UNIVERZITET U BANJOJ LUCI
ELEKTROTEHNIČKI FAKULTET

Srđan Jović

IMPLEMENTACIJA I OPTIMIZACIJA ALGORITMA ZA DETEKCIJU POMJERAJA OBJEKTA U VIDEU NA DSP PROCESORU

Diplomski rad

Banja Luka, decembar 2019

Tema:	IMPLEMENTACIJA I OPTIMIZACIJA ALGORITMA ZA DETEKCIJU POMJERAJA OBJEKTA U VIDEU NA DSP PROCESORU
	Ključne riječi:
	Detekcija pomjeraja
	Uparivanje blokova
	DSP procesor
Komisija:	doc. dr Aleksej Avramović, predsjednik
	prof. dr Vladimir Risojević, mentor
	Vladan Stojnić, član

Uz rad je priložen *CD* sa tekstom diplomskog rada i prilozima

Kandidat:

Srđan Jović

UNIVERZITET U BANJOJ LUCI

ELEKTROTEHNIČKI FAKULTET

KATEDRA ZA RAČUNARSTVO I INFORMATIKU

Predmet:	MULTIMEDIJALNI SISTEMI
Tema:	IMPLEMENTACIJA I OPTIMIZACIJA ALGORITMA ZA DETEKCIJU POMJERAJA U VIDEU NA DSP PROCESORU
Zadatak:	Opisati arhitekturu TI DSP C66x procesora, kao i tehnike za optimizaciju te korištenje mogućnosti cl6x TI kompajlera u svrhu optimizacije koda. Opisati algoritam za detekciju pomjeraja objekta u videu korištenjem estimacije pokreta uparivanjem blokova u susjednim frejmovima. Realizovati, te prilagoditi algoritam za detekciju pomjeraja objekta korištenjem estimacije pokreta za TI DSP C66x procesor. Primijeniti taj algoritam na video koji dolazi sa kamere, te rezultate prikazati na HDMI displeju. Analizirati performanse inicijalnog i optimizovanog koda.
Mentor:	prof. dr Vladimir Risojević
Kandidat:	Srđan Jović

Banja Luka, decembar 2019.

Sadržaj

1. UVOD

Još početkom sedamdesetih godina dvadesetog vijeka naučnici u polju računarske tehnike pokazivali su interesovanje za pravac koji danas nazivamo "Računarski vid" (*eng. Computer vision*). Računarski vid jeste naučna disciplina koja se bavi razvojem i istraživanjem metoda koje računar koristi da bi dobio određen nivo shvatanja digitalnih slika ili digitalnog videa. Iz perspektive inženjeringa, pravac istražuje načine koji će automatizovati procese koje ljudski vizuelni sistem može da obavlja. Računarski vid jeste veoma širok pojam, a samo neke od poddisciplina kojima se računarski vid bavi jesu: rekonstrukcije scena, prepoznavanje objekata, detekcija i estimacija pokreta objekta u videu, kao i mnoge druge.

Poddisciplina kojom se ovaj rad bavi jeste detekcija pomjeraja objekta u videu. Kada se priča o detekciji pomjeraja objekta misli se na proces detekcije promjene pozicije objekta u videu u odnosu na njegovo okruženje, kao i detekcije promjene položaja okruženja objekta u odnosu na stacionaran objekat. Primjena detekcije pomjeraja objekta u videu je raznolika, a samo neka od značajnijih mjesta gdje je detekcija zastupljena jesu: sigurnosni sistemi koji detekciju pokreta koriste da bi se detektovale određene aktivnosti krivičnog karaktera kao što su neautorizovani pristupi prostoru koji se smatra zaštićenim, automobilska industrija koja detekciju pomjeraja objekata koristi za detekciju kretanja ostalih vozila koja učestvuju u saobraćaju u neposrednoj blizini, kao i detekcija kretanja pješaka koji se nalaze u neposrednoj blizini vozila. Metoda detekcije pomjeraja koja će biti obrađena u nastavku rada jeste metoda bazirana na estimaciji pokreta uparivanjem blokova iz susjednih frejmova.

