УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
Трг Доситеја Обрадовића 6, 21000 Нови Сад, Република Србија
Деканат: 021 6350-413; 021 450-810; Централа: 021 485 2000
Рачуноводство: 021 458-220; Студентска служба: 021 6350-763
Телефакс: 021 458-133; e-mail: ftndean@uns.ac.rs
ИНТЕГРИСАНИ СИСТЕМ МЕНАЏМЕНТА СЕРТИФИКОВАН ОД:

ДОКУМЕНТАЦИЈА ЗА ПРЕДМЕТ: ЕМБЕДЕД ОПЕРАТИВНИ СИСТЕМИ

ТЕМА ПРОЈЕКТА:

Детекција слободних паркинг места применом функција за обраду слике

ПРОЈЕКАТ ИЗРАДИЛЕ:

Група 3: Ивана Гордић ЕЕ 23/2020
Ана Мокан ЕЕ 37/2020
Нина Антић ЕЕ 47/2020

Садржај

1. Алгоритам рада

1.1 Уводни део

Тема пројекта је проналажење и маркирање слободних паркинг места на видеу надзорне камере паркинга. Такође, омогућено је праћење броја заузетог у односу на укупан број паркинг места. Рад система заснован је на примени различитих функција за обраду слике где су у средишту *Gaussian Blur* и *Adaptive Threshold* као методе обраде слике модификацијом вредности на нивоу пиксела.

1.2 Главне функције

1.2.1 *Gaussian Blur*

Ефекат замућења, *Image blurring* постиже се конволуцијом оригиналне слике са матрицом (кернелом) нископропусног филтра. Користи се за одстрањивање садржаја високе фреквенције, попут шума и ивица. *Gaussian blur* је један специфичан тип филтрирања слике где се за конволуцију користи *Gaussian kernel* који на основу подешавања одређених параметара (величине кернела, стандардне девијације - сигма) добија коефицијенте израчунате Гаусовом функцијом.

Коефицијенти кернела прате Гаусову расподелу, што значи да ће средиште квадратне матрице имати највећу тежину а крајеви мању. Подешавање вредности сигме одређују колико ће се постепено одвијати тај прелаз, а тиме и колико ће "јак" утицај имати суседни пиксели на онај који је тренутно у фокусу, док величина матрице одређује површину суседства које узимамо у обзир при обрађивању тренутног пиксела.

1.2.2 *Adaptive Threshold*

Функција која зависно од вредности прага компарације додељује пикселу одређену вредност - црно или бело. Код *Adaptive Thresholding*-а вредност прага (*threshold*) није унапред дефинисана већ се израчунава у односу на неки мањи регион слике коју обрађујемо и пикселе суседне оном у фокусу. За наше потребе коришћен је Гаусов тип израчунавања прага функције.

1.3 Опис алгоритма

Програм за одређивање броја слободних паркинг места се састоји из неколико делова.

Део програма написан је помоћу *python* програмског језика. Овај део узима фрејмове улазног видеа, фрејм, односно фотографију конвертује у црно-белу фотографију. Ова црно-бела фотографију представља улаз самог *C++* алгоритма.

