to odredilo, izmjereno je vrijeme svakog koraka i međukoraka. Prosječno vrijeme obrade jedne slike u *Releise* načinu prikazano je tablicom 2.

Tablica 2. Prosječno vrijeme obrade jedne slike algoritmom SIFT

KORACI ALGORITMA SIFT			VRIJEME [ms]
1.	Izgradnja prostora mjerila		641.68
	1.1	Konvolucija Gaussovim filtrom	613.95
	1.2	Promjena rezolucije slike	7.05
	1.3	Računanje DoG piramide	20.68
2.	Detekcija i lokalizacija ključnih točaka		40.16
	2.1	Detekcija ekstrema	11.45
	2.2	Određivanje stabilnih kandidata	28.71
3.	Dodjela orijentacije		176.43
	3.1	Određivanje amplitude i kuta gradijenta	158.99
	3.1	Određivanje histograma i orijentacije	17.44
UKUPNO			**858.27**

Testiranje je provedeno za sive slike dimenzija 384x288, a prikazano je prosječno vrijeme kroz pet mjerenja. Kao što se vidi iz tablice, najviše vremena odlazi na izgradnju prostora mjerila. Izmjereno vrijeme je potrebno za generiranje četiri oktave po pet razina mjerila. Povećavanjem broja mjerila i/ili oktava još se više povećava vrijeme izgradnje prostora mjerila, a time i samog algoritma. Taj se odnos može vidjeti iz grafa na slici 21., a prikazano vrijeme potrebno je za obradu i prikaz rezultata jednog slikovnog okvira. Kod izrade grafa korištene su sive slike dimenzija 512x512, kako bi slika zadnje oktave u obje dimenzije bila veća od 30px. Gornja, crvena linija grafa prikazuje da se povećanjem broja razina mjerila po oktavi, povećava i ukupno vrijeme izvođenja. Znači, broj oktava je

konstantan i iznosi pet, a mijenja se samo broj razina mjerila. Ukupan broj mjerila jednak je *levels+3*, te se povećavanjem parametra *levels* (uzduž osi x), povećava i broj mjerila. Druga, plava linija niže pokazuje ovisnost ukupnog vremena izvođenja o broju oktava. Za testiranje generira se pet mjerila po oktavi. Povećavanjem broj mjerila i/ili oktava povećava se broj slika koje se zaglađuju Gaussovim filtrom, a time i broj razina *DoG* piramide. Zbog toga je odlučeno izgradnju prostora mjerila u potpunosti implementirati u okruženju CUDA.



Slika 21. Vrijeme izvođenja algoritma SIFT ovisno o broju mjerila/oktava

Drugi korak implementacije, tj. detekcija i lokalizacija značajki u odnosu na izgradnju prostora mjerila troši vrlo malo vremena, kao što se vidi iz tablice 2. Implementacijom tog koraka za izvođenje na grafičkom procesoru postiglo bi se ubrzanje. Postignuto ubrzanje ne bi mnogo pridonosilo cjelokupnom ubrzanju algoritma SIFT, kao što je to slučaj kod izgradnje prostora mjerila. Iz tih razloga, drugi korak algoritma SIFT neće se implementirati za izvođenje na grafičkom procesoru. Posljednji korak implementacije troši oko trostruko manje vremena u odnosu na prvi korak, ali preko četiri puta više od drugog koraka (vidi tablicu 2.). Testiranjem je izmjereno vrijeme po međukoracima. Time je utvrđeno da ukupnom vremenu dodjele orijentacije uvelike pridonosi računanje amplitude i kuta gradijenta. Razlog tome je što se oni računaju za sve elemente određene slike, kao što je objašnjeno u odjeljku 6.3. Vrijeme ovog međukoraka čini oko 90%

ukupnog vremena trećeg koraka dodjele orijentacije, te će njegovo ubrzanje dati vidljive rezultate. Zbog toga je u ovom radu računanje amplitude i kuta gradijenta implementirano za izvođenje na grafičkom procesoru. Spomenuti dijelovi algoritma SIFT za izvođenje na grafičkom procesoru implementirani su u okruženju CUDA, a njihove implementacije su detaljno objašnjene u idućem poglavlju.

8. Programsko rješenje algoritma SIFT za GPU

8.1. Instalacija CUDA okruženja

Kako bi se iskoristila snaga i performanse koje pružaju moderni grafički procesori u sklopu ovog rada, koristi se programsko okruženje CUDA tvrtke NVIDIA. Glavni uvjet za programiranje grafičkih procesora u okviru okruženja CUDA, predstavlja posjedovanje grafičke kartice NVIDIA, koja podržava navedeno okruženje. Nakon instaliranja odgovarajućeg pogonskog programa (engl. *driver*), potrebno je instalirati CUDA SDK (engl. *CUDA Software Development Kit*) i razvojno okruženje CUDA (engl. *CUDA Toolkit*). Najjednostavniji način korištenja okruženja CUDA, koji je i primijenjen u ovom radu, nalazi se u sklopu razvojnog okruženja Microsoft Visual Studio. CUDA podrška za Visual Studio (engl. *Cuda Wizard*) omogućuje brzu integraciju *nvcc* prevodioca. Funkcije jezgre se prevode u binarni kod i time je omogućeno njihovo izvođenje na grafičkom procesoru. Zahvaljujući CUDA podršci za Visual Studio svi detalji vezani uz prevođenje i izvođenje na grafičkoj kartici skriveni su od korisnika, odnosno programera. Stoga se s korisnikove perspektive programiranje grafičkih procesora gotovo i ne razlikuje od uobičajenog programiranja u okruženju Visual Studio. Svi neophodni tehnički resursi korišteni u ovom radu za programiranje u okruženju CUDA su:

- Grafička kartica GeForce 310M
- GPU Driver za Windows verzija 3.1
- CUDA Software Development Kit verzija 3.1
- CUDA Toolkit verzija 3.1
- Microsoft Visual Studio 2008. Professional
- CUDA Visual Studio Wizard

8.2. Implementacija

Ova je verzija implementacije za izvođenje na grafičkom procesoru (u nastavku se koristi naziv verzija GPU) isto tako ugrađena u ljusku cvsh2. Ljuska omogućava prikaz rezultata na parovima slika, te se na taj način mogu obraditi sve slike iz zadanog niza. Općenito, da bi se slika obradila na grafičkom procesoru potrebno ju je dohvatiti u prethodno alociranu memoriju grafičkog procesora (u nastavku se koristi naziv memorija CUDA). Iz iskustava, temeljem dosadašnjeg rada u okruženju CUDA, poznato je da alociranje i kopiranje podataka u memoriju CUDA zahtijeva dosta vremena. To je još jedan od razloga zašto je odlučeno u potpunosti implementirati izgradnju prostora mjerila u okruženju CUDA. U suprotnom, mnogo vremena bi se gubilo prilikom kopiranja međurezultata između memorije *hosta* i memorije CUDA. U memoriju CUDA će se učitati samo originalna slika, a sve ostale slike biti će generirane na grafičkom procesoru. Osim kopiranja, potrebno je izbjeći i nepotrebna alociranja. Zato se memorija CUDA alocira samo jednom, prilikom obrade početne slike iz niza. Pri tome se alocira memorijski prostor koji odgovara slici za sve oktave i pripadajuća mjerila. Ukoliko je iduća slika iz niza različitih dimenzija, oslobodit će se memorija CUDA, te ponovno zauzeti u skladu sa dimenzijama iduće slike. Važno je naglasiti da CUDA *runtime* alocira u globalnoj memoriji memorijski prostor s početnom adresom koja je višekratnik broja 128 i smješta prvi element polja na tu adresu. Alociranje memorije CUDA implementirano je statičkom metodom `memAlloc`, a oslobađanje metodom `memFree` razreda `MySift`. U slučaju da su sve slike niza istih dimenzija, memorija CUDA se oslobađa pri izlasku iz programa.

Vremenski ključni dijelovi algoritma implementirani su po istom principu opisanom u poglavlju 6.1 i 6.3. Također, koriste se isti parametri i formule. Jedina je razlika što se sve operacije kao što su: zaglađivanje Gaussovim filtrom, smanjivanje i povećavanje rezolucije slike, generiranje *DoG* piramide, te računanje amplitude i kuta gradijenta izvode na grafičkom procesoru. Njihova implementacija nalazi se u datoteci `buildPyramid.cu`. Ovdje se ukratko podsjeća na redoslijed izvođenja navedenih operacija. Kod izgradnje prostora mjerila samo se originalna slika dohvaća u prethodno alociranu memoriju. Dohvaćena slika se pretprocesira Gaussovim filtrom standardne devijacije 0.5 i udvostruči. Dvostruko veća slika se dalje konvoluira po svim mjerilima sa Gaussovim filtrom koji

pripada pojedinoj razini, kao što je to objašnjeno u odjeljku 6.1. Rezultati konvolucije ostaju pohranjeni u memoriji CUDA na odgovarajućim lokacijama čije su početne adrese višekratnici broja 128. Poslije konvoluiranja svih slika prve oktave, smanjuje se rezolucija za faktor dva. Smanjena slika predstavlja osnovnu sliku za iduću oktavu. Sve oktave u nizu generiraju se na isti način. U tako izgrađenoj Gaussovoj piramidi oduzimaju se slike susjednih mjerila po svim oktavama. Slike koje nastaju kao rezultat oduzimanja predstavljaju *DoG* piramidu. Generiranjem *DoG* piramide dolazi se do kraja prvog koraka, te je potrebno pripadajuće slike *DoG* kopirati iz memorije CUDA u memoriju *hosta*. Nastavlja se sa izvođenjem drugog koraka algoritma SIFT na procesoru opće namjene. Rezultat ovog koraka su značajke, kojima samo preostaje dodijeliti orijentaciju. Za izračun orijentacije koriste se vrijednosti amplitude i kuta gradijenta iz okoline značajke. Računanje tih vrijednosti predstavlja posljednji dio koji će se izvoditi na grafičkom procesoru. Implementacije svih vremenski kritičnih dijelova detaljno su objašnjene u nastavku, redoslijedom koji odgovara njihovom izvođenju.