U prethodnom paragrafu može se vidjeti da neki slučajevi upotrebe detekcije pomjeraja objekta u videu zahtijevaju da se proces detekcije izvršava na sistemima koji često ne posjeduju veliku procesorsku moć i gdje je veličina računarskog sistema na kojem se proces detekcije izvršava prostorno ograničena. Za primjer možemo uzeti detekciju pomjeraja objekata u okolini automobila gdje se sama platforma koja izvršava detekciju nalazi u automobilu. Zbog toga se u ovakvim situacijama koriste sistemi koji se nazivaju ugrađeni sistemi (*eng. Embedded systems*), a njihova primarna uloga jeste da obavljaju neku određenu funkciju, najčešće u sklopu nekog većeg mehaničkog ili električnog sistema. Nešto više o ugrađenim sistemima biće rečeno u Glavi 4 nazvanom "Ugrađeni sistemi". Ugrađeni sistem na kojem će se bazirati ovaj rad zasnovan je na TSM320C66x procesorskoj arhitekturi, koja je razvijena od strane kompanije pod nazivom Texas Instruments i koja je specifična po tome što se programi mogu izvršavati na procesoru za digitalnu obradu signala (*eng. DSP - Digital Signal Processor*) koji je optimizovan za operacije koje su česte kada se radi o obradi digitalnih signala.

Predmet ovog rada jeste implementacija algoritma za detekciju pomjeraja objekta u videu i optimizacija istog za platformu koja je bazirana na TMS320C66x procesorskoj arhitekturi. Ovim radom implementiran je navedeni algoritam te su na njega primijenjene određene optimizacije da bi se na kraju poredile performanse algoritma prije i poslije optimizacija. U drugom poglavlju rada dat je pregled pravca računarske tehnike koji se naziva računarski vid i kratak pregled poddisciplina kojima se računarski vid bavi. Kao jedna od poddisciplina kojom se računarski vid bavi jeste detekcija pomjeraja objekta u videu i ona je opisana u trećem po redu poglavlju, gdje se u prvi plan stavlja metoda detekcije zasnovana na estimaciji pokreta uparivanjem blokova između susjednih frejmova video zapisa. Navedeni metod je zasnovan na više različitih metoda pretrage tokom uparivanja blokova, a tri metoda koja su obrađena u ovoj

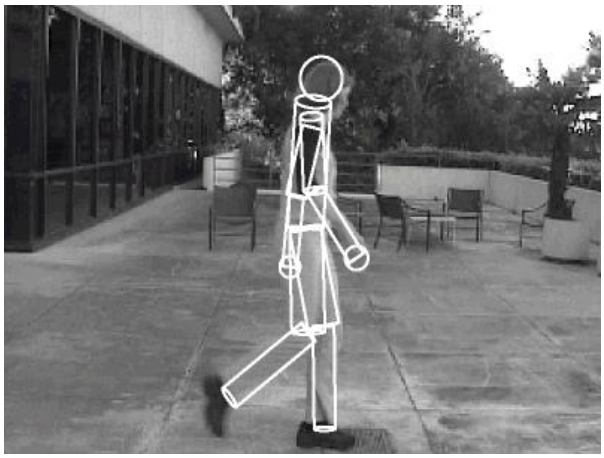
glavi jesu: iscrpna metoda pretrage (*eng. Exhaustive Search*), metoda pretrage u tri koraka (*eng. Three Step Search*) i dijamantska metoda pretrage (*eng. Diamond Search*). Dodatno će biti riječi i o dodatne dvije metode, a to su: šestougaona pretraga (*eng. Hexagon Search*) i logaritamska pretraga (*eng. Logarithmic Search*). Implementacija će se bazirati na prve tri navedene metode, dok se druge dvije spominju samo idejno. Četvrta glava govori uopšteno o ugrađenim sistemima, kao i o platformi TDA2Px koja se koristi kao platforma za realizaciju rada, a nakon toga slijedi pregled DSP procesora i daje uvid u arhitekturu i karakteristike TMS320C66x procesora. Detalji implementacije algoritma za detekciju pomjeraja na datoj hardverskoj arhitekturi dati su u glavi pet, koja daje informativan pregled arhitekture softerskog *framework*-a pod nazivom Processor SDK-Vision u kojem je realizovana implementacija praktičnog dijela. Glava šest sadrži uvid u metode optimizacija koje su korištene pri implementaciji praktičnog dijela, a uključuju optimizacije na nivou algoritma i optimizacije na nivou arhitekture na kojoj je praktični dio razvijen. Testriranje i analiza rezultata dobijenih testiranjem dati su u glavi sedam. U posljednoj glavi nalazi se zaključak rada, gdje se još jednom ističu najznačajniji rezultati rada, mogućnost primjene ovog rada, kao i preporuke za dalji rad na obrađenom problemu.

