Електротехнички факултет

Бања Лука

ИЗВЈЕШТАЈ ПРОЈЕКТНОГ ЗАДАТКА

Системи за дигиталну обраду сигнала

Студент:

Бојан Божић 11123/18

Предметни наставници:

проф. др Младен Кнежић

проф. др Митар Симић

Фебруар, 2023

1. Опис проблема

У склопу пројектног задатка је потребно реализовати систем за манипулацију сликом на развојном окружењу *ADSP-21489*. Систем се састоји из три основна дијела:

- Дио у ком се врши учитавање слике на *ADSP-21489* развојну платформу. Слика која се обрађује ће бити доступна у *.bmp* формату. Потребно је из слике у том формату извући појединачне информације о боји пиксела и тако учитану слику претворити у *gray scale* слику.
- Дио у ком се врши детекција ивица на учитаној слици. Алгоритам за детекцију ивица и његове параметре изабрати на оптималан начин и образложити у извјештају зашто је баш то урађено. Алгоритам треба успјешно да детектује ивице на свим приложеним тестним сликама. Излаз из алгоритма за детекцију ивица резултује новом сликом на којој је могуће идентификовати затворене контуре.
- Дио у ком се врши бојење објеката детектованих у претходном кораку. Извршити кодовање сваког пиксела унутар детектованих затворених контура тако да пикселима који припадају истој контури буде додијељена иста вриједност. Слику са идентификованим ивицама и кодовану слику сачувати у *.bmp* формату.

2. Реализација пројектног задатка

Учитавање слике на *ADSP-21489* развојну платформу је реализовано у оквиру *Cross Core Embedded Studio* развојног окружења. Подаци о учитаној слици који се налазе у заглављу слике чувају се у структури *bmp_header*. Учитавање слике се врши помоћу функције *read_image*.

```
25  typedef struct {
26      int type;
27      int size;
28      int reserved1;
29      int reserved2;
30      int offset;
31      int header_size;
32      int width;
33      int height;
34      int planes;
35      int bits_per_pixel;
36      int compression;
37      int image_size;
38      int x_pixels_per_meter;
39      int y_pixels_per_meter;
40      int colors_used;
41      int colors_important;
42  } bmp_header_t;
```

Слика 1. Приказ *bmp_header* структуре

Архитектура *ADSP-21489* развојне платформе је таква да типове промјењивих као што су *char* и *short* посматра као четворобајтне податке. Из тог разлога, да би били у могућности да читамо дијелове меморије мање од 4 бајта, слику отварамо као текстуалну датотеку и читамо бајт по бајт садржаја слике помоћу функције *fgetc( )*.

Прочитани подаци из заглавља се одговарајућим логичким операцијама и шифтовањем смјештају у одговарајуће промјењиве.

Након завршеног читања заглавља, приступа се читању података о пикселима, односно података о његовим *RGB* компонентама. Читање вриједности компонената се такође врши бајт по бајт помоћу функције *fgetc( )*. С обзиром на то да се свака компонента може представити са 1 бајтом, ради оптимизације меморије један пиксел ћемо посматрати као један четворобајтни податак у који ћемо помоћу логичких операција и операције шифтовања смјештати *RGB* компоненте.

```
for(i = 0; i < (bmp_header.image_size)/3; i++){
    b1 = fgetc(in);
    b2 = fgetc(in);
    pixels[i] = ((fgetc(in) << 16) | (b2 << 8)) | b1;
}
```

Слика 2. Приказ креирања низа пиксела

Gray scale верзија учитане слике се добија на једноставан начин примјеном формуле:

$$gray\ scale\ pixel = 0.3 * R + 0.59 * G + 0.11 * B$$

гдје су са *R*, *G* и *B* означене компоненте пиксела који претварамо у *gray scale* верзију. Затим се израчуната вриједност додијели свакој компоненти пиксела.

```
for (rgbIdx = 0; rgbIdx < (bmp_header.image_size/3); rgbIdx++) {
    temp = (pixels[rgbIdx] & 0x000000FF) * 0.3 + ((pixels[rgbIdx] >> 8) & 0x000000FF) * 0.59 + ((pixels[rgbIdx] >> 16) & 0x000000FF) * 0.11;
    pixels[rgbIdx] = 0;
    pixels[rgbIdx] = temp;
    pixels[rgbIdx] = (pixels[rgbIdx] << 8) | temp;
    pixels[rgbIdx] = (pixels[rgbIdx] << 8) | temp;
}
```

