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

Бања Лука, фебруар 2023

Аутор: Никола Цетић 1117/19

1. Учитавање слике

За учитавање слике на развојну плочу резервисано је 160KB у sram меморији,што је одговара 40к пиксела јер DSP може да адресира минимално 4 бајта, што је довољно за слике које ће бити обрађене у наставку.

```
//-----------------MEMORY MAPPING SECTIONS---------------
extern const uint32 uid;
#pragma section("pixels_3b")
static byte pixels_3b[45001];
int index_pixels_3b;
```

Да би се низ смјестио у sram меморију потребно је мапирати низ у .ldf.

```
    dxe_pixels_3b
      {
            INPUT_SECTIONS( $OBJECTS(pixels_3b) )
      } > mem_sram
```

Учитавање слике вршено је 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.

```
static void to_gray(byte * restrict pixels) {
      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;
            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:

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

Слика 1.

```
void sobel_edge_detector(byte * restrict pixels, byte ** out_pixels, uint32
width,
		uint32 height) {
	uint32 i, j, gx, gy;

	*out_pixels = pixels;		// assigning array to output pointer

	uint32 * restrict out_arr = (uint32 *) heap_malloc(0,
			width * height * sizeof(uint32));
	if (out_arr == NULL) {
		printf("Nije instancirana memorija\n");
		return;
	}
	*out_pixels = out_arr;		// assigning array to output pointer
	uint32 (*edged_pixels_mat_uint32)[width] =
			(uint32 (*)[width]) out_arr; // 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] = gx * gx + gy * gy;
		}
	}
	min_max_normalization(out_arr, width, height);