8.2.1. Konvolucija slike Gaussovim filtrom

Na samom početku potrebno je odrediti Gaussov filtar. Prvo se određuje polumjer filtra, a zatim se računaju svi njegovi elementi po istom principu kao i u verziji CPU. Nakon što je izračunat, filtar se pohranjuje u memoriju konstante kao jednodimenzionalno polje. To omogućuje navođenje ključne riječi `__constant__`.

```
__device__ __constant__ float c_Kernel[KERNEL_W];
```

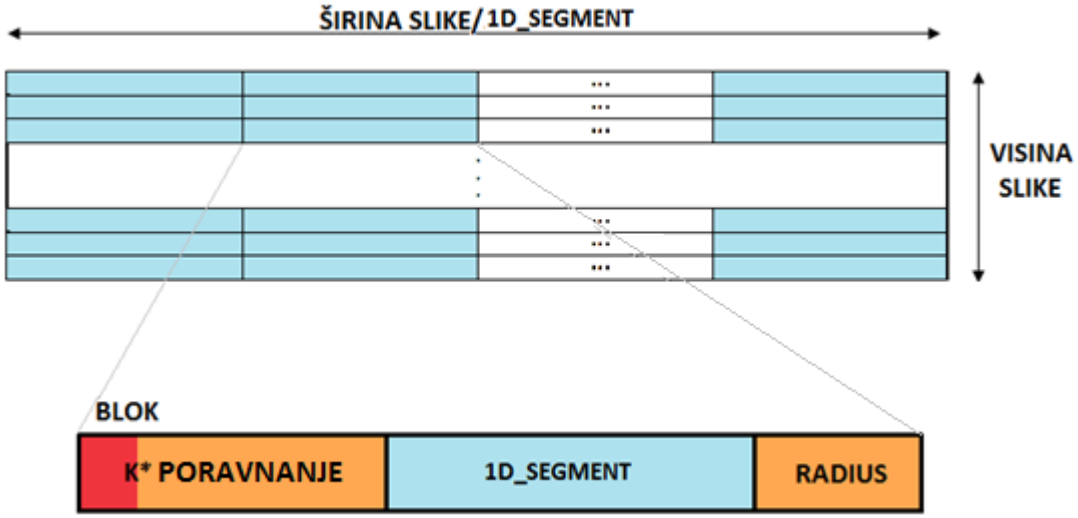
Polumjer, tj. *radijus* filtra predstavlja važan podatak, koristi se prilikom određivanja dimenzija blokova unutar mreže i kod same konvolucije. Konvolucija se izvodi u dva prolaza, prvo se konvoluiraju reci slike Gausovim filtrom, a potom stupci. U oba se prolaza koristi zajednička memorija. Namjera je dohvatiti segment slike iz globalne memorije u zajedničku memoriju, a onda se nad tim segmentom izvodi konvolucija. Korištenjem zajedničke memorije izbjeći će se višestruki pristupi sporoj globalnoj memoriji, što daje dodatno ubrzanje. Prilikom dohvaćanja segmenta slike iz globalne memorije treba posebnu pažnju posvetiti sa kojih se memorijskih lokacija čitaju podaci. Kao što je opisano

u odjeljku 5.4 i 5.5, dretve iste osnove istovremeno dohvaćaju podatke iz globalne memorije. Kada dretve čitaju podatke koji pripadaju različitim segmentima, povećava se broj memorijskih transakcija. Slikovni elementi veličine 4 okteta (*float*) zahtijevaju segmente od minimalno 64 okteta poravnate na lokacije višekratnika broja 64 [14].

Konvolucija po recima

Reci slike su podijeljeni na segmente, u nastavku koristi se naziv *1D_segmenti*. Slika se konvoluira po *1D_segmentima,* tako da je jedna dretva osnove zadužena za računanje konvolucije jednog, njoj centralnog elementa iz *1D_segmenta*. Pozicija dretve unutar osnove odgovara poziciji centralnog elementa unutar promatranog *1D_segmenta* slike. Osnove se sastoje od 32 dretve, a prva dretva osnove uvijek dohvaća element koji se nalazi na poziciji višekratnika 128. Razlog tome je što su podaci poravnati na adresama višekratnika 128 (*k**128), a kako su podaci veličine 4 okteta dobiva se adresa elementa k*128 + *redni_broj_osnove**32*4. Međutim da bi se izračunala konvolucija za centralni element, osim dohvaćanja njegove vrijednosti potrebno je dohvatiti elemente koji se nalaze *radijus* elemenata desno i *radijus* elementa lijevo od centralnog elementa. Time se pristupa elementima van promatranog memorijskog segmenta, te se povećava broj memorijskih transakcija. Jedini način da se to izbjegne je pristupanje podacima koji su i dalje poravnati u memoriji. Svaka će dretva osim centralnog elementa pristupati elementima za 16 mjesta udaljenim lijevo i desno od centralnog. Uzima se broj 16 jer je to zapravo jednako broju dretvi polu-osnove i predstavlja minimalno zadovoljavajuće *poravnanje*. Postavljanjem *poravnavanja* na 16 moguće je obaviti konvoluciju samo sa filtrom *radijusa* manjeg ili jednakog 16. Kako se za generiranje Gaussovog filtra koriste parametri koji u prosjeku daju *radijus* veći od 16, implementirano je iduće rješenje. Svaki put kad se odredi filtar provjeri se da li je veći od *k**16, gdje $k \in 1,..2$. U slučaju da je veći, *poravnanje* se povećava za 16, na taj način implementiran je maksimalan *radijus* filtra jednak 48. Rješenje problema implementirano je funkcijom `convolutionRowsGPU`, gdje se izračunati *radijus* i iznos *poravnanja* predaju funkciji jezgre `ConvRowGPU`. Prije poziva funkcije jezgre potrebno je definirati mrežu dretvi i blokove unutar mreže. Oni se određuju na temelju izračunatih parametara *radijus* i *k*poravnanje*, te veličine

1D_segmenta slike koji mora biti višekratnik broja 16, broja dretvi polu-osnove. Broj blokova unutar mreže u smjeru osi x jedak je broju koji se dobiva dijeljenjem širine slike sa veličinom *1D_segmenta*. U slučaju da kvocijent nije cijeli broj, uzima se prvi veći (to omogućava funkcija `divide`). Broj blokova unutar mreže u smjeru osi y jednak je visini slike (vidi sl. 22.).



Slika 22. Organizacija blokova unutar mreže i prikaz bloka

U funkciji jezgre iz globalne memorije čitaju se podaci veličine *k*poravnanje + 1D_segment + radijus*, te se time postiže poravnanje. U zajedničku memoriju pohranjuju se samo potrebni podaci da bi se obavila konvolucija *1D_segmenta*, tj. ukupno *radijus + 1D_segmenta + radijus* podataka. Temeljem indeksa promatrane dretve određuje se pozicije sa kojih se čita iz globalne memorije *glLoc* i pozicije na koje se upisuje u zajedničku memoriju *sLoc*. Sve dretve za koje su pozicije *sLoc* pozitivne ili nula pohranjuju pročitani podatak globalne memorije na tu poziciju u zajedničku memoriju, za sve ostale dretve pročitani podaci se odbacuju. Da bi se uskladili rezultati konvolucije u okruženju CUDA sa rezultatima verzije CPU, potrebno je računati konvoluciju za prvih i zadnjih *radijus* elementa lijeve i desne strane slike. Za te elemente ne postoje pojedini susjedi, te se odgovarajući elementi zajedničke memorije postavljaju na vrijednosti prvih elemenata sa lijeve ili desne strane ruba. Nakon što su dohvaćeni svi elementi u zajedničku memoriju, slijedi sinkronizacija dretvi. To je zapravo barijera koja osigurava da se dohvate svi podaci prije nego se kreće sa izvođenjem iduće operacije. Svaka dretva na temelju

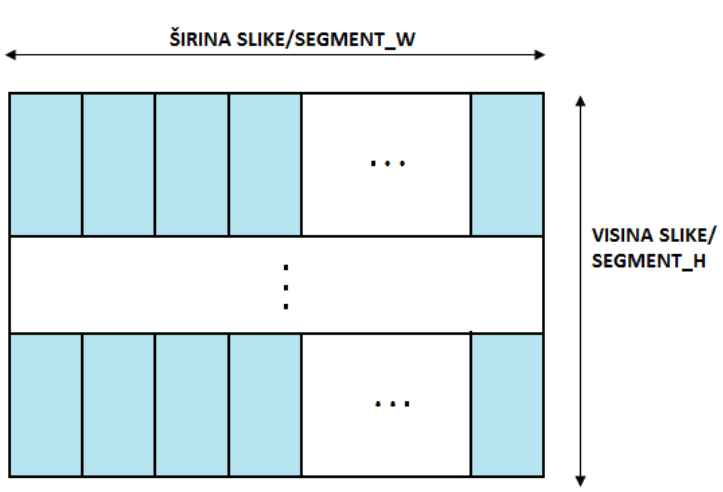
svojeg indeksa zna sa koje pozicije treba čitati elemente zajedničke memorije i gdje treba zapisati dobiveni rezultat konvolucije u globalnoj memoriji. Tako će se za svaki centralni element obaviti 2**radijus*+1 množenja podataka filtra sa odgovarajućim podacima zajedničke memorije. Više dretvi koristit će iste podatke iz zajedničke memorije, ali to ne predstavlja problem jer je dozvoljeno dijeljenje svih podataka definiranih na razini bloka. Opisani izvorni kod konvolucije po recima prikazan je niže.

```
#define IMUL(a,b) __mul24(a,b)
#define SEGMENT_1D    128

template<int i> __device__ float ConvRow(float *s_Data)
{
    return s_Data[i]*c_Kernel[i] + ConvRow<i-1>(s_Data);
}

template<> __device__ float ConvRow<-1>(float *s_Data)
{
    return 0;
}

template<int RADIUS, int ALIGNMENT>
__global__ void ConvRowGPU(float *d_Result, float *d_Data,
                           int width, int height)
{
  __shared__ float s_Data[RADIUS+SEGMENT_1D+RADIUS];

  const int segStart = IMUL(blockIdx.x, SEGMENT_1D);

  const int rowStart = IMUL(blockIdx.y, width);
  const int rowEnd = rowStart + width - 1;
  const int glLoc = threadIdx.x - ALIGNMENT + segStart;
  const int sLoc =  threadIdx.x - ALIGNMENT + RADIUS;

  if (sLoc>=0) {
     (glLoc<0)?s_Data[sLoc]=d_Data[rowStart]:((glLoc>=width)?
     s_Data[sLoc]=d_Data[rowEnd]:s_Data[sLoc=d_Data[rowStart+glLoc]);
  }
  __syncthreads();

  const int segEnd = segStart + SEGMENT_1D - 1;
  const int borders = min(segEnd, width - 1);
  const int writePos = segStart + threadIdx.x;
```

