УНИВЕРЗИТЕТ
У НОВОМ САДУ

ФАКУЛТЕТ
ТЕХНИЧКИХ НАУКА

Трг Доситеја Обрадовића 6, 21000 Нови Сад, Југославија
Деканат: 021 350-413; 021 450-810; Централа: 021 350-122
Рачуноводство: 021 58-220; Студентска служба: 021 350-763
Телефакс: 021 58-133; e-mail: ftndean@uns.ns.ac.yu



# PROJEKAT

# Iz Projektovanja elektronskih uređaja na sistemskom nivou

## TEMA PROJEKTA:

System C model sistema za ekstrahovanje karakterističnih tačaka na fotografiji pomoću SURF algoritma

## TEKST ZADATKA:

Napisati model u system c-u koji predstavlja kompletan sistem namenjen za akceleraciju k SURF algoritma. Definisati koji deo će biti realizovan u softveru, a koji u hardveru, kao i način na koji će se to implementirati.

Mentor:
Nebojša Pilipović

Studenti:
Ristić Dejana
Vig Aleksandar
Žarković Nemanja

U Novom Sadu, 10.05.2024.

# Sadržaj:

# 1. Teorijska analiza

SURF metoda (Speeded Up Robust Features) je brz i robustan algoritam za lokalno, invarijantno predstavljanje sličnosti i poređenje slika. Slično mnogim drugim pristupima zasnovanim na lokalnim deskriptorima, interesne tačke date slike se definišu kao istaknute karakteristike iz prikaza nepromenljive razmere. Ovakva analiza višestrukih razmera je obezbeđena konvolucijom početne slike sa diskretnim jezgrima na nekoliko skala (boks filteri). Drugi korak se sastoji u izgradnji deskriptora invarijantne orijentacije, korišćenjem statistike lokalnog gradijenta (intenzitet i orijentacija). Glavni interes SURF pristupa leži u njegovom brzom izračunavanju operatora koji koriste filtere prozora, čime se omogućavaju aplikacije u realnom vremenu kao što su praćenje i prepoznavanje objekata.

Kao i mnogi pristupi obradi slika, popularna i efikasna metodologija je izdvajanje i upoređivanje lokalnih zakrpa iz različitih slika. Međutim, da bi se dizajnirali brzi algoritmi i dobili kompaktni i lokalno nepromenljivi prikazi, potrebni su neki kriterijumi selekcije i procedure normalizacije. Oskudan prikaz slike je takođe neophodan da bi se izbegla opsežna poređenja u smislu zakrpa koje bi bile skupe u računskom smislu. Glavni izazovi su stoga zadržati najistaknutije karakteristike sa slika (kao što su uglovi, mrlje ili ivice), a zatim izgraditi lokalni opis ovih karakteristika koji je nepromenljiv na bučna merenja, fotometrijske promene ili geometrijsku transformaciju. Problemi se rešavaju od ranih godina kompjuterskog vida, što je rezultiralo veoma bogatom literaturom. Invarijantni lokalni opis slike iz analize na više nivoa je novija tema: SIFT deskriptori – od kojih je SURF u velikoj meri inspirisan – su invarijantni deskriptori slike koji su takođe otporni na šum i fotometrijske promene. Neki algoritmi proširuju ovaj prozor na potpunu invarijantnost transformacije i gustu reprezentaciju. Glavni interes SURF pristupa koji se proučava u ovom radu je njegova brza aproksimacija SIFT metode. Pokazalo se da dele istu robusnost i invarijantnost dok je SURF brži za računanje.

SURF algoritam je sam po sebi zasnovan na dva uzastopna koraka (detekcija karakteristika i opis). Slično mnogim drugim pristupima, kao što je SIFT metoda, detekcija karakteristika u SURF-u se oslanja na skalno-prostornu reprezentaciju, kombinovanu sa diferencijalnim operatorima prvog i drugog reda. Originalnost SURF algoritma je u tome što se ove operacije ubrzavaju upotrebom tehnika prozor (box) filtera.

Pristup za detekciju interesnih tačaka koristi veoma osnovnu aproksimaciju Hesijanove matrice. SURF koristi Hesijanovu matricu zbog njenih dobrih performansi u vremenu računanja i tačnosti. Umesto da koristi drugačiju meru za izbor lokacije i razmere (Hesijan-Laplasov detektor), SURF se oslanja na determinantu Hesijanove matrice za oba. Ako imamo piksel, njegov Hesijan je nešto poput:

$$H\big(f(x,y)\big) = \begin{bmatrix} \dfrac{\partial^2 f}{\partial x^2} & \dfrac{\partial^2 f}{\partial x \partial y} \\ \dfrac{\partial^2 f}{\partial x \partial y} & \dfrac{\partial^2 f}{\partial y^2} \end{bmatrix}$$

Za prilagođavanje bilo kojoj skali, filtrira se slika pomoću Gausovog kernela, tako da je data tačka X = (x,y), Hesijanova matrica X(x, σ) u x na skali σ je definisana kao:

$$\mathcal{H}(x,\sigma) = \begin{bmatrix} L_{xx}(x,\sigma) & L_{xy}(x,\sigma) \\ L_{xy}(x,\sigma) & L_{yy}(x,\sigma) \end{bmatrix}$$

gde je Lxx(x, σ) konvolucija Gausovog izvoda drugog reda sa slikom I u tački x, i slično za Lxy(x, σ) i Lyy(x, σ). Gausovi izvodi su optimalni za analizu skale-prostora, ali u praksi moraju biti diskretizovani i isečeni. Ovo dovodi do gubitka ponovljivosti pri rotaciji slike oko neparnih množioca pi/4. Ova slabost važi za detektore zasnovane na Hesijanu uopšte. Ipak, detektori i dalje dobro rade, a blagi pad performansi ne nadmašuje prednost brzih konvolucija koje donosi diskretizacija i izrezivanje.

