Razvoj integriranog sustava računala i kamere za brzo procesiranje slika pomoću grafičkog procesora

Strahija, Ivan

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

ZAVRŠNI RAD

Ivan Strahija

Zagreb, 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

ZAVRŠNI RAD

Mentor:

Doc. dr. sc. Marko Švaco, mag. ing. mech.

Komentor:

Dr. sc. Filip Šuligoj, mag. ing. mech.

Student:

Ivan Strahija

Zagreb, 2022.

Izjavljujem da sam ovaj rad izradio samostalno koristeći znanja stečena tijekom studija i navedenu literaturu.

Zahvaljujem mentoru doc. dr. sc. Marku Švaci i komentoru dr. sc. Filipu Šuligoju na strpljenju i pomoći pri izradi završnog rada.

Posebno zahvaljujem Jurici, Jean Christopheu, Koti, Nadimeu i Amineu na motivaciji da dovršim ovaj rad.

Ivan Strahija

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE
Središnje povjerenstvo za završne i diplomske ispite
Povjerenstvo za završne i diplomske ispite studija strojarstva za smjerove: proizvodno inženjerstvo, računalno inženjerstvo, industrijsko inženjerstvo i menadžment, inženjerstvo materijala i mehatronika i robotika

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum	Prilog
Klasa: **602 – 04 / 22 – 6 / 1**	
Ur.broj: **15 - 1703 - 22-**	

ZAVRŠNI ZADATAK

Student: **Ivan Strahija** JMBAG: **0035219259**

Naslov rada na hrvatskom jeziku: **Razvoj integriranog sustava računala i kamere za brzo procesiranje slika pomoću grafičkog procesora**

Naslov rada na engleskom jeziku: **Development of an integrated computer and camera system for fast image processing using a graphics processor**

Opis zadatka:

Primjena pametnih kamera u industrijskim okruženjima otvara potpuno nove mogućnosti za inspekciju, mjerenje i druge aplikacije strojnog vida. Procesiranje algoritama strojnog vida može biti računalno zahtjevan zadatak, koji često premašuje mogućnosti CPU-a (eng. „Central Processing Unit") za izvođenje u stvarnom vremenu. Međutim, mnoge operacije strojnog vida učinkovito se preslikavaju na GPU (eng. „Graphical Processing Unit") koji može ubrzati njihovo izvođenje. GPU modul implementiran u OpenCV (eng. „Open Source Computer Vision Library") knjižnicu softvera za računalni vid i strojno učenje, omogućuje iskorištavanje GPU ubrzanja za procesiranje različitih algoritama strojnog vida.

U završnom radu potrebno je razviti i integrirati vlastito rješenje pametne kamere što uključuje sljedeće:

- Projektirati i izraditi (3D print tehnologija) kućište za: kameru s lećom, računalo (SOM – system on a module, Jetson Xavier) i ostalu potrebnu periferiju i kablove,
- Omogućiti automatsku akviziciju slike preko vanjskog signala (eng. „camera trigger"),
- Postaviti operativni sustav i programersko okruženje (C++, Qt, OpenCV s CUDA funkcijama),
- Izraditi vizijsku aplikaciju u C++ koja obuhvaća dohvaćanje slike, procesiranje i detekciju oznaka na brzo putujućem objektu (GPU procesiranje),
- Evaluirati rezultate detekcije i izmjeriti brzinu procesiranja.

U radu je potrebno navesti korištenu literaturu i eventualno dobivenu pomoć.

Zadatak zadan:
30. 11. 2021.
Zadatak zadao:
Doc. dr. sc. Marko Švaco

Datum predaje rada:
1. rok: 24. 2. 2022.
2. rok (izvanredni): 6. 7. 2022.
3. rok: 22. 9. 2022.
Komentor:
Dr. sc. Filip Šuligoj

Predviđeni datumi obrane:
1. rok: 28. 2. – 4. 3. 2022.
2. rok (izvanredni): 8. 7. 2022.
3. rok: 26. 9. – 30. 9. 2022.
Predsjednik Povjerenstva:
Prof. dr. sc. Branko Bauer

SADRŽAJ

POPIS SLIKA

POPIS TABLICA

POPIS KRATICA

Kratica	Značenje
ARM	Advanced RISC Machines
CPU	Central Processing Unit
CRTA	Regionalni centar izvrsnosti za robotske tehnologije
CUDA	Compute Unified Device Architecture
FPS	Frames per second
GPU	Graphical Processing Unit
LED	Light-emitting diode
QR	Quick Response
SD	Secure Digital
SDK	Software development kit
SoM	System on Module

POPIS OZNAKA

Oznaka	Jedinica	Opis
d	[mm]	Udaljenost između dvije naljepnice
d_{min}	[mm]	Minimalna udaljenost između dvije naljepnice
n	[s^{-1}]	Maksimalan broj detektiranih naljepnica u sekundi
$n_{dekodirano}$	[s^{-1}]	Broj dekodiranih naljepnica u sekundi
$n_{detektirano}$	[s^{-1}]	Broj detektiranih naljepnica u sekundi
sk	[mm]	Širina kadra
sn	[mm]	Širina naljepnice
t_{uk}	[s]	Ukupno vrijeme trajanja ciklusa
v	[mm/s]	Brzina kretanja predmeta
vs	[s^{-1}]	Ograničavajuća brzina sustava

SAŽETAK

U završnom radu dano je rješenje problema detekcije brzoputujućih predmeta u proizvodnim i distributivnim centrima u vidu kompletnog vizijskog sustava. Zbog velikih brzina kojima se predmeti mogu kretati u takvim centrima potrebno je razviti vizijski sustav koji omogućuje kvalitetno dohvaćanje i brzu obradu slike. Iz tog razloga, projektirani vizijski sustav, umjesto centralne procesorske jedinice (CPU) i osnovnih OpenCV biblioteka za obradu slike, koristi grafički procesor (GPU) te dodatni CUDA modul za brzu obradu slike na GPU. Završni rad uključuje i postavljanje programskog okruženja na *NVIDIA Jetson Xavier NX* računalu (*System on Module* - SoM) za koje je izrađeno i kućište koje omogućuje laku adaptaciju i montažu cijelog sustava u radnoj okolini. Cijeli sustav, koji uključuje i dvije kamere različitih specifikacija, postavljen je u laboratoriju CRTA-e pored *Fanuc DR-3iB/8L* delta robota. Robot je korišten pri testiranju performansi vizijskog sustava kako bi se promatrani predmet mogao gibati točno određenom brzinom ispred kamera te tako osigurati relevantnost dobivenih rezultata te ponovljivost testiranja. Rezultati provedenog testiranja potvrdili su premisu da kamere koje koriste global shutter dohvaćaju kvalitetniju sliku brzoputujućih objekata od kamera koje koriste rolling shutter. Cijeli sustav uspješno je detektirao i dekodirao oznake na predmetima koji su se gibali brzinama do 4000 mm/s.

Ključne riječi: vizijski sustav, brzo procesiranje slike, *NVIDIA Jetson*, GPU, *OpenCV*, *CUDA*

SUMMARY

This paper presents the integrated computer vision system for the detection of the fast-moving objects in production and distribution centers . Since the objects in these facilities move at great speeds, there is a need for the development of a system that enables high-quality image acquisition and fast image processing. For that reason, the developed system, instead of using the central processing unit (CPU) and standard OpenCV libraries for image processing, uses the graphics processor (GPU) and the additional CUDA module for fast image processing on the GPU. The paper also covers setting up *NVIDIA Jetson Xavier NX* (*System on Module* - SoM) for which a case that enables easy mounting of the whole system in the working environment has also been made. The whole system, which also includes two cameras with different specifications, was set up in the CRTA laboratory by the *Fanuc DR-3iB/8L* delta robot. The robot was used for the testing of the system since it enables moving of the object at a defined speed and hence ensures the relevance of the results and repetable testing. The testing results confirmed the premise that cameras with global shutter capture better quality images of a fast-moving object than cameras with rolling shutter. The system was able to successfully detect and decode the labels on the objects that were moving at a speed of up to 4000 mm/s.

Key words: computer vision, fast image processing, *NVIDIA Jetson*, GPU, *OpenCV*, *CUDA*

1. UVOD

1.1. Definicija problema

Proizvodni i distributivni centri mjesta su na kojima se proizvodi izrađuju i skladište te se iz njih šalju na tržište. Cilj proizvodnih i distributivnih centara je što efikasnije kontrolirati i upravljati tokom proizvoda te u svakom trenutku imati točne podatke o broju i stanju svakog pojedinog proizvoda. Vizijski sustavi često su rješenje za lakše praćenje toka proizvoda kroz proizvodni ili distributivni centar. Kako su ovakvi centri u velikoj mjeri automatizirani, često ih karakterizira i veliki radni obujam te kratka vremena potrebna za proizvod da dođe od početne do krajnje točke procesa. Kratka vremena procesa ujedno znače i velike brzine kretanja proizvoda u procesu. Budući da se proizvodi kreću velikim brzinama, vizijski sustavi u takvim centrima u malom vremenu moraju obraditi veliku količinu podataka koje dobivaju, te iz njih izvući relevantne informacije. Sama brzina kretanja proizvoda kroz proces postavlja zahtjeve za vizijski sustav, odnosno na odabir odgovarajućih kamera, postavljanje radnog okruženja te na brzinu procesuiranja podataka.

U ovom radu prikazan je razvoj integriranog vizijskog sustava koji omogućuje brzu obradu slike te je direktno primjenjiv za rad na proizvodnim linijama ili u distributivnim centrima.

1.2. Računalni vid i korištene biblioteke

Računalni vid grana je umjetne inteligencija koja se bavi obradom slike, a kao rezultat obrade slike daje korisne informacije te omogućuje reagiranje sustava u ovisnosti o informacijama koje dobiva sa slike. Računalni vid omogućio je znatna unaprjeđenja u širokom spektru industrija, no uz stalan napredak dolaze i novi izazovi. Procesiranje i obrada slike pomoću centralne procesorske jedinice (u nastavku će se koristiti kratica CPU), zbog velike količine podataka koje treba obraditi na slikama visoke rezolucije, često nije dovoljno brzo za izvođenje algoritama u realnom vremenu.

Upravo to je jedan od razloga zašto je NVIDIA 2007. godine izdala platformu CUDA (Compute Unified Device Architecture). To je platforma koja omogućuje korištenje i izvođenje raznih algoritama na grafičkoj procesorskoj jedinici (u nastavku će se koristiti kratica GPU). Korištenje GPU tako omogućuje i bržu izvedbu algoritama za obradu slike te samim time pruža

mogućnost za primjenu tih algoritama u realnom vremenu. Najčešće korišteni algoritmi za obradu slike dio su besplatne i često korištene OpenCV biblioteke.

OpenCV (Open Source Computer Vision Library) je 'open source' biblioteka koja se koristi za razvoj aplikacija računalnog vida i strojnog učenja. Biblioteka se sastoji od više od 2500 optimiziranih algoritama te se konstantno nadograđuje. Neki od često korištenih algoritama su oni za prepoznavanje objekata, za praćenje objekata u realnom vremenu, za izradu 3D slike uz pomoć stereo kamera i za pronalaženje karakterističnih slika iz postojeće baze podataka. OpenCV biblioteka napisana je u C++ programskom jeziku, a ima podršku i za Python, Java i MATLAB programska okruženja. OpenCV biblioteku moguće je koristiti na 4 operativna sustava; Windows, Linux, Android i MacOS. Uz osnovni set algoritama postoji i dodatni OpenCV CUDA modul koji je korišten u ovom radu.

OpenCV CUDA modul je set klasa i funkcija koje iskorištavaju prednosti CUDA platforme i GPU procesuiranja te ih direktno primjenjuju na algoritme za obradu slike. Modul ne uključuje sve OpenCV algoritme za obradu slike koje je moguće izvršavati na CPU, ali uključuje one najkorištenije (stereo vid, filtriranje slike, prepoznavanje predmeta itd.). Iz tog razloga se korištenje OpenCV CUDA modula preporučuje kod izrade aplikacija koje zahtijevaju brzu obradu slike. OpenCV CUDA modul dovoljno je kompilirati pri instalaciji OpenCV paketa te će funkcije iz CUDA modula raditi i bez instalacije osnovnog CUDA Toolkita.

1.3. Elementi vizijskog sustava

1.3.1. NVIDIA Jetson Xavier NX Developer Kit

NVIDIA Jetson Xavier NX Developer Kit je System on Module (SoM) računalo namijenjeno za primjernu u ugradbenim sustavima (embedded systems) s velikom snagom procesiranja što ga čini pogodnim za razvoj aplikacija koje zahtijevaju brzu obradu velike količine podataka. Uz Jetson Xavier NX Developer Kit postoje i druga NVIDIA Jetson računala i moduli koji se razlikuju po specifikacijama i mogućnostima primjene. Oni zajedno čine Jetson platformu koju je NVIDIA dizajnirala za razvojne inženjere te se koriste kako u raznim industrijama diljem svijeta tako i za razvoj korisničkih projekata temeljenih na računalnom vidu i umjetnoj inteligenciji. Jetson Xavier NX Developer Kit prikazan je na slici 1., a njegove tehničke specifikacije dane su u Tablici 1.

Slika 1. NVIDIA Jetson Xavier NX Developer Kit [1]

Tablica 1. Tehničke specifikacije NVIDIA Jetson Xavier NX Developer Kit

CPU	6-core NVIDIA Carmel ARMv8.2
GPU	NVIDIA Volta, 384 CUDA cores
RADNA MEMORIJA	8GB 128-bit LPDDR4x 59.7 GB/s
POHRANA PODATAKA	microSD
POVEZIVOST	Gigabit Ethernet, M.2 Key E (WiFi/BT), M.2 Key M (NVMe)
VIDEO IZLAZ	HDMI, Display Port
DIMENZIJE	103 mm x 90.5 mm x 34 mm

1.3.2. Kamere i objektivi

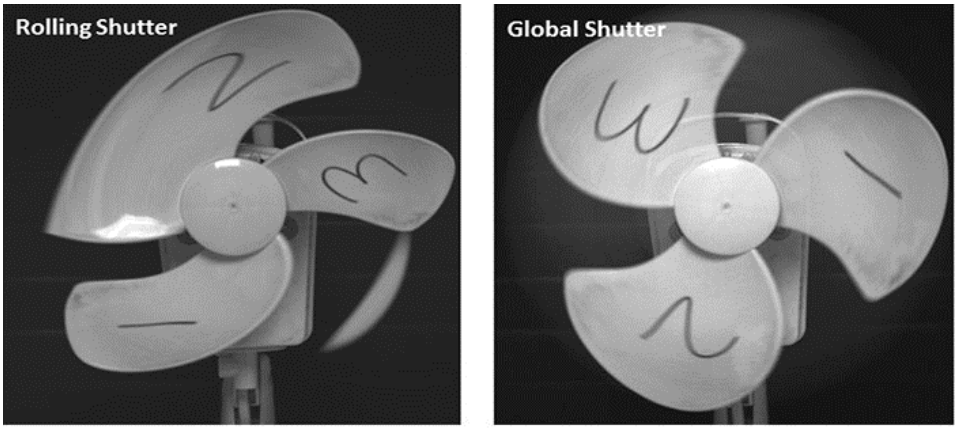
Za dohvaćanje slike korištene su dvije različite kamere; IDS UI-3881LE-C-HQ-AF te Basler acA1300-30gm. Dvije korištene kamere znatno se razlikuju po svojim specifikacijama te će se njihove performanse usporediti u eksperimentalnom dijelu rada. Njihove specifikacije dodatno su pojašnjene u ovom poglavlju. Slika 2. prikazuje navedene kamere.