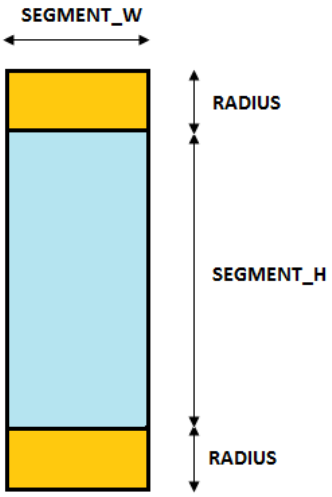
```
    if (writePos <= borders){
        const int sLoc = threadIdx.x + RADIUS;
        d_Result[rowStart + writePos]=ConvRow<2*RADIUS>(s_Data+sLoc-RADIUS);
    }
}
```

Konvolucija po stupcima

Konvolucija po stupcima radi se nad rezultatom konvolucije po recima. Elementi stupca nisu slijedno zapisani u memoriji. U slučaju da je pročitan neki element stupca, idući element udaljen je za iznos širine slike i samim time ne pripada istom segmentu u memoriji. Čitanje takvih ne poravnatih podataka, odnosno podataka različitih segmenata rezultira višestrukim memorijskim transakcijama i zbog velikog broja pristupa znatno narušava performanse. Da bi se to izbjeglo, potrebno je ipak nekako organizirati funkciju jezgre da dretve iste pod-osnove čitaju elemente na slijednim adresama. Zbog toga se umjesto samo elementa stupca iz globalne memorije čita dvodimenzionalan blok elemenata. U implementaciji radi se o bloku slike dimenzija 16x32. Dimenzija x mora odgovarati broju dretvi polu-osnove. Dimenzija y može se postaviti na neki drugi višekratnik broja 16, npr. 48, ali se tada povećava broj pod-segmenata objašnjenih kasnije. Taj blok predstavlja dvodimenzionalni segment slike (u nastavku se koristi naziv *2D_segment*) nad kojim će se izvesti konvolucija. Na slici 23. a) prikazano je kako se slika dijeli u spomenute *2D_segmente* dimenzija *segment_w* x *segment_h*, gdje je *segment_w* jednak 16, a *segment_h* 32. Na identičan način organiziraju se blokovi unutar mreže dretvi, tako da slika 23. a) prikazuje i organizaciju mreže. Da bi se uspješno konvoluiralo *radijus* rubnih elementa promatranog *2D_segmenta*, potrebno je dohvatiti *radijus* elemente iznad i ispod *2D_segmenta*. Zbog toga se iz globalne memorije u zajedničku memoriju dohvaća *2D_segment* zajedno sa *radijus* elemenata iznad i ispod (vidi sl. 23. b).



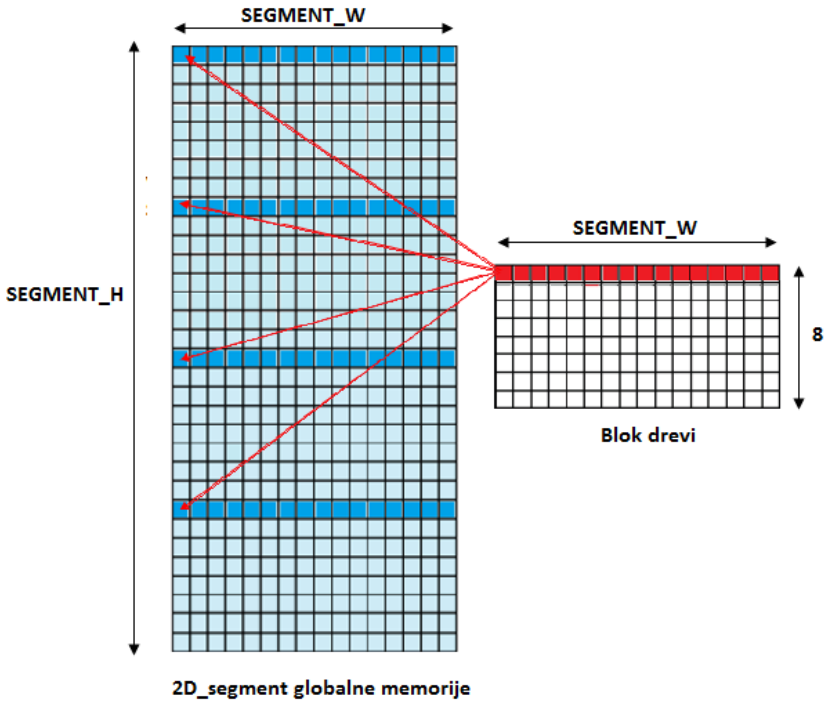
Slika 23. a) Organizacija mreže dretvi



Slika 23. b) Segment slike koji se dohvaća u zajedničku memoriju

Za dohvaćanje potrebnih elemenata i *2D_segmenta* iz globalne memorije u zajedničku, te za računanje konvolucije odgovorne su sve dretve bloka. Radi se o blokovima dimenzija *segment_w* x 8. Dimenzije bloka mogu se postaviti i na npr. *segment_w* x 16 ali se tada, kao što je objašnjeno kasnije, povećava broj memorijskih transakcija unutar bloka. Kako je x dimenzija bloka i *2D_segmenta* jednaka, sve dretve istog retka bloka dohvaćat će jedan redak *2D_segmenta*. Elementi retka promatranog *2D_segment* slijedno su zapisani u globalnoj memoriji, pa će dretve jedne polu-osnove unutar bloka čitati elemente istog segmenta memorije. Tako je smanjen broj memorijskih transakcija. Umjesto 16 memorijskih transakcija, sad je samo jedna i to prilikom čitanja prvog elementa retka. Za blok od 8 redaka to daje 8 memorijskih transakcija. Opisano dohvaćanje elemenata prikazano je na slici 24., gdje crvene strelice pokazuju na redak u globalnoj memoriji (obojan plavo) koji će slijedno čitati sve dretve retka bloka (obojane crveno). Sa slike se vidi da postoji više crvenih strelica. To je zbog toga što je u implementaciji *2D_segment* četiri puta veći od bloka dretvi, te se iste dretve bloka koriste za dohvaćanje elementa slike koji su udaljeni u smjeru osi y za 8 elemenata, odnosno y dimenziju bloka dretvi. Jednostavnije, može se zamisliti da je *2D_segment* podijeljen na pod-segmente koji se u zajedničku memoriju dohvaćaju istim blokom dretvi. Osim *2D_segmenta* u zajedničku memoriju dohvaća se *radijus* elementa iz *2D_segmenta* iznad i ispod promatranog. Ukoliko da se radi o *2D_segmentima* pri vrhu ili pri dnu slike ne

postoji *radijus* elementa koje je potrebno dohvatiti. U tom se slučaju elementi zajedničke memorije na odgovarajućim pozicijama postavljaju na vrijednosti prvih elementa uz rub.



Slika 24. Čitanje slikovnih elemenata iz globalne memorije

Isto kao kod konvolucije po recima, poziva se funkcija `__syncthreads` koja osigurava da se učitaju u zajedničku memoriju svi potrebni elementi prije nego se krene s izračunavanjem konvolucije. Za konvoluciju dohvaćenog *2D_segmenta* slike odgovorne su sve dretve bloka. Svaka dretva na temelju svojih x i y koordinata određuje poziciju zajedničke memorije odakle će čitati elemente i poziciju u globalnoj memoriji kamo će zapisati rezultat. Za svaki slikovni element za kojeg je odgovorna promatrana dretva obavlja se 2**radijus*+1 množenja odgovarajućih elementa zajedničke memorije sa pripadajućim elementima Gaussovog filtra. Kao što je ranije spomenuto, blokovi dretvi su manjih dimenzija od *2D_segmenata*. Iste dretve bloka izračunat će konvoluciju za sve elemente unutar *2D_segmenta* udaljene u smjeru osi y za dimenziju bloka. Odnosno, isto kao kod dohvaćanja podataka, blok dretvi računa konvoluciju za sve pod-segmente promatranog *2D_segmenta*. Opisani izvorni kod konvolucije po stupcima prikazan je niže.

```
#define IMUL(a,b) __mul24(a,b)
#define SEGMENT_W 16
#define SEGMENT_H 32

template<int i> __device__ float ConvCol(float *s_Data)
{
    return s_Data[i*SEGMENT_W]*c_Kernel[i] + ConvCol<i-1>(s_Data);
}

template<> __device__ float ConvCol<-1>(float *s_Data)
{
    return 0;
}

template<int RADIUS>
__global__ void ConvColGPU(float *d_Result, float *d_Data, int width,
                           int height, int sShift, int gShift)
{
  __shared__ float s_Data[SEGMENT_W*(RADIUS + SEGMENT_H + RADIUS)];

  const int segStart = IMUL(blockIdx.y, SEGMENT_H);
  const int segEnd = segStart + SEGMENT_H - 1;
  const int yStart = segStart - RADIUS;
  const int yEnd = segEnd + RADIUS;

  const int xStart = IMUL(blockIdx.x, SEGMENT_W) + threadIdx.x;
  const int xEnd = xStart + IMUL(height-1, width);

  if (xStart<width) {
    int sLoc = IMUL(threadIdx.y, SEGMENT_W) + threadIdx.x;
    int glLoc = IMUL(yStart + threadIdx.y, width) + xStart;

    for (int y = yStart + threadIdx.y; y <= yEnd; y += blockDim.y){

      s_Data[sLoc]=(y<0)?d_Data[xStart]:((y>=height)?d_Data[xEnd]:
      d_Data[glLoc]);

      sLoc += sShift;
      glLoc += gShift;
    }
  }
  __syncthreads();

  if (xStart<width) {
    const int borders = min(segEnd, height - 1);
```

```
    int sLoc = IMUL(threadIdx.y + RADIUS, SEGMENT_W) + threadIdx.x;
    int glLoc = IMUL(segStart + threadIdx.y , width) + xStart;

    for (int y=segStart+threadIdx.y;y<=borders;y+=blockDim.y){
      d_Result[glLoc] = ConvCol<2*RADIUS>(s_Data + sLoc - RADIUS*SEGMENT_W);
      sLoc += sShift;
      glLoc += gShift;
   }
 }
  __syncthreads();
}
```

8.2.2. Povećanje rezolucije slike za faktor 2

Povećanje rezolucije slike za faktor 2 radi se na istom principu kao i u verziji CPU. U prvom prolazu poveća se širina slike, a u drugom prolazu visina. U svakom prolazu vrijednosti susjednih elemenata se linearno interpoliraju, a dobiveni rezultat predstavlja novi slikovni element koji se dodaje između njih. U nastavku objašnjene su funkcije jezgre koje to rade.