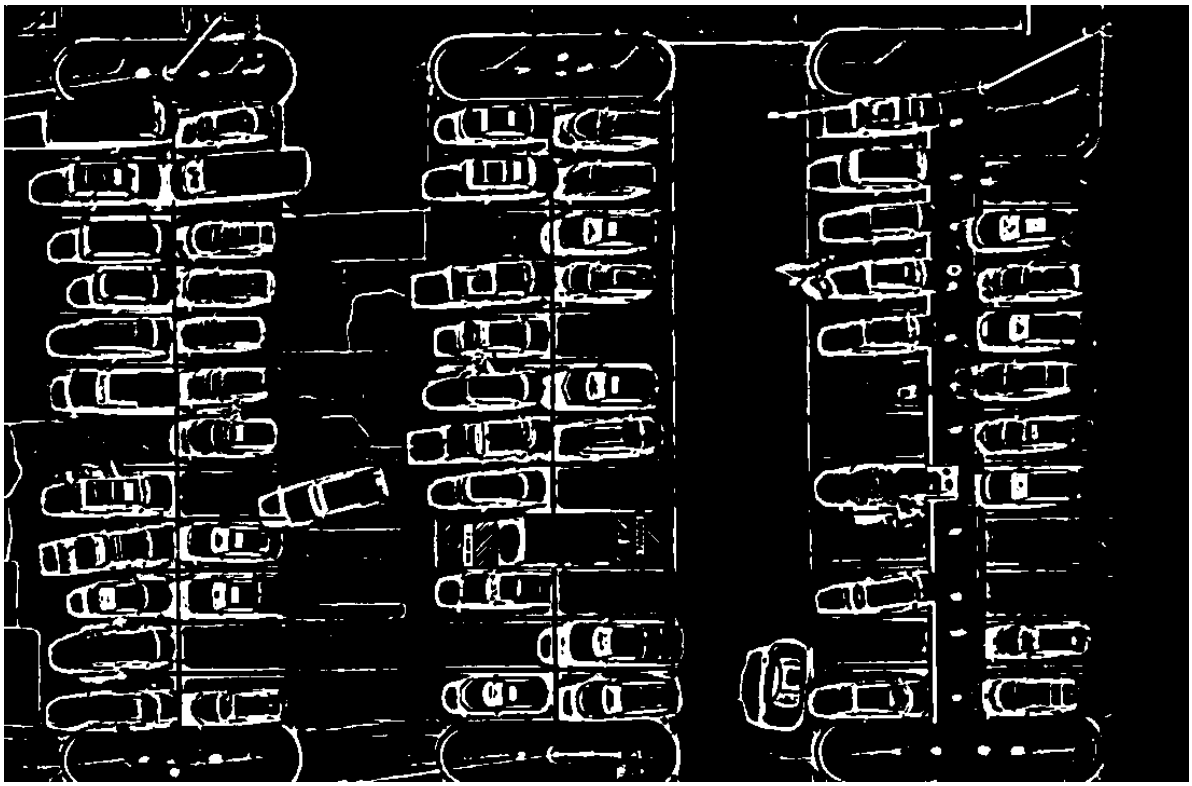
Део кода написан *python* програмским језиком такође прима и број слободних паркинг места (излаз *C++* алгоритма), те врши њихово обележавање на фотографији и приказује поменуту фотографију Овај део кода није био укључен у профајлирање перформанси.

Сам *C++* алгоритам за проналазак слободних паркинг места се састоји из следећих сегмената:

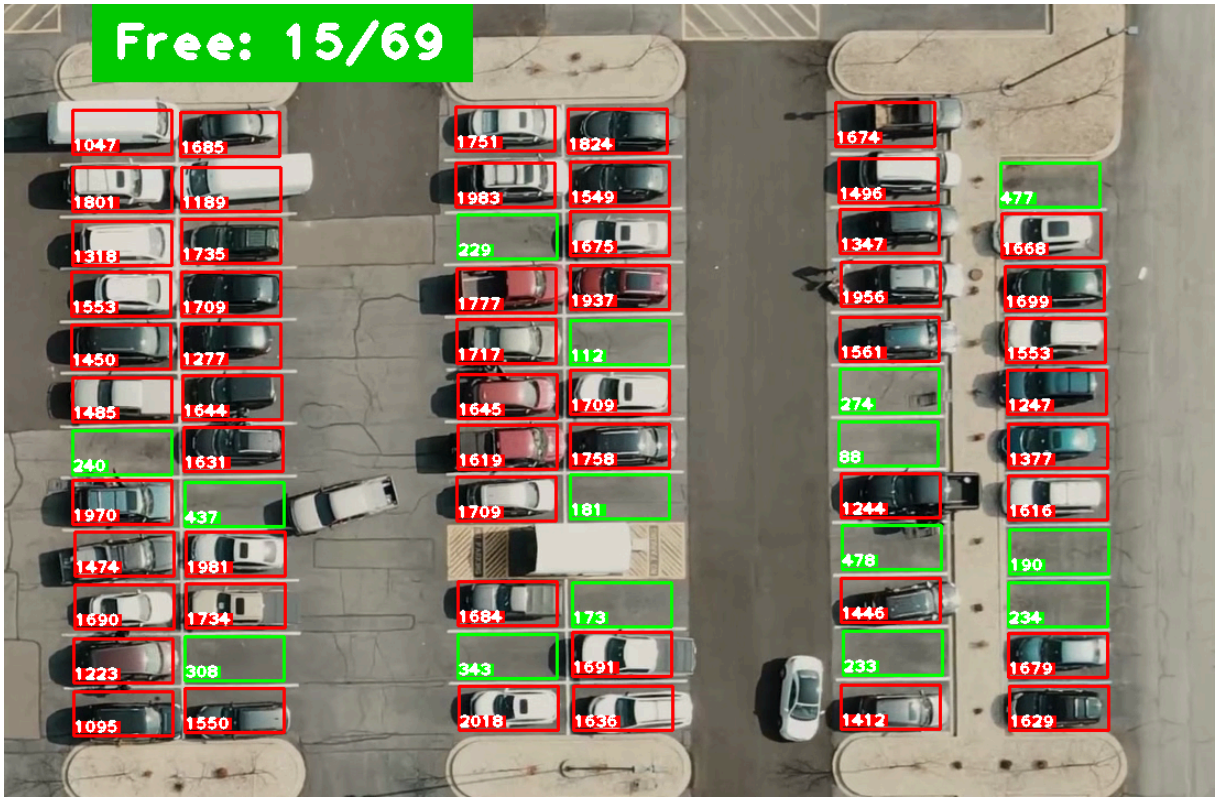
1. Стварање Гаусовог кернела за замућење фотографије уз помоћ Гаусове функције, врши се два пута: први пут са кернелом величине 3 пута 3, а затим са величином кернела од 15 пута 15, до 25 пута 25, пропорционално величини фотографије (функција ***gaussianBlur***).
2. Примена Гаусовог кернела на фотографију, односно замућење фотографије, (функција ***applyGaussian***), позиви ове функције врше се унутар функције ***gaussianBlur,***
3. Одређивање вредности прага у односу на који пиксели фотографије постају или црни или бели, врши се у односу на вредности пиксела у околини одређеној величином кернела при другом замућивању фотографије (функција ***adaptiveThreshold***); други позив ***gaussianBlur*** функције одиграва се унутар ***adaptiveThreshold*** функције,
4. Проширивање области фотографије на којима се налазе бели пиксели (функција ***dilateImage***),
5. Проналазак слободних паркинг места, (функција ***checkParkingSpace***), унутар ове функције се врши отварање текстуалног фајла са координатама паркинг места, издвајање делова фотографије означених тим координатама позивом функције ***cropImage***. Затим се врши пребројавање белих пиксела на издвојеним деловима фотографије функцијом ***countNonZero*** и поређење броја белих пиксела са прослеђеном граничном вредношћу како би се одредило да ли је паркинг место слободно. Ова гранична вредност представља потребан број белих пиксела у оквиру једног паркинг места, да би се оно сматрало заузетим. Она се утврђује експериментално и зависи од величине слике.

Функција ***checkParkingSpace*** враћа низ нула и јединица, где јединице означавају заузета паркинг места, а нуле слободна паркинг места. Овај низ представља излаз алгоритма.