Kreiranje SURF deskriptora odvija se u dva koraka. Prvi korak se sastoji od fiksiranja ponovljive orijentacije na osnovu informacija iz kružnog regiona oko ključne tačke. Zatim konstruišemo kvadratni region poravnat sa izabranom orijentacijom i iz njega izdvajamo deskriptor SURF.

Cilj deskriptora je da obezbedi jedinstven i robustan opis karakteristike slike, na primer, opisivanjem distribucije intenziteta piksela u okolini tačke interesovanja. Većina deskriptora se stoga izračunava na lokalni način, pa se dobija opis za svaku tačku od interesa koja je prethodno identifikovana.

## 2. Specifikacija

SURF algoritam napisan je u c++ programskom jeziku uz pomoć OpenCV biblioteke i u ovom poglavlju će biti objašnjene funkcije koje su relevantne za ubrzanje samog algoritma u hardveru. Fotografija je ograničena na rezoluciju 128x128.

### *1.int sc_main*

```
int sc_main (int argc, char **argv)
{
    int samplingStep = 2; // Initial sampling step (default 2)
    int octaves = 4; // Number of analysed octaves (default 4)
    double thres = 4.0; // Blob response treshold
    bool doubleImageSize = false; // Set this flag "true" to double the image size
    int initLobe = 3; // Initial lobe size, default 3 and 5 (with double image size)
    int indexSize = 4; // Spatial size of the descriptor window (default 4)
    struct timezone tz; struct timeval tim1, tim2; // Variables for the timing measure

    // Read the arguments
    ImLoad ImageLoader;
    int arg = 0;
    string fn = "../data/out.surf";
    Image *im=NULL;
    while (++arg < argc) {
        if (! strcmp(argv[arg], "-i"))
            im = ImageLoader.readImage(argv[++arg]);
        if (! strcmp(argv[arg], "-o"))
            fn = argv[++arg];
    }

    // Start measuring the time
    gettimeofday(&tim1, &tz);

    // Create the integral image
    Image iimage(im, doubleImageSize);

    //inicijalizacija
    //initializeGlobals(image, false, 4);

    // Start finding the SURF points
    cout << "Finding SURFs...\n";

    // These are the interest points
    vector< Ipoint > ipts;
    ipts.reserve(300);
```

```
    // Extract interest points with Fast-Hessian
    FastHessian fh(&iimage, /* pointer to integral image */
                   ipts,
                   thres, /* blob response threshold */
                   doubleImageSize, /* double image size flag */
                   initLobe * 3 /* 3 times lobe size equals the mask size */,
                   samplingStep, /* subsample the blob response map */
                   octaves /* number of octaves to be analysed */);


    fh.getInterestPoints();

    // Initialise the SURF descriptor
    initializeGlobals(&iimage, doubleImageSize, indexSize);
    // Get the length of the descriptor vector resulting from the parameters
    vLength = getVectLength();

    // Compute the orientation and the descriptor for every interest point
    for (unsigned n=0; n<ipts.size(); n++){
        setIpoint(&ipts[n]); // set the current interest point
        assignOrientation(); // assign reproducible orientation
        makeDescriptor(); // make the SURF descriptor
    }
    // stop measuring the time, we're all done
    gettimeofday(&tim2, &tz);

    // save the interest points in the output file
    saveIpoints(fn, ipts);

    // print some nice information on the command prompt
    cout << "Detection time: " <<
        (double)tim2.tv_sec + ((double)tim2.tv_usec)*1e-6 -
        (double)tim1.tv_sec - ((double)tim1.tv_usec)*1e-6 << endl;

    delete im;

    return 0;
}
```

Slika 2.1a i 2.1b funkcija sc_main

Funkcija sc_main je glavna funkcija programa koja pokreće proces detekcije i opisivanja karakterističnih tačaka na ulaznoj slici. Prvo se učita ulazna slika pa se zatim kreira integralna slika na osnovu učitane. Karakteristične tačke na slici se detektuju korišćenjem Fast-Hessian metode, i nakon detekcije, svakoj od njih se dodeljuje orijentacija. Za svaku tačku se generišu deskriptori i na kraju se detektovane tačke sa njihovim deskriptorima čuvaju u izlaznom fajlu.

### *2.void saveIpoints*

```
// Save the interest points to a regular ASCII file
void saveIpoints(string sFileName, const vector< Ipoint >& ipts)
{
    ofstream ipfile(sFileName.c_str());
    if( !ipfile ) {
        cerr << "ERROR in loadIpoints(): "
             << "Couldn't open file '" << sFileName << "'!" << endl;
        return;
    }


    double sc;
    unsigned count = ipts.size();

    // Write the file header
    ipfile << VLength + 1 << endl << count << endl;

    for (unsigned n=0; n<ipts.size(); n++){
        // circular regions with diameter 5 x scale
        sc = 2.5 * ipts[n].scale; sc*=sc;
        ipfile  << ipts[n].x /* x-location of the interest point */
                << " " << ipts[n].y /* y-location of the interest point */
                << " " << 1.0/sc /* 1/r^2 */
                << " " << 0.0     //(*ipts)[n]->strength /* 0.0 */
                << " " << 1.0/sc; /* 1/r^2 */

        // Here should come the sign of the Laplacian. This is still an open issue
        // that will be fixed in the next version. For the matching, just ignore it
        // at the moment.
        ipfile << " " << 0.0; //(*ipts)[n]->laplace;

        // Here comes the descriptor
        for (int i = 0; i < VLength; i++) {
            ipfile << " " << ipts[n].ivec[i];
        }
        ipfile << endl;
    }

    // Write message to terminal.
    cout << count << " interest points found" << endl;
}
```

Slika 2.2 funkcija saveIpoints

Ova funkcija čuva detektovane tačke i njihove deskriptore u izlazni fajl. Svaka tačka se čuva sa svojim koordinatama, skalom i vektorom deskriptora.