Povećanje broja elementa širine za faktor 2

Na samom početku u funkciji `resizeUp` definiraju se dimenzije mreže dretvi i dimenzije bloka. U implementaciji korišteni su blokovi dimenzija 16x16. Znači broj dretvi u x, y smjeru jednak je broju dretvi polu-osnove. Kao što je ranije objašnjeno, dretve iste polu-osnove zajedno pristupaju globalnoj memoriji, pa su blokovi ovih dimenzija najuobičajeniji i koriste se u mnogim primjerima. Podjelom slike na blokove definira se mreža. Preciznije, broj blokova u smjeru osi x jednak je broju koji se dobije dijeljenjem širine slike sa x dimenzijom bloka. Broj blokova u smjeru y jednak je kvocijentu visine slike i dimenzije bloka. U slučaju da kvocijent nije cijeli broj uzima se prvi veći. Nakon što je definirana mreža i blok, poziva se funkcija jezgre `upSampleWidth`. Svaka dretva bloka na temelju svojih x, y koordinata unutar bloka određuje koji element slike čita iz globalne memorije. Tako se množenjem: y koordinate bloka, dimenzije bloka i širine slike dobiva

pomak na slici u smjeru osi y. Množenjem x koordinate bloka sa dimenzijom bloka nalazi se pomak u smjeru osi x. Na taj način pronalazi se dio slike za kojeg su zadužene dretve promatranog bloka. Sada dretve pomoću svojih y, x koordinata pronalaze pripadajući slikovni element unutar bloka. Svaka dretva dohvaća iz globalne memorije vrijednost svojeg elementa i vrijednost prvog susjeda u smjeru osi x. Dohvaćene vrijednosti se linearno interpoliraju, a dobiveni rezultat predstavlja novi slikovni element. Vrijednosti pripadajućeg i novog elementa spremaju se natrag u globalnu memoriju. Slika koja se sprema je dimenzija *visina x (širina*2)*, te je potrebno izračunati novu lokaciju u globalnoj memoriji. Nova lokacija računa se na identičan način kao i kod čitanja, ali sa dvostruko većom širinom slike, x koordinatom dretve i x dimenzijom bloka. Izvorni kod ove funkcije prikazan je niže.

```
#define IMUL(a,b) __mul24(a,b)

__global__ void upSampleWidth(float *d_Data,float *d_Rez, int width)
{
   const int glLoc = threadIdx.x + IMUL(blockIdx.x, blockDim.x) +
       IMUL(threadIdx.y,width) + IMUL(IMUL(blockIdx.y, blockDim.y), width);
   const int glDest = IMUL(threadIdx.x,2) + IMUL(IMUL(blockIdx.x,blockDim.x),2)
       + IMUL(IMUL(threadIdx.y,width),2) + IMUL(IMUL(blockIdx.y, blockDim.y),
       IMUL(width,2));

   d_Rez[glDest] = d_Data[glLoc];
   d_Rez[glDest+1] = 0.5*(d_Data[glLoc] + d_Data[glLoc +1]);

}
```

Povećanje broja elemenata visine za faktor 2

Nakon što je slika povećana dva puta u smjeru osi x, na istoj slici nastavlja se s povećanjem u smjeru osi y. Slika se povećava na isti način, samo se ovdje prolazi po stupcima slike. Prilikom implementacije razmatrana je metoda da se slika razlomi na dvodimenzionalne segmente, koji bi se dohvaćali u zajedničku memoriju. Sličan postupak bio je kod konvolucije po stupcima. Ovdje svaka dretva osim pripadajućeg elementa

dohvaća samo još jedan element i to onaj koji je udaljen za širinu slike. Zato se broj memorijskih transakcija za svaku dretvu poveća samo za jedan. Eksperimentiranjem je utvrđeno da se u ovom slučaju korištenjem zajedničke memorije ne postiže ubrzanje. Za dodatno čitanje jednog elementa po dretvi ne isplati se dohvaćati segmente u zajedničku memoriju. Povećanje visine slike implementirano je na isti način kao i povećanje širine. Svaka će dretva dohvaćati vrijednost njoj pripadajućeg elementa i susjednog elementa ispod. Interpolacijom se dobiva nova vrijednost koja će se pridijeliti novom elementu slike. Potrebno je odrediti i nove lokacije u globalnoj memoriji. One se računaju kao i kod povećanja širine, samo se ovdje uzimaju dvostruko veće vrijednosti svih parametara koji određuju pomak u smjeru osi y. Na kraju, dretva sprema vrijednost pripadajućeg elementa i novog elementa kojeg je upravo izračunala.

```
#define IMUL(a,b) __mul24(a,b)

__global__ void upSampleHeight(float *d_Data,float *d_Rez, int width)
{
   const int gLoc = threadIdx.x + IMUL(blockIdx.x,blockDim.x) +
       IMUL(threadIdx.y,width) + IMUL(IMUL(blockIdx.y,blockDim.y),width);
   const int yDest = threadIdx.x + IMUL(blockIdx.x, blockDim.x) +
       IMUL(IMUL(threadIdx.y,2),width) + IMUL(blockIdx.y,IMUL(IMUL
       (blockDim.y,2),width));

   d_Rez[yDest] = d_Data[gLoc];
   d_Rez[yDest+width] = 0.5*(d_Data[gLoc] + d_Data[gLoc + width]);

}
```

8.2.3. Smanjivanje rezolucije slike za faktor 2

Rezolucija slike smanjuje se na identičan način kao u verziji CPU. Odbacuje se svaki drugi element retka ili stupca. Slikovni elementi se selektiraju u blokovima 16x16. Znači da je broj blokova unutar mreže u smjeru osi x jedak *širina_slike*/16, a u smjeru y *visina_slike*/16. Koordinate promatrane dretve odgovaraju koordinatama slikovnih elemenata unutar promatranog bloka. Za svaku se dretvu provjeravaju njezine x, y koordinate. Ako su one djeljive sa dva, pronašla se ispravna dretva. Ispravna će dretva

pročitati pripadajući element iz originalne slike i spremiti ga na odgovarajuću lokaciju nove smanjene slike. Smanjenjem slike smanjuje se njezina širina, a i dimenzije bloka. Navedene parametre potrebno uzeti u obzir prilikom računanja nove lokacije u globalnoj memoriji. Isto tako se smanjuje broj slikovnih elemenata, a time i broj dretvi. Zato se za računanje nove lokacije koriste dvostruko manje koordinate dretvi.

```
#define IMUL(a,b) __mul24(a,b)

__global__ void downSample(float *d_Data,float *d_Rez, int width, int newWidth)
{
   int xs= threadIdx.x;
   int ys= threadIdx.y;

   if((xs%2==0) && (ys%2==0)){
      const int gLoc = threadIdx.x +IMUL(blockIdx.x, blockDim.x) +
       IMUL(threadIdx.y,width) + IMUL(blockIdx.y, IMUL(blockDim.y, width));
      const int gDest = threadIdx.x*0.5 +blockIdx.x* blockDim.x*0.5 +
       threadIdx.y*0.5*newWidth + blockIdx.y* blockDim.y*0.5 * newWidth;
      d_Rez[gDest] = d_Data[gLoc];
   }
}
```

8.2.4. Oduzimanje slika zaglađenih Gaussovim filtrom

Radi se o vrlo jednostavnoj implementaciji. Slika se opet dijeli u blokove 16x16. Odnosno, blokovi se sastoje od 256 dretvi. Svaka dretva temeljem svojih x, y koordinata dohvaća dva slikovna elementa iz dvije različite slike. Prvi element dohvaća se iz slike koja odgovara slici više razine mjerila, a drugi element nalazi se na istoj lokaciji u slici nižeg mjerila. Dretva od vrijednosti elementa višeg mjerila oduzima vrijednost elementa nižeg mjerila. Dobiveni se rezultat sprema na istu poziciju nove slike. Nakon što svoj posao obave sve dretve po svim blokovima, dobiva se nova slika *DoG*. Implementacija funkcije jezgre prikazana je niže.

```
#define IMUL(a,b) __mul24(a,b)

__global__ void subtract(float *d_Result, float *d_Data2, float *d_Data1,
                         int width, int height){

      const int x = IMUL(blockIdx.x, blockDim.x) + threadIdx.x;
      const int y = IMUL(blockIdx.y, blockDim.y) + threadIdx.y;
      int n = IMUL(y, width) + x;
      d_Result[n] = d_Data2[n] - d_Data1[n];

}
```