Слика 1. Изглед фотографије након замућења Гаусовим кернелом 25x25

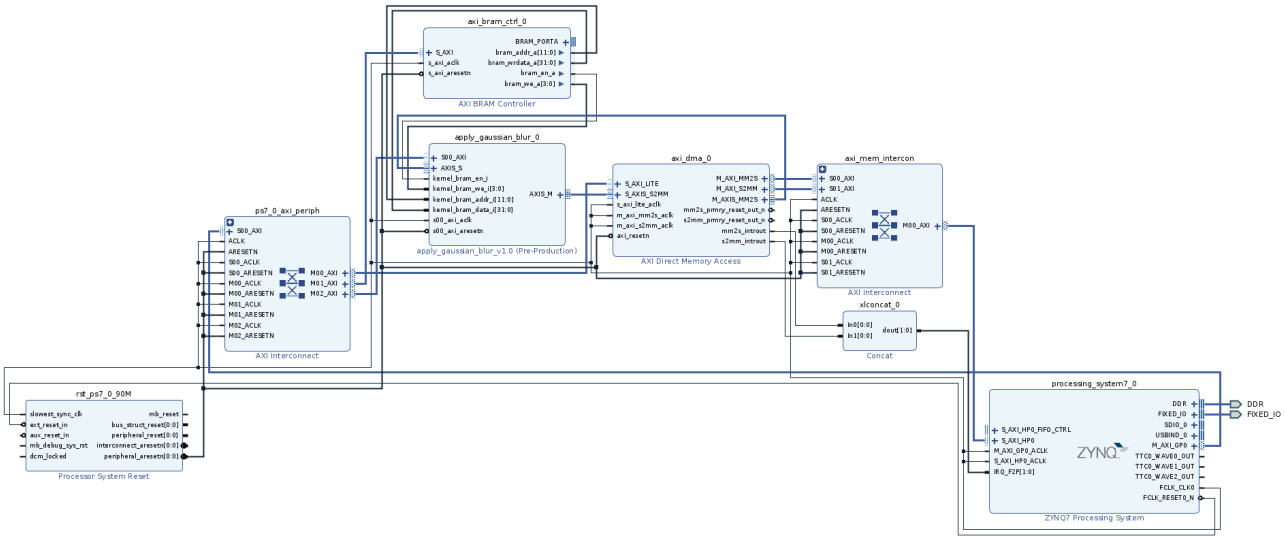


Слика 2. Изглед фотографије након примене прага у оквиру функције adaptiveThreshold



Слика 3. Приказ слободних паркинг места у python скрипти

2. Спецификација хардвера



Слика 4. Блок схема система након повезивања унутар алата Vivado IP Integrator

Компоненте које чине интегрисани систем овог пројекта су: наше *IP* језгро, *apply_gaussian_blur*, које примењује Гаусов кернел на фотографију у циљу замагљивања фотографије, *Zynq 7 Processing System, AXI DMA (Direct Memory Access)* и *Axi Bram Controller*.

Да би се омогућила комуникација процесора са *IP*-ем, успостављена је *Axi Lite* магистрала која служи за упис и читање регистара за стартовање, ресетовање и провере завршености рада (*ready*) *IP*-а, као и за подешавање димензије кернела.

РЕГИСТАР	АДРЕСА
start - slv_reg0	0
kernel_size - slv_reg1	4
restart - slv_reg2	8
ready - slv_reg3	12

Табела 1. Преглед Axi Lite регистара

AXI Bram Controller служи за уписивање вредности Гаусовог кернела у *Kernel BRAM*, који је подкомпонента *apply_gaussian_blur IP*-а.

DMA служи за упис пиксела фотографије коју је потребно обрадити у *Img BRAM,* који је такође подкомпонента *apply_gaussian_blur IP*-а. Такође служи и за упис пиксела обрађене, резултантне фотографије из *IP*-а у *DDR* меморију. Потребно је конфигурисати *DMA*, што се ради путем *Axi Lite* магистрале која спаја *DMA* и процесор. При овој конфигурацији, уписују се почетне адресе *DDR* меморије са које је потребно прочитати улазну слику и на коју је

потребно уписати излазну слику. Такође, уписује се и величина *DMA* трансфера у бајтовима, односно колико бајтова након почетне адресе је потребно прочитати или уписати.