Слика 3. Приказ претварања пиксела у *gray scale*

За детекцију ивица могуће је користити више врста алгоритама. Неки од постојећих су *Kirsch*-ов, *Sobel*-ов и *Prewitt*-ов. Сва три наведена алгоритма имају исти принцип. Да би детектовали ивице потребно је извршити конволуцију одговарајућег кернела за сваки алгоритам са *gray scale* сликом. *Sobel*-ов алгоритам је алгоритам који детектује ивице у различитим правцима и има ниске временске захтјеве за вршење рачунских операција. За детекцију ивице се врше двије конволуције два кернела са сликом. Један кернел служи за детекцију вертикалних ивица а други за детекцију хоризонталних ивица. Резултујућа мапа ивица се добија комбинацијом ове двије мапе ивица. *Kirsch*-ов алгоритам је сложенији алгоритам који користи 8 различитих кернела за детекцију ивица у више смјерова. Добар је у детекцији ивица које су под угловима таквим да нису поравнате са хоризонталном и вертикалном осом. Има

доста веће временске захтјеве у односу на *Sobel*-ов алгоритам. *Prewitt*-ов алгоритам је алгоритам сличан *Sobel*-овом алгоритму, али користи другачије конволуционе кернеле. Може бити мање прецизан у односу на остале алгоритме у неким случајевима. За израду овог пројектног задатка кориштен је *Sobel*-ов алгоритам.

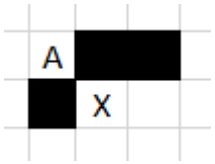
```
#pragma section("seg_sdram1")
int mask_x[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1} };

#pragma section("seg_sdram1")
int mask_y[3][3] = { {-1, -2, -1}, {0, 0, 0}, {1, 2, 1} };
```

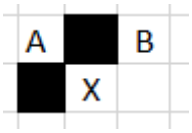
Слика 4. Дефинисање кернела потребних за детекцију ивица

У функцији *convolve* се врши описани процес детекције ивица за све пикселе осим рубних. Из тог разлога се након завршетка конволуција врши обрада рубних пиксела. Како је потребно да они буду црни, са двије *for* петље се пролази кроз рубне пикселе и њихова вриједност поставља на вриједност бијеле боје јер се након тога позива функција *black_and_white* која ће проћи кроз све пикселе и поставити њихову вриједност на вриједност бијеле или црне боје. Ако је пиксел претходно био сив или црн он ће након ове функције постати бијел, а ако је био бијел постаће црн. На овај начин добијамо црне пикселе на мјестима гдје су ивице и руб слике, а бијеле пикселе на свим осталим мјестима.

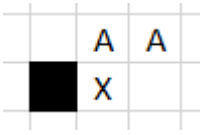
Бојење, односно кодовање детектованих објеката бојом се врши у оквиру функције *code_pixels*. Функција пролази кроз све пикселе и примјењује следећи алгоритам. Провјеравамо да ли је тренутно посматрани пиксел црне боје, ако јесте у питању је или детектована ивица или руб слике па тај пиксел остаје црне боје. Уколико је тренутни пиксел бијеле боје, потребно је провјерити које су боје његови сусједни пиксели који су већ обрађени и у зависности од њихове вриједности промијенити вриједност тренутно посматраног пиксела. На наредним сликама биће приказане стања сусједних пиксела које се провјеравају и од којих зависи вриједност коју ћемо уписати у тренутно посматрани пиксел.



Слика 5.1



Слика 5.2



Слика 5.3

Сва три наведена примјера представљају случај када је претходни пиксел у односу на тренутни посматрани пиксел црне боје, уколико није црн тренутно посматрани пиксел ће имати вриједност претходног пиксела. На слици 5.3 имамо примјер када је претходни пиксел црн а пиксели изнад тренутно посматраног су неке боје A, у овом случају тренутно посматрани пиксел (означен са X на све три слике) узима вриједност пиксела изнад себе, тј вриједност A. На слици 5.2 имамо примјер када је претходни пиксел црне боје као и пиксел изнад тренутно посматраног пиксела. У

том случају тренутно посматрани пиксел (пиксел X) добиће вриједност првог наредног пиксела (боја B) у односу на пиксел изнад њега, под претпоставком да он није црне боје. На слици 5.1 видимо трећи случај када је претходни пиксел, као и пиксел изнад и његов наредни пиксел црне боје. Тренутно посматрани пиксел тада добија нову вриједност, тј вриједност промјењиве *codeValue* чија је почетна вриједност задана хексадецималним записом 0x000000A0 која се, када се следећи пут појави потреба за другом бојом мијења шифтовањем у лијево за 5 бита. Ова вриједност је изабрана произвољно.