8.2.5. Računanje amplitude i kuta gradijenta

Računanje amplitude i kuta gradijenta je posljednji kritični dio algoritma SIFT koji je implementiran u okruženju CUDA. Kao i u prethodnim implementacijama radi se o blokovima dimenzija 16x16. Svaka dretva bloka dohvaća četiri susjedna elementa. Elementi se dohvaćaju iz globalne memorije i spremaju u zajedničku memoriju, koja se odlikuje vrlo niskim vremenom odziva. Zajednička memorija se koristi zbog toga što će više dretvi koristiti iste elemente. Veličina zajedničke memorije veća je od veličine bloka i iznosi 18x18. Razlog je tome što je zajednička memorija definirana na razini bloka, pa je osim elemenata pripadajućeg bloka potrebno u zajedničku memoriju pohraniti i prvi element sa svake strane bloka. Jedino će se na taj način moći izračunati vrijednost amplitude i kuta gradijenta za rubne elemente bloka. Amplituda i kut gradijenta računaju se po izrazima 1.21 i 1.22, a svaka dretva treba dohvatiti elemente: lijevo, desno, iznad i ispod promatranog elementa. Prilikom dohvaćanja elementa ispod i iznad pristupa se podacima različitih memorijskih segmenata. To rezultira sa dodatne dvije memorijske transakcije po dretvi, no još se uvijek dobiva zadovoljavajuće ubrzanje. U namjeri da se to izbjegne, razmatrana je implementacija dohvaćanja dvodimenzionalnog segmenta slike u zajedničku memoriju. Ovdje bi opet trebalo za sve rubne elemente znati sve susjede, pa i one elemente koji pripadaju susjednom bloku. Zbog toga bi se implementacija dijelila u dva prolaza. Jedan bi bio po recima slike i računanja gradijenta u smjeru osi x, a drugi bi prolaz bio po stupcima i računale bi se vrijednosti elemenata gradijenta u smjeru osi y. Rezultat bi bile dvije nove „slike“ gdje bi na poziciji svakog elementa stajale izračunate

vrijednosti gradijenta. Izračunate vrijednosti trebalo bi ponovno dohvaćati iz globalne memorije, pa je eksperimentiranjem utvrđeno je da ova implementacija ne bi dala ubrzanje. Zbog toga je odlučeno koristiti jednostavniju, prethodno objašnjenu implementaciju čiji kod je priložen u nastavku.

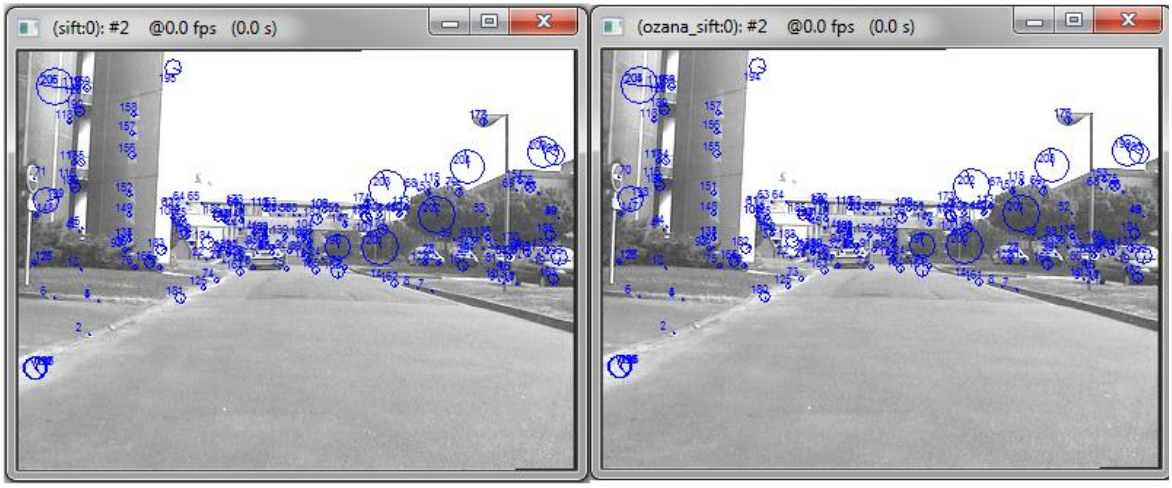
```
#define IMUL(a,b) __mul24(a,b)

__device__ float computeOri(float x){

   if(x>=0)
      return fmod(x,float(2*M_Pi));
   else
      return 2*M_Pi + fmod(x, float(2*M_Pi)) ;
}

__global__ void computeGrad(float *d_Mag, float *d_Ori, float *d_Data, int
                            width, int height ){

   __shared__ float s_Data[18][18];

   const int xx = IMUL(blockIdx.x, blockDim.x) + threadIdx.x;
   const int yy = IMUL(blockIdx.y, blockDim.y) + threadIdx.y;
   int n = IMUL(yy, width) + xx;

   int x=threadIdx.x+1;
   int y=threadIdx.y+1;

   if (xx<width-1 && xx>0 && yy<height-1 && yy>0){

      s_Data[x-1][y] = d_Data[n - 1];
      s_Data[x +1][y] = d_Data[n + 1];
      s_Data[x][y +1] = d_Data[n + width];
      s_Data[x][y - 1] = d_Data[n  - width];

       __syncthreads();

      float Gx = s_Data[x + 1][y] - s_Data[x - 1][y];
      float Gy = s_Data[x][y+1] - s_Data[x][y-1];

      d_Ori[n]= computeOri(atan2f(Gy, Gx));
      d_Mag[n]= sqrtf((Gx*Gx)+(Gy*Gy));
    }

}
```

9. Analiza rezultata algoritma SIFT za GPU

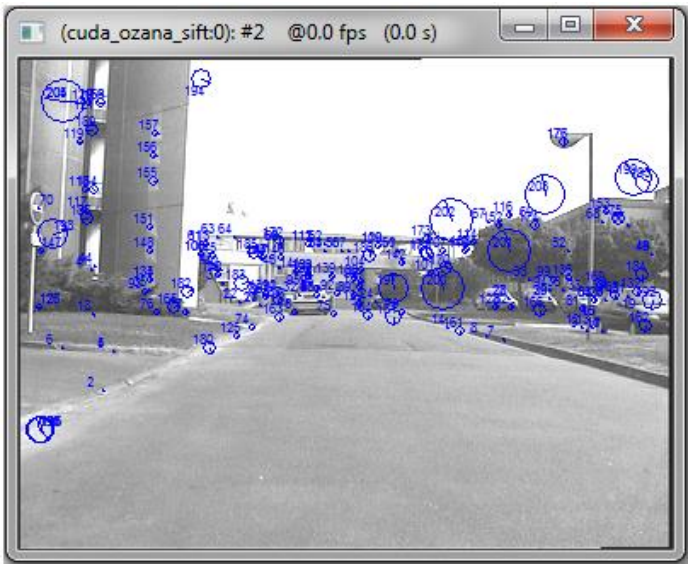
U ovom poglavlju prikazani su rezultati algoritma SIFT čiji se pojedini dijelovi izvode na grafičkom procesoru. Rezultati su uspoređeni sa prethodnim verzijama implementacija za procesor opće namjene. Provedena je analiza vremena iz koje se vidi koliko ključni dijelovi, implementirani u okruženju CUDA, pridonose ubrzanju cjelokupnog algoritma SIFT.

9.1. Prikaz rezultata

Usporedba rezultata verzije CPU, implementacije A. Vedaldija i promatrane verzije GPU prikazana je na slici 25. Kako su slike dimenzija 384x288, testiranje je provedeno za četiri oktave sa pet mjerila (*levels* je 2). U sve je tri verzije vrijednost praga za odbacivanje ekstrema niskog kontrasta postavljena na 0.03, dok je prag za odbacivanje kandidata slabo lokaliziranih uz rub jednak 10.



Slika 25. a) Rezultati verzije A. Vedaldija Slika 25.b) Rezultati verzije CPU

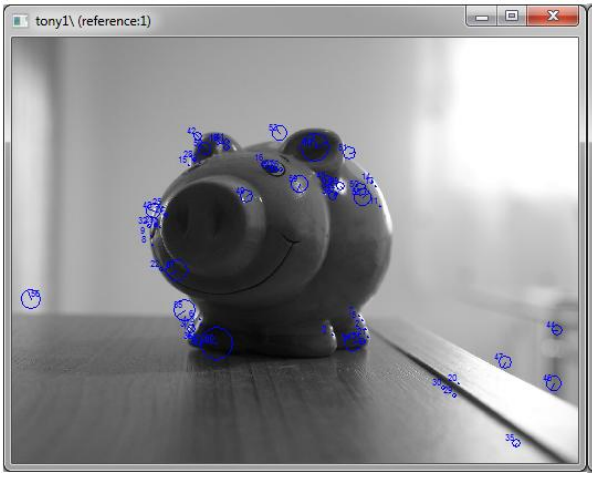


Slika 25. c) Rezultati verzije GPU

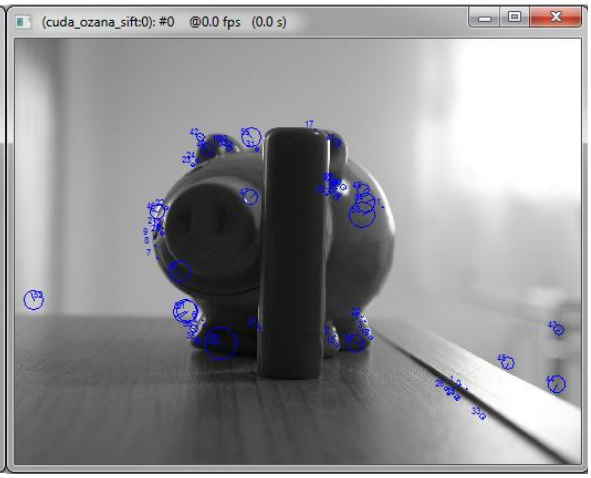
Kao što se vidi sa slike 25. a) verzija GPU pronalazi značajke na istim mjestima kao verzija CPU i implementacija A. Vedaldija. Broj detektiranih značajki u sve tri verziji jednak je 174. Vrijednost orijentacije maksimalno odstupa oko 3 stupnja. Navedena odstupanja su posljedica zaokruživanja prilikom računanja. Temeljem toga može se reći da rezultati verzije GPU odgovaraju rezultatima verzije CPU i implementacije A. Vedaldija.

Niže su prikazani primjeri koji dokazuju invarijantnost algoritma SIFT obzirom na promjene u skaliranju, rotaciji i zaklonjenosti objekata.