2. RAČUNARSKI VID

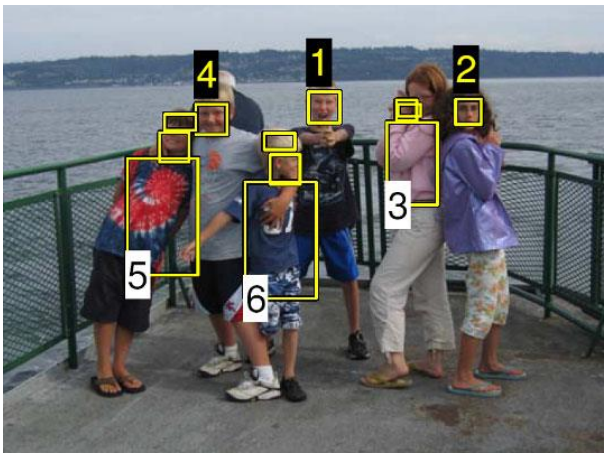
Sam pojam detekcije pomjeraja objekta u videu ne bi se istraživao da naučnici u sedamdesetim godinama prošlog vijeka nisu pokazali određen nivo interesovanja u polju računarskog vida. Ljudska bića svakodnevno opažaju trodimenzionalnu strukturu svijeta oko sebe i to rade sa velikom lakoćom. Ljudi bez ikakvih problema mogu da zaključe kakvog je neki predmet obilika, koliko se ljudi nalazi na fotografiji, pa čak i kojega pola i koje starosti su ti ljudi sa fotografije. Perceptivni psiholozi proveli su decenije pokušavajući da shvate kako vizuelni sistem kod čovjeka funkcioniše i iako su uspjeli da shvate neke od osnovnih principa, kompletan način funkcionisanja vizuelnog sistema čovjeka i danas ostaje misterija [1].

Naučnici u polju računarskog vida razvili su računarske metode za rekonstruisanje trodimenzionalnog oblika objekata sa slika na osnovu dovoljno velikog broja slika istog objekta iz različitih uglova posmatranja. Danas je moguće pratiti kretanje objekata iza kojih se nalaze kompleksne pozadine, kao što je prikazano Slikom 2.1. Na Slici 2.2 prikazana je mogućnost određivanja osoba koje se nalaze na fotografiji tako što se koriste kombinacije metoda za detekciju i prepoznavanje lica, odjeće i kose. Međutim, bez obzira na sva ova tehnološka dostignuća današnjice, i dalje je nezamslivo da računar analizira i interpretira fotografiju na nivou dvogodišnjeg djeteta [1].

Postavlja se pitanje zašto je računarski vid toliko kompleksan domen i težak za razumijevanje? Odgovor proizilazi iz činjenice da je vid „inverzan“ problem, što znači da je potrebno odrediti neke nepoznate promijenljive, a na raspolaganju su nam informacije koje nisu dovoljne da se te promijenljive odrede sa lakoćom. U poljima računarske grafike i fizike najčešće se koriste *forward* modeli koji na osnovu informacija o kretanju objekata, prelamanja, odbijanja i projektovanja svjetlosti na ravne i zakrivljene površine modeluju okruženje. Sa druge strane se u računarskom vidu isto tako pokušava raditi obrnuta procedura, gdje se na osnovu jedne ili više fotografija dobijaju informacije kao što su oblik, osvjetljenost, distribucija boja na objektima, kao i način i pravac kretanja objekata sa fotografije ili videa [1].