3. Развој драјвера

У овом одељку биће описане основне структуре коришћене при имплементацији драјвера као и функција за рад са фајловима, којим се посредује пренос података између корисничког и кернел простора. Кернел простор обухвата рад са хардвером што у нашем случају представља процесор за обраду слике (IP блок). У оквиру хардвера издвајају се три целине са којима софтвер треба да размењује информације: IP језгро (*apply_gaussian*), BRAM меморија за скалдиштење коеф кернела (*kernel_bram*) и *DDR* меморија којој се приступа директно(*axi_dma*).

Структура platform_driver

Основна је иницијализациона структура хардверског система. Састоји се од више поља где треба издвојити *.driver* што чини засебну структуру помоћу које повезујемо уређаје са њиховим називом (*.compatible* поље). *.probe* и *.remove* поља представљају њима придружене функције, које се позивају приликом убацивања модула и служе за проверу и одбацивање поменутих уређаја.

```
// My driver struct
static struct platform_driver my_driver = {
    .driver =
        {
                .name = DRIVER_NAME,
                .owner = THIS_MODULE,
                .of_match_table = gaussian_of_match,
        },
        .probe = gaussian_probe,
        .remove = gaussian_remove,
};
```

Слика 5. Структура platform_driver

Додавање и брисање новог кернел модула

gaussian_probe извршава иницијализацију и алокацију потребних ресурса за рад са уређајем. Врши се редом: резервисање меморије за тражени *device,* добављање физичког меморијског опсега како би се резервисао потребан адресни опсег у кернел простору, затим се врши мапирање на виртуалне адресе. У конкретном случају потребно је применити *probe* на тру уређаја која се разликују по *minor* броју. Додатно, за *axi_dma* потребно је преузети информацију о броју прекида како би се могла регистровати прекидна рутина. *gaussian_init* функција позива се приликом додавања модула у кернел. Служи за алоцирање управљачких бројева које додељујемо сваком од уређаја из хардвера са којима ће се радити. Затим се стварају “node” фајлови унутар /dev директоријума за сваки од уређаја. Сваки уређај се креира помоћу *device_create* функције унутар које се као параметар прослеђује *struct class,* међутим, како више уређаја користи исти драјвер користи се једна класа. Када се наиђе на постојећу вредност *.compatible* поља из стабла уређаја позива се *gaussian_probe* функција. *gaussian_remove* и gaussian_exit су инверзне функције функцијама *gaussian_probe* и *gaussian_init* где је потребно ослободити заузете ресурсе и избрисати додат кернел модул.

Структура за операције са фајловима

Како је Линукс мото “све је фајл”, потребно је омогућити да се када кориснику ради неку операцију са фајлом, то манифестује неком изменом у хардверу у кернел простору. Након успешног додавања модула унутар директоријума /dev видимо следеће фајлове: *apply_gaussian, kernel_bram, axi_dma.*

```
// File ops
static struct file_operations my_fops =
{
    .owner = THIS_MODULE,
    .open = gaussian_open,
    .release = gaussian_close,
    .read = gaussian_read,
    .write = gaussian_write,
    .fasync = gaussian_fasync,
    .mmap =(void *)gaussian_dma_mmap //npr kao void *
};
```

Слика 6. Структура file_operations

Имплементиране су следеће функције:

- ***gaussian_open, gaussian_close:*** Функције које се позивају када корисник отвара и затвара фајлове који одговарају компонентама хардвера којима желимо да приступамо.

- ***gaussian_read:*** Функција се позива када корисник жели да чита из фајлова и разликују се три случаја зависно од компоненте којој желимо да приступимо. У оквиру *gaussian_core* може се читати само из *ready* регистра који се налази на офсету 12, док се елементи из *kernel_bram* не могу прочитати. Такође, читање *axi_dma* је имплементирано на други начин те је се не може читати на класичан начин.

