Универзитет у Бањој Луци

Електротехнички Факултет

Системи за дигиталну обраду сигнала

# ПРОЈЕКТНИ ЗАДАТАК

*Сегментација слике употребом алгоритма за детекцију ивица на ADSP-21489 развојној платформи*

Аутор: Никола Цетић 1117/19

Бања Лука, фебруар 2023

## 1. Учитавање слике

За учитавање слике на развојну плочу резервисано је 40Kb у sram меморији, што је довољно за слике које ће бити обрађене у наставку. Учитавање слике вршено помоћу функције дате у задатку пројекта, са малим измјенама, због неких ограничења самог DSP. Јер када отворимо фајл са “wb” чита по 4 бајта иако је иницирано да буде прочитан само 1 бајт.

## *2.* Претварање у црно-бијелу слику

Претварање у црно-бијелу слику урађено је тежинском методом тј.

Grayscale = 0.299R + 0.587G + 0.114B

За разлику од трокомпонентне слике која се налази у sram, црно-бијела се налази у меморији самог чипа.

void to\_gray(void) {

    bytesPerPixel = 1;

    gray\_pix\_arr = (byte \*) heap\_malloc(0, height \* width);

    if (gray\_pix\_arr == NULL) {

        printf("Nije instancirana memorija\n");

        return;

    }

    int pix\_position;

    int num\_iter = height \* width;

    for (int i = 0; i < num\_iter; i++) {

        pix\_position = i \* 3;

        //                 ------RED--------                 --------GREEN--------               ------BLUE-------

        gray\_pix\_arr[i] = (char) ((float) pixels[pix\_position + 2] \* 0.299

                + (float) pixels[pix\_position + 1] \* 0.587

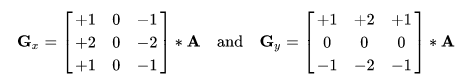
                + (float) pixels[pix\_position] \* 0.114);

    }

}

## 3. Детекција ивица

За детекцију ивица кориштен је Собелов оператор који има 2 кернела:



Слика 1.

void sobel\_edge\_detector(byte \*pixels, byte \*\*out\_pixels, uint32 width,

        uint32 height) {

    uint32 i, j, gx, gy;

    \*out\_pixels = pixels;         // assigning array to output pointer

    uint32 \*edged\_pixels\_arr\_uint32 = (uint32 \*) heap\_malloc(0,

            width \* height \* sizeof(uint32));

    if (edged\_pixels\_arr\_uint32 == NULL) {

        printf("Nije instancirana memorija\n");

        return;

    }

    uint32 (\*edged\_pixels\_mat\_uint32)[width] =

            (uint32 (\*)[width]) edged\_pixels\_arr\_uint32; // converting array to matrix

    int32 mx[3][3] = { { -1, 0, 1 }, { -2, 0, 2 }, { -1, 0, 1 } };

    int32 my[3][3] = { { -1, -2, -1 }, { 0, 0, 0 }, { 1, 2, 1 } };

    for (i = 1; i < height - 1; i++) {

        for (j = 1; j < width - 1; j++) {

            gx = convolution(pixels, mx, i, j, width);

            gy = convolution(pixels, my, i, j, width);

            edged\_pixels\_mat\_uint32[i][j] = sqrt(gx \* gx + gy \* gy);

        }

        // printf("Convolved!\n");

    }

    // printf("Done with convolution!\n");

    min\_max\_normalization(edged\_pixels\_arr\_uint32, width, height);

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

        pixels[i] = edged\_pixels\_arr\_uint32[i]; //converting from 32bit to 8bit

        // printf("%d",edged\_pixels\_arr[i]);

    }

    heap\_free(0,edged\_pixels\_arr\_uint32);