Slika 2.1 – Algoritam za praćenje kretanja osobe sa kompleksnom pozadinom



Slika 2.2 – Kombinacija algoritama za detekciju lica i odjeće

2.1 Istorija

Ideja da bi računarski vid trebao biti jednostavan koncept potiče još iz ranih dana vještačke inteligencije, kada se vjerovalo da je kognitivni dio inteligencije dosta kompleksniji za shvatanje od perceptualnog dijela. U to vrijeme naučnici vještačke inteligencije i robotike (na institutima kao što su Masačusetski tehnološki institut (*eng. MIT – Massachusetts Institute of Technology*), Stanford univerzitet (*eng. Stanford University*) i Karnedž Melon univerzitet (*eng. CMU – Carnegie Mellon University*)) vjerovali su da će problem vizuelnog ulaza biti samo jednostavan korak na putu do rješavanja nekih mnogo kompleksnijih problema kao što je svhvatanje na visokom nivou (*eng. high-level reasoning*).

U posljednjih četrdeset godina polje računarskog vida znatno je napredovalo, a kratak istorijski pregled najznačajnijih dostignuća u polju računarskog vida dat je u Tabeli 2.1.

Tabela 2.1 – Istorijski pregled dostignuća u polju računarskog vida [1]

Period	*Dostignuća*
1970e	Winston (1975) i Hanson i Riesman (1978) objavljuju radove na temu rekonstrukcije trodimenzionalne strukture svijeta na osnovu fotografija; Huffman (1971), Clowes (1971), Waltz (1975) objavljuju radove na temu razvoja algoritama za linearno labeliranje
1980e	Canny (1986), Nalwa i Binford (1986) objavljuju radove o detekciji ivica i kontura; Procesiranje trodimenzoinalnih podataka (*eng. Three dimensional data processing*), uključuje pribavljanje, spajanje, modelovanje i prepoznavnaje, aktivno se istražuje tokom osamdesetih godina i obrađeno je u radovina koje objavljuju Besl i Jain (1985), Faugeras i Hebert (1987)i mnogi drugi;
1990e	Metode optičkog toka (*eng. Optical Flow*) nastavljaju da se obrađuju u radovima koje objavljuju Nagel i Enkelmann (1986); Aktivno se istražuju *multi-view stereo* algoritmi u radovima koje objavljuju Seitz i Dyer (1999), kao i Kutulakos i Seitz (2000); Algoritmi za praćenje (*eng. tracking algorithms*) su se takođe unaprijedili i obrađeni su u radovima koje objavljuju Kass, Witkin i Terzopoulos (1988), kao i Balke i Isard (1998);
2000e	Razvijaju se mnogobrojne tehnike koje danas spadaju pod disciplinu koja se naziva komputaciona fotografija (*eng. computational photography*), a u koje spadaju spajanje slika (*eng. image stiching*), fotografije visokog dinamičkog opsega (*eng. HDR – High Dynamic Range*); Sinteza tekstura i *quilting* obrađeni u radovima koje objavljuju Efros i Freeman (2001), Kwatra, Shodl, Essa (2003); Tehnike proepoznavanja zasnovane na obilježjima (*eng. feature based recognition*) obrađene su u radovima Ponce, Hebert i Schmid (2006); Metode računraskog vida bazirane na mašinskom učenju;

2010e Primjena konvolucionih neuronskih mreža u svrhe prepoznavanja objekata, detekciji objekata i semantičkoj segmentaciji slika. Neke od najznačajnijih arhitektura jesu AlexNet (2012) razvijena od grupe SuperVision, GoogLeNet (2014) koji je razvijen od strane kompanije Google, ResNet (2015) i mnoge druge;

2.2 Primjena

Pojam računarskog vida je širok i obuhvata veliki broj poddisciplina, pa je samim tim i način na koji se znanja iz ove oblasti primjenjuju raznolik. Samo neke od primjena računarskog vida u današnje vrijeme jesu:

- Optičko prepoznavanje karaktera (*eng. OCR - Optical character recognition*) – koristi se kod automatskog prepoznavanja registarskih tablica na slikama;
- Inspekcija mašina (*eng. Machine Inspection*) – brza inspekcija dijelova koji se koriste u auto i avio industriji da bi se osigurao maksimalni kvalitet konačnog proizvoda;
- Izgradnja trodimenzionalnih modela (*eng. 3D Model Building*) – automatizovani sistemi koji generišu trodimenzionalne modele reljefa na osnovu aero fotografija;
- Automobilska sigurnost (*eng. Automotive safety*) – detekcija neočekivanih prepreka kao što su pješaci na ulici u uslovima gdje klasične tehnike vida nisu baš najbolje rješenje;
- Obrada medicinskih fotografija (*eng. Medical imaging*) – jedna od primjena u ovoj oblasti jeste obrada slika u cilju proučavanja morfologije ljudkog mozga dok stari;
- Prepoznavanje otiska prsta i biometrija (*eng. Fingerprint Recognition and Biometrics*) – za automatsku autentikaciju baziranu na skeniranju otiska prsta, kao i za forenzičke aplikacije;
- Nadzor (*eng. Surveillnace*) – detekcija pomjeraja objekta u videu koristi se za nadgledanje uljeza, analizu saobraćaja, kao i na primjer za praćenje stanja bazena u svrhu detekcije utapanja;