### *3.void initializeGlobals*

```
//Inicijalizacija globalnih promenljivih
void initializeGlobals(IImage *im, bool dbl = false, int insi = 4) {
    _iimage = im;
    _doubleImage = dbl;
    _IndexSize = insi;
    _MagFactor = 12 / insi; // Pretpostavka na osnovu prvobitne logike
    _OriSize = 4; // Pretpostavljena vrednost
    _VecLength = _IndexSize * _IndexSize * _OriSize; // Izračunavanje na osnovu datih vrednosti
    _width = im->getWidth();
    _height = im->getHeight();
    // Inicijalizacija _Pixels, _lookup1, _lookup2...
    createLookups(); // Popunjava _lookup1 i _lookup2 tabele


    double** tempPixels = _iimage->getPixels();
    _Pixels.resize(_height);
    for (int i = 0; i < _height; ++i) {
        _Pixels[i].resize(_width);
        for (int j = 0; j < _width; ++j) {
            _Pixels[i][j] = static_cast<num_f>(tempPixels[i][j]);
        }
    }

    /*for (int i = 0; i < _height; ++i) {
        for (int j = 0; j < _width; ++j) {
            cout << "Pixels[" << i << "][" << j << "]" << " = " << _Pixels[i][j] << endl;
        }
    }*/

    // allocate _index
    _index.resize(_IndexSize, std::vector<std::vector<num_f>>(_IndexSize, std::vector<num_f>(_OriSize, 0.0f)));

    // initial sine and cosine
    _sine = 0.0;
    _cose = 1.0;
}

int getVectLength() {
    return _VecLength;
}

void setIpoint(Ipoint* ipt) {
    _current = ipt;
}
```

Slika 2.3 funkcija initializeGlobals

Ova funkcija inicijalizuje globalne promenljive i priprema okruženje za rad algoritma. Konkretno, ona postavlja početne vrednosti za promenljive kao što su dimenzije _IndexSize, _MagFactor i _OriSize, kao i pravi lookup tabele _lookup1 i _lookup2 za kasniju upotrebu.

### *4.void assignOrientation*

```
void assignOrientation() {
    scale = (1.0+_doubleImage) * _current->scale;
    x = (int)((1.0+_doubleImage) * _current->x + 0.5);
    y = (int)((1.0+_doubleImage) * _current->y + 0.5);

    int pixSi = (int)(2*scale + 1.6);
    const int pixSi_2 = (int)(scale + 0.8);
    double weight;
    const int radius=9;
    double dx=0, dy=0, magnitude, angle, distsq;
    const double radiussq = 81.5;
    int y1, x1;
    int yy, xx;

    vector< pair< double, double > > values;
    for (yy = y - pixSi_2*radius, y1= -radius; y1 <= radius; y1++, yy+=pixSi_2){
        for (xx = x - pixSi_2*radius, x1 = -radius; x1 <= radius; x1++, xx+=pixSi_2) {
            // Do not use last row or column, which are not valid
            if (yy + pixSi + 2 < _height && xx + pixSi + 2 < _width && yy - pixSi > -1 && xx - pixSi > -1) {
                distsq = (y1 * y1 + x1 * x1);

                if (distsq < radiussq) {
                    weight = _lookup1[(int)distsq];
                    dx = get_wavelet2(_iimage->getPixels(), xx, yy, pixSi);
                    dy = get_wavelet1(_iimage->getPixels(), xx, yy, pixSi);

                    magnitude = sqrt(dx * dx + dy * dy);
                    if (magnitude > 0.0){
                        angle = atan2(dy, dx);
                        values.push_back( make_pair( angle, weight*magnitude ) );
                    }
                }
            }
        }
    }

    double best_angle = 0;
```

```
    if (values.size()) {
        sort( values.begin(), values.end() );
        int N = values.size();

        float d2Pi = 2.0*M_PI;

        for( int i = 0; i < N; i++ ) {
            values.push_back( values[i] );
            values.back().first += d2Pi;
        }

        double part_sum = values[0].second;
        double best_sum = 0;
        double part_angle_sum = values[0].first * values[0].second;

        for( int i = 0, j = 0; i < N && j<2*N; ) {
            if( values[j].first - values[i].first < window ) {
                if( part_sum > best_sum ) {
                    best_angle  = part_angle_sum / part_sum;
                    best_sum = part_sum;
                }
                j++;
                part_sum += values[j].second;
                part_angle_sum += values[j].second * values[j].first;
            }
        else {
            part_sum -= values[i].second;
            part_angle_sum -= values[i].second * values[i].first;
            i++;
        }
        }
    }
    _current->ori = best_angle;
}
```

Slika 2.4a i 2.4b funkcija assignOrientation

Ova funkcija je odgovorna za dodeljivanje orijentacije svakoj detektovanoj tački na slici. Orijentacija tačke je ključna informacija za kasnije generisanje deskriptora, jer omogućava da se tačke pravilno usmere u odnosu na okolinu.

Funkcija koristi okolinu detektovane tačke da bi odredila dominantnu orijentaciju, a ta okolina se uzima kao kvadratna oblast oko tačke, čiji je centar sama tačka. Orijentacija se određuje analizom gradijenta u okolini tačke. Koristi se mera nagiba gradijenta kako bi se odredio pravac najvećeg nagiba, što predstavlja dominantnu orijentaciju tačke.