Упис нове слике, након детектовања ивица и бојења, врши се на сличан начин као и учитавање слике на развојну платформу. Креира се нова слика као текстуална датотека ради уписивања бајт по бајт а затим се позове функција *write_image* која коришћењем одговарајућих логичких операција и шифтовања уписује прво заглавље *bmp* слике а након тога и обрађене пикселе.

Низови који се користе за реализацију пројектног задатка *pixels* и *outputPixels* смјештени су у кориснички креирану секцију заједно са кернелима за реализацију *Sobel*-овог алгоритма.

```
#pragma section("seg_sdram1")
unsigned int pixels[250000];

#pragma section("seg_sdram1")
unsigned int outputPixels[250000];

#pragma section("seg_sdram1")
int mask_x[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1} };

#pragma section("seg_sdram1")
int mask_y[3][3] = { {-1, -2, -1}, {0, 0, 0}, {1, 2, 1} };
```

Слика 6. Приказ смјештања низова у кориснички креирану меморијску секцију

```
dxe_seg_sdram1
{

    INPUT_SECTIONS( $OBJECTS(seg_sdram1) )

} > mem_sram
```

Слика 7. Приказ кориснички креиране секције у датотеци *app.ldf*

3. Резултати

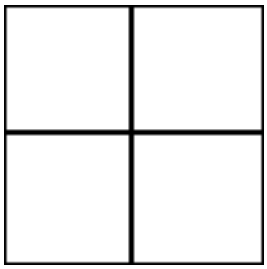
Као тестне слике коришћене су наредне двије слике:

Слика 8. img.bmp

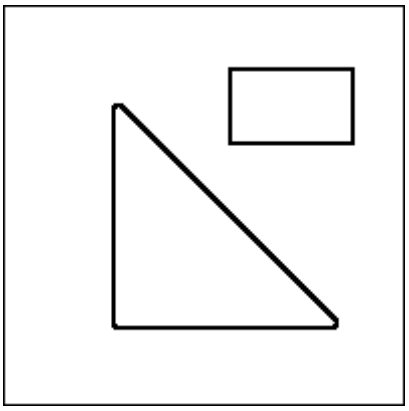
Слика 9. img2.bmp

Приказане слике су доступне у Debug директоријуму у оквиру пројекта који се налази на гит репозиторијуму.

Након *gray scale* обраде и детекције ивица добијамо наредне слике:

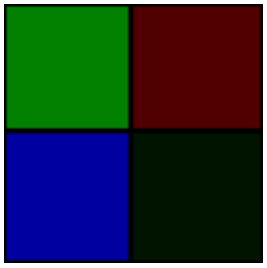


Слика 10. img.bmp након детекције ивица

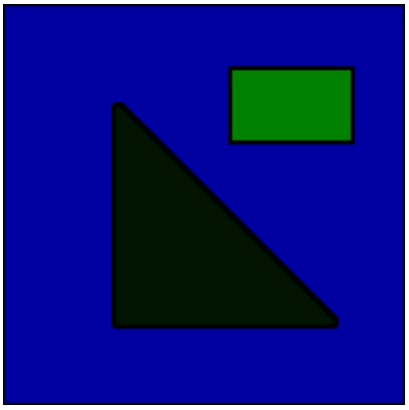


Слика 11. img2.bmp након детекција ивица

Након кодовања претходних слика бојом добијамо наредне слике:



Слика 12. img.bmp након кодовања бојом



Слика 13. img2.bmp након кодовања бојом

На основу добијених резултата можемо закључити да реализовани алгоритам даје добре резултате на тестираним сликама.