Ovaj rad će se primarno fokusirati na posljednju primjenu navedenu u prethodnoj listi, a to je detekcija pomjeraja objekta u videu. Naredna glava daje detaljan uvid u detekciju pomjeraja objekta u videu, kao i u tehnike i algoritme koji se koriste za postizanje ovog cilja.

3. DETEKCIJA POMJERAJA OBJEKTA U VIDEU

Potreba da se vrši detekcija pomjeraja objekta u video zapisu postaje sve češća, a broj discpilina koje imaju koristi od metoda detekcije pomjeraja raste svakodnevno. Skoro svi moderni sistemi za nadgledanje (*eng. surveillance systems*) danas posjeduju mogućnost praćenja objekta koji se kreće na sceni, a da bi praćenje objekta bilo moguće potrebno je detektovati svaki vid kretanja tog objekta u videu. Video sistemi za nadzor koji rade u realnom vremenu imaju najviše koristi od metoda detekcije pomjeraja objekta, mada, nije ni rijetkost da se metodi detekcije pomjeraja koriste i za obradu već snimljenog video sadržaja. Na Slici 3.1 je prikazan jedan interesantan primjer takve primjene - detekcija pomjeraja i kretanja igrača na video snimku fudbalske utakmice, gdje se dobijene informacije o kretanju objekata (u ovom slučaju igrača) koristi za analizu i poboljšavanje pozicioniranja individualnih igrača, kao i cijelog tima.

Slika 3.1 – Detekcija pomjeraja igrača na fudbalskoj utakmici

Generalno, proces detekcije pomjeraja objekta u videu smatra se procesom kojim se potvrđuje promjena pozicije objekta ili objekata u odnosu na položaj pozadine, ili promjena pozadine u odnosu na objekat [3]. Kroz prethodnih nekoliko decenija, predloženo je i razvijeno nekoliko tehnika da bi se ispunio ovaj cilj. Naravno, ne postoji savršena metoda detekcije pomjeraja koja je otporna na sve probleme sa kojima se susreće tokom procesa detekcije, a koji se najčešće tiču uslova osvjetljena scene i dinamične promjene istog, brzine kretanja objekta ili objekata na sceni, kao i otpornost na razne vidove šuma koji neprestano ometaju proces detekcije. U nastavku glave obrađene su dvije metode detekcije pomjeraja

objekta u videu: detekcija pomjeraja bazirana na razlici frejmova (*eng. Image Substraction*) i detekcija pomjeraja bazirana na estimaciji pokreta uparivanjem blokova (*eng. Block-matching Motion Estimation*). Prva metoda je obrađena samo teorijski, dok se implementacija u nastavku rada (Glava 6) bazira na drugoj metodi, koja je iz toga razloga obrađena sa više detalja.

3.1 Razlika frejmova

Razlika frejmova (*eng. Image substraction*) jeste jedna od jednostavnijih i popularnijih tehnika koje se danas koriste da bi se detektovao pomjeraj u videu. Jednostavno, razlika frejmova se može predstaviti izrazom:

$$\Delta I(i,j) = I_{Curr}(i,j) - I_{Prev}(i,j), \tag{3.1}$$

gdje *i* predstavlja indeks reda frejma, *j* predstavlja indeks kolone frejma, $I_{Curr}(i,j)$ predstavlja matricu intenziteta piskela trenutnog frejma, a $I_{Prev}(i,j)$ predstavlja matricu intenziteta piksela prethodnog frejma. $\Delta I(i,j)$ predstavlja matricu koja se dobije razlikom intenziteta piksela na odgovarajućim pozicijama za dva uzastopna frejma.