Slika 2. IDS *UI-3881LE-C-HQ-AF i* Basler *acA1300-30gm* kamere [2] [3]

Sa IDS kamerom korištena je Corning *C-S-25H0-096* leća, a sa Basler kamerom je korištena Edmund *C-Series 25* leća. Navedene kamere i leće korištene su kako bi se mogla napraviti usporedba rada aplikacije s različitom opremom. Slika 3. prikazuje navedene leće.

Slika 3. Corning C-S-25H0-096 i Edmund C-Series 25 leće [4] [5]

Korištena IDS kamera može dohvaćati sliku visoke rezolucije te se u ovom radu koristi kako bi se pokazala mogućnost brzog procesiranja velike količine podataka (slika visoke rezolucije) putem GPU. Korištena Basler kamera može dohvaćati sliku manje rezolucije, no upotrebljena je zato što koristi global shutter koji je pogodniji za snimanje brzoputujućih predmeta. Kamere s rolling shutterom sliku dohvaćaju 'red po red' što znači da između snimanja prvog i zadnjeg reda piksela na slici prođe određeno vrijeme. Kod snimanja brzoputujućih predmeta to može izazvati distorziju slike budući da se predmet ne nalazi u istom položaju u trenutku u kojem se dohvaća prvi i u trenutku u kojem se dohvaća posljednji red piksela slike. Zato se za snimanje brzoputujućih predmeta preporučuje korištenje kamera s global shutterom. Te kamere istovremeno dohvaćaju cijelu sliku (otuda i naziv global) te su stoga pogodnije za navedenu primjenu. U ovom radu korištena su oba tipa kamere kako bi se kroz testiranje mogla pokazati njihova razlika. Slika 4. prikazuje razliku u slici dohvaćenoj s rolling i global shutterom.



Slika 4. Usporedba slike dohvaćene kamerom s rolling i global shutterom [6]

S IDS kamerom korištena je leća koja ima mogućnost autofokusa, dok je s Basler kamerom korištena leća kod koje je potrebno ručno namjestiti fokus na fiksnu vrijednost. Prednost korištenja leće s autofokusom jest mogućnost brzog i automatskog izoštravanja slike na različitim udaljenostima od kamere.

Tablica 2. daje usporedbu bitnih značajki navedenih kamera. Tablica 3. daje usporedbu značajki navedenih leća.

Tablica 2. Usporedba karakteristika korištenih kamera [2] [3]

	IDS *UI-3881LE-C-HQ-AF*	**Basler *acA1300-30gm***
Tip senzora	CMOS Color	Monochrome
Shutter	Rolling shutter	Global shutter
Rezolucija	3088x2076 px	1296x966 px
Frame rate	58 FPS	30 FPS
Lens mount	S-mount	C-Mount
Povezivost	USB Type-C	Ethernet

Tablica 3. Usporedba karakteristika korištenih leća [4] [5]

	Corning *C-S-25H0-096*	**Edmund *C-Series 25***
Focal length	9.6 mm	25 mm
Focus range	70 mm - ∞	100 mm - ∞
Autofokus	Da	Ne

1.4. Fanuc DR-3iB/8L

Za pomicanje predmeta korišten je Fanuc DR-3iB/8L delta robot. Delta roboti su roboti paralelne kinematike često korišteni u industriji za obavljanje 'pick and place' zadataka. Najčešće su fiksno montirani iznad pokretnih traka te, zbog inovativnog dizajna i male inercije ovih robota, velikom brzinom obavljaju potrebne zadaće. Nedostatak ovih robota je mala nosivost te se iz tog razloga koriste u industrijama u kojima je naglasak na povećanju produktivnosti, a ne na baratanju velikim masama. Uz to valja napomenuti kako je doseg ovakvih robota znatno ograničen zbog same konstrukcije te se radno područje smanjuje kako se po vertikalnoj osi odmiče od njegove baze. Fanuc DR-3iB/8L prikazan je na slici 5. Tablica 4. prikazuje najbitnije tehničke karakteristike Fanuc DR-3iB/8L robota.

Slika 5. Fanuc DR-3iB/8L delta robot [7]

Tablica 4. Tehničke karakteristike Fanuc DR-3iB/8L robota [7]

Stupnjevi slobode	4
Preciznost	±0.03 mm
Maksimalna linearna brzina	5500 mm/s
Nosivost	8kg
Doseg	1600 mm
Masa	170 kg

1.5. Metodologija rada

U sklopu završnog rada potrebno je razviti vlastito rješenje za vizijski sustav što uključuje i projektiranje potrebnih kućišta za pojedine elemente te izradu eksperimentalnog postava. Za početak su modelirani potrebni elementi eksperimentalnog postava (kućišta i postolja) te su izrađeni metodom 3D ispisa. Nakon izrade je slijedila montaža eksperimentalnog postava. Zatim je postavljeno programersko okruženje na NVIDIA Jetson računalu. Postavljen je operativni sustav te su instalirani potrebni programi i biblioteke. Nakon uspješnog postavljanja programerskog okruženja slijedi izrada programskog rješenja za akviziciju i obradu slike pomoću grafičkog procesora. Kod programa napisan je pomoću osnovnih C++ naredbi te dodatnih funkcija iz OpenCV biblioteke. Po izradi slijedi i samo testiranje programa, modifikacija parametara i konačna evaluacija dobivenih rezultata.

2. PROJEKTIRANJE VIZIJSKOG SUSTAVA

2.1. Izrada i postavljanje eksperimentalnog okruženja

Za modeliranje kućišta i nosača korišten je programski paket SolidWorks. Kod modeliranja kućišta za Jetson Xavier NX obraćena je pozornost na ostavljanje dovoljno otvora kako bi zrak mogao nesmetano cirkulirati i hladiti sami uređaj. Uz to, s donje strane kućišta predviđeno je mjesto za filter prašine koji je također modeliran i dodan u sklop. Kod modeliranja kućišta naglasak je stavljen na kompaktnost i mogućnost lake montaže neovisno o radnoj okolini. Iz tog razloga su modelirani i dodatni imobilizatori koji onemogućuju pomicanje Jetson računala nakon što se postavi na željenu poziciju. Slika 6. prikazuje kućište i imobilizatore montirane pored pokretne trake s robotom.

Slika 6. Kućište i imobilizatori

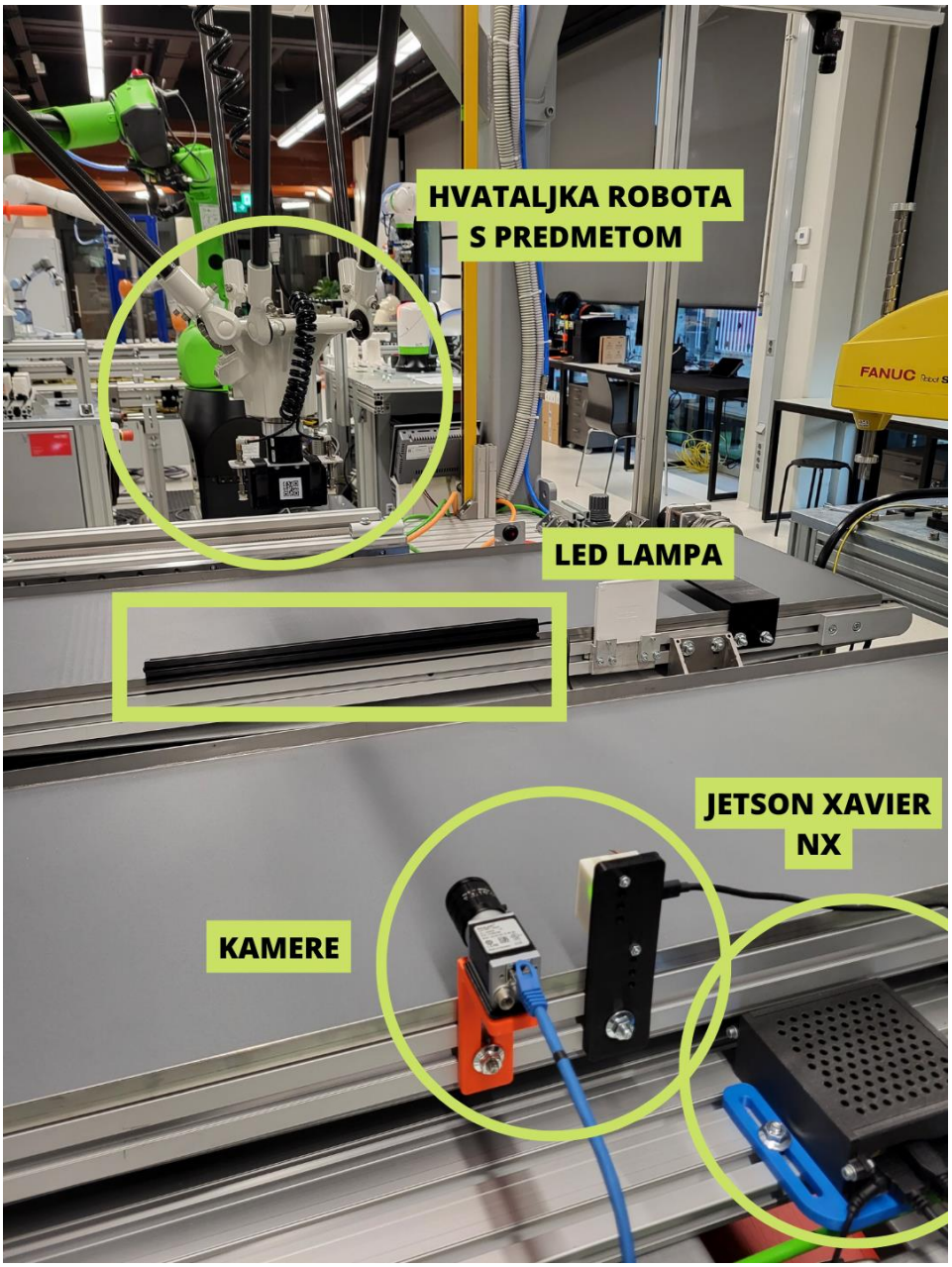
Nosač za IDS kameru napravljen je tako da se već postojeće kućište kamere može lako montirati na njega i to na različite visine. Nosač je zatim vijkom pričvršćen na aluminijski profil pored pokretne trake. Nosač za Basler kameru također je vijkom pričvršćen na aluminijski profil te je podesiv po visini. Slika 7. prikazuje montirane nosače s kamerama pored pokretne trake s robotom.

Slika 7. Nosači za kamere

Od ostale opreme korištena je i LED lampa koja pruža kvalitetno osvjetljenje radnog prostora te omogućava bolje uvjete rada i bolje rezultate izvedbe samog programa. Usporedba performansa sustava u različitim uvjetima osvjetljenja dana je kasnije. Slika 8. prikazuje LED lampu montiranu pored pokretne trake s robotom.

Slika 8. LED lampa

Cijeli sustav montiran je pored Fanuc DR-3iB/8L delta robota u laboratoriju CRTA-e. Delta robot koristio se kako bi se provelo testiranje i mogućnost brzog dohvaćanja i obrade slike vizijskog sustava. Na slici 9. prikazan je cijeli eksperimentalni postav.



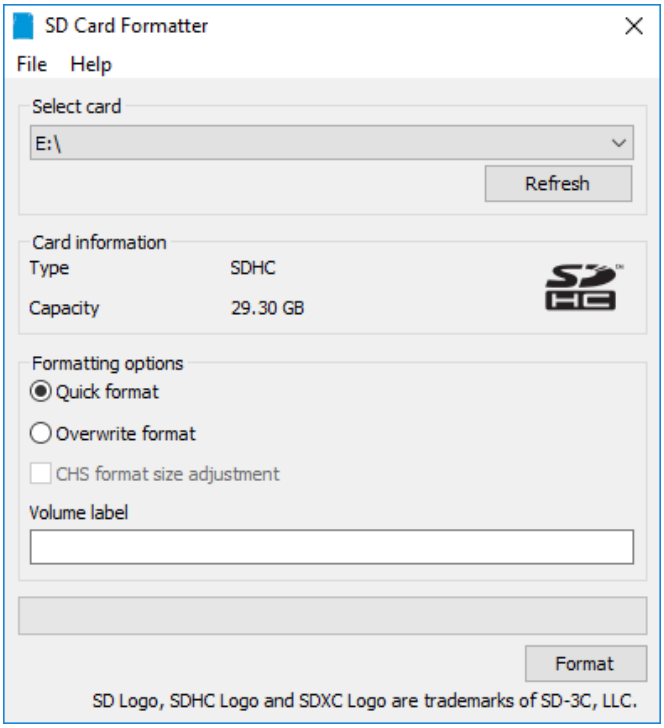
Slika 9. Eksperimentalni postav u laboratoriju CRTA-e

2.2. Postavljanje operativnog sustava

NVIDIA Jetson Xavier NX dolazi bez instaliranog operativnog sustava, stoga je prvi korak postavljanje operativnog sustava prema uputama s NVIDIA web stranice. Nakon uspješnog postavljanja operativnog sustava potrebno je instalirati potrebne programe i biblioteke kako bi se moglo krenuti s izradom vizijske aplikacije. Svaki korak instalacije i postavljanja programerskog okruženja detaljno je opisan u sljedećim poglavljima.

