Univerzitet u Banjoj Luci
Elektrotehnički fakultet
Sistemi za digitalnu obradu signala

PROJEKTNI ZADATAK

Segmentacija slike upotrebom algoritma za detekciju ivica na ADSP-21489 razvojnoj platformi

Autor: Aleksandar Bešlić, 1198/17

Banja Luka, februar 2023

1. Učitavanje slike

Na razvojnu platformu ADSP-21489 učitava se slika u .bmp formatu. Sve informacije o slici čuvaju se u strukturi *IMAGE*, kao što su visina, širina, broj bajtova potrebni za predstavljanje jednog piksela kao i sam niz piksela (pod piksela). Za niz piksela rezervisano je 4MB memorije na sdram čipu. Nastavku je prikazana u struktura IMAGE.

```
typedef struct img
{
   unsigned int width;
   unsigned int height;
   unsigned int bytesPerPixel;
   unsigned char *pixels;
} IMAGE;
```

2. Pretvaranje u *gray scale*

Nakon učitavanja slike, sliku je potrebno pretvoriti u *grey scale* sliku. Posmatrati ćemo dva metoda za konverziju, i luminiscentni metod kao i srednju vrijednost *RGB* komponenti.

U nastavku je prikazan isječak kôda koji konvertuje sliku u *gray scale* primjenom luminiscentno metoda. Sam metod vise odgovara načinu kako ljudsko oko funkcioniše.

```
for (i = 0u; i < size; ++i)
{
  /* Grayscale conversion 0.11*B + 0.59*G + 0.3*R */
  gray_pixels[i] = 0.11f *pixels[i*onePixel] + 0.59f *pixels[i*onePixel + 1u] + 0.3f *pixels[i*onePixel +
2u];
}
```

U nastavku prikazan je isječak kôda koji konvertuje sliku u *gray scale* primjenom srednje vrijednosti *RGB* komponenti ovo ćemo smatrati dolje riješene u smislu brzine izvršavanja.

```
for (i = 0u; i < imageSize; ++i)
 {
      B = pixels[i * onePixel];
      G = pixels[i * onePixel + 1u];
      R = pixels[i * onePixel + 2u];
      gray_pixels[i] = (R + G + B) / 3u;
}
```

3. Detekcija ivica

Za detektovanje ivica koristićemo 2D konvoluciju sa Laplasovim operatorom za detekciju ivica. Laplasov kernel odabran je iz razloga što u jednom prolazu možemo detektovati ivice iz različitih pravaca. Sam kernel aproksimira drugi izvod funkcije. Nakon primjene ovakvog kernela ivica se nalazi na mjestu gdje funkcija presjeca x-osu (*„zero crosing edge“).* U nastavku su prikazani neki primjeri Laplasovih kernela.

$$\begin{bmatrix} -1 & 0 & -1 \\ 0 & 4 & 0 \\ -1 & 0 & -1 \end{bmatrix}$$
a)

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$
b)

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$
c)

$$\begin{bmatrix} 1 & 4 & 1 \\ 4 & -20 & 4 \\ 1 & 4 & 1 \end{bmatrix}$$
d)

Slika 1. a) Laplas kernel v1, b) Laplas kernel v2, c) Quick kernel, d) Najbolji kernel

Od datih kernela najbolje se pokazao kernel d) iz slike 1. Jedan način da se poboljša detekcija ivica jeste da se slika prvo izravna Gusovim filterom pa tek onda primjeni Laplasov kernel ili da se koristi kombinovani operator *LoG* kernel. U nastavku je prikazan kôd koji implementira 2D konvoluciju.

```
unsigned int centerX = k_col  / 2u;
unsigned int centerY = k_row / 2u;
for(i = 0u; i < row; ++i){
   for(j = 0u; j < column; ++j){
      for(m = 0u; m < k_row; ++m){
          mm = k_row - 1u - m;  /* row index of flipped kernel */

           for(n = 0u; n < k_col; ++n){
              nn = k_col - 1u - n;  /* column index of flipped kernel */

               /* index of input signal, used for checking boundary */
               ii = i + (centerY - mm);
               jj = j + (centerX - nn);

               /* ignore input samples which are out of bound */
               if((ii < row) && (jj < column))
                 output[i*column + j] += pixels[ii * column + jj] * kernel[mm * k_col + nn]; }
          }
   }
}
```