}

## 4. Кодовање затворених контура

Кодовање затворених контура вршено је connected Component алгоритмом, веома добро описаног на [линку](https://www.codeproject.com/Articles/336915/Connected-Component-Labeling-Algorithm). За разлику од структуре података кориштене на сајту, овде ће бити кориштен низ од HEIGHT\*WIDTH елемената, гдје ће сваки од њих бити свој родитељ у почетном случају. Прије почетка извршавања алгоритма потребно је слику превести у бинарни облик. Тј. Ивице ће бити кодоване са 0, а остали простор са 1. Приказано у коду испод.

|  |
| --- |
| // binary conversion  #pragma SIMD\_for      for (int i = 0; i < w \* h; i++) {          if (edge\_im[i] > 10)              edge\_im[i] = 0;          else              edge\_im[i] = 1;      }  byte (\*edge\_mat)[w] = (byte (\*)[w]) edge\_im;      uint32 \*parent\_arr = (uint32 \*) heap\_malloc(0, w \* h \* sizeof(uint32));      if (parent\_arr == NULL) {          printf("Nije instancirana memorija\n");          return;      }  #pragma vector\_for      for (int i = 0; i < w\*h; i++)          // Everyone is it's own parent      {          parent\_arr[i] = i;      } |

Постоје двије имплементације овог алгоритма. Прва је да тражимо комшијски пиксел са најмањом вриједношћу. Те ћемо тај пиксел прогласити за родитељски пискелу који се провјерава и додијелимо му вриједност родитеља. Овај тип реализације алгоритма има додатну провјеру а то је да морамо провјеравати и да ли је комшијски пиксел ивица тј. Да ли је једнак 0. С тиме у вези извршена је оптимизација самог алгоритма тако да се тражи комшијски пиксел са највећом вриједношћу кога ћемо прогласити за родитељски. Овиме смо избацили провјеру ивице. У коду се ова 2 алгоритма могу покренути тако што се за први случај откоментарише

#define LABELING\_V1, а за други случај #define LABELING\_V2.

Након завршене итерације преко слике слиједи тражење коријенског родитеља. Пошто је ријеч о низу само ћемо итерирати до коријенског. Код у наставк у је за случај да се користи

#define LABELING\_V2.

    for (i = w\*h; i >= label\_counter; i--) {

        j = i;

        while (parent\_arr[j] != j) {

            j = parent\_arr[j];

        }

        parent\_arr[i] = j;

    }

## 5. Бојење слике

Након завршеног кодовања слике, бојење је једноставно. За различите вриједности пиксела, додијелимо различиту боју.

|  |
| --- |
| void colorImage(byte \* labeled\_im, byte\* colored\_im, uint32 w, uint32 h) {      uint32 num\_iter = w \* h;      uint32 counter = 1;      for (int i = 0; i < num\_iter; i++) {          counter = i \* 3;          //\*((int\*) (colored\_im + counter)) = colormap[labeled\_im[i]].number;          colored\_im[counter] = colormap[labeled\_im[i]].r;          colored\_im[counter+1] = colormap[labeled\_im[i]].g;          colored\_im[counter+2] = colormap[labeled\_im[i]].b;      }  } |

## 6. Оптимизација

За оптимизацију кориштене су методе као што је векторизација петљи, одмотавање петљи, те познавање величине слике ради олакшања компајлеру да оптимизује код.

У коду испод приказана је функција за конволуцију без икакве оптимизације, као и функција са одмотаном петљом. Побољшања у перформансама су итекако примјетна, поготово када се користи заједно са унапријед познатом величином слике.

#ifdef CONVOLUTION\_NO\_OPT

static inline uint32 convolution(byte \*pixels, int32 kernel[3][3], uint32 row, uint32 column, uint32 width)

{

    byte(\*pix\_mat)[width] = (byte(\*)[width])pixels;

    uint32 i, j, sum = 0;

    for (i = 0; i < 3; i++)

    {

        for (j = 0; j < 3; j++)

        {

            sum += kernel[i][j] \* pix\_mat[i + row - 1][j + column - 1];

        }

    }

    return sum;