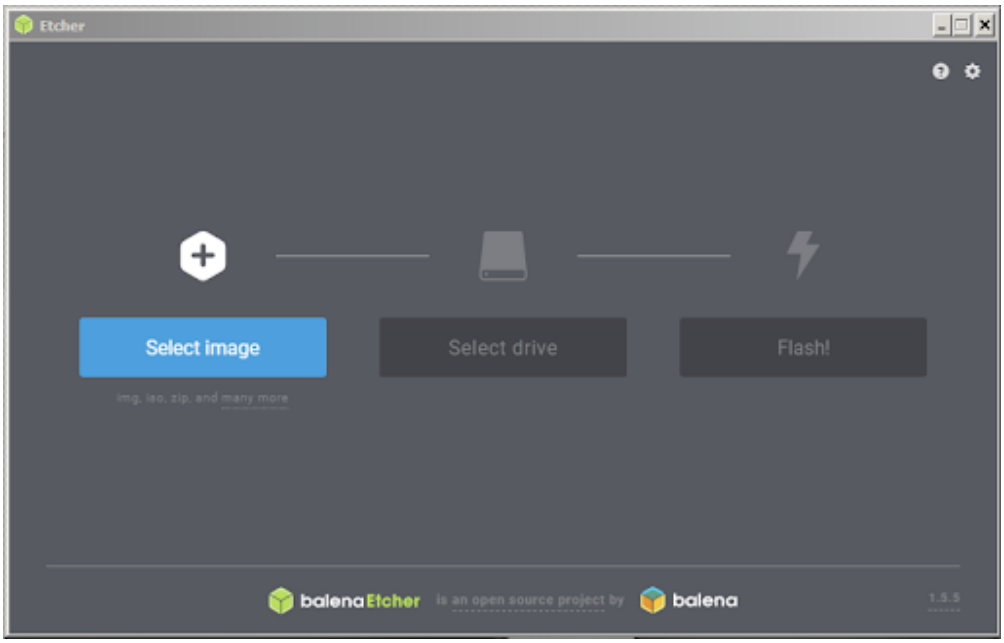
2.2.1. Postavljanje operativnog sustava

NVIDIA je za Jetson uređaje pripremila JetPack SDK paket koji se sastoji od Linux Ubuntu operacijskog sustava te dodatnih biblioteka za ubrzano procesuiranje podataka putem grafičkog procesora (Tensor RT, cuDNN, CUDA). Za instalaciju na Jetson Xavier NX potrebno je pripremiti SD karticu te na nju učitati sliku željenog JetPack SDK paketa. Za ovaj rad korišten je Jetpack 4.6.1 SDK koji uključuje Linux Ubuntu 18.04 operativni sustav. Za početak, s NVIDIA stranice [8] potrebno je preuzeti datoteku željenog JetPack SDK-a. Zatim je potrebno formatirati SD karticu na koju želimo učitati sliku paketa. NVIDIA preporučuje korištenje SD Card Formatter [9] programa čije je sučelje intuitivno i lako za korištenje. Prikaz sučelja SD Card Formatter programa dan je na slici 10.



Slika 10. Sučelje programa SD Card Formatter

Nakon formatiranja kartice, na nju je potrebno učitati sliku JetPack SDK paketa. Za to je preporučeno koristiti program balenaEtcher [10] čije je sučelje također intuitivno i lako se koristi. Treba odabrati željenu datoteku koju želimo učitati na željeni disk (u ovom slučaju to je preuzeta JetPack 4.6.1 SDK datoteka koju želimo učitati na SD karticu) te pokrenuti učitavanje slike. Slika 11. prikazuje sučelje programa balenaEtcher.



Slika 11. Sučelje programa balenaEtcher

Nakon toga potrebno je SD karticu s učitanom slikom JetPack SDK paketa umetnuti u utor Jetson Xavier NX računala te ga uključiti. Kod prvog pokretanja potrebno je postaviti operativni sustav te izabrati korisničko ime i lozinku. Jetson Xavier NX može se koristiti kao i svako drugo računalo s Ubuntu operativnim sustavom. Bitno je obratiti pozornost na arhitekturu procesorske jedinice (Tablica 1.) te kod instalacije programa i biblioteka odabrati verzije koje su prilagođene za ARM arhitekturu.

2.2.2. Instalacija potrebnih programa i biblioteka

Instalacija potrebnih programa i biblioteka vrši se putem Ubuntu Terminala. U ovom poglavlju slijedno su navedene naredbe te dodatna pojašnjenja za instalaciju pojedinih programa i biblioteka potrebnih za izradu vizijske aplikacije. Valja naglasiti kako predloženi način instalacije nije i jedini mogući te se u izvorima mogu pronaći i drugačije upute, no ovaj način bio je najjednostavniji. Komande unesene u Terminal u nastavku bit će uokvirene kako bi se mogle lako razlikovati od teksta koji ih pojašnjava.

Valja navesti kako je sve navedeno potrebno raditi kao *sudo* (superuser do) korisnik što daje korisniku sva prava kod instalacije i pristupanju datotekama. Prvo je potrebno provjeriti postoje li moguće nadogradnje za već instalirane programe i biblioteke i po potrebi ih nadograditi. Naredbe *update* i *upgrade* služe za to.

```
$ sudo apt-get update
$ sudo apt-get upgrade
```

Nakon toga nastavlja se sa instalacijom Qt Creator platforme putem koje će biti razvijena i vizijska aplikacija.

```
$ sudo apt install qtcreator
$ sudo apt install build-essential
$ sudo apt install qt5-default
$ sudo apt install qt5-doc qt5-doc-html qtbase5-doc-html qtbase5-examples
```

Zatim se instalira IDS Software Suite kako bi se omogućila povezivost korištene IDS kamere s Jetson Xavier NX računalom te automatska akvizicija slike. Potrebne datoteke su preuzete sa službene IDS stranice [11]. Instalacija se provela koristeći 'archive file' s navedene stranice.

Potrebno je preuzeti navedenu datoteku s dane stranice te ju zatim raspakirati. Nakon toga treba pokrenuti skriptu koja će sama dovršiti instalaciju. U nastavku je dan općeniti kod za instalaciju. Uglate zagrade nije potrebno pisati. Potrebno je samo umjesto njih upisati verziju IDS Software Suitea i arhitekturu procesora za koji je i preuzeta datoteka s IDS stranice [10] te u idućoj naredbi ime direktorija u kojem je raspakirana datoteka. U ovom radu korišten je IDS Software Suite 4.96.1. za ARMv8 64-bit arhitekturu procesora, a datoteka je raspakirana u direktoriju Downloads.

```
$ sudo tar xvf ueye_[verzija_programa]_[arhitektura_procesora].tar
$ sudo ./[direktorij_u_kojem_je_raspakirana_tar_datoteka]/ueyesetup -i all
```

Time je dovršen postupak instalacije IDS Software Suite programa, no prije prvog pokretanja potrebno je jednom manualno pokrenuti i zaustaviti 'daemon' programe koji će se koristiti.

```
$ sudo /etc/init.d/ueyeusbdrc start
$ sudo /etc/init.d/ueyeusbdrc stop
```

Potom slijedi instalacija OpenCV paketa. Prije same instalacije potrebno je provjeriti jesu li instalirane potrebne biblioteke, te ako nisu, instalirati ih putem sljedeće komande.

```
$ sudo apt install build-essential cmake git pkg-config libgtk-3-dev libavcodec-dev libavformat-dev libswscale-dev libv4l-dev libxvidcore-dev libx264-dev libjpeg-dev libpng-dev libtiff-dev gfortran openexr libatlas-base-dev python3-dev python3-numpy libtbb2 libtbb-dev libdc1394-22-dev
```

Zatim je potrebno klonirati OpenCV i OpenCV contrib repozitorije sa GitHuba. Ovim naredbama klonirati će se najnovije verzije repozitorija, no moguće je po želji klonirati i starije verzije.

```
$ sudo mkdir ~ /opencv_build && cd ~/opencv_build
$ git clone https://github.com/opencv/opencv.git
$ git clone https://github.com/opencv/opencv_contrib.git
```

Nakon kloniranja repozitorija potrebno je kreirati privremeni direktorij u kojem će se izvršiti instalacija.

```
$ cd ~/opencv_build/opencv
$ mkdir build && cd build
```

Nakon toga potrebno je pozvati CMake te uz instalaciju osnovnih, dodati i potrebne CUDA biblioteke kod instalacije OpenCV paketa. Treba obratiti pozornost kod definiranja CUDA arhitekture računala. U slučaju instalacije na Jetson Xavier NX potrebno je odabrati '7.2' arhitekturu.

```
$ cmake -D CMAKE_BUILD_TYPE=RELEASE \
-D ENABLE_FAST_MATH=1 \
-D CUDA_FAST_MATH=1 \
-D WITH_CUBLAS=1 \
-D WITH_CUDA=ON \
-D BUILD_opencv_cudacodec=ON \
-D WITH_CUDNN=ON \
-D OPENCV_DNN_CUDA=ON \
-D CUDA_ARCH_BIN=7.2 \
-D CMAKE_INSTALL_PREFIX=/usr/local \
-D INSTALL_C_EXAMPLES=ON \
-D OPENCV_GENERATE_PKGCONFIG=ON \
-D OPENCV_EXTRA_MODULES_PATH=~/opencv_build/opencv_contrib/modules \
-D WITH_QT=ON \
```

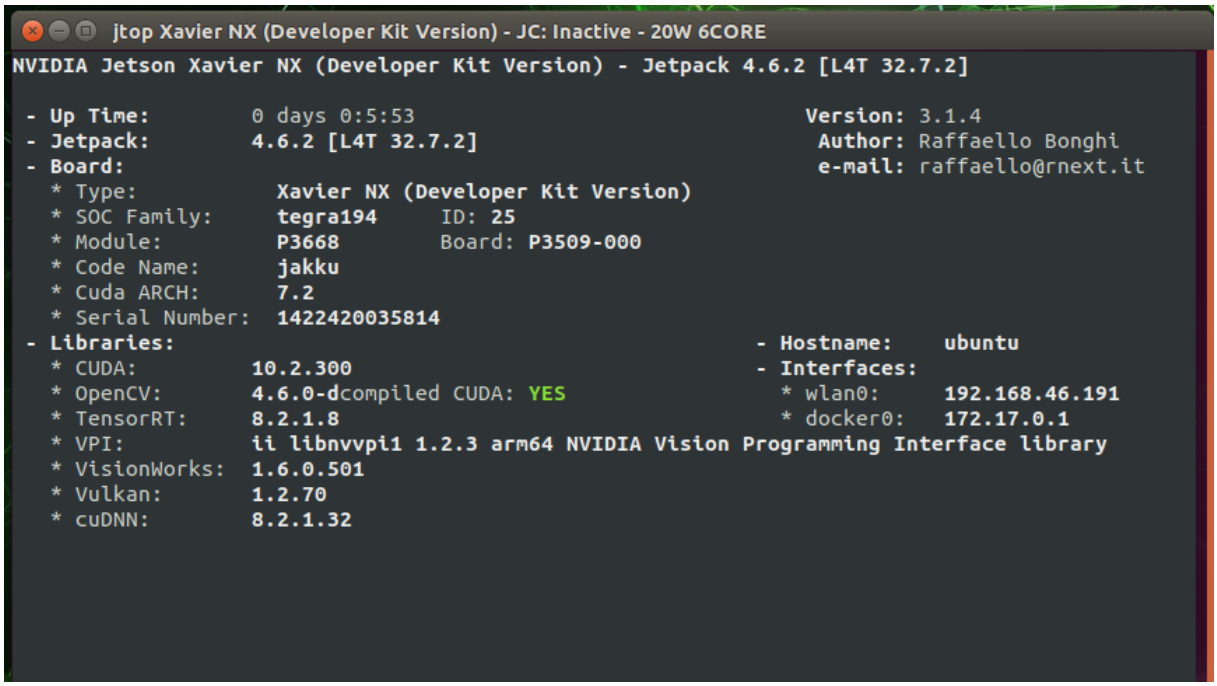
Nakon što je CMake komanda izvršena potrebno je pokrenuti proces kompiliranja. Prije kompilacije potrebno je znati broj jezgri procesora na korištenom računalu te na mjesto slova x upisati broj jezgri umanjen za jedan. Uglate zagrade ne treba pisati. Budući da Jetson Xavier NX koristi šestojezgreni procesor (Tablica 1.) u ovom konkretnom slučaju upisan je broj pet.

```
$ sudo make -j[x]
```

Potom slijedi instalacija OpenCV paketa.

```
$ sudo make install
```

Za provjeru instalacije OpenCV paketa korišten je Jetson Stats [12] program preuzet sa GitHub repozitorija. Slika 12. prikazuje instaliranu verziju OpenCV paketa i dodatnih biblioteka.



Slika 12. Verzija instaliranog OpenCV paketa i dodatnih biblioteka

3. PROGRAMSKO RJEŠENJE ZA BRZO PROCESIRANJE SLIKA POMOĆU GRAFIČKOG PROCESORA

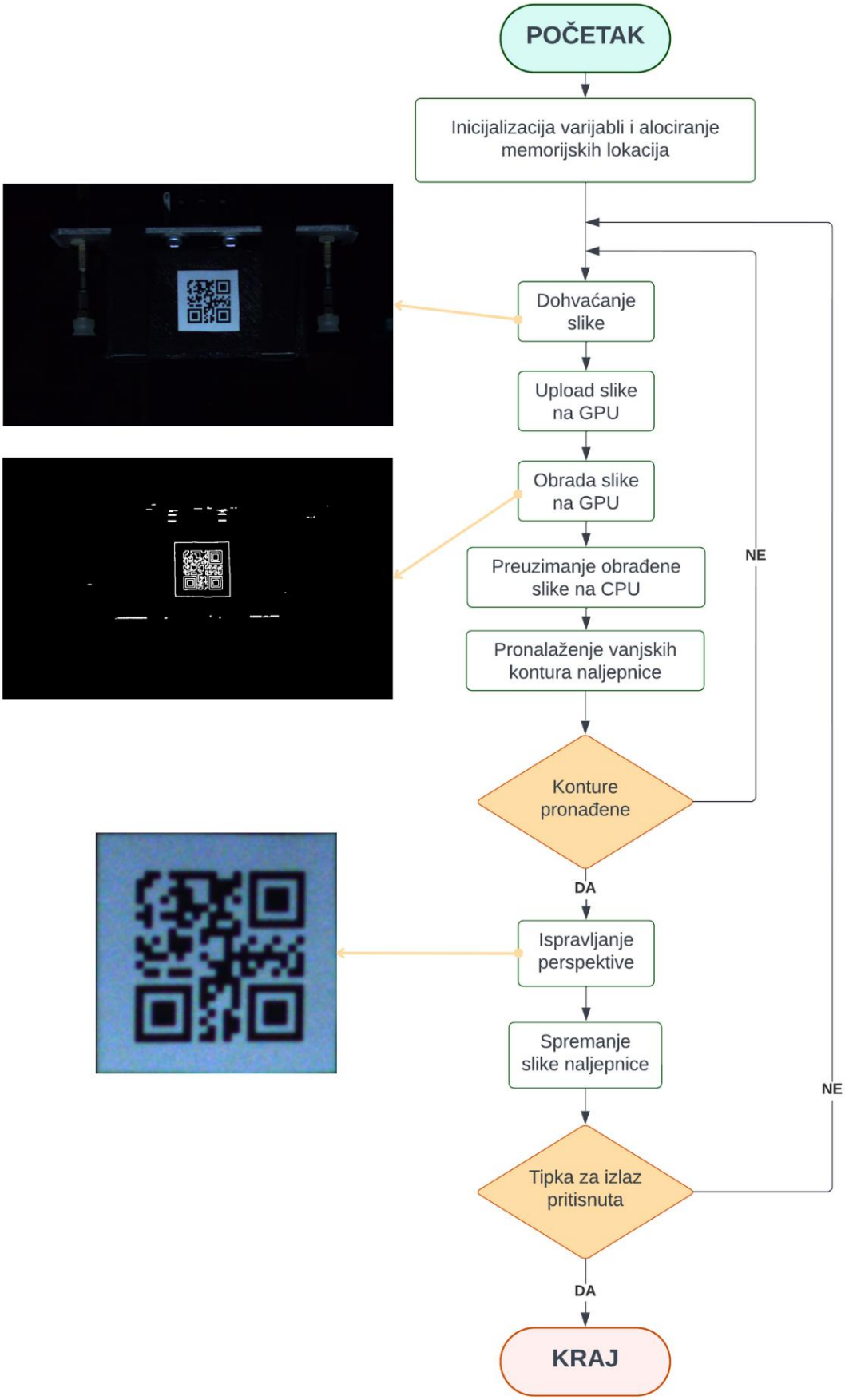
Vizijska aplikacija za dohvaćanje slike, procesiranje, detekciju naljepnica i dekodiranje QR kodova s detektiranih naljepnica napisana je u C++ programskom jeziku, koristeći Qt razvojno okruženje. U ovom poglavlju objašnjene su postojeće i korisničke funkcije korištene za izradu vizijske aplikacije.