4. Kodovanje zatvorenih kontura

Kodovanje zatvorenih kontura radi se tako da se svakom pikselu u zatvorenoj konturi dodjeli jedinstven kod. Kodovanje je izvršeno primjenom *flood fill* algoritma, koji je baziran na *BFS* algoritmu za pretragu grafova. U nastavku prikazana je implementacija algoritma za kodovanje.

```
code_segments:
   for (i = 0u; i < height; i++)
      for (j = 0u; j < width; j++)
         /* Check if the pixel is empty and if that pixel isn't already color  then calls flood fill function */
         if (pixels[i * width + j] == 0u && codes[i * width + j] == 0u) {
            flood_fill(image, j, i, next_code, codes);
            next_code = (next_code + 1u) % NUM_COLORS;}

/* Assign the color codes to the image */
   for (i = 0u; i < size; i++)
          pixels[i] = codes[i];
```

```
flood_fill:
   queue[rear][0] = x;
   queue[rear][1] = y;
   rear++;
   codes[y * width + x] = color;
   while (front != rear) {
      int current_x = queue[front][0];
      int current_y = queue[front][1];
      front++;
      for (i = -1; i <= 1; i++)
         for (j = -1; j <= 1; j++)
            xx = current_x + i;
            yy = current_y + j;

            /* ignore out of bound positions */
            if (((xx < 0) || (xx >= width)) || ((yy < 0) || (yy >= height)))
               continue;

/* If the pixel is empty and if that pixel isn't already color then we color it and put the positions the
queue */
            if (pixels[yy * width + xx] == 0u && codes[yy * width + xx] == 0u) {
               codes[yy * width + xx] = color;
               queue[rear][0] = xx;
               queue[rear][1] = yy;
               rear++; }}
```

Broj kontura koji se mogu kodovati zavisi od broja definisanih boja (*NUM_COLOR).*

5. Bojenje piksela

Nakon završenog kodovanja slijedi bojenje piksela. Sam kôd kojim je piksel kodovan odgovara jednoj boji. U nastavku prikazan je kôd koji boji piksele.

```
/* Coloring each pixel */
for (i = 0u; i < size; i++){
      color_index = codes[i];

      if (color_index < NUM_COLORS){
         /* Setting RGB values */
         pixels[i * onePixel]       = C_MASK_B(colors[color_index]);
         pixels[i * onePixel + 1u] = C_MASK_G(colors[color_index]);
         pixels[i * onePixel + 2u] = C_MASK_R(colors[color_index]);
      }
   }
```

Progres segmentacije slike može se pratiti preko dioda na razvojnoj ploči, tako nakon svakog koraka jedna dioda će se uključiti.

6. Optimizacija

Optimizacija je obavljena primjenom različitih metoda, kao što su vektorizacija petlji, branch predikcija, korištenjem kompajlerskih optimizacija, primjenom efikasnih varijanti funkcija.

Što se tiče konverzije u *gray scale* tu smo koristili drugi metod konverzije koji u sebi ne sadrži operacije koje koriste *float* varijable. Pored toga uveli smo ključnu riječ *restrict* koja govori da pokazivači na niz ne pokazuju na istu memorijsku lokaciju, i dodali smo pragmu *const* koja govori kompajleru da funkcija ne mijenja globalne varijable.

```
#pragma const
void convert_to_grayscale_dsp(IMAGE *originalImage, IMAGE *grayscaleImage)
…
unsigned char * restrict pixels = originalImage->pixels;
unsigned char * restrict gray_pixels = grayscaleImage->pixels;
```

Za optimizaciju konvolucije uradili smo sljedeće, zamijenili smo redoslijed *for* petlji tako da unutrašnje petlje imaju veći broj ponavljanja. Uveli smo i vektorizaciju petlji, *branch* predikciju kao i ključnu riječ *restrict* i pragmu *const*. Pa tako imamo sljedeći kod:

```
#pragma vector_for
   for(m = 0u; m < k_row; ++m){
      mm = k_row - 1u - m; /* row index of flipped kernel */

      #pragma vector_for
      for(n = 0u; n < k_col; ++n) {
         nn = k_col - 1u - n;  /* column index of flipped kernel */

         for(i = 0u; i < row; ++i) {
            for(j = 0u; j < column; ++j){
               /* index of input signal, used for checking boundary */
               ii = i + (centerY - mm);
               jj = j + (centerX - nn);

               /* ignore input samples which are out of bound */
               if(expected_true((ii < row) && (jj < column)))
                  pixelsO[i*column + j] += pixelsI[ii * column + jj] * kernel[mm * k_col + nn];}}}
}
```