Orijentacija se izražava u radijanima, gde je 0 orijentacija ka x osi, a vrednost od pi/2 orijentacija ka y osi.

Tačno određivanje orijentacije je ključno za kasnije generisanje deskriptora, jer deskriptori moraju biti usmereni u odnosu na orijentaciju tačke radi invarijantnosti na rotaciju slike.

Precizna orijentacija omogućava bolje usklađivanje tačaka između različitih slika, što je bitno za aplikacije kao što su pretraživanje slika i mapiranje objekata.

### *5.makeDescriptor*

```
void makeDescriptor() {
    _current->allocIvec(_VecLength);

    // Initialize _index array
    for (int i = 0; i < _IndexSize; i++) {
        for (int j = 0; j < _IndexSize; j++) {
            for (int k = 0; k < _OriSize; k++) {
                _index[i][j][k] = 0.0;
            }
        }
    }

    // calculate _sine and co_sine once
    _sine = sin(_current->ori);
    _cose = cos(_current->ori);

    // Produce _upright sample vector
    createVector(1.65*(1+_doubleImage)*_current->scale,
                 (1+_doubleImage)*_current->y,
                 (1+_doubleImage)*_current->x);

    int v = 0;
    for (int i = 0; i < _IndexSize; i++){
        for (int j = 0; j < _IndexSize; j++){
            for (int k = 0; k < _OriSize; k++)
            _current->ivec[v++] = _index[i][j][k];
        }
    }

    normalise();
}
```

Slika 2.5 funkcija makeDescriptor

Ova funkcija generiše deskriptore za svaku detektovanu tačku na slici. Deskriptori opisuju okolinu svake tačke i koriste se za identifikaciju i upoređivanje tačaka između različitih slika. Nakon generisanje deskriptori se normalizuju radi invarijantnosti na osvetljenje.

### *6.void createVector*

```
void createVector(double scale, double row, double col) {
    int i, j, iradius, iy, ix;
    double spacing, radius, rpos, cpos, rx, cx;
    int step = MAX((int)(scale/2 + 0.5),1);

    iy = (int) (row + 0.5);
    ix = (int) (col + 0.5);

    double fracy = row-iy;
    double fracx = col-ix;
    double fracr =   _cose * fracy + _sine * fracx;
    double fracc = - _sine * fracy + _cose * fracx;

    // The spacing of _index samples in terms of pixels at this scale
    spacing = scale * _MagFactor;

    // Radius of _index sample region must extend to diagonal corner of
    // _index patch plus half sample for interpolation.
    radius = 1.4 * spacing * (_IndexSize + 1) / 2.0;
    iradius = (int) (radius/step + 0.5);

    // Examine all points from the gradient image that could lie within the _index square.
    for (i = -iradius; i <= iradius; i++)
        for (j = -iradius; j <= iradius; j++) {

        // Rotate sample offset to make it relative to key orientation.
        // Uses (x,y) coords.  Also, make subpixel correction as later image
        // offset must be an integer.  Divide by spacing to put in _index units.
        rpos = (step*(_cose * i + _sine * j) - fracr) / spacing;
        cpos = (step*(- _sine * i + _cose * j) - fracc) / spacing;

        // Compute location of sample in terms of real-valued _index array
        // coordinates.  Subtract 0.5 so that rx of 1.0 means to put full
        // weight on _index[1] (e.g., when rpos is 0 and _IndexSize is 3.
        rx = rpos + _IndexSize / 2.0 - 0.5;
        cx = cpos + _IndexSize / 2.0 - 0.5;

        // Test whether this sample falls within boundary of _index patch
        if (rx > -1.0 && rx < (double) _IndexSize  &&
            cx > -1.0 && cx < (double) _IndexSize) {

            int r = iy + i*step;
            int c = ix + j*step;

            AddSample(r, c, rpos, cpos, rx, cx, int(scale));
         }
    }
```

Slika 2.6 funkcija createVector

Ova funkcija generiše vektor deskriptora za određenu detektovanu tačku na slici. Ovaj vektor opisuje lokalnu strukturu okoline tačke i koristi se kao deskriptor tačke.

Funkcija prolazi kroz svaki piksel u lokalnoj okolini detektovane tačke. Za svaki od piksela, raćuna se gradijent koristeći Sobel operatore za parcijalne derivacije po x i y osama. Gradijenti se zatim koriste za određivanje orijentacije teksture u okolini tačke. Na osnovu orijentacija magnituda gradijenta, generiše se vektor deskriptora koji opisuje raspodelu gradijenata u okolini tačke.

Vektori deskriptora moraju biti dovoljno detaljni da se razlikuju među različitim tačkama i da omoguće pouzdano upoređivanje tačaka između različizih slika.

### *7.void AddSample i void PlaceInIndex*

```
void AddSample(int r, int c, double rpos,
                  double cpos, double rx, double cx, int step) {
    double weight;
    double dx, dy;
    double dxx, dyy;
    double ori1, ori2;

    // Clip at image boundaries.
    if (r < 1+step  ||  r >= _height - 1-step  || c < 1+step  ||  c >= _width - 1-step) {
        return;}

    weight = _lookup2[num_i(rpos * rpos + cpos * cpos)];

    dxx = weight*get_wavelet2(_Pixels, c, r, step);
    dyy = weight*get_wavelet1(_Pixels, c, r, step);
    dx = _cose*dxx + _sine*dyy;
    dy = _sine*dxx - _cose*dyy;

    if (dx < 0) ori1 = 0;
    else ori1 = 1;

    if (dy < 0) ori2 = 2;
    else ori2 = 3;

    PlaceInIndex(dx, ori1 , dy, ori2 , rx, cx);
}
```