- ***gaussian_write:*** Функција се позива када корисник жели да упише у фајлове и разликују се три случаја зависно од компоненте којој желимо да приступимо. У оквиру *gaussian_core* може се уписати на локације које одговарају конфигурацијама регистара start, reset, kernel_size чији су офсети дати у таблици дела о хардверу.У *kernel_bram* је могуће поставити вредности на 32-битне магистрале података где се од корисника добијају и адреса и податак. Код *axi_dma* разликују се два случаја: иницијални упис и упис ради допуњавања *BRAM* меморије IP блока која служи за кеширање података.

- ***gaussian_fasync:*** Такође омогућено је и асинхроно обавештавање као механизам индикације кориснику да се десио прекид, што у нашем случају даје знак да је завршена иницијализација и да IP може бити стартован за рад.

- ***gaussian_dma_mmap:*** Меморијско мапирање има значаја приликом преноса велике количине информација што је случај са преносом слика велике резолуције путем *DMA* интерфејса. ммап функција искоришжена је ради мапирања физичких адреса на виртуалне, како би свака адреса коју корисник позове одговарала оној у меморији тако

да не морамо копирати целу слику у кернел простор и тиме побољшавамо рад процесора.

Рад са DMA (Direct Memory Access)

Рад са ДМА захтева поштовање посебних корака конфигурације регистара ДМА контролера како би се подаци слати и примати. Због тога било је потребно имплементирати посебне фје за иницијализацију трансакције код читања и уписа, као и гункције за читање и упис ДМА.

- ***static irqreturn_t dma_isr(int irq,void*dev_id)***: Омогућени су прекиди који ће сигализирати завршетак уписа у меморију, односно да је TX бафер примио све податке послате од старне корисника. Улога прекида је да можемо разликовати случај иницијализације и даљег уписивања у *BRAM* меморију одакле процесуирајући блок преузима податке које обрађује.
- ***int dma_init(void __iomem *base_address)*:** Ради иницијализацију регистара *MM2S* і S*2MM* који омогућују слање и примање трансакција од процесора ка периферији и обратно. Више информација о конфигурацијама регистара и значењу сетовања појединачних бита унутар сваког регистара видети у литератури.
- ***u32 dma_simple_write(dma_addr_t TxBufferPtr, u32 max_pkt_len, void __iomem*base_address)***
- ***u32 dma_simple_read(dma_addr_t RxBufferPtr, u32 max_pkt_len, void __iomem*base_address)***

  Функције које редом врше конфигурацисање регистара *MM2S* і S*2MM* ради уписа односно читања из меморије. Овде је могуће подесити величину података која се преноси као и офсет адресе одакле почиње трансакција. Ове параметре прослеђује корисник посредством *gaussian_write* функције, позивом функције уписа у фајл из корисничког простора.

4. Развој апликације

Након тестирања рада основних функција унутар драјвера, било је неопходно написати апликацију која би тестирала рад читавог система за улазне параметре које корисник прослеђује. Из апликације је могуће проследити:
слику која ће бити обрађена,
величину кернела као и коефицијенте потребне за обраду слике,
команде за ресет/стартовање процесуирајућег блока.
Идеја је била да апликација тестира систем на начин који је рађен у апликацији у делу пројекта Пројектовање сложених дигиталних система.
Следи листа коришћених функција и њихова објашњења:

- ***void write_reg (const int reg, const int value)***
  Функција која служи за конфигурисање регистара унутар ИП блока. Као први параметар прослеђује се макро назив регистра коме одговара адреса на коју желимо уписати вредност, која је прослеђена као други параметар.
- ***void write_kernel_bram(int pos, unsigned int kern_coeff)***

  Функција која служи за уписивање претходно израчунатих вредности коефицијената Гаусовог кернела у *kernel_bram*. Као први параметар прослеђује се адреса на коју уписујемо (помножена фактором 4), а као други децимални број конвертован у целобројни запис (бинарна репрезентација је иста док се тумачење вредности разликује).