3.1. Postavljanje projekta

Za *build* aplikacije korišten je *qmake* alat koji dolazi uključen u Qt razvojnom okruženju. Alat *qmake* putem *.pro* datoteke daje uputu C++ kompajleru koje repozitorije i biblioteke treba pozvati kod kompiliranja aplikacije. Cijela *.pro* datoteka dodana je kao prilog ovom završnom radu. Najbitnije je naglasiti kako putem `INCLUDEPATH` i `LIBS` naredbi treba dodati potrebne OpenCV, CUDA i uEye biblioteke kako bi se one mogle u glavnom kodu dodati putem C++ naredbe `#include` te bi se kod mogao izvršavati pozivajući korištene funkcije iz uključenih biblioteka. Putem `SOURCE` naredbe definiraju se datoteke koje sadrže glavni kod programa koji će se izvršavati (u ovom radu to je datoteka *main.cpp*). Jednom kad je *.pro* datoteka ispravno konfigurirana, može se koristiti za sve projekte koji uključuju iste biblioteke. Treba naglasiti kako se korištena *.pro* datoteka priložena ovom radu ne može koristiti na drugim računalima bez prilagodbe budući da bi se lokacije biblioteka te njihove verzije mogle razlikovati od onih korištenih u ovom radu. Stoga je potrebno modificirati `INCLUDEPATH` i `LIBS` naredbe kako bi bile valjane na računalu na kojem se koriste. Uz to, postoji mogućnost korištenja *cmake* alata umjesto *qmake* alata koji je korišten u ovom radu. Alat *cmake* ima istu funkciju kao i *qmake* alat, no njihove se sintakse razlikuju. U drugim izvorima moguće je pronaći kako postaviti projekt korištenjem *cmake* alata. U sklopu ovog rada to nije obrađeno budući da je *qmake* alat bio intuitivniji za korištenje, a odrađuje istu funkciju kao i *cmake* alat.

3.2. Algoritam za online obradu slike i detekciju naljepnica

Za početak je dan dijagram toka za online obradu slike i detekciju naljepnica na brzo putujućem objektu. Svaki od koraka bit će detaljno objašnjen te će se ukratko opisati korištene funkcije iz OpenCV paketa. Korisničke funkcije bit će dodatno opisane kako bi se kod mogao u potpunosti razumjeti. Za bolje razumijevanje koda preporučuje se čitanje detaljnih opisa korištenih funkcija sa službene OpenCV internetske stranice [13].


POČETAK
Inicijalizacija varijabli i alociranje memorijskih lokacija
Dohvaćanje slike
Upload slike na GPU
Obrada slike na GPU
Preuzimanje obrađene slike na CPU
Pronalaženje vanjskih kontura naljepnice
Konture pronađene
NE
DA
Ispravljanje perspektive
Spremanje slike naljepnice
NE
Tipka za izlaz pritisnuta
DA
KRAJ

Čitav kod (*main.cpp*) ove vizijske aplikacije dan je u prilogu. U nastavku je temeljito objašnjen i prikazan svaki dio čitavog koda.

Na početku je potrebno putem naredbe **`#include`** dodati sve biblioteke koje će se koristiti u aplikaciji.

```
#include <iostream>
#include <string>
#include <stdio.h>
#include <opencv2/opencv.hpp>
#include "opencv2/objdetect/objdetect.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <opencv2/core/cuda.hpp>
#include <opencv2/cudaimgproc.hpp>
#include <opencv2/cudafilters.hpp>
#include <opencv2/cudaarithm.hpp>
#include "ueye.h"
```

Potrebno je inicijalizirati varijable i alocirati potrebne memorijske lokacije u koje će se spremati slike na grafičkoj i centralnoj procesorskoj jedinici. Tako se ubrzava rad same aplikacije budući da je memorija unaprijed alocirana te se ne treba alocirati svaki puta kod slanja slike na GPU i CPU. Putem naredbe **`Mat`** definiraju se memorijske lokacije u koje će se spremati slike na CPU, a putem naredbe **`cuda::GpuMat`** definiraju se memorijske lokacije u koje će se spremati slike na GPU. Inicijalizirana su širina i visina slike koju dohvaćamo s kamere (1920x1080), širina i visina slike u koju spremamo ispravljenu sliku naljepnice (600x600) te broj slika naljepnice koje spremamo i kasnije učitavamo u algoritam za offline dekodiranje QR koda sa spremljenih slika. Inicijalizirani su i vektori točaka **`initialPoints`** te **`arrangedPoints`** u koje će se spremati koordinate vrhova detektirane naljepnice.

```
//inicijalizacija
int height = 1080; //širina i visina dohvaćene slike
int width = 1920;
int w = 600; //širina i visina naljepnice
int h = 600;
int b = 0;
int noimg = 20; //ukupan broj spremljenih slika naljepnice
vector<Point> initialPoints, arrangedPoints;
Mat img(height, width, CV_8UC3);
Mat imgprocessedcpu;
Mat imgWarped(w, h, CV_8UC3, Scalar(0, 0, 0));
cuda::GpuMat imggpu, imggray, imgblur, imgthr, imgcanny, imgprocessed;
```

Po uzoru na kod za inicijalizaciju uEye kamera koji je objavljen na GitHub stranici [14], dodane su korisničke funkcije za inicijalizaciju kamere i dohvaćanje slike s kamere. Za inicijalizaciju

kamere dodana je funkcija **initializeCameraInterface** koja kao argument prima adresu kamere koju želimo koristiti. Unutar funkcije provjerava se je li kamera uspješno inicijalizirana te se putem varijabli **colorMode** i **displayMode** postavljaju način rada kamere i način prikaza slike. Za ovaj slučaj odabran je način rada kamere u boji (**IS_CM_BGR8_PACKED**) te spremanje i prikaz slike iz RAM memorije uređaja (**IS_SET_DM_DIB**). U funkciji **getFrame** alocira se memorija u koju se sprema slika putem naredbe **is_AllocImageMem.** Aktiviranje memorije i dohvaćanje slike vrši se naredbama **is_SetImageMem** te **is_FreezeVideo**. Također se unutar funkcije slika prebacuje u **Mat** format u kojem se kasnije vrši obrada slike. Memorija u koju je slika spremljena oslobađa se putem naredbe **is_FreeImageMem** kako bi se nova slika mogla ponovno spremiti.

```
void initializeCameraInterface(HIDS* hCam_internal) {
    // Inicijaliziranje kamere
    INT nRet = is_InitCamera(hCam_internal, NULL);
    if (nRet == IS_SUCCESS) {
        cout << "Camera initialized!" << endl;
    }
    // Postavljanje ColorMode-a kamere
    INT colorMode = IS_CM_BGR8_PACKED;
    nRet = is_SetColorMode(*hCam_internal, colorMode);
    if (nRet == IS_SUCCESS) {
        cout << "Camera color mode succesfully set!" << endl;
    }
    //Postavljanje DisplayMode-a kamere
    INT displayMode = IS_SET_DM_DIB;
    nRet = is_SetDisplayMode(*hCam_internal, displayMode);
}

Mat getFrame(HIDS* hCam, int width, int height, Mat& mat) {
    // alociranje memorije
    char* pMem = NULL;
    int memID = 0;
    is_AllocImageMem(*hCam, width, height, 24, &pMem, &memID);
    // Aktiviranje memorije za sliku i dohvaćanje slike
    is_SetImageMem(*hCam, pMem, memID);
    is_FreezeVideo(*hCam, IS_WAIT);
    // Prebacivanje slike u mat format
    VOID* pMem_b;
    int retInt = is_GetImageMem(*hCam, &pMem_b);
    if (retInt != IS_SUCCESS) {
        cout << "Image data could not be read from memory!" << endl;
    }
    memcpy(mat.ptr(), pMem_b, mat.cols * mat.rows * 3);
    is_FreeImageMem(*hCam, pMem, memID);
    return mat;
}
```

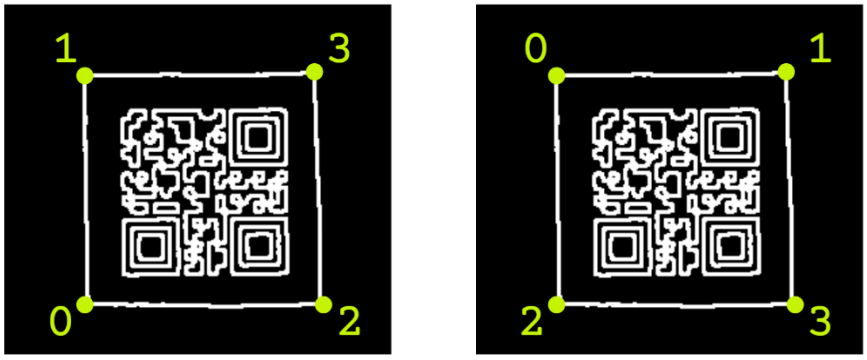
Definirana je i funkcija **getContours** koja vraća vektor rubnih točaka najveće zatvorene konture na slici koju kao ulaz prima u **Mat** formatu. U funkciji se definira vektor **contours** koji će se sastojati od vektora točaka svih vanjskih kontura pronađenih na slici. Zatim se poziva OpenCV funkcija **findContours** koja u vektor **contours** sprema sve vektore točaka vanjskih kontura pronađenih na slici. Definira se vektor **conPoly** u koji će se spremati vektori svih točaka aproksimiranih kontura pomoću postojeće OpenCV funkcije **approxPolyDP**. Također, definira se i vektor **biggest** u koji će se spremiti najveća detektirana kontura te će to ujedno biti i izlaz koji vraća funkcija **getContours**. Uz to je potrebno i definirati varijablu **maxArea** preko koje će se tražiti najveća kontura. Kroz for petlju prolazi se kroz sve konture spremljene u vektor **contours**. OpenCV funkcija **contourArea** računa površinu koju pojedina kontura zatvara te, ako je ta površina veća od određene (u ovom slučaju je odabrana vrijednost 15000 budući da se detekcija vrši na slikama rezolucije 1296x966 ili većim, a naljepnica zauzima veliki dio površine slike), koristeći postojeću OpenCV funkciju **approxPolyDP** postojeću konturu aproksimiramo najbližim n-terokutom te spremamo vrijednosti njegovih vanjskih točaka u vektor **conPoly**. Ako je površina te konture veća od trenutno maksimalne zabilježene površine u varijabli **maxArea**, te je ta kontura aproksimirana četverokutom, uzimamo četiri vrha te aproksimirane konture te ih bilježimo u vektor **biggest**. Također, spremamo površinu te konture u varijablu **maxArea**. Postupak ponavljamo kroz for petlju sve dok ne pretražimo sve konture sa slike.

```
vector<Point> getContours(Mat image) {

    vector<vector<Point>> contours;
    vector<Vec4i> hierarchy;
    findContours(image, contours, hierarchy, RETR_EXTERNAL, CHAIN_APPROX_SIMPLE);
    vector<vector<Point>> conPoly(contours.size());
    vector<Rect> boundRect(contours.size());
    vector<Point> biggest;
    int maxArea = 0;
    for (int i = 0; i < contours.size(); i++){
        int area = contourArea(contours[i]);
        if (area > 15000) {
            float peri = arcLength(contours[i], true);
            approxPolyDP(contours[i], conPoly[i], 0.02 * peri, true);
            if (area > maxArea && conPoly[i].size() == 4) {
                biggest = {conPoly[i][0],conPoly[i][1],conPoly[i][2]
,conPoly[i][3]};
                maxArea = area;
            }
        }
    }
    return biggest;
}
```

Definira se i funkcija **rearrange** koja kao ulaz prima vektor točaka te na izlazu vraća taj vektor točaka poredan tako da zadovoljava uvjete OpenCV funkcije **warpPerspective**. Potrebno je poredati točke tako da gornja lijeva točka konture bude spremljena kao prva, gornja desna točka konture kao druga, donja lijeva točka konture kao treća i donja desna točka konture kao četvrta točka u izlaznom vektoru. Zato u funkciji definiramo vektor točaka **newPoint** te vektore **sumPoints** i **subPoints** u koje ćemo spremati zbroj i razliku vrijednosti x i y koordinata točaka. For petljom zbrajamo i oduzimamo x i y koordinate svake od četiri vršnih točaka konture te ih naredbom **push_back** dodajemo na kraj vektora **sumPoints** i **subPoints**. Budući da za gornju lijevu točku vrijedi da joj je zbroj x i y koordinata minimalan, za gornju desnu točku vrijedi da joj je razlika x i y koordinata maksimalna, za donju lijevu točku vrijedi da joj je razlika x i y koordinata minimalna te za donju desnu točku vrijedi da joj je zbroj x i y koordinata maksimalan (ishodište se nalazi u gornjem lijevom kutu slike, ne radi se o apsolutnim vrijednostima kod računanja razlike) na sljedeći način možemo pravilno poredati točke. Naredbom **push_back** prvo na kraj vektora **newPoints** stavljamo točku čiji je zbroj x i y koordinata minimalan. Zatim istom naredbom na kraj istog vektora stavljamo točku čija je razlika x i y koordinata maksimalna te tako redom. Tako će vektor **newPoints** sadržati vršne točke najveće konture ispravno poredane kako bi naredba **warpPerspective** mogla dati ispravan izlaz.

```
vector<Point> rearrange(vector<Point> points) {

    vector<Point> newPoints;
    vector<int>  sumPoints, subPoints;

    for (int i = 0; i < 4; i++) {
        sumPoints.push_back(points[i].x + points[i].y);
        subPoints.push_back(points[i].x - points[i].y);
    }

    newPoints.push_back(points[min_element(sumPoints.begin(), sumPoints.end()) -
sumPoints.begin()]);
    newPoints.push_back(points[max_element(subPoints.begin(), subPoints.end()) -
subPoints.begin()]);
    newPoints.push_back(points[min_element(subPoints.begin(), subPoints.end()) -
subPoints.begin()]);
    newPoints.push_back(points[max_element(sumPoints.begin(), sumPoints.end()) -
sumPoints.begin()]);

    return newPoints;
}
```

Na slici 13. prikazane su detektirane rubne točke konture te njihovi indeksi prije i poslije pozivanja `rearrange` funkcije.