Na slici 26. a) broj detektiranih značajki referentne slike je 62. Bez promjene položaja kamere snimljena je iduća slika b), gdje je dodan novi objekt koji djelomično zaklanja objekt u pozadini. Testiranjem je detektirano nešto manje, točnije 57 značajki. Pri tome se većina značajki i dalje pronalazi na lokacijama koje odgovaraju točkama referentne slike, a orijentacija se kod nekih značajki razlikuje samo u par stupnjeva. Dobiveni rezultati dokazuju invarijantnost algoritma SIFT obzirom na zaklonjenost objekta.

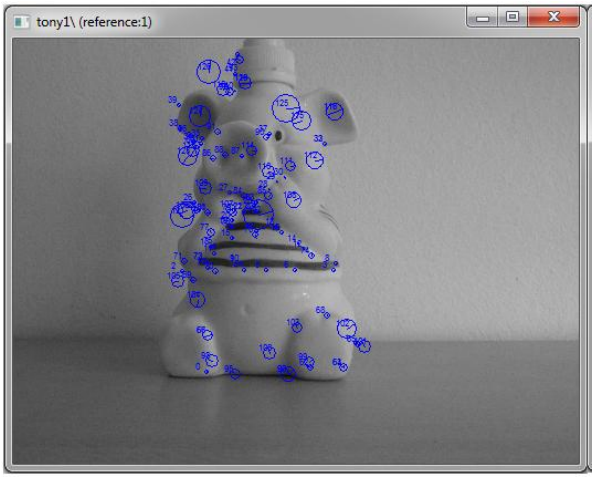


Slika 26. a) Referentna slika

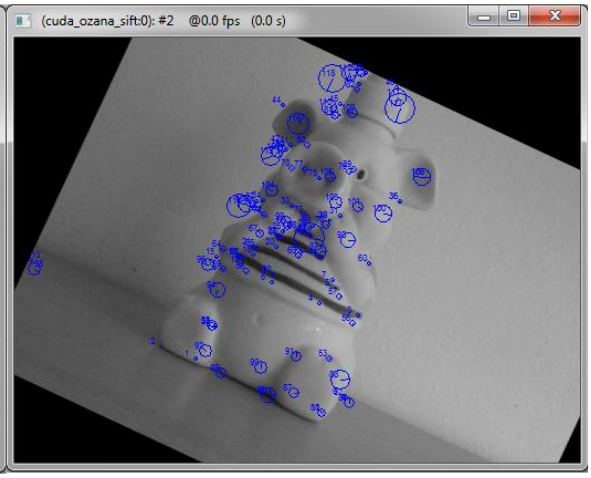


Slika 26. b) Slika zaklonjenog objekta

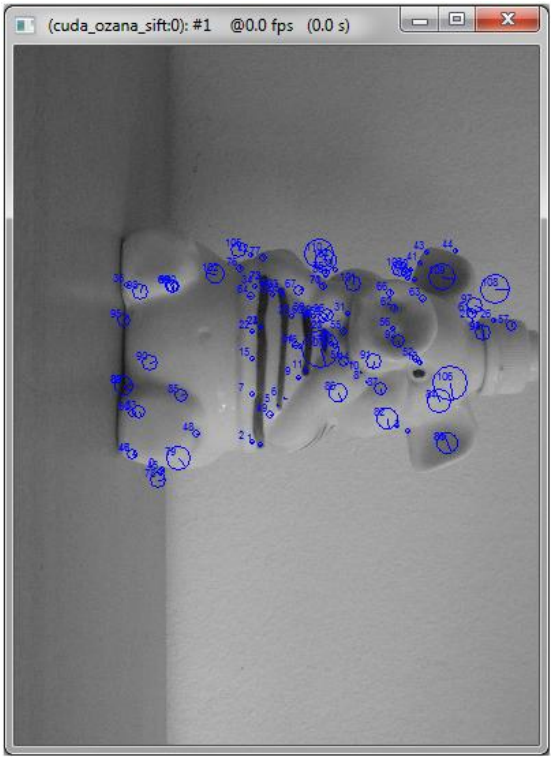
Slijedi primjer kojim se pokazuje invarijantnost algoritma SIFT obzirom na rotaciju. Na slici 27. a) prikazana je referentna slika, slika b) rezultat je rotacije cijele referentne slike za 25º, a slika c) je rezultat rotacije za 90º. Testiranje je provedeno kroz četiri oktave sa četiri mjerila. Broj značajki referentne slike je 119. Poslije rotacije za 25º detektirano je 108, a nakon rotacije za 90º 101 značajka . Većina značajki pronalazi se na istim pozicijama u sve ti slike.



Slika 27. a) Referentna slika



Slika 27. b) Slika zarotirana za 25º



Slika 27. c) Slika zarotirana za 90°

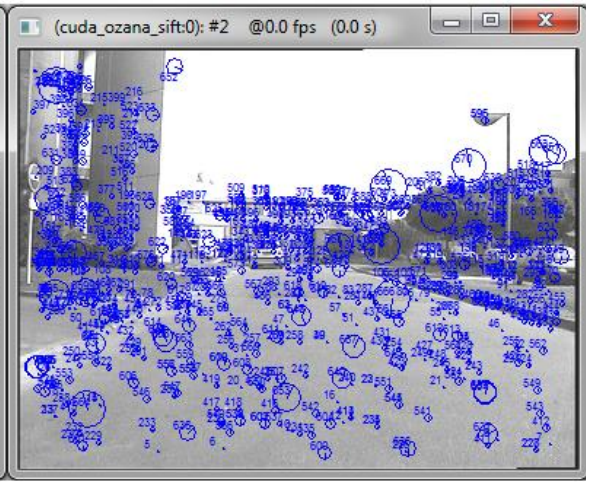
Posljednji primjer testiranja izveden je na slikama prometa (vidi sl. 28. a i b). Isto kao što je to u verziji CPU, usporedit će se prva i stota slika niza. Broj ključnih značajki referentne slike jedak je 166, dok je u približenoj slici detektirano nešto manje, točnije 162. Većina značajki referentne slike detektira se i na približenoj slici b). Pri tome razlika između vrijednosti orijentacije podudarnih značajki iznosi maksimalno 3°.

Slika 28. a) Referentna slika

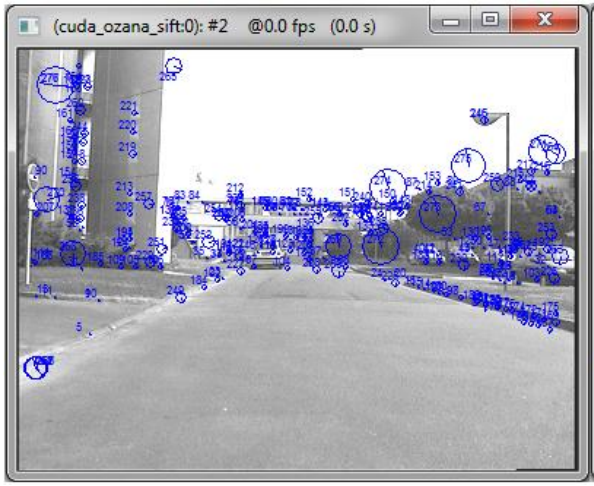
Slika 28. b) Stota slika niza

Nakon što je prikazan rad samog algoritma, potrebno je vidjeti u kojim koracima i u kolikom se broju detektiraju značajke.

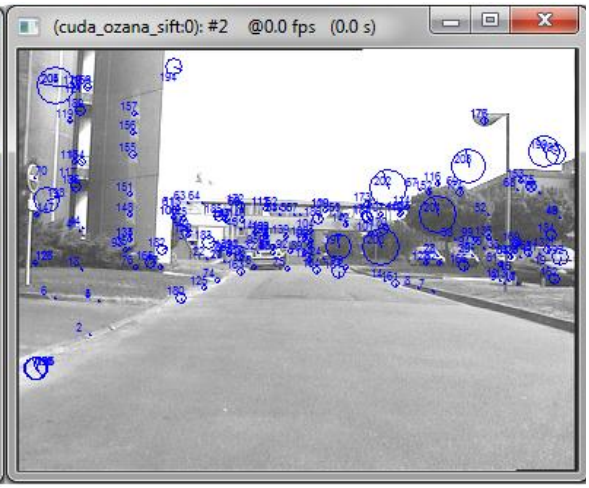
Slika 29. a) Originalna slika



Slika 29. b) Ekstremi detektirani u DoG slikama



Slika 29 c) Nakon odbacivanja kandidata minimalnog kontrasta

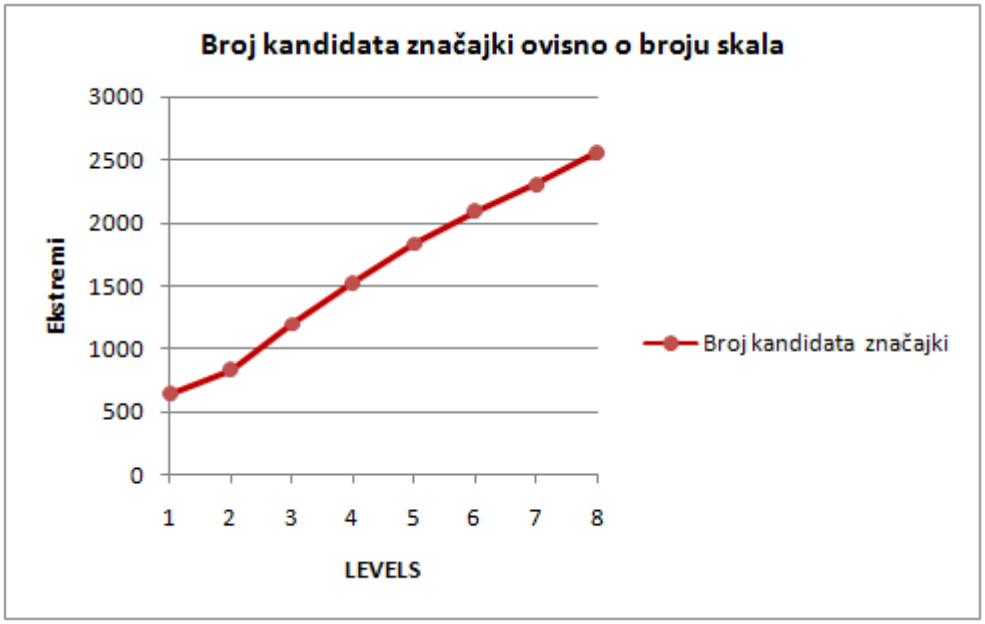


Slika 29. d) Poslije odbacivanja kandidata slabo lokaliziranih uz rub

Slika 29. a) prikazuje originalnu sivu sliku dimenzija 384x288. Rezultati slike b) odgovaraju pronađenim ekstremima *26-susjedstva* na slikama *DoG*. Broj detektiranih i prikazanih ekstrema je 610, a u nastavku biti će prikazano kako taj broj ovisi o broju razina mjerila. Dodavanjem praga za kontrast iznosa 0.3, odbacuju se svi ekstremi niskog kontrasta. Tako je broj detektiranih ekstrema na slici c) jednak 244, a odbačeno je 366 kandidata. Finalna slika odgovara slici d). Ovdje se odbacuju svi kandidati slabo lokalizirani uz rub. To se može vidjeti usporedbom slika b) i c), uz desni rub ceste. Na taj način

odbačeno je još 70 kandidata, što znači da je ukupan broj značajki detektiranih u prva tri koraka jednak 174.