Slika 3.2 – Rezultat razlike dva uzastopna frejma u videu

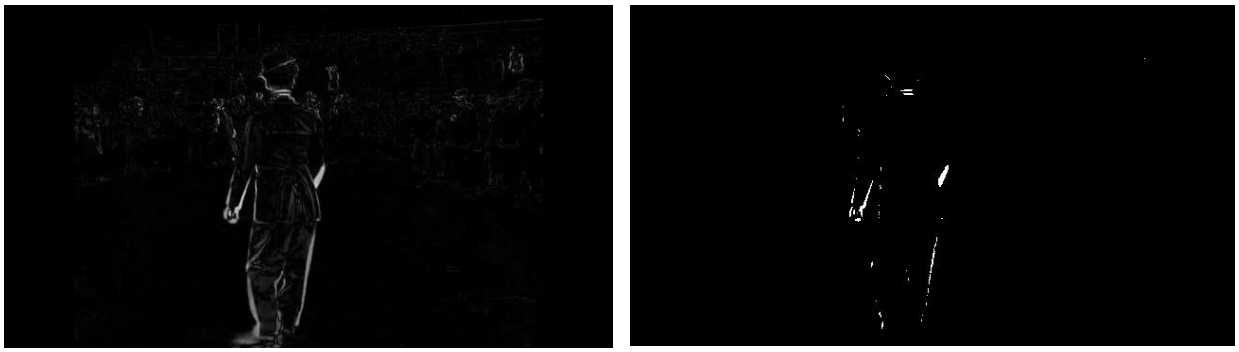
Kao što se može vidjeti na Slici 3.2, rezultat razlike dva uzastopna frejma u videu jeste intenzitetska slika na kojoj vrijednost intenziteta piksela na pozicijama gdje nije bilo razlike iznosi nula ili približno nula, dok se na pozicijama gdje je postojala određena razlika između frejmva nalaze pikseli intenziteta koji je veći od nula. Na osnovu intenziteta piskela na rezultantnoj slici može se zaključiti da li je došlo do pomjeraja objekata na slici, i ako jeste, na kojoj lokaciji frejma se taj pomjeraj desio.

Iako prilikom detekcije pomjeraja baziranoj na razlici dva uzastopna frejma dobijamo zadovoljavajuće rezulate kada se radi o statičnom frejmu (pomjeraj pozadine i objekata u pozadini je nepostojeći ili minimalan), ovaj način detekcije pomjeraja ne daje zadovoljavajuće rezultate kada se isti princip koristi na video sa pozadinom koja se pomjera. Isto tako, problem predstavlja i promjena osvjetljenja scene, a tada je razlika intenziteta piksela postojeća na svim pozicijama na frejmu. Ovaj problem se može ublažiti tako što se koristi metoda usrednjavanja razlika više uzastopnih frejmova, ali ovaj rad se u nastavku neće osvrtati na navedeno poboljšanje. Takođe, jedan od problema na koji ovaj pristup nailazi jeste šum pozadine koji ne treba da bude detektovan kao pomjeraj. Način na koji se ovaj šum može otkloniti jeste korištenjem praga intenziteta na frejmu razlike intenziteta. Korištenje praga dato je izrazom:

$$\Delta I(i,j) = 0, \text{ ako važi } \Delta I(i,j) < t, \tag{3.2}$$

$$\Delta I(i,j) = 1, \text{ ako važi } \Delta I(i,j) > t,$$

gdje t predstavlja prag odbacivanja šuma. U zavisnosti od vrijednosti praga, rezultantna razlika će biti manje ili više tolerantna na šum. Prevelika vrijednost praga t može da odbaci i korisne informacije pa ne postoji najbolja vrijednost praga, te se ta vrijednost određuje eksperimentalno u zavisnosti od situacije. Rezulati filtriranja šuma pragom dati su na Slici 3.3.