Slika 13. Detektirane rubne točke konture i njihovi indeksi prije (lijevo) i poslije (desno) pozivanja funckije `rearrange`

Definira se i funkcija **`getWarp`** koja na ulazu prima **`Mat`** sliku, vektor točaka, te dva broja koja predstavljaju širinu i visinu **`Mat`** slike koju funkcija daje na izlazu. Definiraju se i točke vektora **`src`** koji se sastoji od 4 ispravno poredanih vršnih točaka konture te točke vektora **`dst`** koje su zapravo četiri vršne točke slike u koju želimo spremiti detektiranu naljepnicu. Pomoću OpenCV funkcije **`getPerspectiveTransform`** se u matricu **`matrix`** sprema matrica transformacija danih točaka (spremljenih u vektore **`src`** i **`dst`**). Nakon toga poziva se OpenCV funkcija **`warpPerspective`** koja kao ulaz prima ulaznu sliku u **`Mat`** formatu, matricu transformacija točaka te širinu i visinu izlazne slike te vraća transformiranu sliku u **`Mat`** formatu.

```
Mat getWarp(Mat image, vector<Point> points, float w, float h) {

    Point2f src[4] = { points[0],points[1],points[2],points[3] };
    Point2f dst[4] = { {0.0f,0.0f},{w,0.0f},{0.0f,h},{w,h} };
    Mat matrix = getPerspectiveTransform(src, dst);
    warpPerspective(image, imgWarp, matrix, Point(w, h));

    return imgWarp;
}
```

Nakon definiranja potrebnih korisničkih funkcija ulazi se u glavni dio programa. Pomoću kreirane funkcije **`initializeCameraInterface`** inicijalizira se kamera te se pomoću ugrađene OpenCV funkcije **`cuda::printCudaDeviceInfo(0)`** ispisuju informacije o korištenom uređaju (računalu) za izvođenje aplikacije. Slijedi definiranje potrebnih parametara za obradu slike putem GPU. Pomoću OpenCV naredbe **`cuda::createGaussianFilter`** definiramo parametre Gaussian filtera koji će se primijeniti na sliku. Pomoću OpenCV naredbe

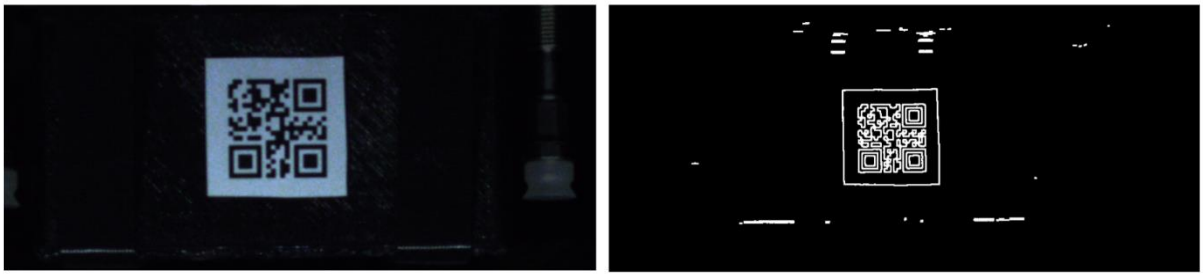
`cuda::createCannyEdgeDetector` definira se detektor rubova koji će se kasnije koristiti. OpenCV naredbom `getStructuringElement` definira se element koji će se koristiti za dilataciju slike. Kreira se i filter za dilataciju pomoću OpenCV naredbe `cuda::createMorphologyFilter`. Parametri za navedene filtere nisu dodatno objašnjavani zato što su specifični za svaku pojedinu primjenu te se više o njihovim vrijednostima može pronaći na službenim OpenCV internetskim stranicama [13].

```
initializeCameraInterface(&hCam);
cuda::printCudaDeviceInfo(0);
Ptr<cuda::Filter> gaussian;
Ptr<cuda::Filter> dilate;
gaussian = cuda::createGaussianFilter(CV_8UC1, CV_8UC1, Size(1, 1), 1);
Ptr<cuda::CannyEdgeDetector> C = cuda::createCannyEdgeDetector(25, 35, 1, false);
Mat element = getStructuringElement(MORPH_RECT, Size(3, 3));
dilate = cuda::createMorphologyFilter(MORPH_DILATE, CV_8UC1, element);
```

Nakon toga ulazi se u while petlju koja se izvršava sve do pritiska tipke za izlaz (`q`). Naredbom `getTickCount()` u varijablu `start` spremamo vrijeme pokretanja tajmera za kasnije određivanje FPS-a. Korištenjem kreirane korisničke funkcije `getFrame` dohvaća se slika s kamere te se u `Mat` formatu sprema u varijablu `img`. Pomoću naredbe `upload` slika iz varijable `img` učitava se u `GpuMat` formatu u varijablu `imggpu`. Kad je slika učitana na GPU kreće i obrada same slike. Pomoću OpenCV naredbe `cuda::cvtColor` slika u boji prebacuje se u sliku sivih tonova. Zatim se kreirani Gaussian filter naredbom `apply` primjenjuje na tu sliku kako bi se rubovi mogli lakše detektirati. Naredbom `detect` kreiranim Canny Edge detektorom traže se rubovi slike. Naposlijetku se ponovno naredbom `apply` primjenjuje dilate filter kreiran ranije kako bi se podebljali detektirani rubovi. Slika se potom naredbom `download` preuzima nazad na CPU u sliku `imgprocessedcpu Mat` formata.

```
auto start = getTickCount();

getFrame(&hCam, img.cols, img.rows, img);

imggpu.upload(img);

cuda::cvtColor(imggpu, imggray, COLOR_BGR2GRAY);
gaussian->apply(imggray, imgblur);
C->detect(imgblur, imgcanny);
dilate->apply(imgcanny, imgprocessed);

imgprocessed.download(imgprocessedcpu);
```

Slika 14. prikazuje sliku koju kamera dohvaća (`img`) te obrađenu sliku (`imgprocessedcpu`).



Slika 14. Slika koju kamera dohvaća (lijevo) i obrađena slika (desno)

Kad je slika obrađena, nastavlja se s traženjem kontura na obrađenoj slici. Pozivanjem definirane korisničke funkcije `getContours`, koja na ulazu prima sliku `imgprocessedcpu`, u vektor `initialPoints` spremaju se početne rubne točke najveće detektirane konture na slici `imgprocessedcpu`. Ako je vanjska kontura pronađena, pozivanjem kreirane korisničke funkcije `rearrange`, koja kao ulaz prima vektor točaka spremljen u vektor `initialPoints`, pravilno se raspoređuju rubne točke detektirane konture te se takve spremaju u vektor `arrangedPoints`. Ako vektor `arrangedPoints` nije prazan, pozivanjem korisničke funkcije `getWarp`, koja kao ulaz prima izvornu sliku `img`, vektor poredanih koordinata rubnih točaka `arrangedPoints` te širinu i visinu izlazne slike `w` i `h`, u varijablu `imgWarped` sprema se detektirana i ispravljena slika naljepnice. Ako je broj već spremljenih slika naljepnica manji od unaprijed određenog i zapisanog u varijablu `noimg`, detektirana naljepnica trenutno spremljena u varijablu `imgWarped` trajno se sprema u mapu projekta kako bi se kasnije mogla vršiti detekcija QR kodova sa spremljenih slika naljepnica.

```
initialPoints = getContours(imgprocessedcpu);
if (!initialPoints.empty()) {
	arrangedPoints = rearrange(initialPoints);
	//warping
	if (!arrangedPoints.empty()) {
		imgWarped = getWarp(img, arrangedPoints, w, h);
		//spremanje slike
		if (b <= noimg) {
			b = b + 1;
			string ime = to_string(b) + ".tif";
			imwrite(ime, imgWarped);
		}
	}
}
```

Kad se dovrši obrada slike i spremanje slike naljepnice naredbom **getTickCount,** u varijablu **end** sprema se trenutno vrijeme za određivanje FPS-a. Iz spremljenih vremena u varijablama **start** i **end** računa se FPS (frames per second) te se sprema u varijablu **fps.** Naredbom **putText** ispisuje se vrijednost zabilježena u varijabli **fps** na početnu sliku **img**.

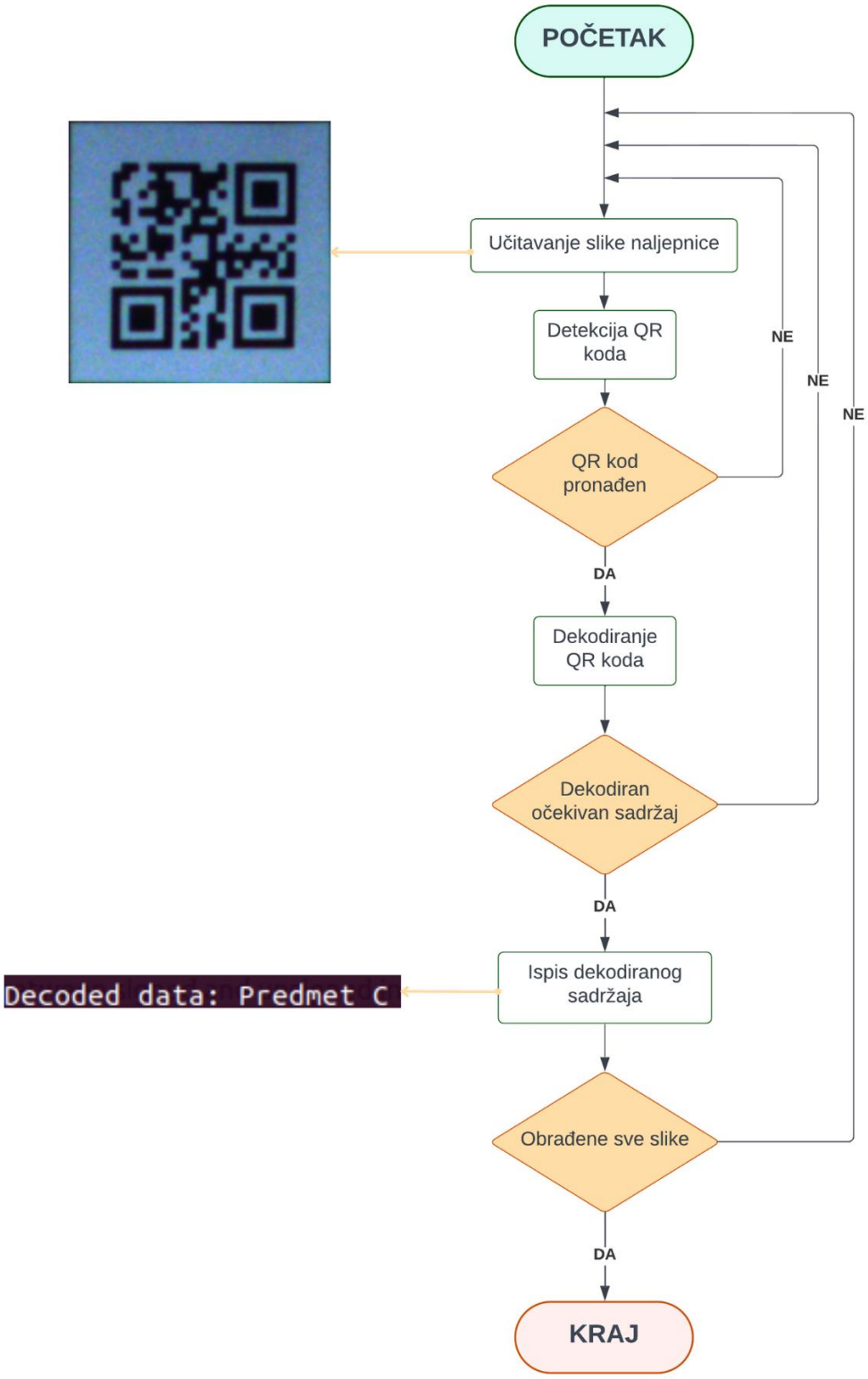
```
auto end = getTickCount();
auto totalTime = (end - start) / getTickFrequency();
auto fps = 1 / totalTime;

putText(img, "FPS: " + to_string(int(fps)), Point(20, 40), FONT_HERSHEY_SIMPLEX,
1, Scalar(0, 2, 237), 3, false);
```

Slijedi prikaz originalne slike, obrađene slike te izdvojene slike naljepnice naredbom **imshow**. Za ubrzanje rada aplikacije prikaz slika može se i isključiti. Kako je navedeno ranije, ovaj proces se ponavlja u while petlji sve dok ne pritisnemo tipku **q**. Pritiskom tipke **q** izlazi se iz while petlje, naredbom **is_ExitCamera** gasi se kamera te slijedi kod za offline dekodiranje QR kodova sa spremljenih slika naljepnice.

```
imshow("Image", img);
imshow ("Procesirana slika", imgprocessedcpu);
imshow("Naljepnica", imgWarped);

//izlaz iz while petlje
if (waitKey(2) == 'q') {
      break;
}
}     //zatvorena zagrada while petlje
is_ExitCamera(hCam);
```

3.3. Algoritam za offline dekodiranje QR koda sa detektiranih naljepnica

Za početak je dan dijagram toka za offline dekodiranje QR koda sa spremljenih slika detektiranih naljepnica. Svaki od koraka bit će detaljno objašnjen te će se ukratko opisati korištene funkcije iz OpenCV paketa.



Dekodiranje započinje inicijalizacijom ugrađene OpenCV funkcije `QRCodeDetector`. Inicijalizira se i vektor točaka `points` u koji će se spremati detektirane rubne točke QR kodova. Definira se varijabla `count` te se njena vrijednost postavlja u 0. Pomoću te varijable provjeravat će se broj ispravno dekodiranih QR kodova. Naredbom `getTickCount` u varijablu

startqr spremamo vrijeme početka obrade detektiranih naljepnica s QR kodovima kako bismo na kraju mogli ispisati ukupno potrebno vrijeme za dekodiranje QR kodova.

```
QRCodeDetector decoder = QRCodeDetector();
vector<Point> points;
int count = 0;
auto startqr = getTickCount();
```

Korištenjem for petlje prolazi se kroz sve spremljene slike te ih se pomoću naredbe **imread** učitava u varijablu **qr**. Pozivom OpenCV funkcije **detectAndDecode** u varijablu **predmet** sprema se dekodirani sadržaj s trenutno učitane slike u varijabli **qr** te se u vektor točaka **points** spremaju rubne točke detektiranog QR koda s iste slike. Ako vektor točaka **points** nije prazan (odnosno QR kod je uspješno detektiran na slici), te ako dekodirani sadržaj QR koda odgovara očekivanom (u ovom slučaju vrši se detekcija QR kodova koji sadrže tekst 'Predmet A', 'Predmet B' te 'Predmet C') naredbom **cout** ispisuje se dekodirani sadržaj QR koda spremljen u varijablu **predmet**. U tom slučaju također se povećava i vrijednost varijable **count** kako bi se kasnije mogao ispisati broj uspješno detektiranih i dekodiranih QR kodova.

```
for (int a = 1; a <= b - 1; a++) {
	string imea = to_string(a) + ".tif";
	Mat qr = imread(imea, IMREAD_UNCHANGED);
	string predmet = decoder.detectAndDecode(qr, points);
	if (!points.empty()) {
		if (predmet.rfind("P", 0) == 0) {
		    cout << "Decoded data: " << predmet << endl;
		    count = count + 1;
		}
	}
}
```

4. TESTIRANJE SUSTAVA

Vizijski sustav postavljen je u laboratoriju CRTA-e pored Fanuc DR-3iB/8L delta robota. Delta robot je korišten kako bi se putem upravljačke jedinice robota mogle zadati točne brzine kretanja predmeta s naljepnicom te tako dobiti relevantne rezultate testiranja pri različitim brzinama. Kamere su postavljene na udaljenosti od 70 cm od predmeta s naljepnicom te je na toj udaljenosti kadar koji hvata IDS kamera širine 25 cm dok je kadar koji hvata Basler kamera širine 12 cm.