Na slici 30. prikazan je graf na temelju kojeg se vidi kako broj kandidata značajki ovisi o ukupnom broju mjerila unutar oktave. Testiraju se sive slike prometa dimenzija 384x288, te je broj oktava postavljen na četiri. U smjeru osi x nalazi se broj intervala na koji se dijeli oktava, međutim ukupan broj mjerila unutar oktave jednak je *levels+3*. Kao što se vidi sa grafa povećanjem broja mjerila po oktavi detektira se veći broj kandidata značajki. Radi se o kandidatima detektiranim nakon *26-susjedstva*. D. Lowe kaže da se time nužno ne povećava stabilnost algoritma. Razlog tome je što se na taj način u prosjeku detektira veliki broj nestabilnih značajki, koje karakterizira niža ponovljivost [9]. Zato se većina tih kandidata odbacuje dodavanjem praga za kontrast i praga za slabo lokalizirane rubove.



Slika 30. Broj kandidata značajki ovisno o broju mjerila

9.2. Analiza vremena

U prethodnom odjeljku pokazano je kako verzija CPU i verzija GPU daju identične rezultate. Ključna razlika, a ujedno i cilj implementacije verzije GPU bilo je ubrzati

izvođenje algoritma SIFT. Detaljnom analizom u poglavlju 7.2 određeni su dijelovi algoritma verzije CPU koji troše najviše vremena. Implementacijom tih kritičnih dijelova u okruženju CUDA postignuto je ubrzanje, a detaljna analiza po koracima i međukoracima prikazana je u tablici 3. Kako bi se rezultati mjerenja mogli kasnije usporediti sa prethodno prikazanim rezultatima verzije CPU, uzeti su isti parametri. Detaljnije, radi se o sivim slikama prometa dimenzija 384x288. Značajke se detektiraju kroz četiri oktave sa pet mjerila. Prikazano vrijeme je prosječno vrijeme obrade jedne slike kroz pet mjerenja.

Tablica 3. Prosječno vrijeme kritičnih dijelova prilikom obrade jedne slike algoritmom SIFT

KRITIČNI DIJELOVI ALGORITMA SIFT			**VRIJEME [ms]**
1.	Izgradnja prostora mjerila		58.08
	1.1	Dohvaćanje originalne slike u memoriju CUDA	0.66
	1.2	Konvolucija Gaussovim filtrom	16.82
	1.3	Povećanje rezolucije slike za faktor 2	0.06
	1.4	Smanjivanje rezolucije slike za faktor 2	0.07
	1.5	Računanje *DoG* slika	0.21
	1.6	Dohvaćanje *DoG* slika u memoriju *hosta*	26.48
+		Ostali dio programa za CPU	13.78
2.	Dodjela orijentacije		46.35
	2.1	Računanje amplitude i kuta gradijenta	0.26
	2.2	Dohvaćanje amplitude i kuta gradijenta u memoriju hosta	16.19
+		Ostali dio programa za CPU	29.9
UKUPNO			**104.43**

Kao što se vidi iz tablice 3., obrada određenih dijelova algoritma na grafičkom procesoru izuzetno je brza. Kod izgradnje prostora mjerila relativno dosta vremena troši se na onaj dio koji se izvodi na procesoru opće namjene. Tu se misli na dio programa u

koji je ugrađena implementacija u okruženju CUDA. Ukupno vrijeme trećeg koraka, dodjele orijentacije je vrijeme koje se troši izvođenjem na procesoru opće namjene. Prvenstveno se to odnosi na onaj dio koji nije implementiran u okruženju CUDA, izgradnja histograma i izračun konačne vrijednosti orijentacije. Ubrzanje koje se dobiva je još uvijek zadovoljavajuće. U tablici 4. može se vidjeti usporedba vremena po koracima implementacije CPU i GPU. Prikazano vrijeme predstavlja prosječno vrijeme obrade jednog slikovnog okvira kroz pet mjerenja. Verzija GPU prikazuje ukupno vrijeme potrebno za obradu pojedinog koraka, koje uključuje izvođenja na procesoru opće namjene i grafičkom procesoru.

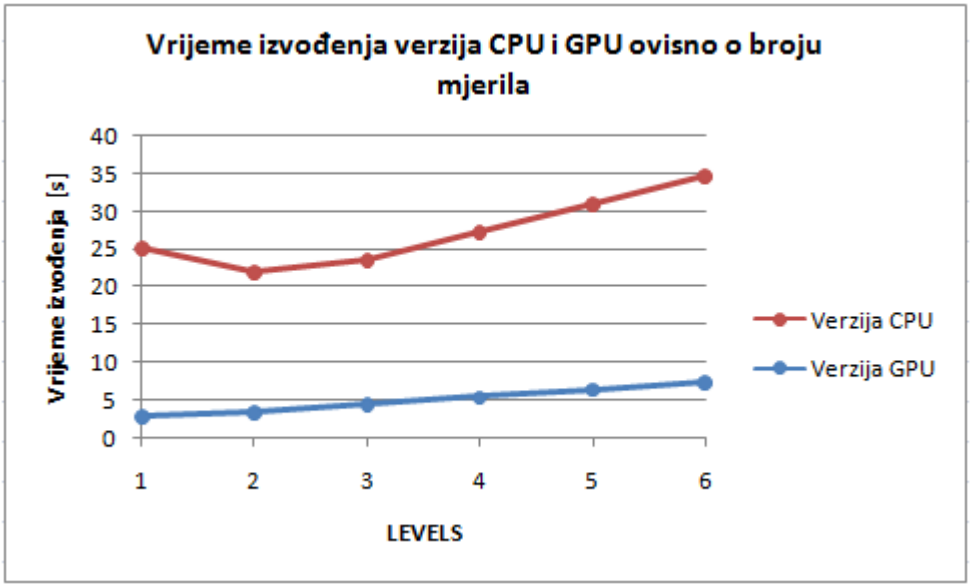
Tablica 4. Usporedba vremena izvođenja verzije CPU i GPU

KORACI ALGORITMA		VERZIJA CPU [ms]	VERZIJA GPU [ms]
1.	Izgradnja prostora mjerila	641.68	58.08
2.	Detekcija i lokalizacija ključnih točaka	40.16	
3.	Dodjela orijentacije	176.43	46.35
UKUPNO		**858.27**	**144.59**

Kao što se vidi iz tablice 4., drugi korak algoritma SIFT nije implementiran u okruženju CUDA. Prilikom analize vremena verzije CPU utvrđeno da on oduzima najmanje vremena, zato se nije krenulo s njegovom izgradnjom za grafički procesor.

Dosad je uspoređivano vrijeme verzija CPU i GPU po koracima i međukoracima algoritma SIFT. Analiza ukupnog ubrzanja verzije GPU dana je u nastavku.

Na slici 31. prikazana je usporedba ukupnog vremena izvođenja niza od deset slikovnih okvira ovisno o broju mjerila (*levels*+3). Za testiranje se koriste slike dimenzija 384x288, a broj oktava je stalan i iznosi četiri.

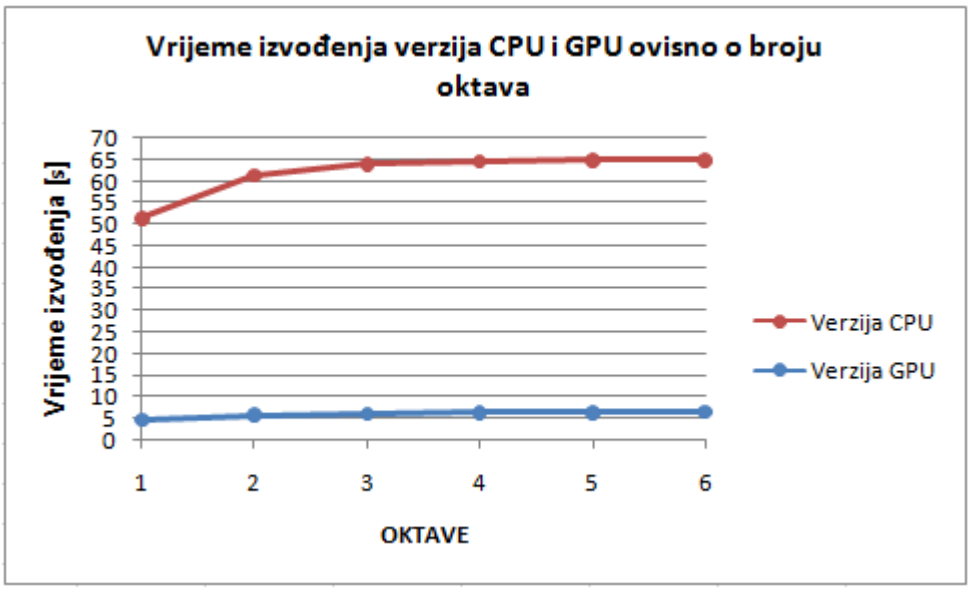


Slika 31. Usporedba verzije CPU i GPU temeljem broja mjerila za 10 slikovnih okvira

Temeljem grafa vidi se da je velika razlika između vremena izvođenja kada je ukupan broj mjerila jedak četiri (*levels* je 1). Razlog je tome što se u tom slučaju generira Gaussov filtar sa najvećim radijusom koji iznosi 45. To oduzima puno vremena prilikom izvođenja verzije CPU. Verzija GPU ne daje veliko ubrzanje povećanjem broja mjerila. Kao što je objašnjeno u prethodnom odjeljku, povećanjem broja mjerila povećava se broj slika *DoG* i detektira se daleko veći broj ekstrema, odnosno kandidata značajki. Selekcija kandidata obavlja se u drugom koraku algoritma SIFT, koji nije implementiran u okruženju CUDA. Da bi se nastavilo izvođenje na procesoru opće namjene potrebno je slike *DoG* kopirati iz memorije CUDA u memoriju *hosta*. Na kopiranje slika troši se prilično vremena. Isto tako veći broj ekstrema zahtijevat će više rada u drugom koraku, a time i više vremena. Ovdje dolazi do gubitka vremena verzije GPU. Kao budući rad savjetuje se implementacija drugog koraka, tj. detekcija i lokalizacija značajki u okruženju CUDA.