	heap_free(0,pixels);
}
```

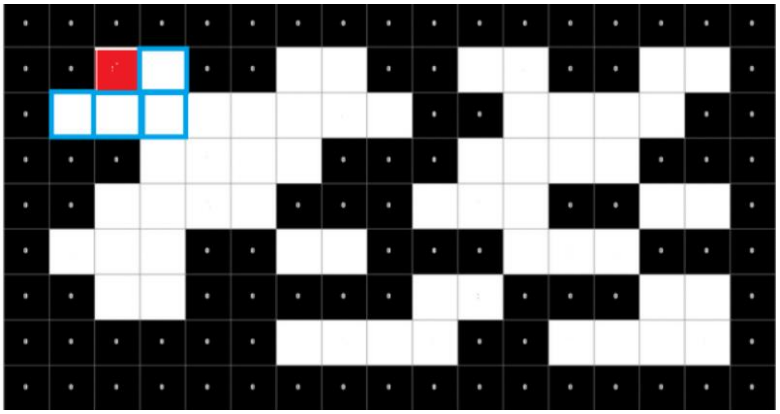
Приказана је функција која врши детекцију ивица. Примијетимо да се функција за конволуцију често позива. Оптимизација ове функције биће показана у наставку тачка 6.

4. Кодовање затворених контура

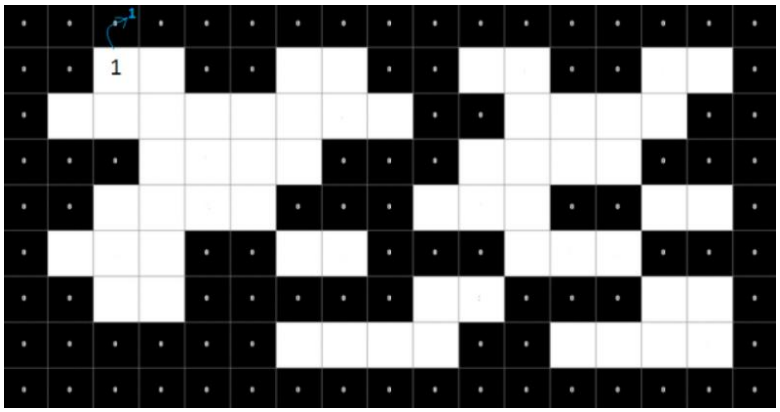
Кодовање затворених контура вршено је connected Component алгоритмом. Да би биле објашњене и оптимизације што се тичу самог алгоритма биће објашњен и сам алгоритам.

За рад овог алгоритма потребно је слику претворити у бинарну. У почетку алгоритма креирамо варијаблу која нам говори о броју лабела на слици, у почетку је то 1 јер сигурно имамо

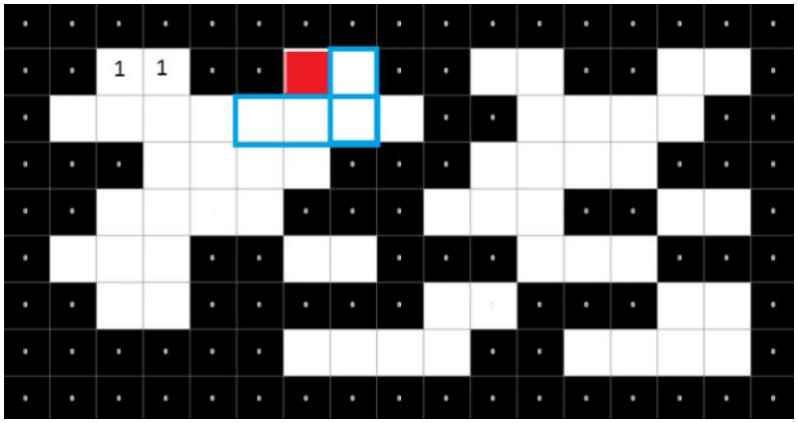
једну лабелу. Алгоритам се извршава тако што при наиласку на први неивични пиксе провјерава његове комшијске пикселе који нису ивице те одређује на овом примјеру закључује да ниједан комшијски пиксел лабелисан те додјељује себи вриједности коју је држала варијабла за тренутни број лабела на слици, а то је тренутно 1.



Слика 1. Примјер слике

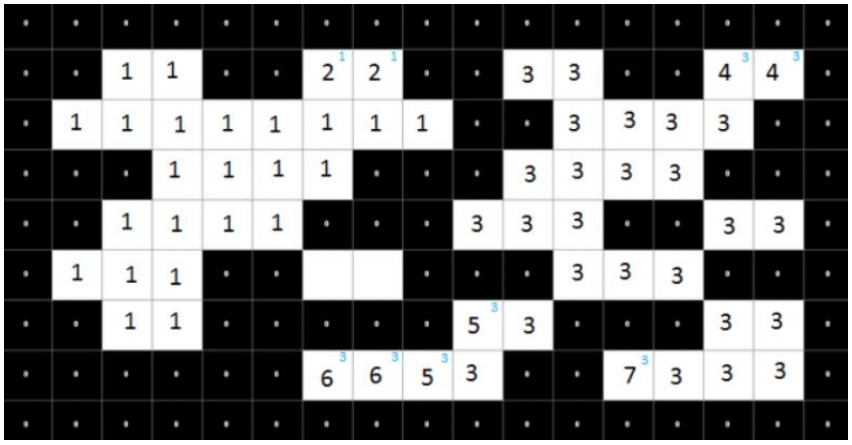


Слика 2. Први пиксел лабелисан



Слика 3. Случај када нема лабелисаних комшијских пиксела

Као што видимо на слици 3. Овај алгоритам провјерава да ли су и доњи пиксели лабелисани, што сигурно неће бити ако алгоритам ради одозго према доле. Самим тиме једна од оптимизација је избацивање те провјере.



Слика 4. Након попуњавања цијеле слике лабелама

Након што је слика попуњена са вриједностима лабела видимо да нпр лабеле 3 и 4 припадају истом објекту али имају различите вриједности лабеле те је потребно пронаћи родитељску лабелу. У овом случају је то лабела са мањој вриједношћу лабеле. Приликом извршавања овог алгоритма примјетно је да је потребно провјеравати да ли је пиксел ивица или није. У то случају урађено је побољшање алгоритма тако да се узме за родитељску лабелу пиксел са већом вриједношћу лабеле, а пошто смо ивице кодовали са 0 сигурно оне неће бити родитељске јер су најмање(Још једна од оптимизација).