Testiranje je provedeno tako da predmet ispred kamere prolazi 50 puta određenom brzinom te program sprema slike detektirane naljepnice. Slika naljepnice se sprema samo ako je na slici uspješno detektirana čitava naljepnica. Nakon toga, vrši se dekodiranje sadržaja spremljenih slika naljepnice te se, u ovisnosti o mogućnosti dekodiranja QR koda, evaluira kvaliteta spremljenih slika naljepnice. Ako je QR kod sa spremljene slike uspješno dekodiran, kvaliteta slike smatra se zadovoljavajućom. Testiranje je provedeno s kvadratnim naljepnicama površine 4 i 9 cm^2 te brzinama kretanja predmeta od 2000, 2500, 3000, 3500 i 4000 mm/s. Obje kamere nalaze se na istoj udaljenosti od predmeta.

Program za trenutnu obradu i spremanje slike naljepnice izvršava se na GPU brzinom dovoljnom da obradi između 38 i 45 slika u sekundi rezolucije 1296x966 piksela, te između 30 i 35 slika u sekundi rezolucije 1920x1080 piksela. Program za dekodiranje spremljenih slika izvršava se na CPU te može obraditi do 5 slika u sekundi, te se iz tog razloga ne izvodi paralelno s programom za obradom slike, već po završetku njegovog izvođenja. Na brzinu i kvalitetu obrade slike znatno utječu i osvjetljenje te vrijeme ekspozicije kamere. Kod testiranja je korišteno dodatno osvjetljenje kako bi se vrijeme ekspozicije kamere moglo smanjiti, a naljepnica i dalje ostati jasno vidljiva.

4.1. Testiranje brzine i kvalitete detekcije naljepnice s IDS kamerom

IDS kamera dohvaća sliku u boji visoke rezolucije (1920x1080 piksela) 58 puta u sekundi (58 FPS). IDS kamera koristi rolling shutter. Kadar koji dohvaća IDS kamera dan je na slici 15. U Tablici 5. dani su rezultati testiranja dobiveni s IDS kamerom i naljepnicom površine 9 cm^2, a u Tablici 6. rezultati testiranja dobiveni s naljepnicom površine 4 cm^2. Predmet ispred kamere prolazi 50 puta.

Slika 15. Kadar koji dohvaća IDS kamera

Tablica 5. Rezultati testiranja s IDS kamerom i naljepnicom površine 9 cm²

Brzina kretanja predmeta s naljepnicom [mm/s]	Broj spremljenih slika naljepnice	Broj uspješno dekodiranih slika naljepnice
2000	50	50
2500	50	50
3000	50	50
3500	47	47
4000	45	44

Tablica 6. Rezultati testiranja s IDS kamerom i naljepnicom površine 4 cm²

Brzina kretanja predmeta s naljepnicom [mm/s]	Broj spremljenih slika naljepnice	Broj uspješno dekodiranih slika naljepnice
2000	50	44
2500	50	35
3000	50	32
3500	47	17
4000	45	8

4.2. Testiranje brzine i kvalitete detekcije naljepnice s Basler kamerom

Basler kamera dohvaća sliku sivih tonova rezolucije 1296x966 piksela 30 puta u sekundi (30 FPS). Basler kamera koristi global shutter. Kadar koji dohvaća Basler kamera dan je na slici 16. U Tablici 7. dani su rezultati testiranja dobiveni s Basler kamerom i naljepnicom površine 9 cm^2, a u Tablici 8. rezultati testiranja dobiveni s naljepnicom površine 4 cm^2. Predmet ispred kamere prolazi 50 puta.

Slika 16. Kadar koji dohvaća Basler kamera

Tablica 7. Rezultati testiranja s Basler kamerom i naljepnicom površine 9 cm^2

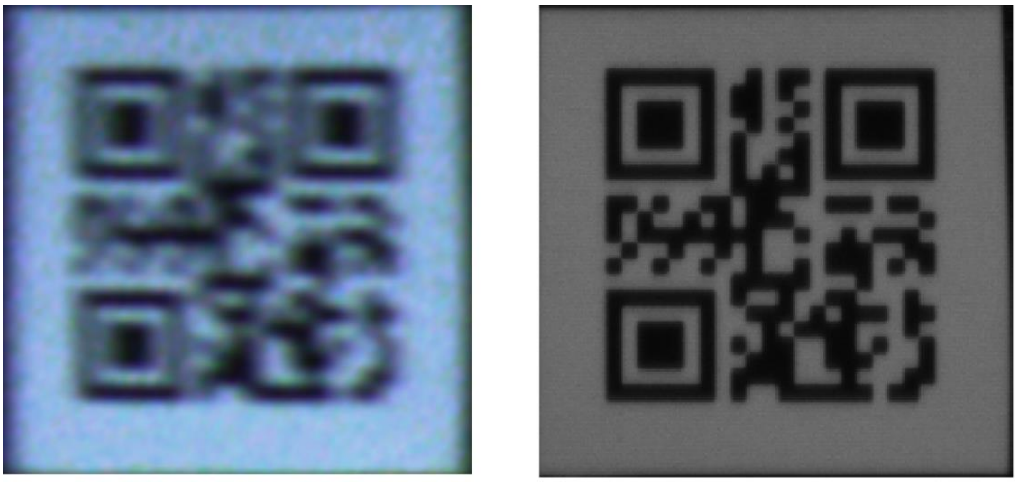
Brzina kretanja predmeta s naljepnicom [mm/s]	Broj spremljenih slika naljepnice	Broj uspješno dekodiranih slika naljepnice
2000	68	68
2500	54	54
3000	50	50
3500	50	50
4000	48	48

Tablica 8. Rezultati testiranja s Basler kamerom i naljepnicom površine 4 cm^2

Brzina kretanja predmeta s naljepnicom [mm/s]	Broj spremljenih slika naljepnice	Broj uspješno dekodiranih slika naljepnice
2000	64	64
2500	55	55
3000	50	50
3500	49	49
4000	45	45

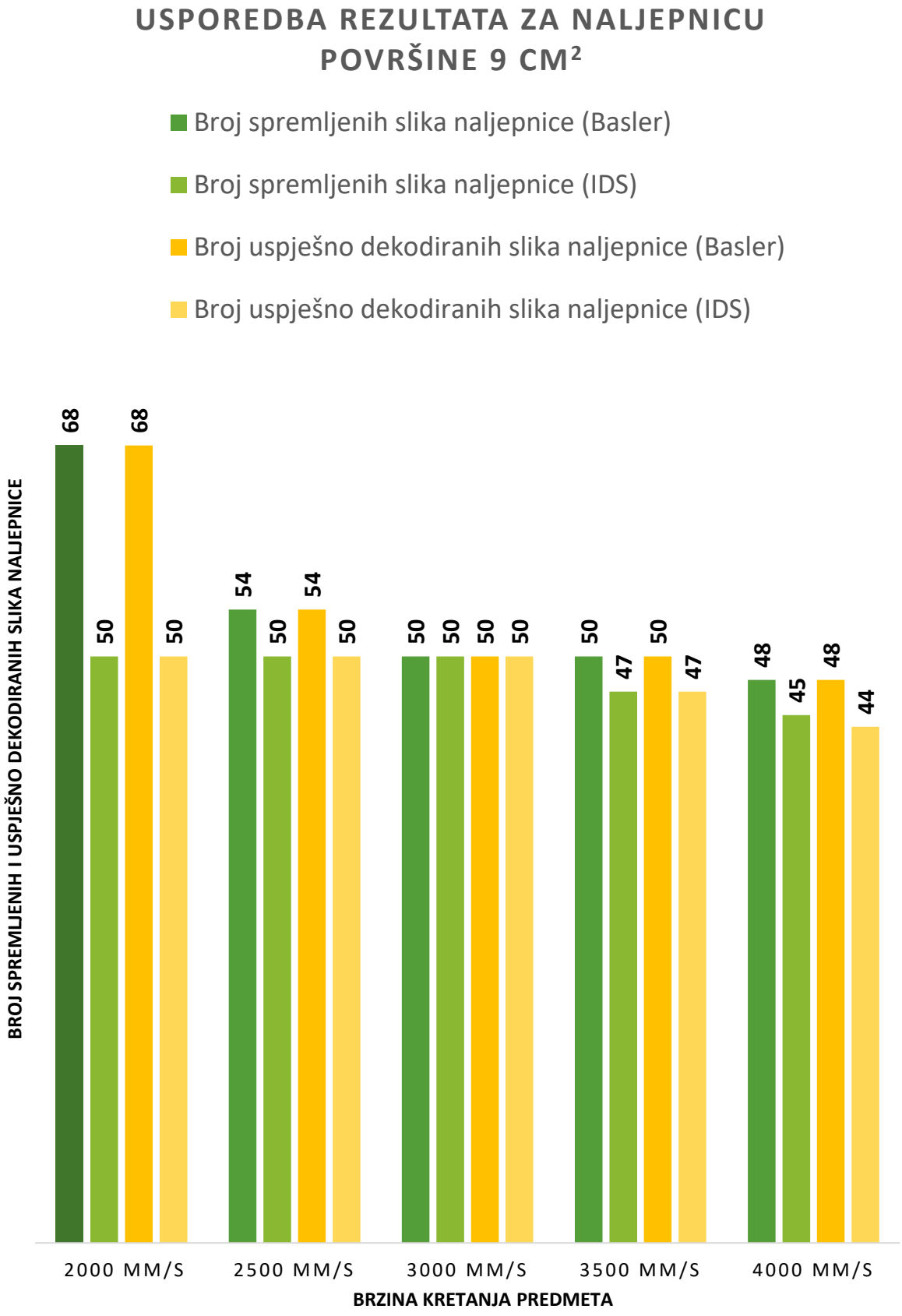
4.3. Usporedba i evaluacija rezultata testiranja

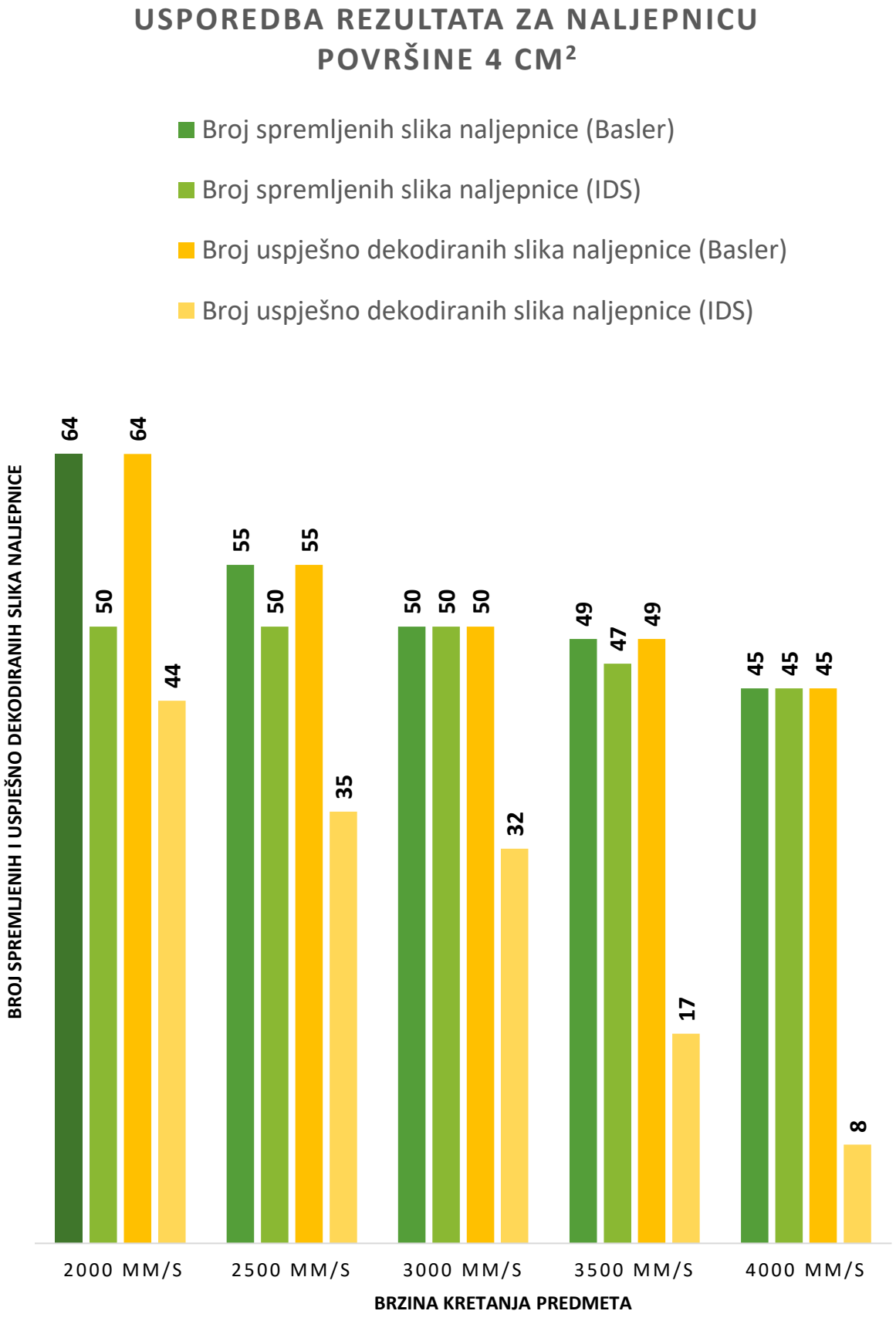
Dobiveni rezultati testiranja ukazuju na prednost korištenja kamere s global shutterom kod snimanja brzo putujućih predmeta. Iako IDS kamera dohvaća sliku veće rezolucije, zbog velike brzine kretanja predmeta, slika je djelomično deformirana te je zbog te deformacije nemoguće uspješno dekodirati QR kod. Iz rezultata je vidljivo kako povećanje brzine kretanja predmeta ne utječe na mogućnost detektiranja same naljepnice, no znatno utječe na kvalitetu dohvaćene slike s IDS kamerom koja koristi rolling shutter. Slika 17. prikazuje usporedbu spremljene slike naljepnice površine 4cm^2 pri brzini kretanja predmeta od 4000 mm/s snimljenu s IDS i Basler kamerom.