Ostale funkcije su na sličan način optimizovane. Pa tako za kodovanje imamo:

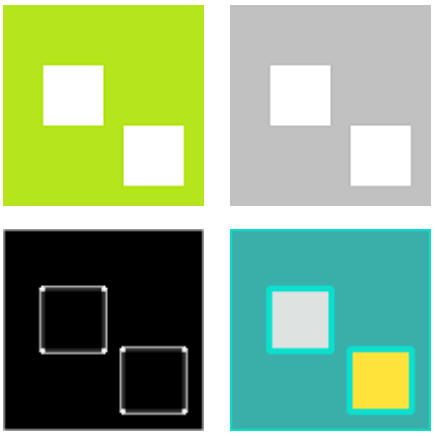
```
code_segments:
#pragma vector_for
   for (i = 0u; i < height; i++)
      for (j = 0u; j < width; j++)
         /* Check if the pixel is empty and if that pixel isn't already color  then calls flood fill function */
         if (expected_true((pixels[i * width + j] == 0u) && (codes[i * width + j] == 0u)))
                        ...
   /* Assign the color codes to the image */
   #pragma SIMD_for
   for (i = 0u; i < size; i++)
         ...
```

```
flood_fill:
  while (front != rear)
     ...
           /* ignore out of bound positions */
           if  (expected_false(((xx < 0) || (xx >= width)) || ((yy < 0) || (yy >= height))))
                 ...
           if (expected_true((pixels[yy * width + xx] == 0u) && (codes[yy * width + xx] == 0u)))
                ...
```

Najveći *„bottle neck“* u kôdu sigurno predstavljaju procesi kodovanja i detekcije ivica. Jedna bolja varijanta za detekciju ivica jeste da se umjesto 2D konvolucije koristi 2D *FFT*, što bi dovelo do daljnjeg poboljšanja performansi. Što se tiče kodovanja najbolja varijanta bi bila da se pomoćni nizovi čuvaju u što bližim i bržim memorijama na procesoru. U cikličnom pitanju ova dva procesa najviše zahtijevaju procesorskog vremena, stim da vrijeme potrebno da se jedna slika pročita i upiše znatno nadmašuje vrijeme potrebno za njenu obradu, riječ je o minutima za čitanje i upis a za obradu riječ o milisekundima.

7. Rezultati

Slika rezolucije 100x100:

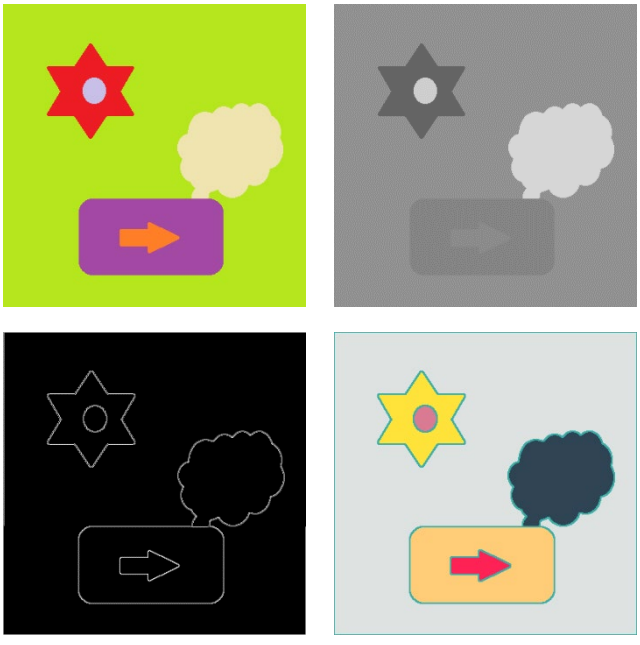


Slika 2. Proces obrade slike 100x100

Tabela 1. Broj ciklusa potrebnih za pojedine korake za sliku 100x100

Slika 100x100	*Gray scale*	*Detekcija ivica*	*Kodovanje piksela*	*Bojenje piksela*
Početni kôd	*1.443.455*	*13.763.794*	*13.822.113*	*1.664.164*
Početni kôd + 100% Kom.	*1.313.103*	*5.696.716*	*8.270.895*	*1.011.382*
Optimizovani kôd	*861.865*	*4.836.014*	*8.185.203*	*411.714*

Slika rezolucije 400x400:

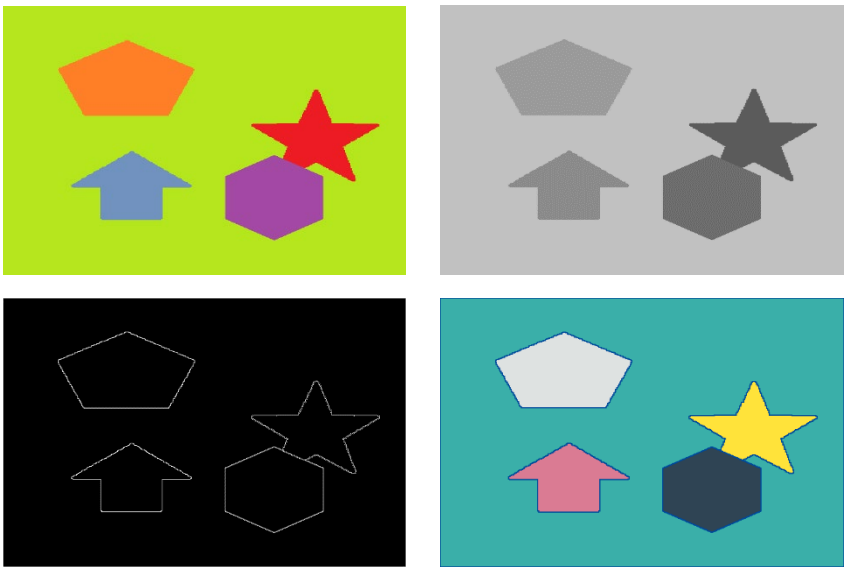


Slika 3. Proces obrade slike rezolucije 400x400

Tabela 2. Broj ciklusa potrebnih za pojedine korake za sliku 400x400

Slika 400x400p	*Gray scale*	*Detekcija ivica*	*Kodovanje piksela*	*Bojenje piksela*
Početni kôd	*23.092.627*	*221.766.087*	*239.630.752*	*26.629.755*
Početni kôd + 100% Kom.	*21.010.560*	*131.007.283*	*172.099.573*	*20.882.018*
Optimizovani kôd	*13.789.024*	*120.545.325*	*168.382.051*	*11.370.028*

Slika rezolucije 720x480:

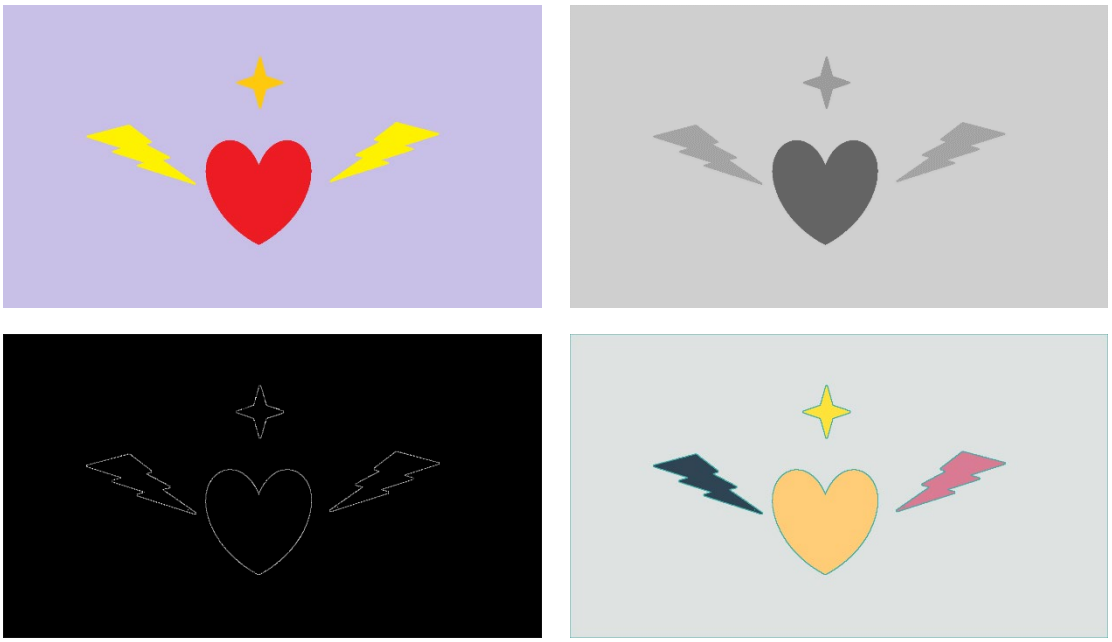


Slika 4. Proces obrade slike 720x480

Tabela 3. Broj ciklusa potrebnih za pojedine korake za sliku 720x480

Slika 720x480	*Gray scale*	*Detekcija ivica*	*Kodovanje piksela*	*Bojenje piksela*
Početni kôd	*49.879.998*	*409.667.103*	*511.024.972*	*49.750.314*
Početni kôd + 100% Kom. Opt	*45.382.978*	*214.848.967*	*359.657.889*	*35.992.896*
Optimizovani kôd	*29.784.133*	*168.623.209*	*348.285.427*	*14.229.006*

Slika rezolucije 1280x720:



Slika 5. Proces obrade slike 1280x720

Tabela 4. . Broj ciklusa potrebnih za pojedine korake za sliku 1280x720

Slika 1280x720	*Gray scale*	*Detekcija ivica*	*Kodovanje piksela*	*Bojenje piksela*
Početni kôd	*103.187.340*	*1.285.044.971*	*1.649.838.878*	*158.617.954*
Početni kôd + 100% Kom. Opt	*92.868.606*	*656.868.438*	*1.039.398.493*	*111.032.568*
Optimizovani kôd	*61.912.404*	*449.765.740*	*989.903.327*	*44.413.027*

8. Uputstvo za upotrebu

int read_image(char *fileName, IMAGE *image)
Za učitavanje *.bmp* slike koristi se funkcija *read_image*, kojoj je potrebno proslijediti ime slike i pokazivač na strukturu *IMAGE*. Sam niz u kojem se čuvaju pikseli potrebno je alocirati prije samog pokretanja funkcije. Postoji dodatni način za učitavanja slike a to je da se slika učita na razvojnu platformu u obliku *.h* fajla, u priloženom kôdu nalazi se jednostavan program koji će konvertovati *.bmp* sliku tako da se ona može smjestiti u *.h* fajl. Ovo je znatno brži način učitavanja stim da je pogodan za slike manjih dimenzija.

int write_image(char *fileName, IMAGE *image)
Za pisanje *.bmp* slike koristi se funkcija *write_image*, kojoj je potrebno proslijediti ime slike i pokazivač na strukturu *IMAGE*. Za čuvanje među rezultata prije pokretanja ove funkcije potrebno je pozvati funkciju *prep_for_writing*, koja će sliku među rezultata pretvoriti sliku koja se može sačuvati na disku.

void convert_to_grayscale(IMAGE *originalImage, IMAGE *grayscaleImage)
Za konverziju slike u gray scale koristi se funkcija *convert_to_grayscale*, kojoj je potrebno proslijediti pokazivač na učitanu sliku kao i pokazivač za izlaznu gray scale sliku. Sam niz u kojem se čuvaju pikseli potrebno je alocirati prije samog pokretanja funkcije.

void edge_detection(IMAGE *input, IMAGE *output, char *kernel, unsigned int k_row, unsigned int k_col);
Za detektovanje ivica na slici koristi se funkcija *edge_detection*, kojoj je potrebno proslijediti pokazivač na *gray scale* sliku, pokazivač za izlaznu sliku (prethodno alociran niz piksela), pokazivač na kernel matricu (koja je u obliku niza), broj kolona i redova kernel matrice. Ova funkcija može se koristiti za različite stvari koje uključuju 2D konvoluciju, sve zavisi od tipa kernela koji se proslijedi.

void code_segments(IMAGE *image, unsigned char *codes , int *queue[2]);
Za kodovanje segmenata koristi se funkcija *code_segments*, kojoj je potrebno proslijediti pokazivač na sliku na kojoj su već detektovane ivice, kao i pomoćne nizove koji su prethodno alociran na veličinu slike.

void color_segments(IMAGE *input, IMAGE *output)
Za bojenje kodovanih piksela koristi se funkcija *color_segments*, kojoj je potrebno proslijediti pokazivač na kodovanu sliku i pokazivač na izlaznu obojenu sliku. Niz u kojem se čuvaju obojeni pikseli potrebno je alocirati prije poziva funkcije.

Pored datih funkcija postoje i optimizovane verzije ovih funkcija koje su označene sa *_dsp* u nazivu. Imaju iste zahtjeve i funkcionalnosti.

Dati kôd je u skladu* sa MISRA-C:2004 standardom.