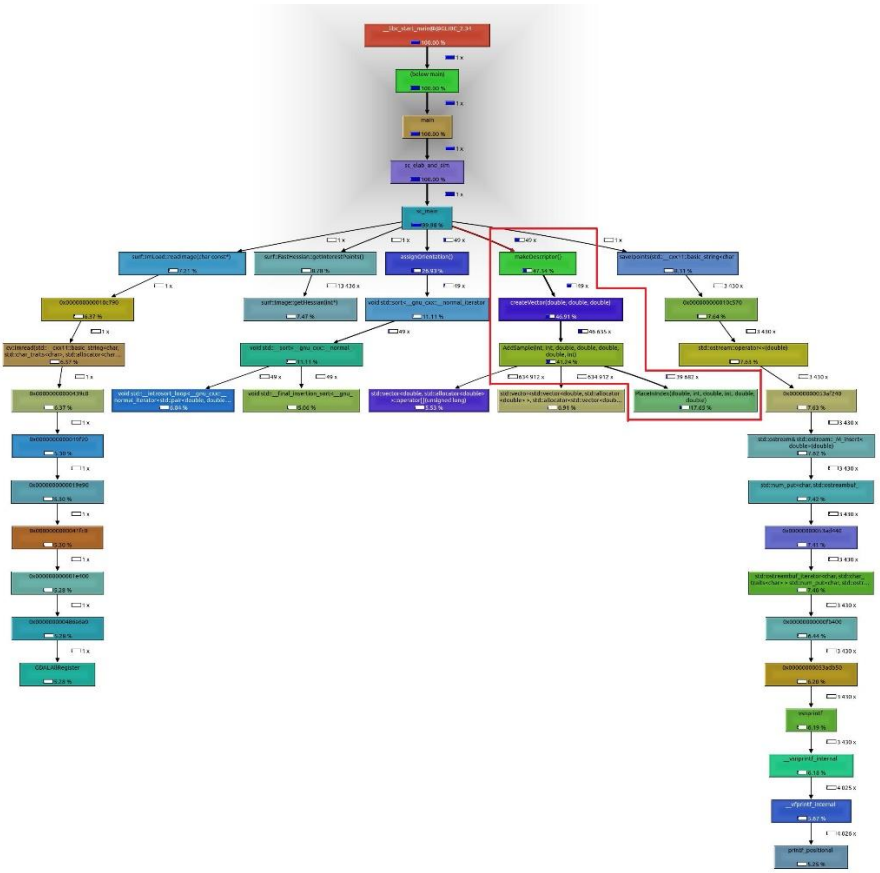
```
void PlaceInIndex(double dx, int ori1, int dy, int ori2, int rx, int cx) {

    int ri = std::max(0, std::min(static_cast<int>(_IndexSize - 1), static_cast<int>(rx)));
    int ci = std::max(0, std::min(static_cast<int>(_IndexSize - 1), static_cast<int>(cx)));


    // Izračunavanje frakcionih delova i težina
    double rfrac = rx - ri;
    double cfrac = cx - ci;

    rfrac = std::max(0.0f, std::min(float(rfrac), 1.0f));
    cfrac = std::max(0.0f, std::min(float(cfrac), 1.0f));

    double rweight1 = dx * (1.0 - rfrac);
    double rweight2 = dy * (1.0 - rfrac);
    double cweight1 = rweight1 * (1.0 - cfrac);
    doublee cweight2 = rweight2 * (1.0 - cfrac);

    // Pre nego što pristupamo _index, proveravamo da li su ri i ci unutar granica
    if (ri >= 0 && ri < _IndexSize && ci >= 0 && ci < _IndexSize) {
        _index[ri][ci][ori1] = cweight1;
        _index[ri][ci][ori2] = cweight2;
    }


}
```

Slika 2.7a i 2.7b funkcije AddSample i PlaceInIndex

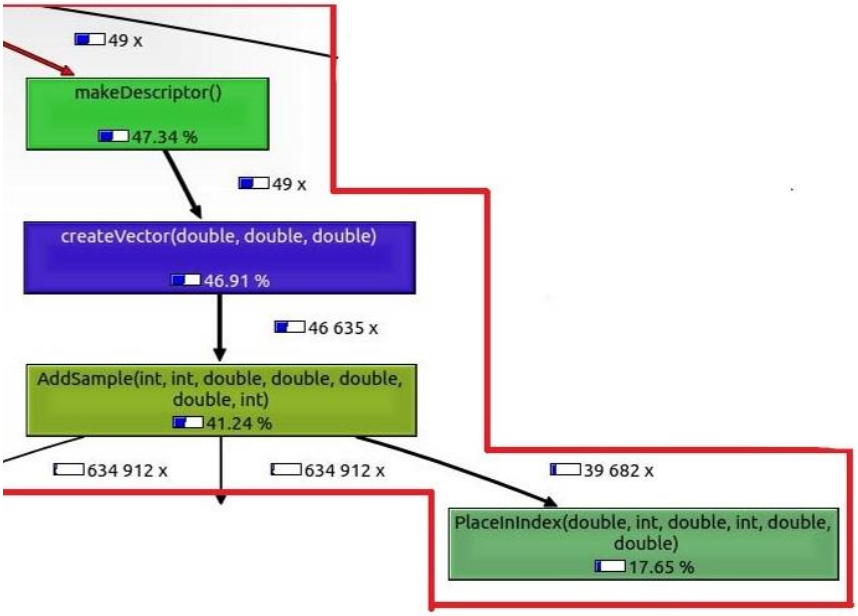
Ove funkcije zajedno služe za dodavanje uzorka u matricu indeksa (_index) tokom procesa generisanja deskriptora tačaka.