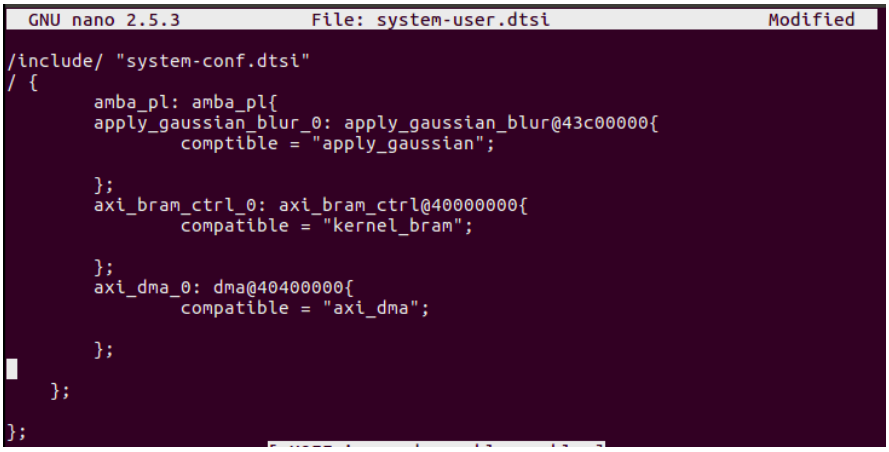
- ***int read_ready_reg()***
  Функција која служи за ишчитавање вредности из статусног регистра *ready* као индикације да је завршена обрада улазне слике.

Како корисник није у могућности да барата са прекидима, а даљи ток апликације зависи од њега било је потребно увести систем асинхроног обавештавања путем *sighandler* функције.

Коришћен директан приступ меморији, а слика која се шаље је великог формата било је потребно извршити меморијско мапирање. У те сврхе коришћена је функција *mmap*. За слање података коришћена је функција *memcpy*.

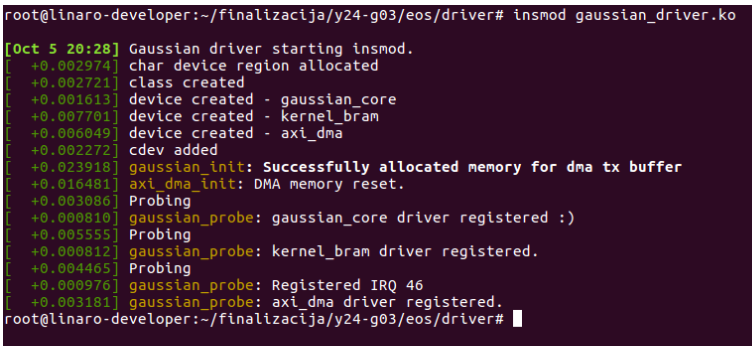
5. Тестирање рада на плочици

Како би подизање оперативног система на *Zybo* плочици био омогућен било је потребно инсталирати Petalinux ОС и придружене алате на SD картицу. Како би подесили систем за наш дизајн било је неопходно додавање .xsa фајла и измене у погледу *compatible* поља стабла уређаја како би делови хардвера којима приступамо били препознати од стране драјвера.