Slika 3.3 – Rezultantni frejm prije i nakon filtriranja šuma uz pomoć praga

3.2 Estimacija pokreta uparivanjem blokova

Estimacijom pokreta smatra se proces kojim se određuju vektori pomjeraja koji opisuju pomjeranje objekata između dva uzastopna frejma. Isto tako se može reći da vektori pomjeraja opisjuju kako se jedan frejm transformiše u drugi. Ovaj problem nije dobro definisan zato što slike zapravo predstavljaju projekciju trodimenzionalne scene u dvodimenzionalnu ravan pa korespondencija između pokreta objekata u 3D okruženju i 2D ravni nije jednoznačna. Međutim ako je broj frejmova u videu veliki, onda se može pretpostaviti da je količina pomjeraja objekata između dva uzastopna frejma mala što omogućava dovoljno dobru estimaciju koja se može iskoristiti da bi se detektovao pomjeraj objekta u videu [2]. Veliki je broj algoritama koji se koriste da bi se odredili vektori pomjeraja, a neki od njih su: uparivanje blokova, određivanje optičkog toka, fazna korelacija, piksel-rekurzivni algoritmi, algoritmi zasnovani na detekciji i uparivanju obilježja, itd. Ovaj rad se u nastavku fokusira na algoritam uparivanja blokova. Estimacija pokreta pored detekcije pomjeraja objekta ima i druge primjene, a primarno se koristi kao korak prilikom komprimovanja video sadržaja. U nastavku, rad se isključivo fokusira na primjenu estimcije pokreta radi detekcije pomjeraja objekta u videu.

Algoritam uparivanja blokova u opštem slučaju pronalazi vektore pomjeraja blokova između dva frejma – referentni frejm i trenutni frejm. U slučaju primjene estimacije pokreta radi detekcije pomjeraja, smatra se da su ta dva frejma uzastopna ako nije drugačije naglašeno, a referenciraće se kao trenutni i prethodni frejm. Prvi korak algoritma jeste da se trenutni frejm podijeli na blokove fiksne veličine. Veličina blokova se bira pod pretpostavkom da se prostorno bliski pikseli kreću na sličan način [2]. Za svaki blok *B* na trenuntom frejmu potrebno je odrediti odgovarajući blok sa prethodnog frejma tako da mjera nesličnosti blokova poprima minimalnu vrijednost. Mjera nesličnosti dva bloka odnosi se na različitost blokova u zavisnosti od razika vrijednosti piksela blokova na istim pozicijama. Za mjeru nesličnosti blokova najčešće se koriste suma apsolutnih razlika intenziteta piksela između frejmova ili suma kvadrata razlika. Mjera nesličnosti je u ovom slučaju veća i izraženija ako se vrijednosti piksela blokova na istim pozicijama razlikuju, dok je manja ako se radi o istim ili približno istim vrijednostima. Bitno je naglasiti da mjera nesličnosti u ovom kontekstu ne povlači striktno i vizuelnu razliku blokova. Suma apsolutnih razlika definisana je izrazom:

$$D(d_1, d_2) = \sum_{(x,y)\in B} |s(x,y,k) - s(x + d_1, y + d_2, k-1)| \tag{3.3}$$

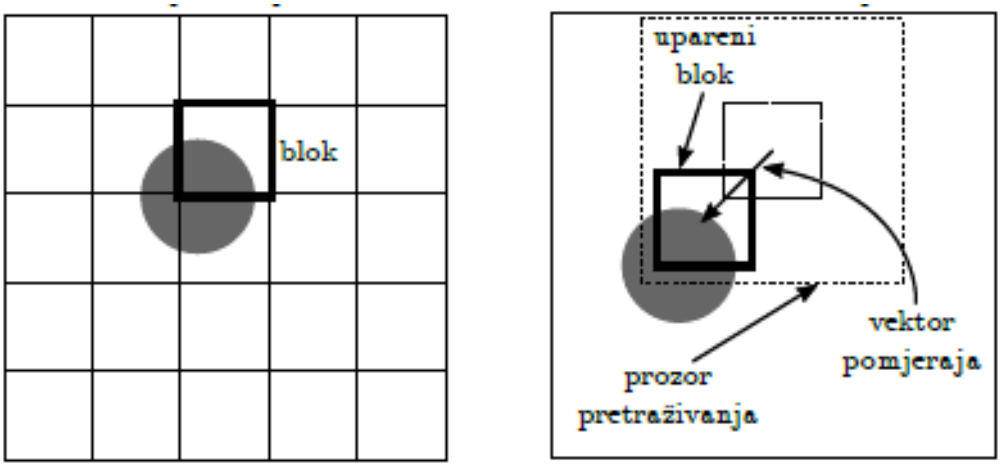
dok je suma kvardata razlika data izrazom:

$$D(d_1, d_2) = \sum_{(x,y)\in B} [s(x,y,k) - s(x + d_1, y + d_2, k-1)]^2 \tag{3.4}$$

gdje k predstavlja indeks trenutnog frejma, a $[d_1, d_2]^T$ predstavlja vektor pomjeraja između blokova koje posmatramo. U nastavku rada razmotrene su tri metode određivanja najsličnijeg bloka sa prethodnog frejma, a to su: iscrpna(sekvencijalna) metoda pretrage, metoda pretrage u tri koraka i dijamantska metoda pretrage.