Funkcija AddSample prolazi kroz piksele u lokalnoj okolini tačke i uzima njihove gradijente. Gradijenti se skaliraju težinama koje zavise od udaljenosti piksela od centra uzorka, kako bi se izbegle nepotrebne interpolacije. Skalirani gradijenti se prosleđuju funkciji PlaceInIndex. Ona dodaje skalirane gradijente u matricu indeksa na odgovarajuće pozicije koristeći interpolaciju.

# 3. Rezultati profajliranja



Slika 3.1 grafički prikaz profajlinga




Slika 3.2 izdvojene funkcije koje oduzimaju najviše vremena

U ovom poglavlju govori se o profajliranju koje je odrađeno pomoću vallgrind alata za profajliranje. Na osnovu rezultata sa slika 3.1 i 3.2 vidi se da najviše procesorskog vremena uzima funkcija makeDescriptor(). Pošto se u funkciji makeDescriptor() poziva funkcija createVector(double, double, double) koja oduzima skoro čitavo vreme koje oduzima makeDescriptor(), u hardveru će biti realizovana for petlja u kojoj se poziva funkcija AddSample(int, int, double, double, double, double, int) i funkcija PlaceInIndex(double, int, double, int, double, double) koja se poziva u funkciji AddSample.

## 4. Bitska analiza

Ovo poglavlje govori o puštanju system c simulacije nakon što su vrednosti svih int-ova i double-ova unutar funkcija koje su profajliranjem određene kao one koje zauzimaju najviše vremena, zamenjene sa system c sc_int i sc_fixed tipom.

Analizom bita dobijamo da treba da koristimo umesto tipa double sc_fixed<48,30> a umesto tipa int sc_int<11>.

```
typedef sc_dt::sc_int<11> num_i;
typedef sc_dt::sc_fixed<48, 30, sc_dt::SC_TRN, sc_dt::SC_SAT> num_f;
```

Slika 4.1 definisani tipovi podataka

```
    for (i = -iradius; i <= iradius; i++)
        for (j = -iradius; j <= iradius; j++) {

        // Rotate sample offset to make it relative to key orientation.
        // Uses (x,y) coords.  Also, make subpixel correction as later image
        // offset must be an integer.  Divide by spacing to put in _index units.
        rpos = (step*(_cose * i + _sine * j) - fracr) / spacing;
        cpos = (step*(- _sine * i + _cose * j) - fracc) / spacing;

        // Compute location of sample in terms of real-valued _index array
        // coordinates.  Subtract 0.5 so that rx of 1.0 means to put full
        // weight on _index[1] (e.g., when rpos is 0 and _IndexSize is 3.
        rx = rpos + _IndexSize / 2.0 - 0.5;
        cx = cpos + _IndexSize / 2.0 - 0.5;

        // Test whether this sample falls within boundary of _index patch
        if (rx > -1.0 && rx < (double) _IndexSize  &&
            cx > -1.0 && cx < (double) _IndexSize) {

            num_i r = iy + i*step;
            num_i c = ix + j*step;

            AddSample(r, c, rpos, cpos, rx, cx, int(scale));

         }
    }


}
```

Slika 4.2 for petlja u kojoj se poziva AddSample unutar bitske analize

```
void AddSample(num_i r, num_i c, num_f rpos,
                    num_f cpos, num_f rx, num_f cx, num_i step) {
    num_f weight;
    num_f dx, dy;
    num_f dxx, dyy;
    num_i ori1, ori2;

    // Clip at image boundaries.
    if (r < 1+step  ||  r >= _height - 1-step  || c < 1+step  ||  c >= _width - 1-step) {
        return;}

    weight = _lookup2[num_i(rpos * rpos + cpos * cpos)];

    dxx = weight*get_wavelet2(_Pixels, c, r, step);
    dyy = weight*get_wavelet1(_Pixels, c, r, step);
    dx = _cose*dxx + _sine*dyy;
    dy = _sine*dxx - _cose*dyy;

    if (dx < 0) ori1 = 0;
    else ori1 = 1;

    if (dy < 0) ori2 = 2;
    else ori2 = 3;

    PlaceInIndex(dx, ori1 , dy, ori2 , rx, cx);
}
```

Slika 4.3 funkcija AddSample unutar bitske analize

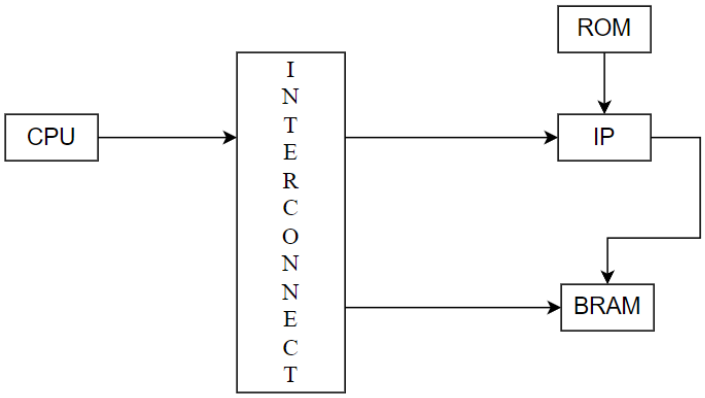
```
void PlaceInIndex(num_f dx, num_i ori1, num_f dy, num_i ori2, num_f rx, num_f cx) {

    num_i ri = std::max(0, std::min(static_cast<int>(_IndexSize - 1), static_cast<int>(rx)));
    num_i ci = std::max(0, std::min(static_cast<int>(_IndexSize - 1), static_cast<int>(cx)));


    // Izračunavanje frakcionih delova i težina
    num_f rfrac = rx - ri;
    num_f cfrac = cx - ci;

    rfrac = std::max(0.0f, std::min(float(rfrac), 1.0f));
    cfrac = std::max(0.0f, std::min(float(cfrac), 1.0f));

    num_f rweight1 = dx * (1.0 - rfrac);
    num_f rweight2 = dy * (1.0 - rfrac);
    num_f cweight1 = rweight1 * (1.0 - cfrac);
    num_f cweight2 = rweight2 * (1.0 - cfrac);

    // Pre nego što pristupamo _index, proveravamo da li su ri i ci unutar granica
    if (ri >= 0 && ri < _IndexSize && ci >= 0 && ci < _IndexSize) {
        _index[ri][ci][ori1] = cweight1;
        _index[ri][ci][ori2] = cweight2;
    }


}
```