Slika 17. Usporedni prikaz spremljene slike naljepnice pri brzini kretanja predmeta od 4000 mm/s snimljene s IDS (lijevo) i Basler kamerom (desno)

Grafički prikaz dobivenih rezultata s obje kamere za obje naljepnice dan je u nastavku.




USPOREDBA REZULTATA ZA NALJEPNICU POVRŠINE 4 CM²
Broj spremljenih slika naljepnice (Basler)
Broj spremljenih slika naljepnice (IDS)
Broj uspješno dekodiranih slika naljepnice (Basler)
Broj uspješno dekodiranih slika naljepnice (IDS)
BROJ SPREMLJENIH I USPJEŠNO DEKODIRANIH SLIKA NALJEPNICE
64
50
64
44
55
50
55
35
50
50
50
32
49
47
49
17
45
45
45
8
2000 MM/S
2500 MM/S
3000 MM/S
3500 MM/S
4000 MM/S
BRZINA KRETANJA PREDMETA

Sustav radi u slobodnom načinu rada (bez okidača) te je kod provođenja testiranja s Basler kamerom pri brzinama većim od 3000 mm/s u dohvaćenim kadrovima naljepnica ponekad bila samo djelomično vidljiva. Do toga je dolazilo budući da se testiranje provodilo s naljepnicama širina 2 i 3 cm, a kadar Basler kamere pokrivao je kadar širine 12 cm kako je i prije navedeno. Budući da Basler kamera dohvaća 30 slika u sekunda, pri brzinama većim od 3000 mm/s predmet se u vremenu između dohvaćanja dvije slike pomicao za više od 100 mm te bi se ponekad dohvatila dva uzastopna kadra s djelomično vidljivom naljepnicom. Do toga nije dolazilo kod dohvaćanja slike s IDS kamerom, no zbog rolling shuttera kojeg koristi ta kamera, dohvaćene slike naljepnice bile su lošije kvalitete. Kako bi se izbjegli slučajevi nepotpune detekcije naljepnice predlaže se implementacija vanjskog okidača koji sinkronizirano s kretnjom robota šalje signal kameri te tako osigurava okidanje slike u trenutku kada se čitava naljepnica nalazi u kadru. Primjer kadra s djelomično vidljivom naljepnicom snimljen Basler kamerom pri brzini kretanja predmeta od 4000 mm/s dan je na slici 18.

Slika 18. Prikaz kadra s djelomično vidljivom naljepnicom snimljen Basler kamerom pri brzini kretanja predmeta od 4000 mm/s

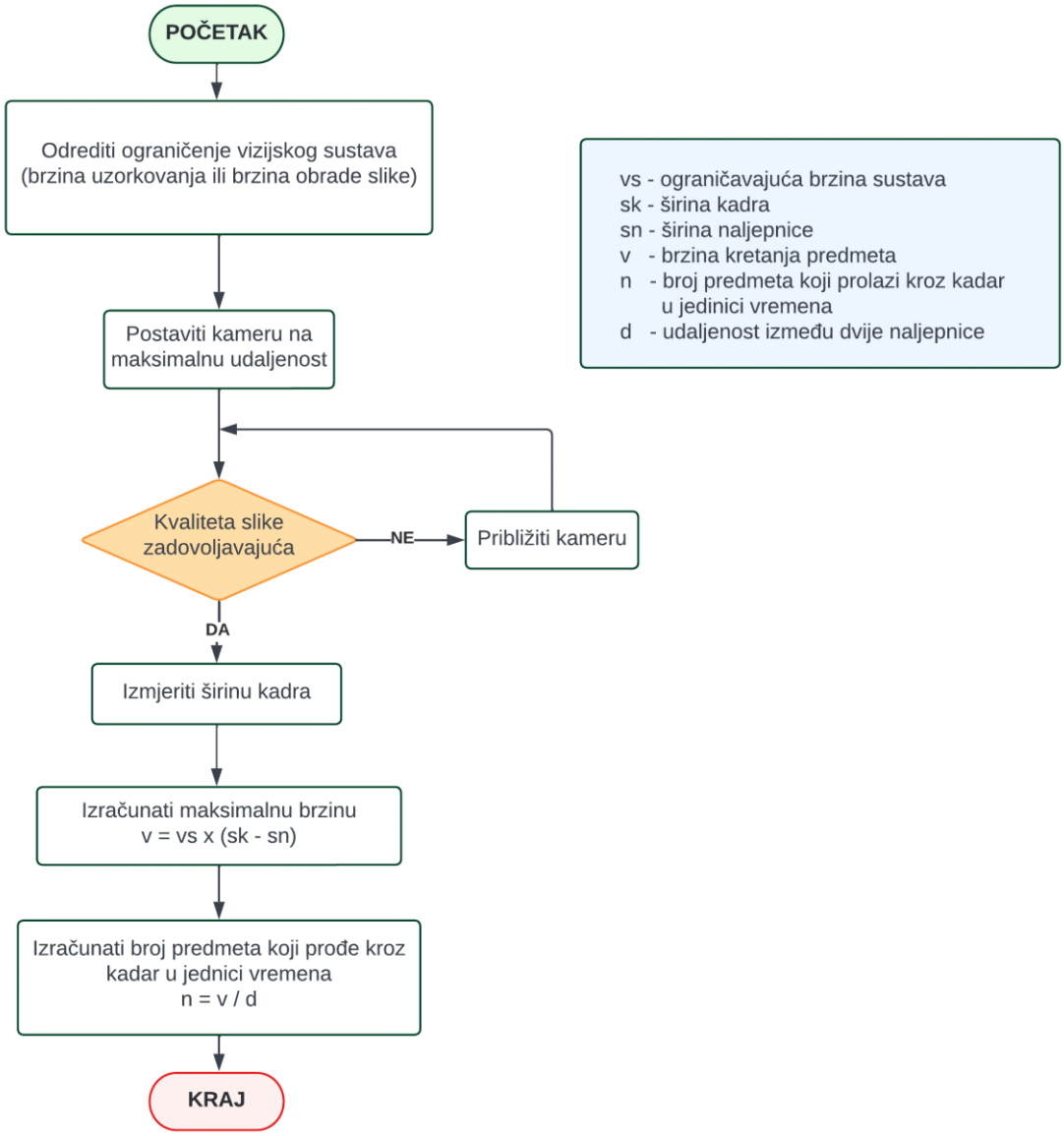
4.4. Mogućnosti i ograničenja sustava

Provedeno testiranje pokazuje mogućnosti, ali i ograničenja praktične primjene ovog vizijskog sustava. Rezultati provedenog testiranja pokazuju nadmoć korištenja kamere s global shutterom pri snimanju brzo putujućih objekata.

Korištena Basler kamera dohvaća sliku 30 puta u sekundi što predstavlja i prvo ograničenje sustava. Budući da izrađena vizijska aplikacija obrađuje između 38 i 45 slika u sekundi (rezolucije 1296x966 px), što je više slika u sekundi nego što kamera dohvaća, zaključujemo da brzina obrade slike ne predstavlja ograničenje ovog sustava. Prema rezultatima testiranja vidljivo je da maksimalna brzina kretanja predmeta pri kojoj je svaki od njih uspješno detektiran iznosi 3000 mm/s. Ako se maksimalna brzina kretanja predmeta podijeli s ograničavajućom brzinom uzorkovanja slike (u ovom slučaju je to 30 FPS), dobiva se udaljenost koju predmet prođe između uzorkovanja i obrade dvije uzastopne slike. U ovom slučaju ta udaljenost iznosi 100 mm. Budući da širina kadra koju Basler kamera u eksperimentalnom postavu dohvaća iznosi 120 mm, a širina manje naljepnice korištene u testiranju iznosi 20 mm, potvrđuje se i matematička osnova po kojoj je osigurana detekcija naljepnice pri svakom prolasku predmeta. Maksimalan broj spremljenih slika naljepnice ovisi o dostupnoj memoriji sustava. Jedna spremljena slika naljepnice zauzima 500 kilobajta prostora. Samo dekodiranje spremljenih slika izvršava se na CPU nakon što se program za obradu i spremanje slika završi. Program za dekodiranje može obraditi do 5 slika u sekundi.

Kod postavljanja vizijskog sustava treba obratiti pozornost na obrnuto proporcionalan odnos kvalitete dohvaćene slike naljepnice s obzirom na udaljenost naljepnice od kamere. Postavljanjem predmeta dalje od kamere možemo povećati brzinu kojom se predmeti kreću te tako povećati produktivnost, no ograničavajući faktor prema kojem treba postaviti čitav sustav će uvijek biti brzina uzorkovanja ili brzina obrade slike. Ako se brzina predmeta poveća iznad maksimalne nije moguće osigurati okidanje i obradu slike za svaki od prolazećih predmeta te se može dogoditi slučaj djelomične detekcije naljepnice prikazan ranije.

U nastavku je dan dijagram toka za određivanje maksimalnog mogućeg broja objekata koje vizijski sustav može obraditi u jedinici vremena te izračun maksimalnog broja objekata koje ovaj vizijski sustav može uspješno detektirati bez pojave djelomične detekcije naljepnice u jedinici vremena za prikazani eksperimentalni postav.



Ograničavajući faktor u provedenom testiranju bila je brzina uzorkovanja slike Basler kamere (30 FPS). Širina kadra *(sk)* koji dohvaća Basler kamera iznosi 120 mm. Širine naljepnica *(sn)* s kojima se provodilo testiranje iznose 20 i 30 mm. Za ovaj slučaj promatrat će se rezultati dobiveni za naljepnicu širine 20 mm. Dobiva se da maksimalna brzina predmeta *(v)* pri kojoj se sve naljepnice i dalje mogu uspješno detektirati iznosi 3000 mm/s, što potvrđuju i rezultati testiranja. Udaljenost između dvije naljepnice *(d)* mora biti barem tolika da se ni u jednom trenutku dvije naljepnice cijelom površinom ne nalaze u kadru kojeg dohvaća kamera. Za maksimalnu učinkovitost sustavu navedenu udaljenost potrebno je minimizirati. Minimalni iznos udaljenosti između dvije naljepnice dobiva se oduzimanjem širine naljepnice od širine kadra. U ovom slučaju minimalna udaljenost između dvije naljepnice iznosi 100 mm.

Dijeljenjem maksimalne brzine predmeta s udaljenosti između dvije naljepnice dobiva se maksimalan broj predmeta *(n)* koji sustav može procesuirati u jedinici vremena. Slijedi kako testirani sustav može detektirati do 30 naljepnica u sekundi.

$$vs = 30\ \mathrm{s}^{-1}$$

$$sk = 120\ \mathrm{mm}$$

$$sn = 20\ \mathrm{mm}$$

$$v = vs \cdot (sk - sn) = 30 \cdot (120 - 20) = 3000\ \mathrm{mm/s} \quad (1)$$

$$d = d_{min} = sk - sn = 120 - 20 = 100\ \mathrm{mm} \quad (2)$$

$$n = \frac{v}{d} = \frac{3000}{100} = 30\ \mathrm{s}^{-1} \quad (3)$$

Iz prikazanog postupka jasno je vidljivo kako sustav može procesuirati maksimalno onu količinu podataka koju definira ograničavajuća brzina sustava. Za testirani sustav ograničavajući faktor bila je brzina uzorkovanja slike kamere te je u tom slučaju maksimalan broj predmeta koji sustav može obraditi u jedinici vremena jednak brzini uzorkovanja slike kamere.

Vizijska aplikacija izrađena u sklopu ovog završnog rada vrši i dekodiranje sadržaja sa spremljenih slika naljepnice. Prema ranije navedenom, dekodiranje sadržaja vrši se na CPU brzinom do 5 slika u sekundi. Kako se u memoriji sustava ne bi nakupljao velik broj slika preporučuje se dekodiranje spremljenih slika u kraćim vremenskim intervalima te brisanje dekodiranih slika. Budući da se dekodiranje ne odvija paralelno s obradom slike, ovaj vizijski sustav za vrijeme dekodiranja spremljenih slika ne može istovremeno i dohvaćati sliku s kamere te spremati nove slike detektiranih naljepnica. Iz tog razloga, za vrijeme dekodiranja slika, predmeti moraju stajati.

Uz provedeni izračun maksimalnog broja detektiranih naljepnica u jedinici vremena provest će se i izračun maksimalnog broja detektiranih i dekodiranih naljepnica u jedinici vremena. Budući da se detekcija naljepnica i dekodiranje sadržaja ne odvijaju paralelno, vremena potrebna za odrađivanje pojedine zadaće programa potrebno je zbrojiti te se tako dobiva ukupno vrijeme ciklusa detekcije i dekodiranja sadržaja naljepnice. Ako sustav radi pri maksimalnoj brzini kretanja predmeta u jednoj sekundi detektira se 30 naljepnica. Zatim je potrebno 6 sekundi (5 naljepnica po sekundi) da bi se sadržaj 30 detektiranih naljepnica dekodirao. Za to vrijeme ostatak sustava mora stajati. Spremljene slike se nakon dekodiranja brišu te proces detekcije naljepnica može ponovno započeti. Zbrajanjem vremena potrebnim za detekciju i dekodiranje

30 naljepnica dobivamo ukupno vrijeme trajanja ciklusa koje iznosi 7 sekundi. Prema tome, sustav može uspješno detektirati i dekodirati do 4 naljepnice u sekundi. Slijedi kako u jednom satu sustav može detektirati i dekodirati do 14400, a u jednom danu do 345600 naljepnica.

$$n_{detektirano} = 30\ \mathrm{s}^{-1}$$
$$n_{dekodirano} = 5\ \mathrm{s}^{-1}$$
$$t_{uk} = \frac{n_{detektirano} + n_{dekodirano}}{n_{dekodirano}} = 7\ \mathrm{s} \quad (4)$$

Tablica 9. prikazuje najbitnije karakteristike testiranog vizijskog sustava. Tablica 10. prikazuje podatke o mogućnostima detekcije i dekodiranja naljepnica s Basler kamerom. Podaci o mogućnostima detekcije i dekodiranja naljepnica s IDS kamerom nisu prikazani budući da su inferiorni prema onima dobivenim za Basler kameru. IDS kamera je korištena u ovom radu kako bi se pokazala razlika između performansi kamera koje koriste global i rolling shutter.