}

#endif

#ifdef CONVOLUTION\_UNROLL

static inline uint32 convolution(byte \*pixels, int32 kernel[3][3], uint32 row,

        uint32 column, uint32 width) {

    byte (\*pix\_mat)[width] = (byte (\*)[width]) pixels;

    uint32 i, j, sum = 0;

    sum += kernel[0][0] \* pix\_mat[row - 1][column - 1];

    sum += kernel[0][1] \* pix\_mat[row - 1][1 + column - 1];

    sum += kernel[0][2] \* pix\_mat[row - 1][2 + column - 1];

    sum += kernel[1][0] \* pix\_mat[1 + row - 1][column - 1];

    sum += kernel[1][1] \* pix\_mat[1 + row - 1][1 + column - 1];

    sum += kernel[1][2] \* pix\_mat[1 + row - 1][2 + column - 1];

    sum += kernel[2][0] \* pix\_mat[2 + row - 1][column - 1];

    sum += kernel[2][1] \* pix\_mat[2 + row - 1][1 + column - 1];

    sum += kernel[2][2] \* pix\_mat[2 + row - 1][2 + column - 1];

    return sum;

}

#endif

## 7. Резултати

Слика резолуције 95x88:

|  |  |
| --- | --- |
|  |  |
|  |  |

*Слика 2. Оригинал, „црно-бијела“(DSP има неки проблем), детектоване ивице, обојена слика*

*Табела 1. Број циклуса по дијеловима алгоритма за слику 95х88.*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Слика 95х88 | Учитавање слике | Конверзија у црно-бијелу слику | Детекција ивица | Кодовање | Бојење | Упис слике | УКУПНО |
| Без оптимизације, прва верзија кодовања | 1,084,151 | 1,003,346 | 8,717,982 | 3,252,143 | 982,344 | 1,311,457 | 17,219,122 |
| Без оптимизације,  Друга верзија кодовања | 1,085,870 | 1,003,345 | 8,718,686 | 3,062,922 | 982,343 | 1,312,506 | 17,030,603 |
| Са оптимизацијом | 1,084,195 | 668,939 | 705,591 | 594,657 | 585,239 | 1,311,517 | 5,791,379 |
| Позната величина слике | 1,084,021 | 668,938 | 463,808 | 566,305 | 585,237 | 1,311,462 | 5,521,076 |

## 8. Упутство за употребу

У главној функцији су редом позиване функције које обрађују слику на задати начин. Да би се примијенила свака од оптимизација понаособ, потребно је на почетку фајла откоментарисати неки од следећих линија, а друге закоментарисати.

//-----------------------COMPILING------------------------

const char \* filename = "100x100.bmp";

//NO OPTIMIZATION

#define READ\_1

#define WRITE\_1

#define GRAY\_1

#define CONVOLUTION\_NO\_OPT

#define NORMALIZATION\_NO\_OPT

#define EDGE\_NO\_OPT

#define LABELING\_V1

#define COLOR\_IMAGE\_NO\_OPT

// //OPTIMIZED LABELING FUNCTION

// #define READ\_1

// #define WRITE\_1

// #define GRAY\_1

// #define CONVOLUTION\_NO\_OPT

// #define NORMALIZATION\_PRAGMA

// #define EDGE\_NO\_OPT

// #define LABELING\_V2

// #define COLOR\_IMAGE\_NO\_OPT

// //OPTIMIZED CONVOLUTION FUNCTION

// #define READ\_1

// #define WRITE\_1

// #define GRAY\_1

// #define CONVOLUTION\_UNROLL

// #define NORMALIZATION\_PRAGMA

// #define EDGE\_NO\_OPT

// #define LABELING\_V2

// #define COLOR\_IMAGE\_NO\_OPT

//KNOWN IMAGE SIZE

// #define KNOWN\_IMAGE\_SIZE\_READ

// #define GRAY\_1

// #define CONVOLUTION\_UNROLL

// #define KNOWN\_IMAGE\_SIZE\_EDGE\_OPTIMIZED

// #define KNOWN\_IMAGE\_SIZE\_LABELING\_OPTIMIZED

// #define KNOWN\_IMAGE\_SIZE\_NORMALIZATION

// #define WRITE\_1

// #define COLOR\_IMAGE\_NO\_OPT

//========================================================

//--------USED ONLY WHEN KNOW\_IMAGE\_SIZE IS DEFINED-------

#define H 100

#define W 100

//========================================================

За случај гдје желимо користи познату величину слике потребно је промијенити #define H, #define W у одговарајуће вриједности.