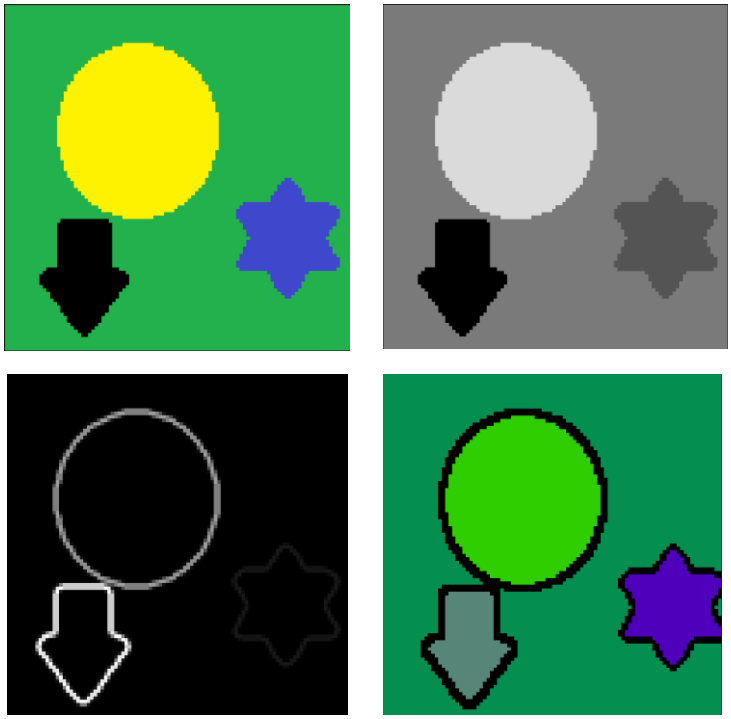
5. Бојење слике

Бојење слике се може вршити на 2 начина. Тако да се креира низ од по 3 компоненте RGB те преко лукап табеле боји. Други начин је много бржи. BMP формат у свом заглављу има сегмент у коме се уписује лукап табела за црно-бијелу слику која има само по 1 бајт за сваки пиксел. У коду је креирана лукап табела те је само усписана у крајњу слику. Нпр за слику 100x100 првим начином слика заузима 30МБ а другим 11МБ.

6. Оптимизација

За потребе демонстрације оптимизације биће приказани резултати за слику величине 100x100.

За почетак приказани су резултати за извршавање алгоритма без оптимизација. На учитавање и чување није био фокус током оптимизације. Те вријеме извршавања читавог програма укључује и учитавање и чување слике са сваког сегмента обраде, док сама обрада траје много краће.



Слика 5. Примјер слике 100x100

СЕГМЕНТ АЛГОРИТМА	БРОЈ ЦИКЛУСА
Конверзија у црно-бијелу слику	1 200 147
Детекција ивица	9 981 693
Кодовање	3 316 211

Као референтна тачка у табели су приказани резултати за случај без чак и компајлерске оптимизације оптимизације.

СЕГМЕНТ АЛГОРИТМА	БРОЈ ЦИКЛУСА
Конверзија у црно-бијелу слику	750 128
Детекција ивица	1 331 737
Кодовање	763 142

Након укључене компајлерске оптимизације за перформансе видимо много боље резултате. Али покушаћемо да их још побољшамо.

Као први вид оптимизације пришло се рјешавању проблема око детекције ивице. Примјећено је уско грло код извршавања функције конволуције. Покушано је пар начина рјешавања проблема. Први је одмотавање петље. Које је чак показало лошије резултате него без одмотавања. А ако погледамо у документацији од SHARC процесоре, није препоручено

ручно одмотавање него да се то препусти компајлеру. У наставку је приказан код за оба начина са и без одмотавања.