4. Оптимизација ресурса

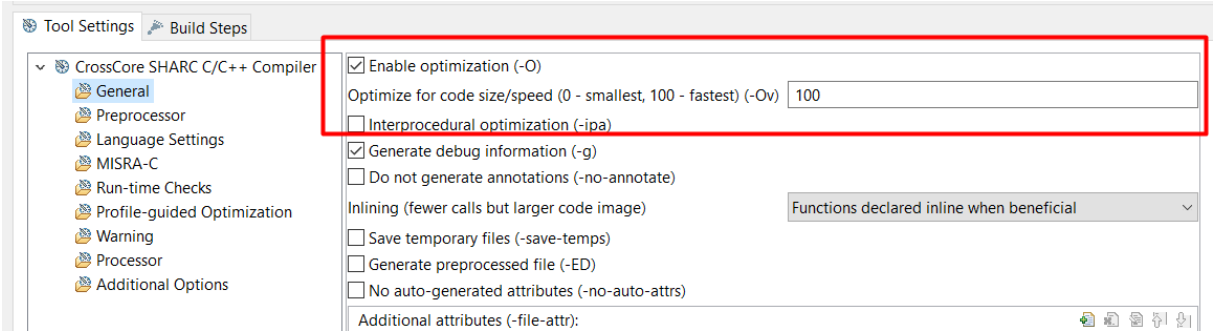
За праћење броја циклуса потребних да се изврше све фазе обраде слике користимо заглавље *cycle_count*. Ово заглавље садржи два макора којим се омогућава приступ садржају *EMUCLK* регистра, који приказује број извршених циклуса: *START_CYCLE_COUNT(S)* и *STOP_CYCLE_COUNT(T, S)* где су параметри *S* и *T* типа *cycle_t*. Дефинисан је и макро за испис промјењиве типа *cycle_t: PRINT_CYCLES(String,T)*.

За случај без оптимизације број потребних циклуса за извршење појединачних фаза обраде износи:

- Учитавање слике – 2 041 511 циклуса
- *Gray scale* – 2 100 017 циклуса
- *Sobel*-ов алгоритам – 8 562 220 циклуса
- Црно бијела трансформација – 748 398 циклуса
- Упис слике са ивицама – 3 269 537 циклуса
- Кодовање бојом – 1 524 199 циклуса
- Упис кодоване слике – 3 269 559 циклуса

Примјећујемо из резултата да највише ресурса заузима алгоритам за детекцију ивица, што је и очекивано јер алгоритам врши двије конволуције и корекцију рубних пиксела слике.

У наредном случају омогућили смо оптимизацију по брзини:



Слика 14. Омогућење оптимизације по брзини

Исти ефекат би постигли коришћењем претпроцесорске директиве:

#pragma optimize_for_speed

Сада након покретања имамо наредне резултате за број потребних циклуса:

- Учитавање слике – 1 841 439 циклуса
- *Gray scale* – 566 684 циклуса
- *Sobel*-ов алгоритам – 2 600 465 циклуса
- Црно бијела трансформација – 500 034 циклуса
- Упис слике са ивицама – 3 002 634 циклуса
- Кодовање бојом – 1 434 916 циклуса

- Упис кодоване слике – 3 002 637 циклуса

Примјетимо сада значајно смањење броја циклуса потребно за извршавање појединих фаза обраде слике. Највише примјетна промјена је број циклуса потребан за детекцију ивица који се са нешто више од 8 500 000 циклуса смањио на око 2 600 000 циклуса.

У наредном случају омогућићемо претпроцесорску директиву:

#pragma no_vectorization

На овај начин дајемо наредбу компајлеру да не врши векторизацију петље јер паралелно извршавање више од једне итерације у петљи може да успори извршавање петљи са веома малим бројем итерација.

Сада након покретања имамо наредне резултате за број потребних циклуса:

- Учитавање слике – 2 041 653 циклуса
- *Gray scale* – 2 100 013 циклуса
- *Sobel*-ов алгоритам – 8 562 221 циклуса
- Црно бијела трансформација – 748 404 циклуса
- Упис слике са ивицама – 3 269 510 циклуса
- Кодовање бојом – 1 524 201 циклуса
- Упис кодоване слике – 3 269 546 циклуса

Видимо да нисмо добили побољшање у односу на случај без оптимизације.

У наредном случају омогућићемо претпроцесорску директиву:

#pragma SIMD_for

Сада након покретања имамо наредне резултате за број потребних циклуса:

- Учитавање слике – 2 041 641 циклуса
- *Gray scale* – 2 100 014 циклуса
- *Sobel*-ов алгоритам – 8 562 222 циклуса
- Црно бијела трансформација – 748 402 циклуса
- Упис слике са ивицама – 3 269 551 циклуса
- Кодовање бојом – 1 524 204 циклуса
- Упис кодоване слике – 3 269 523 циклуса

Као и у слуају без векторизације видимо да нисмо добили ниаква побољшања у овом случају у односу на случај без оптимизације.

5. Приједлог побољшања

Тренутна реализација овог задатка је ограничена меморијом развојне платформе и може обрађивати слике чија је величина, односно производ ширине и висине, мања од 500 000. Могућност обраде слика већих димензија би се могла постићи блоковским учитавањем дијелова слике, затим обрадом учитаног блока, чувањем обрађеног блока у излазну слику и након тога понављање поступка са наредним блоком, све док се не обради цијела слика.

6. Литература

1. *“Image processing in C”, Dwayne Phillips*