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 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

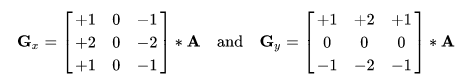
+ (**float**) pixels[pix\_position] \* 0.114);

}

}

## 3. Детекција ивица

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



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);

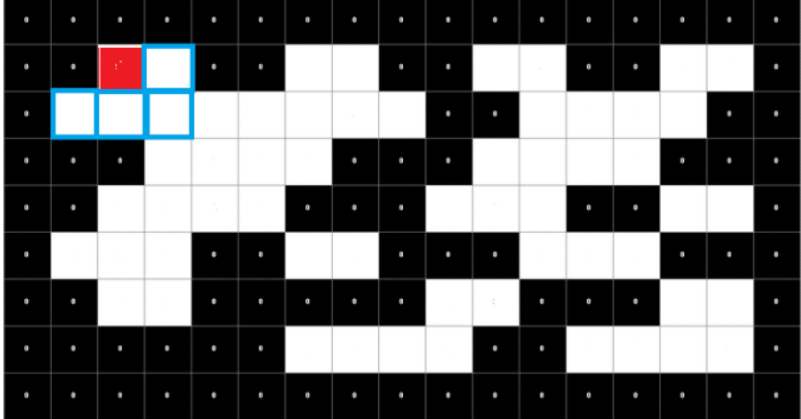
}

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

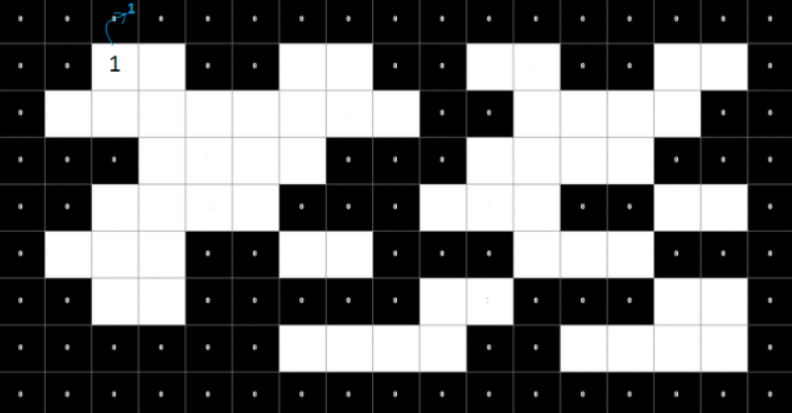
## 4. Кодовање затворених контура

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

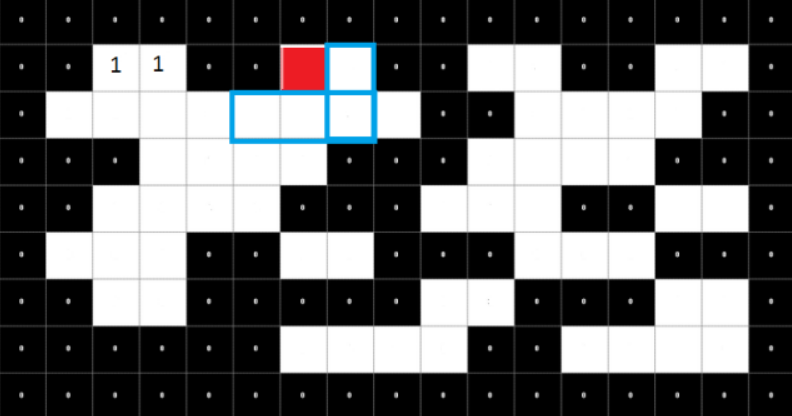
За рад овог алгоритма потребно је слику претворити у бинарну. У почетку алгоритма креирамо варијаблу која нам говори о броју лабела на слици, у почетку је то 1 јер сигурно имамо једну лабелу. Алгоритам се извршава тако што при наиласку на први неивични пиксе провјерава његове комшијске пикселе који нису 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

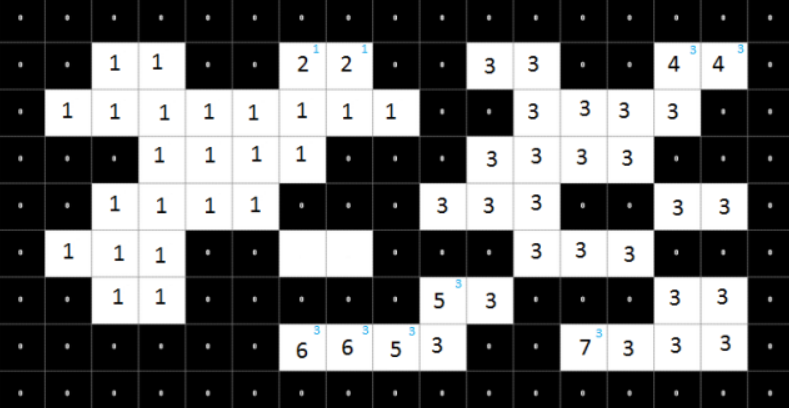


Слика 2. Први пиксел лабелисан



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 100х100 првим начином слика заузима 30МБ а другим 11МБ.

## 6. Оптимизација

За потребе демонстрације оптимизације биће приказани резултати за слику величине 100х100.

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 100х100

|  |  |
| --- | --- |
| **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 кориштења гранања боље се окренути хардверској имплементацији. Ова функција је позивана унутар функције за детекцију ивица као и за кодовање, тако да су примјетна побољшања у обе функције.

**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;

}

}

|  |  |
| --- | --- |
| **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 кориштењем 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 кориштења 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 кориштењем ДМА контролера за упис пиксела из sram меморије у интерну меморију уређаја.
* Могло се прибјећи кориштењу рачунања конволуције у фреквенцијском домену кориштењен 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 кориштењем 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 100х100 нису тестиране, тако да нису ни препоручене.

## 9. Литература

[1] [Алгоритам повезаних компонената](https://www.codeproject.com/Articles/336915/Connected-Component-Labeling-Algorithm)

[2] [SharcCompilerLoops](https://el.etfbl.net/pluginfile.php/70292/mod_resource/content/1/SharcCompilerLoops.pdf)

[3] [SharcCompilerGuidelines](https://el.etfbl.net/pluginfile.php/70293/mod_resource/content/1/SharcCompilerLoopGuidelines.pdf)