```
#ifdef CONVOLUTION_NO_OPT
uint32 convolution(byte * restrict 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
uint32 convolution(byte * restrict 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
```

СЕГМЕНТ АЛГОРИТМА	**БРОЈ ЦИКЛУСА**
Конверзија у црно-бијелу слику	750 129
Детекција ивица	1 696 670
Кодовање	763 123

Резултати након ручног одмотавања петље

Након неуспјешног одмотавања петље приступило се другом начину рјешавања а то је да векторизујемо вањску петљу а унутрашњу одмотамо. Видимо да смо овиме смањили број циклуса за 100к у односу на ону без икакве оптимизације.

СЕГМЕНТ АЛГОРИТМА	**БРОЈ ЦИКЛУСА**
Конверзија у црно-бијелу слику	750 129
Детекција ивица	1 197 262
Кодовање	763 123

Резултат након прагма директиве и одмотавања унутрашње петље

```
uint32 convolution(byte *restrict pixels, int32 kernel[3][3], uint32 row, uint32
column, uint32 width)
{
	byte(*pix_mat)[width] = (byte(*)[width])pixels;
	uint32 i, j, sum = 0;
	#pragma vector_for
	for (i = 0; i < 3; i++)
	{
		sum += kernel[i][0] * pix_mat[i + row - 1][column - 1];
		sum += kernel[i][1] * pix_mat[i + row - 1][1 + column - 1];
		sum += kernel[i][2] * pix_mat[i + row - 1][2 + column - 1];
	}
	return sum;
}
```

Ако погледамо у функцији за детекцију ивица видимо да се функција за конволуцију позива много пута, те би било боље да је прогласимо за INLINE како би била искомпајлирана на сваком мјесту позива, без непотребног позивања много пута.

```
#pragma inline
static uint32 convolution(byte * restrict pixels_a, int32 kernel[3][3], uint32
row, uint32 column, uint32 width_a)
{
	byte(*pix_mat)[width_a] = (byte(*)[width_a])pixels_a;
	int i, j;
	int sum = 0;
	for (i = 0; i < 3; i++)
	{
		sum += kernel[i][0] * pix_mat[i + row - 1][column - 1];
		sum += kernel[i][1] * pix_mat[i + row - 1][1 + column - 1];
		sum += kernel[i][2] * pix_mat[i + row - 1][2 + column - 1];
	}
	return sum;
}
```

СЕГМЕНТ АЛГОРИТМА	**БРОЈ ЦИКЛУСА**
Конверзија у црно-бијелу слику	750 129
Детекција ивица	544 140
Кодовање	763 123

Може се закључити да смо овиме много побољшали алгоритам чак више од 2 пута.

Даље можемо примијетити да бисмо могли искористити уграђене функције за проналажење минималног и максималног броја приликом нормализације. За разлику од коришћења гранања боље се окренути хардверској имплементацији. Ова функција је позивана

```
void min_max_normalization(uint32 *pixels, uint32 width, uint32 height) {
	uint32 min = INT_MAX, max = 0, i;
	uint32 length = height * width;
	for (i = 0; i < length; i++) {
		if (pixels[i] < min) {
			min = pixels[i];
		}
		if (pixels[i] > max) {
			max = pixels[i];
		}
	}
	double sub = max - min;
	double ratio;
	for (i = 0; i < length; i++) {
		ratio = (double) (pixels[i] - min) / sub;
		pixels[i] = ratio * 255;
	}
}

void min_max_normalization(uint32 * restrict pixels, uint32 width, uint32
height) {
	uint32 min_v = INT_MAX, max_v = 0, i;
	uint32 length = height * width;
	int value=0;
	for (i = 0; i < length; i++) {
		value = pixels[i];
		max_v = max(value,max_v); //hardware accelerated function
		min_v = min(value,min_v); //hardware accelerated function
	}
	double sub = max_v - min_v;
	double ratio;
	for (i = 0; i < length; i++) {
		ratio = (double) (pixels[i] - min_v) / sub;
		pixels[i] = ratio * 255;
	}
}
```

унутар функције за детекцију ивица као и за кодовање, тако да су примјетна побољшања у обе функције.

СЕГМЕНТ АЛГОРИТМА	БРОЈ ЦИКЛУСА
Конверзија у црно-бијелу слику	750 129
Детекција ивица	514 160
Кодовање	733 143

Резултат након хардверске оптимизације

Као што је речено у објашњењу алгоритма за кодовање ако измијенимо мало алгоритам можемо добити значајна побољшања у извршавању.

СЕГМЕНТ АЛГОРИТМА	**БРОЈ ЦИКЛУСА**
Конверзија у црно-бијелу слику	750 129
Детекција ивица	514 160
Кодовање	632 818

Резултати након измјене алгоритма

Побољшања су веома велика. Само избацивање додатне провјере услова.