Slika 4.4 funkcija PlaceInIndex unutar bitske analize

# 5. Virtuelna platforma

Virtuelna platforma predstavlja softverski model čija je svrha da bude zamena za hardverski prototip u ranim fazama razvoja modela. Softverski modeli bitnih delova sistema se povezuju sa ciljem da se dobije model celokupnog sistema koji može da se izvršava. TLM(Transaction Level Modeling) standard olakšava pravljenje virtuelnih platformi.

CPU i IP komuniciraju putem interconnecta i koriste TLM za prenos podataka.




Slika 5.1 blok šema virtuelne platforme

```
374         while(!done)
375         {
376             if(ready)
377             {
378                 need_start = 1;
379             }
380
381
382             if (need_start)
383             {
384                 write_hard_int(addr_start,1);
385                 need_start = 0;
386             }
387
388             while(ready)
389             {
390                 ready = read_hard_int(addr_ready);
391                 cout<< "IP is about to start" << endl;
392                 if (!ready)
393                     write_hard_int(addr_start,0);
394             }
395
396             ready = read_hard_int(addr_ready);
```

```
402         if(ready)
403         {
404             index1d.clear();
405
406             for (int i = 0; i < 64 ; i++) {
407                 offset += sc_core::sc_time(DELAY+DELAY, sc_core::SC_NS);
408                 index1d.push_back(read_mem(addr_index1+i));
409             }
410
411             int index_1d_index = 0;
412             for (int i = 0; i < _IndexSize; ++i) {
413                 for (int j = 0; j < _IndexSize; ++j) {
414                     for (int k = 0; k < 4; ++k) {
415                         _index[i][j][k] = index1d[index_1d_index++];
416                     }
417                 }
418             }
419
420             /*for (int i = 0; i < _IndexSize; ++i) {
421                 for (int j = 0; j < _IndexSize; ++j) {
422                     for (int k = 0; k < 4; ++k) {
423                         cout << "IZ CPU-a _index[" << i << "]" << "[" << j << "]" << "[" << k <<
424                     }
425                 }
426             }
427
428             cout << "////////////////////////////////////////////////////////////" << endl;*/
429
430             ready = 0;
431             need_start = 0;
432             done = 1;
433         }
434     }
435 }
```

Slika 5.2 i 5.3 komunikacija CPU-a sa IP-em

```
Ip::Ip(sc_module_name name) :
    sc_module(name),
    ready(1)

{
    SC_THREAD(proc);
    _index.resize(_IndexSize, std::vector<std::vector<num_f>>(_IndexSize, std::vector<num_f>(4, 0.0f)));
    _lookup2.resize(40);
    interconnect_socket.register_b_transport(this, &Ip::b_transport);
    cout << "IP constructed" << endl;
}
```

```
if (start == 1 && ready == 1)
{
    ready = 0;
    offset += sc_time(DELAY, SC_NS);
}

else if (start == 0 && ready == 0)
{
```

Slika 5.4 i 5.5 IP

```
313             cout << "Entry from IP to memory completed" << endl;
314             ready = 1;
```

Slika 5.6 završena hardverska obrada

Postoje 3 flega (done, ready i need_start). Inicijalno postavljamo done i need_start na 0, a ready na 1. Kada je done = 0 program ulazi u while petlju. Ready je inicijalno 1 pa će need_start dobiti vrednost 1 i program šalje hardveru da je start = 1. U konstruktoru je specificirano da je ready u hardveru jednak jedinici. Kada su i ready i start na jedinici, ready postaje 0. U CPU-u ready je i dalje jednak jedinici, pa program ulazi u while(ready) petlju i ready uzima vrednost registra addr_ready koja se čita iz hardvera, što znači da će ready u tom trenutku dobiti vrednost 0. Ako je ready 0, ispunjen je if uslov i hardveru šaljemo 0 kao vrednost registra addr_start, start iz hardvera postaje 0, i ispunjen je else if uslov jer su i ready i start 0. Kada je ispunjen ovaj uslov hardver počinje da obrađuje sve što se u njemu nalazi i nakon završetka obrade, ready dobija vrednost 1, uz pomoć čega IP obaveštava CPU da je završio sa obradom.

U registre start i ready kao i registre scale, iradius, fracr, fracc, spacing, step, _sine, _cose, iy i ix podatke šaljemo iz procesora.

Podaci koji su nam potrebni za obradu koji se nalaze u _lookup2 nizu smešteni su u rom memoriju i odatle ih čitamo.

_Pixels niz iz procesora upisujemo u BRAM, zatim iz BRAM-a čitamo u hardveru, dok _index niz iz hardvera upisujemo u BRAM i šaljemo CPU-u.