Osim po broju mjerila napravljena je usporedba verzije CPU i GPU temeljem broja oktava. Kako bi se testiranje izvelo kroz šest oktava koriste se slike dimenzija 512x512. Broj mjerila po oktavi je konstantan i iznosi četiri. Na grafu niže prikazano je prosječno vrijeme potrebno za obradu deset slikovnih okvira kroz pet mjerenja. Na taj način dobiju se dvostruko bolji rezultati, gdje je verzija GPU u prosjeku oko 9.4 puta brža od verzije CPU. Povećanjem broja oktava povećava se i broj slika *DoG* koje je potrebo kopirati u

memoriju *hosta*. Time se još jednom ukazuje kako bi implementacija drugog koraka algoritma SIFT u okruženju CUDA uvelike pridonijela ubrzanju.



Slika 32. Usporedba verzije CPU i GPU temeljem broja oktava za 10 slikovnih okvira

10. Zaključak

Kroz ovaj rad prikazano je traženje i izlučivanje značajki koje služe za ustanovljavanje korespondencije između slika. Iz širokog skupa metoda odabran je algoritam SIFT koji pronalazi ključne značajke invarijantne obzirom na promjene rotacije, svjetline, šuma, mjerila i djelomično na afine transformacije. Razmotrena je teorijska osnova algoritma SIFT i implementirana su prva tri koraka u programskom jeziku C++. Dobiveni rezultati su značajke sa pripadajućom lokacijom, mjerilom i orijentacijom. Profiliranjem izvođenja ove implementacije ustanovljeni su kritični dijelovi koji troše najviše vremena prilikom izvođenja na procesoru opće namjene.

Ubrzanje kritičnih dijelova postiglo se njihovom implementacijom u okruženju CUDA, čime je omogućeno izvođenje na grafičkom procesoru. Tako je u okruženju CUDA implementiran prvi korak, izgradnja prostora mjerila, te računanje amplitude i kuta gradijenta što čini sastavni dio računanja orijentacije značajke. Dobivene značajke u potpunosti odgovaraju značajkama iz osnovne implementacije (C++). Analizom ubrzanja utvrđeno je kako ono uvelike ovisi o broju oktava i mjerila prilikom izgradnje prostora mjerila. Razlog je što se povećavanjem broja mjerila i oktava povećava broj slika *DoG*, a time i broj detektiranih kandidata značajki. Povećavanjem broja slika *DoG* više vremena odlazi na njihovo dohvaćanje u memoriju *hosta*. Na taj se način povećava vrijeme izvođenja drugog koraka algoritma SIFT. U budućem se radu savjetuje implementirati drugi korak, detekcija i lokalizacija značajki u okruženju CUDA. Preporuča se implementirati posljednji korak algoritma SIFT, izgraditi deskriptor i pridružiti ga svakoj pojedinoj značajki.

11. Literatura

[1] Bakamović J. Određivanje korespondentnih točakana slikamadobivenih stereo kamerom, Diplomski rad, Fakultet elektrotehnike i računarstva, 2010

[2] Ballard B. i Chris M. Difference of Gaussian ScaleSpace Pyramids for SIFT Feature Detection, *Final Project Report*, 3(2007)

[3] Dostal, D. Primjene opisnika značajki u računalnom vidu, Seminar, Fakultet elektrotehnike i računarstva, 2010

[4] Hess R. SIFT Feature Detector, 2010., *A C implementation of a SIFT image feature* detector, http://web.engr.oregonstate.edu/~hess/ , 20.3.2011.

[5] Kirk, D. i Hwu, W. CUDA Programming Model, 2006-2008*., Chapter 2: CUDA Programming Model*, http://sites.google.com/site/cudaiap2009/materials-1/cuda-textbook, 15.04.2011.

[6] Kirk, D. i Hwu, W. CUDA Threading Model, 2006-2008., *Chapter 3: CUDA CUDA Threading Model*, http://sites.google.com/site/cudaiap2009/materials-1/cuda-textbook, 16.04.2011.

[7] Kirk, D. i Hwu, W. CUDA Memory Model, 2006-2008*., Chapter 4: CUDA Memory Model*,http://sites.google.com/site/cudaiap2009/materials-1/cuda-textbook, 17.04.2011.

[8] Lindeberg, T. Scale-space theory: A basic tool for analysing structures at different scales. Journal of Applied Statistics. 2 (1994), 225-270

[9] Lowe, D.G. Distinctive Image Features from Scale-Invariant Keypoints. International Journal of Computer Vision. 60 (2004), 91–110

[10] Lowe, D.G. Demo Software: SIFT Keypoint Detector, 2005, http://www.cs.ubc.ca/~lowe/keypoints/, 4.04.2011.

[11] Lukovac, B. Sazrijevanje računalnog vida: Automatsko pronalaženje korespondencija, Seminar, Fakultet elektrotehnike i računarstva, 2008

[12] NVIDIA Corporation, Best Practices Guide, 19.05.2010., *NVIDIA CUDA C Best Practices Guide Version 3.1,*http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_BestPracticesGuide_3.1.pdf, 13.04.2011.

[13] NVIDIA Corporation, Programming Guide, 22.10.2010., NVIDIA CUDA Programming Guide Version 3.2, http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf, 15.04.2011.

[14] Podlozhnyuk, V. Image Convolution with CUDA, 23.06.2007., *Image Convolution with CUDA,*http://www.naic.edu/~phil/hardware/nvidia/doc/src/convolutionSeparable/doc/convolutionSeparable.pdf, 05.04.2011.

[15] Sangyoon, S.L. CUDA Convolution, 08.10.2008, *GPU Programming CUDA Convolution*, http://www.evl.uic.edu/sjames/cs525/final.html, 7.04.2011.

[16] Šegvić, S. i Kalafatić, Z. Analiza kretanja u slikovnoj ravnini, predavanja iz predmeta Dinamička analiza 3D scena, Fakultet elektrotehnike i računarstva, 2011.

[17] Trbojević, A. Izvedba algoritama računalnog vida na grafičkim procesorima, Završni rad, Fakultet elektrotehnike i računarstva, 2010.

[18] Tuytelaars, T. i Mikolajczyk, K. Local Invariant Feature Detectors: A Survey. Foundations and Trends in Computer Graphics and Vision, 3 (2007), 177-280

[19] Utkarsh, S. SIFT: Scale Invariant Feature Transform, 2010, 2010/05/sift-scale-invariant-feature-transform/, 04.03.2011

[20] Vedaldi, A. An open implementation oft he SIFT detector and descriptor. Technical Report *070012,* UCLA CSD (2007)

[21] Vedaldi A. SIFT Tutorial, VLFeat: An open source computer vision in C, 2005, http://www.vlfeat.org/overview/sift.html, 1.03.2011.

[22] Wu, C. SiftGPU: A GPU Implementation of Scale Invariant Feature Transform, 12.10.2010., http://www.cs.unc.edu/~ccwu/siftgpu/, 25.03.2011.

[24] Živković O. Programska okruženja za programiranje na grafičkim procesorima, Seminar, Fakultet elektrotehnike i računarstva, 2010

12. Sažetak/Abstract

Naslov

Izlučivanje istaknutih točaka slike u ekstremima konvolucijskog odziva razlike Gaussovih funkcija

Sažetak

U ovom radu razmatra se ustanovljavanje korespondencije među parovima slika. Kao pristup rješavanju problema odabran je algoritam SIFT koji slovi kao jedna od najstabilnijih metoda u tom području. Prikazana je teorijska strana algoritma, a programska realizacija u jeziku C++ obuhvaća prva tri koraka algoritma SIFT. Kritični dijelovi algoritma SIFT implementirani su u okruženju CUDA, te je na taj način omogućeno njihovo izvođenje na grafičkom procesoru. Dobiveni su rezultati koji u potpunosti odgovaraju prvoj verziji implementacije (C++). Analizom brzine izvođenja utvrđeno je da je implementacija u okruženju CUDA od pet do deset puta brža. Definirana su moguća poboljšanja koja bi znatno ubrzala izvođenje algoritma SIFT.

Ključne riječi

značajke, detekcija značajki, SIFT, CUDA, GPU

Title

Extraction of image keypoints as scale-space extreme of the DoG response

Abstract

This work deals with finding similarities between image pairs. SIFT algorithm has been chosen as the best candidate for solving the problem. It is known to be the most stable method for this problem. The work first deals with theoretical part of the algorithm. A C++ implementation, which consists of it's first three steps, was developed. Critical parts of the SIFT algorithm have been implemented in CUDA framework which enables their processing on a graphical processing unit. The results obtained by testing and analysis are identical to the first (C++) implementation. Further speed analysis showed CUDA implementation to be five to ten times faster than the standard (C++) implementation. Additional upgrades, which could increase the performance even further, have also been described.

Keywords

Keyponts, detection of keypoints, SIFT, CUDA, GPU