```
Бржи код!!!
if (mat_val[i - 1][j - 1] > max_neighbour) {
      max_neighbour = mat_val[i - 1][j - 1];
}
if (mat_val[i - 1][j] > max_neighbour) {
      max_neighbour = mat_val[i - 1][j];
}


Спорији код!!!
if (mat_val[i-1][j-1] > EDGE_VAL && mat_val[i-1][j-1] < min_neighbour)
{
      min_neighbour = mat_val[i-1][j-1];
}
if (mat_val[i-1][j] > EDGE_VAL && mat_val[i-1][j] < min_neighbour)
{
      min_neighbour = mat_val[i-1][j];
}
```

Овде није крај. Ако узмемо претпоставку да ће на слици бити мање ивица него пиксела који припадају објекту можемо још побољшати овај алгоритам коришћењем директиве `expected_false` и `expected_true`.

СЕГМЕНТ АЛГОРИТМА	**БРОЈ ЦИКЛУСА**
Конверзија у црно-бијелу слику	750 129
Детекција ивица	514 160
Кодовање	589 772

Резултати коришћења `expected_false` и `expected_true` директива.

Ако узмемо да нам је позната величина слике коју обрађујемо у току компајлирања можемо добити чак и боље перформансе. Видимо да када компајлер познаје више података у вријеме компајлирања у стању је да изгенерише и ефикаснији код.

СЕГМЕНТ АЛГОРИТМА	**БРОЈ ЦИКЛУСА**
Конверзија у црно-бијелу слику	750 129
Детекција ивица	504 610
Кодовање	580 785

Резултати након познавања величине слике у току компајлирања

Приједози додатних побољшања:

- Конверзија слике би се могла реализовати коришћењем ДМА контролера за упис пиксела из sram меморије у интерну меморију уређаја.
- Могло се прибјећи коришћењу рачунања конволуције у фреквенцијском домену коришћењен акцелератора за брзу фуријеову трансформацију.

Напомена: Покушано је поштовање Мисра 2004 стандарда али због индексирања низа, који је у 2012-ој верзији поправљен, није било могуће испоштовати.

```
uint32 convolution(byte * restrict pixels, int32 kernel[3][3], uint32 row,
uint32 column, uint32 width)
{
	byte(*pix_mat)[width] = (byte(*)[width])pixels;/* ovde javi gresku*/
	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;
}
```

Од додатних функционалности које су приложене у задатку урађено је:

- Обрада слике различитих димензија
- Обрада било којих слика у заданом оквиру величине(Алгоритам у C на личном рачунару ради са било којом величином различитих комплексности)
- Визуелни рпиказ сегментације (Бојење поља умјесто њиховог кодовања)
- Приказ прогреса обраде коришћењем диода на плочи

7. Упутство за употребу

```
/*--------USED ONLY WHEN KNOW_IMAGE_SIZE IS DEFINED-------*/
#define H 100
#define W 100
/*=========================================================*/

/*-----------------------COMPILING-----------------------*/
extern const char * filename;

/*
#define READ_1 ||KNOWN_IMAGE_SIZE_READ
#define WRITE_1
#define GRAY_1
#define CONVOLUTION_NO_OPT || CONVOLUTION_UNROLL || CONVOLUTION_PRAGMA ||
CONVOLUTION_PRAGMA_INLINE
#define NORMALIZATION_NO_OPT || NORMALIZATION_HARDWARE ||
KNOWN_IMAGE_SIZE_NORMALIZATION
#define EDGE || KNOWN_IMAGE_SIZE_EDGE_OPTIMIZED
#define LABELING_V1 || LABELING_V2 || LABELING_V2_EXPECTED ||
KNOWN_IMAGE_SIZE_LABELING_OPTIMIZED
*/



/*NO OPTIMIZATION*/
#define READ_1
#define WRITE_1
#define GRAY_1
#define CONVOLUTION_NO_OPT
#define NORMALIZATION_NO_OPT
#define EDGE
#define LABELING_V1
#define COLOR_IMAGE_NO_OPT
```

Да би се овај пројекат користио потребно је са гитхаб платформе преузети сам пројекат и у фајлу Segmentacija_slike.h мијењати дефиниције по избору, које су горе наведене и коментарисане(Нису све комбинације тестиране). У фајлу Segmentacija_slike.c је потребно навести назив слике у filename. Могуће је овај код покренути у симулацији као и на самој платформи. Слике величине веће од 100x100 нису тестиране, тако да нису ни препоручене.

9. Литература

[1] Алгоритам повезаних компонената

[2] SharcCompilerLoops

[3] SharcCompilerGuidelines