## 6. Performanse sistema

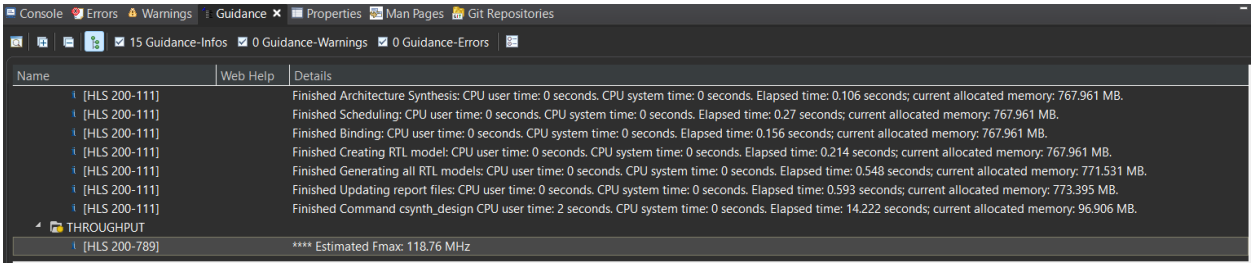
Da bi se odredila frekvencija sistema, kritična funkcija se realizuje u jednoj iteraciji. Kod se nalazi na slici ispod, i rezultat pokretanja simulacije nam govori da je vrednost frekvencije sistema oko 118MHz.

```
#include <math.h>
#include <stdio.h>

double _lookup2[40] = { .939411163330078125, .82902908325195312 5, .7316131591796875, .6456451416015625, .569782257080078125, .50283050537109375, .443744659423828
int _IndexSize = 4;
int iradius = 23;
double _cose = 0.93941116330078125, _sine = .829029083251953125;
int step = 4;
double fracr = 4.29283, fracc = 3.28324;
double spacing = 3.50605;
int iy = 2; ix = 3;
double _Pixels[4] = { 4797.886272430419921875, 8336.73332977294921875, 10852.898036956787109375, 1873.266666412353515625};

void hard(double _index[4][4][4]) {
    double rpos = (step*(_cose + _sine) - fracr) / spacing;
    double cpos = (step*(- _sine + _cose) - fracc) / spacing;

    double rx = rpos + _IndexSize / 2.0 - 0.5;
    double cx = cpos + _IndexSize / 2.0 - 0.5;

    if (rx > -1.0 && rx < (double) _IndexSize &&
        cx > -1.0 && cx < (double) _IndexSize) {

        int r = iy + step;
        int c = ix + step;
        int ori1 = 1, ori2 = 2;
        int ri = 2, ci = 3;
        int scale = 2;
        int _height = 129, _width = 129;

        int addSampleStep = scale;
```

```
        double weight;
        double dxx1, dxx2, dyy1, dyy2;
        double dx, dy;
        double dxx, dyy;
        double rfrac, cfrac;
        double rweight1, rweight2, cweight1, cweight2;

        if (r >= 1 + addSampleStep && r < _height - 1 - addSampleStep && c >= 1 + addSampleStep && c < _width - 1 - addSampleStep) {

            weight = _lookup2[(int)(rpos * rpos + cpos * cpos)];

            dxx1 = _Pixels[0];
            dxx2 = _Pixels[1];
            dyy1 = _Pixels[2];
            dyy2 = _Pixels[3];

            dxx = weight * (dxx1 - dxx2);
            dyy = weight * (dyy1 - dyy2);
            dx = _cose * dxx + _sine * dyy;
            dy = _sine * dxx - _cose * dyy;

            if (dx < 0) ori1 = 0;
            else ori1 = 1;

            if (dy < 0) ori2 = 2;
            else ori2 = 3;
```

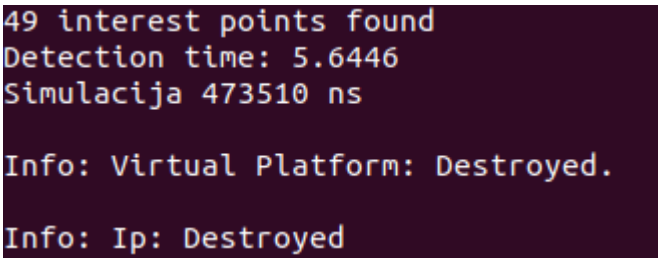
```
            if (rx < 0) ri = 0;
            else if (rx >= _IndexSize) ri = _IndexSize - 1;
            else ri = rx;

            if (cx < 0) ci = 0;
            else if (cx >= _IndexSize) ci = _IndexSize - 1;
            else ci = cx;

            rfrac = rx - ri;
            cfrac = cx - ci;

            if (rfrac < 0.0) rfrac = 0.0;
            else if (rfrac > 1.0) rfrac = 1.0;

            if (cfrac < 0.0) cfrac = 0.0;
            else if (cfrac > 1.0) cfrac = 1.0;

            rweight1 = dx * (1.0 - rfrac);
            rweight2 = dy * (1.0 - rfrac);
            cweight1 = rweight1 * (1.0 - cfrac);
            cweight2 = rweight2 * (1.0 - cfrac);

            if (ri >= 0 && ri < _IndexSize && ci >= 0 && ci < _IndexSize) {
                _index[ri][ci][ori1] = cweight1;
                _index[ri][ci][ori2] = cweight2;
            }
        }
    }
}
int main() {
    double _index[4][4][4];

    hard(_index);

    return 0;
}
```

Slika 6.1 Kod koji je korišćen za određivanje frekvencije




Slika 6.2 Maksimalna frekvencija sistema

Da bi se odredio THROUGHPUT, ispisuje se u terminalu ukupno akumulirano vreme simulacije.




Slika 6.3 akumulirano vreme simulacije

Dobijeno je da se može obraditi oko 211 slika u sekundi.