Tablica 9. Karakteristike vizijskog sustava

Brzina uzorkovanja slike (IDS)	58 FPS
Brzina uzorkovanja slike (Basler)	30 FPS
Brzina obrade slike (IDS)	30 – 35 FPS
Brzina obrade slike (Basler)	38 – 45 FPS
Ograničavajuća brzina sustava (IDS)	30 FPS
Ograničavajuća brzina sustava (Basler)	30 FPS

Tablica 10. Podaci o mogućnostima detekcije i dekodiranja naljepnica sa Basler kamerom

Širina kadra	120 mm
Širina naljepnice	20 mm
Maksimalna brzina kretanja predmeta	3000 mm/s
Minimalna udaljenost između naljepnica	100 mm
Maksimalan broj detektiranih naljepnica	30 s^{-1}
Maksimalan broj detektiranih i dekodiranih naljepnica	4 s^{-1}

5. ZAKLJUČAK

Algoritmi za obradu slike neprestano se razvijaju te su se područja njihove primjene znatno proširila. Implementacijom ubrzanih algoritama za obradu slike putem GPU vizijski sustavi postaju kvalitetno rješenje za praćenje i dobivanje informacija i o brzo putujućim predmetima.

U ovom radu prikazan je postupak razvoja i postavljanja kompletnog vizijskog sustava koji služi za brzu detekciju oznaka na predmetima te za to koristi algoritme koji se izvršavaju na GPU. Kroz rad je dan i usporedni prikaz kamera s rolling i global shutterom te se kroz rezultate provedenog testiranja potvrdila ranije navedena premisa o prednostima korištenja kamere s global shutterom kod snimanja brzo putujućih predmeta.

Prema podacima dobivenim u testiranju može se zaključiti o prednostima integriranja ovakvog sustava u proizvodne i distributivne centre. Limitirajući faktor samog sustava jest brzina dekodiranja slika na CPU koji može obraditi do 5 slika u sekundi. Program za detekciju naljepnica može obraditi i do 45 slika u sekundi, no ograničenje na 30 slika u sekundi postavlja brzina uzorkovanja slike korištene Basler kamere. Kroz testiranje je pokazano kako Basler kamera koja koristi global shutter može dohvatiti kvalitetnu sliku naljepnice na predmetu koji se giba brzinom do 4000 mm/s. Kod postavljanja vizijskog sustava treba obratiti pozornost na osvjetljenje radnog prostora te prema predloženom postupku eliminirati mogućnost dohvaćanja slike s djelomično vidljivom naljepnicom.

Budući da se program za obradu slike na GPU i program za dekodiranje spremljene slike na CPU ne mogu izvršavati paralelno na istom uređaju, brzina ovog sustava limitirana je brzinom procesuiranja spremljenih slika naljepnice na CPU. Rezultati testiranja pokazali su da sustav u eksperimentalnom postavu može uspješno detektirati do 30 naljepnica u sekundi. Uračuna li se u vrijeme procesa i dekodiranje slika naljepnice, sustav uspješno može detektirati i dekodirati do 4 naljepnice u sekundi, odnosno do 14400 naljepnica u satu.

Razvijeni vizijski sustav moguće je unaprijediti implementacijom vanjskog okidača koji bi, sinkronizirano s kretanjem predmeta, kameri slao signal te osiguravao okidanje slike dok se predmet u potpunosti nalazi u kadru. Na taj način eliminirali bi se slučajevi djelomične detekcije oznaka do kojih može doći u slobodnom načinu rada pri većim brzinama. Uz to, umjesto na internu memoriju sustava, slike detektiranih naljepnica moguće je spremati na dijeljenu memoriju te na drugom računalu paralelno vršiti dekodiranje spremljenih slika bez prekidanja procesa dohvaćanja novih slika naljepnice. Na taj način znatno bi se smanjilo ukupno vrijeme procesa detekcije i dekodiranja naljepnice.

LITERATURA

[1] https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/

[2] https://en.ids-imaging.com/store/ui-3881le-af.html

[3] https://mvcamera.com/product/camera/basler-ace/gige/aca1300-30gm/

[4] C-S-25H0-096 Auto Focus Lens Module | M12 Auto Focus Lens | Corning

[5] 25mm C Series Fixed Focal Length Lens | Edmund Optics

[6] Rolling Shutter vs Global Shutter sCMOS Camera Mode- Oxford Instruments (oxinst.com)

[7] https://www.fanuc.eu/be/en/robots/robot-filter-page/delta-robots/dr-3ib-series/dr3ib-8l

[8] JetPack SDK | NVIDIA Developer

[9] SD Memory Card Formatter | SD Association (sdcard.org)

[10] balenaEtcher - Flash OS images to SD cards & USB drives

[11] Download details - IDS Imaging Development Systems GmbH (ids-imaging.com)

[12] https://github.com/rbonghi/jetson_stats

[13] OpenCV: OpenCV modules

[14] tryout_code_snippets/IDS_uEye_camera.cpp at master · StevenPuttemans/tryout_code_snippets (github.com)

PRILOZI

I. Repozitorij - https://github.com/IvanStrahija/JetsonXavierNX_MachineVisionSystem

II. Kod

a) postavke.pro

```
QT -= gui
QT += core
CONFIG += c++11 console
CONFIG -= app_bundle
DEFINES += QT_DEPRECATED_WARNINGS
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target

DESTDIR = $$system(pwd)
OBJECTS_DIR = $$DESTDIR/Obj
# C++ flags
QMAKE_CXXFLAGS_RELEASE =-03
SOURCES += main.cpp
QMAKE_CFLAGS_ISYSTEM = -I
CUDA_DIR = /usr/local/cuda-10.2
INCLUDEPATH += /usr/local/include/opencv4/
INCLUDEPATH += $$CUDA_DIR/include
INCLUDEPATH += -I/usr/include
INCLUDEPATH += -I/usr/local/cuda-10.2/include
QMAKE_LIBDIR += $$CUDA_DIR/lib64
LIBS += -lcudart -lcuda
LIBS += -L$$PWD/../../../../usr/lib/ -lueye_api
LIBS += -L/usr/local/lib -lopencv_stitching -lopencv_calib3d -
lopencv_core -lopencv_dnn -lopencv_features2d -lopencv_flann -
lopencv_gapi -lopencv_highgui -lopencv_imgcodecs -lopencv_imgproc -
lopencv_ml -lopencv_objdetect -lopencv_photo -lopencv_video -
lopencv_videoio -lopencv_cudaimgproc -lopencv_cudafilters -
lopencv_cudaarithm
CUDA_ARCH = sm_72
```

b) main.cpp

```
#include <iostream>
#include <string>
#include<stdio.h>
#include <opencv2/opencv.hpp>
#include "opencv2/objdetect/objdetect.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include <opencv2/core/cuda.hpp>
#include <opencv2/cudaimgproc.hpp>
#include <opencv2/cudafilters.hpp>
#include <opencv2/cudaarithm.hpp>
#include "ueye.h"
```

```
using namespace std;
using namespace cv;

void initializeCameraInterface(HIDS* hCam_internal){
    // Inicijaliziranje kamere
    INT nRet = is_InitCamera (hCam_internal, NULL);
    if (nRet == IS_SUCCESS){
        cout << "Camera initialized!" << endl;
    }
    //SENSORINFO m_sInfo;             // sensor information struct
        //is_GetSensorInfo(*hCam_internal, &m_sInfo);

    // Postavljanje ColorMode-a kamere
    INT colorMode = IS_CM_BGR8_PACKED;
    nRet = is_SetColorMode(*hCam_internal,colorMode);

    if (nRet == IS_SUCCESS){
        cout << "Camera color mode succesfully set!" << endl;
    }

    //Postavljanje DisplayModea - alociranje memorije za dohvacanje
slike
    INT displayMode = IS_SET_DM_DIB;
    nRet = is_SetDisplayMode (*hCam_internal, displayMode);

}

Mat getFrame(HIDS* hCam, int width, int height, Mat& mat) {
    // alociranje memorije
    char* pMem = NULL;
    int memID = 0;
    is_AllocImageMem(*hCam, width, height, 24, &pMem, &memID);



 // Aktiviranje memorije za sliku i dohvacanje slike
    is_SetImageMem(*hCam, pMem, memID);
    is_FreezeVideo(*hCam, IS_WAIT);

    // Prebacivanje slike u mat format
    VOID* pMem_b;
    int retInt = is_GetImageMem(*hCam, &pMem_b);
    if (retInt != IS_SUCCESS) {
        cout << "Image data could not be read from memory!" << endl;
    }
    memcpy(mat.ptr(), pMem_b, mat.cols*mat.rows*3); // dodano *3 za
BGR, inace grayscale
    is_FreeImageMem(*hCam, pMem, memID);

        return mat;
}
vector<Point> getContours(Mat image) {

    vector<vector<Point>> contours;
```

```
    vector<Vec4i> hierarchy;

    findContours(image, contours, hierarchy, RETR_EXTERNAL,
CHAIN_APPROX_SIMPLE);
    vector<vector<Point>> conPoly(contours.size());
    vector<Rect> boundRect(contours.size());

    vector<Point> biggest;
    int maxArea=0;
    for (int i = 0; i < contours.size(); i++)
    {
        int area = contourArea(contours[i]);  //podesit area > xyz za
primjenu (na drugoj traci barem 8000)

        if (area > 5000) {
            float peri = arcLength(contours[i], true);
            approxPolyDP(contours[i], conPoly[i], 0.02 * peri, true);

            if (area > maxArea && conPoly[i].size()==4 ) {
                biggest = { conPoly[i][0],conPoly[i][1] ,conPoly[i][2]
,conPoly[i][3] };
                maxArea = area;
            }
        }
    }
    return biggest;
}

vector<Point> rearrange(vector<Point> points)
{
    vector<Point> newPoints;
    vector<int>  sumPoints, subPoints;

    for (int i = 0; i < 4; i++)
    {
        sumPoints.push_back(points[i].x + points[i].y);
        subPoints.push_back(points[i].x - points[i].y);
    }

    newPoints.push_back(points[min_element(sumPoints.begin(),
sumPoints.end()) - sumPoints.begin()]); // 0
    newPoints.push_back(points[max_element(subPoints.begin(),
subPoints.end()) - subPoints.begin()]); //1
    newPoints.push_back(points[min_element(subPoints.begin(),
subPoints.end()) - subPoints.begin()]); //2
    newPoints.push_back(points[max_element(sumPoints.begin(),
sumPoints.end()) - sumPoints.begin()]); //3

    return newPoints;
}

Mat imgWarp;
Mat getWarp(Mat image, vector<Point> points, float w, float h )
{
```

```
    Point2f src[4] = { points[0],points[1],points[2],points[3] };
    Point2f dst[4] = { {0.0f,0.0f},{w,0.0f},{0.0f,h},{w,h} };

    Mat matrix = getPerspectiveTransform(src, dst);
    warpPerspective(image, imgWarp, matrix, Point(w, h));

    return imgWarp;
}

int main()
{
    HIDS hCam = 0;
    initializeCameraInterface(&hCam);
    cuda::printCudaDeviceInfo(0);
    cout << "\n Press 'q' to decode saved images.\n"<< endl;

    //inicijalizacija
    int height = 1080; //sirina i visina dohvacene slike
    int width = 1920;
    int w = 600; //sirina i visina naljepnice

    int h = 600;
    int b = 1;   //inicijalizacija za spremanje slika (ime=b.tif)
    int noimg = 50; //broj spremljenih slika naljepnice
    Mat img (height, width, CV_8UC3); // image(height, width, type)
    Mat imgprocessedcpu,test;
    Mat imgWarped(w, h, CV_8UC3, Scalar(0, 0, 0));
    cuda::GpuMat warmupgpu, imggpu,imggray, imgblur,imgthr,
imgcanny,imgprocessed;

    //za procesiranje slike
    cv::Ptr<cv::cuda::CannyEdgeDetector> C =
cv::cuda::createCannyEdgeDetector(120,170,3,false); //podesiti
vrijednosti s obzirom na osvijetljenje, gledati imgprocessedcpu
    cv::Mat element = cv::getStructuringElement
(cv::MORPH_RECT,cv::Size(3,3));
    cv::Ptr<cv::cuda::Filter> dilate;


     dilate = cv::cuda::createMorphologyFilter
(cv::MORPH_DILATE,CV_8UC1,element);
    vector<Point> initialPoints, arrangedPoints;


    while(true){
        //dohvacanje slike
        auto start = getTickCount();
        getFrame(&hCam, img.cols, img.rows, img);

        //upload na gpu
        imggpu.upload(img);

        //procesiranje slike
        cuda::cvtColor(imggpu, imggray, COLOR_BGR2GRAY);
```

```
        C->detect(imggray, imgcanny);
        dilate->apply(imgcanny,imgprocessed);

        //download na cpu
        imgprocessed.download(imgprocessedcpu);

        //findcontours
        initialPoints = getContours(imgprocessedcpu);
        if (!initialPoints.empty()){
            arrangedPoints = rearrange(initialPoints);

            //warping
            if (!arrangedPoints.empty()){
                imgWarped = getWarp(img, arrangedPoints, w, h);
                //save images

            if (b <= noimg){

                    string ime = to_string(b)+ ".tif";
                    imwrite(ime, imgWarped);
                    b= b+1;

                }
            }
        }

        //FPS
        auto end = getTickCount();
        auto totalTime = (end - start) / getTickFrequency();
        auto fps = 1 / totalTime;

        //prikaz rezultata
        putText(img, "FPS: " + to_string(int(fps)), Point(20, 40),
FONT_HERSHEY_SIMPLEX, 1, Scalar(0, 2, 237), 3, false);
        imshow("Image", img);
        imshow ("Procesirana slika", imgprocessedcpu);
        imshow ("Naljepnica", imgWarped);

        //exit
        if(waitKey(2) == 'q'){
            break;
        }
    }
        is_ExitCamera(hCam);

        //qr code detection
        QRCodeDetector decoder = QRCodeDetector();
        std::vector<Point> points;
        int count = 0;
        int failcount = 0;
        int goodcount = 0;
        auto startqr = getTickCount();

        for (int a=1; a <= b-1; a++){
```

```
            string imea = to_string(a)+ ".tif";
            Mat qr = imread(imea,IMREAD_UNCHANGED);
            std::string predmet = decoder.detectAndDecode(qr, points);
                        if (!points.empty()) {

                            if (predmet.rfind("P",0)==0) {
                                count = count +1;
                                cout <<"Decoded data: " <<predmet<<" :
" << count << endl;
                                goodcount=goodcount +1;

                            }
                        else {cout <<"Fail :" << count << endl;
                                    count = count +1;
                                    failcount = failcount +1;}

                        }
        }

        auto endqr = getTickCount();
        auto totalTimeqr = (endqr - startqr) / getTickFrequency();
        cout << "\nSuccessfuly decoded " << goodcount << " out of "<<
b-1 <<" images.\n" << "Failed to decode " << failcount << " images.
\n" << 100*goodcount/(b-1) << "% success rate" << endl;

        cout << "Total time for QR code decoding elapsed: " <<
totalTimeqr << "seconds \n" << endl;


        return 0;
}
```