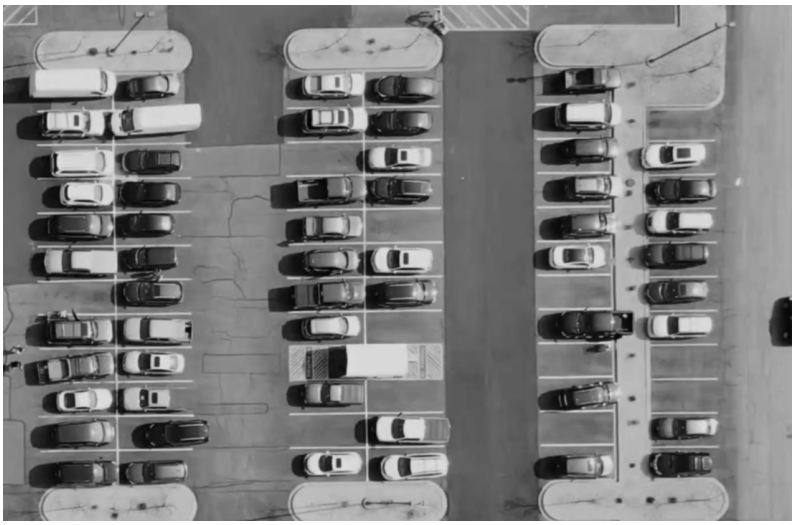
Слика 7. - Измењена compatible поља за стабло уређаја које се генерише за наш систем

Након потребних конфигурација и инсталација додатних алата за рад са кернел модулима, извршено је додавање модула у кернел (*insmod)* након чега се јављају пропратне поруке функција које се позивају из драјвера као знак да је додавање успешно обављено. Такође, унутар /dev директоријума могу се наћи изгенерисани фајлови који одговарају називима уређаја које смо тражили.



Слика 8 - Поруке у терминалу приликом позива insmod

Након компајлирања, покренута је и апликација којом је тестиран рад хардвера за обраду слике, где резултати следе у наставку.



Слика 9. - Изглед слике након обраде кернелом веичине 3

Слика 10. - Изглед слике након обраде кернелом величине 25

```
root@linaro-developer:~/finalizacija/y24-g03/eos/postprocessing# make run
./main

************************************************
        There are 12 free parking spaces.
------------------------------------------------
```

Слика 11. - Испис резултата - броја слободних паркинг места у терминалу

Напомена: На *сликама 9 и 10* су прикази резултата у виду *png* слика добијени форматирањем кроз *python* код како би биле уочљиве измене над сликом које ради хардвер. Резултати које заправо добијамо су целобројни децимални бројеви који одговарају пикселима слике. Додатно, формат улазне слике је мало другачији, јер је хардвер предвиђен да обрађује по два пиксела истовремено. Сваки елемент улазног низа садржи по два пиксела конкатенирана у једну вредност. Излазни фајл садржи појединачне пикселе и шаље се на даљу обраду софтверском алгоритму у *C++*.

На *слици 11* види се резултат даље софтверске обраде слике коју добијамо од хардвера и која треба да да вредност броја слободних паркинг места. Како смо овај пут узели слику са 12 слободних паркинг места, овим је потврђен тај резултат и тиме је доказано да наш хардвер заиста обавља успешно свој задатак.

6. Литература

[1] Материјали са предавања и вежби из предмета Ембедед оперативни системи, https://www.elektronika.ftn.uns.ac.rs/embeded-operativni-sistemi/specifikacija/specifikacija-predmeta/ (4.10.2024.)

[2] Документација за развој кернел платформских драјвера, https://www.kernel.org/doc/html/v5.0/driver-api/ (30.9.2024.)

[3] Документација о динамичком мапирању меморије код директног приступа меморији https://docs.kernel.org/core-api/dma-api.html (29.9.2024.)

[4] Додатна документација за инсталацију Petalinux -а и потребних алата https://docs.amd.com/search/all?query=PetaLinux+Tools+Documentation&content-lang=en-US (25.9.2020)

[5] Изворни *python* алгоритам: https://www.youtube.com/watch?v=caKnQlCMIYI&t=2001s (12.3.2024.)

[5] D. Gajski, S. Abdi, A. Gerstlauer, G. Schirner, *Embedded System Design Modeling, Synthesis and Verification,* Springer 2009

[8] Варијанта *C++* кода за Гаусово замућивање фотографије:

https://github.com/Zeljko9999/GaussianBlur (20.4.2024.)