Што се тиче црно-бијеле слике, као што је наведено испод слике 2, наведен је чудан проблем са плочом и симулацијом.

    if (t == GRAY) {

        int zero = 0;

        int counter = 0;

        for (unsigned char i = 0; i < 256; i++) {

            fwrite(&counter, 1, 1, outputFile);

            fwrite(&counter, 1, 1, outputFile);

            fwrite(&counter, 1, 1, outputFile);

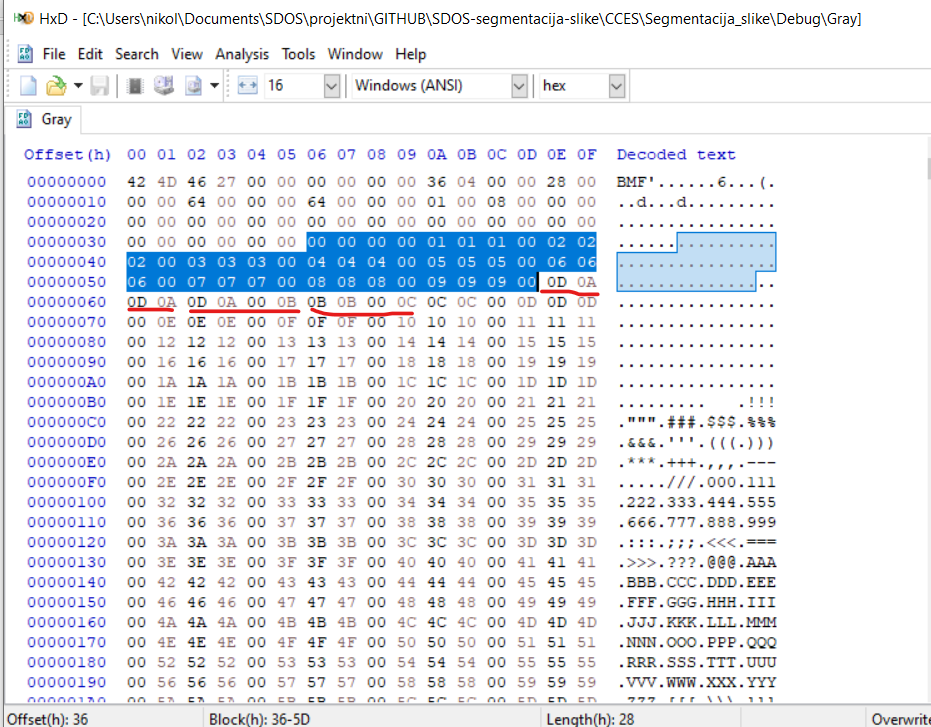
            fwrite(&zero, 1, 1, outputFile);

            counter = counter + 1;

        }

    }

Наиме овај дио кода треба да направи палету боја које се уписују у фајл. Оне морају да имају 256 вриједност тј од 0 до 255 редом. Као што можете да видите јако једноставна функција. Уписује 00 00 00 00,01 01 01 00,02 02 02 00,03 03 03 00 и тако све до FF FF FF 00. Е сада, када је у фајл потребно уписати 0А 0А 0А 00, деси се јако чудна ствар. Упише 0D 0A 0D 0A. Што је скорз погрешно, а можете и видјети у коду изнад да је јако једноставан код. Погрешно изгледа слика, али таква је каква је РАДИ, јер су у меморији исправни подаци.



Слика 3. Чудна ствар код уписа у фајл