3.2.1 Iscrpna metoda pretrage

U opštem slučaju da bi algoritam uparivanja blokova našao blok sa prethodnog frejma koji najbolje odgovara referentnom bloku sa trenutnog frejma, potrebno je da se poredi sa svim blokovima prethodnog frejma, što je veoma zahtjevno računarski pa se prostor pretrage u praksi ograničava takozvanim prozorom pretrage. Prozor pretrage je prostor koji se dobija tako što se na blok sa prethodnog frejma koji odgovara pozicijom bloku trenutnog frejma doda fiksno određeno proširenje sa svake od četiri strane bloka, čime površina prozora pretrage postaje veća od površine bloka. Ograničavanje oblasti koja se pretražuje zasniva se na pretpostavci da su pomjeraji objekta mali između susjednih frejmova i da će upareni blok biti pronađen u neposrednoj blizini referentnog bloka [2]. Uobičajeno je da se proširenje u horizontalnom i vertikalnom pravcu ograniči na veličinu bloka. Tako da ako se uzme da su dimenzije bloka $p \times p$, onda horizontalne i vertikalne vrijednosti proširenja mogu imati vrijednosti iz intervala $[-p, p]$, pa je oblast pretraživanja dimenzija $(2p + 1) \times (2p + 1)$ [2]. Na Slici 3.4 predstavljeni su trenutni frejm i prethodni frejm sa prozorom pretrage i rezultantnim blokom pretrage.



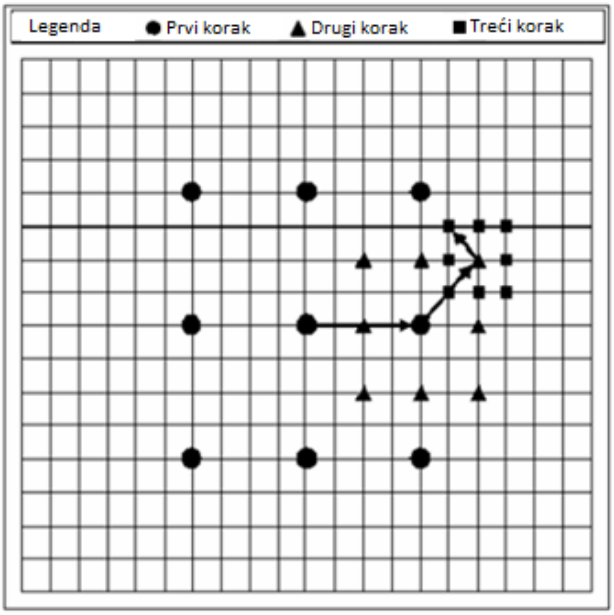
Slika 3.4 – Lijevo je prikazan treuntni frejm i blok koji se obrađuje, desno je prikazan prethodni frejm sa određenim prozorom pretrage i uparenim blokom [2]

Proces pretrage unutar prozora odvija se klizanjem referentnog bloka sa trenutnog frejma preko svih blokova koji se nalaze unutar prozora pretrage na prethodnom frejmu i za svaki par se računa mjera nesličnosti. Referentni blok se pomjera za po jedan piksel horizontalno do se ne dostigne kraj reda prozora pretrage, nakon čega se vraća na početak i pomjera se za jedan piksel nadole u naredni red. Proces se ponavlja za sve redove. Blok iz prozora pretrage koji je dao minimalnu vrijednost mjere nesličnosti uzima se kao najbolji rezultat, a vektor pomjeraja za referentni blok dobija se kao razlika koordinata u lijevom gornjem uglu referentnog bloka i bloka koji je određen kao najbolje poklapanje unutar prozora pretrage. Uzmimo za primjer referentni blok čiji gornji lijevi piksel posjeduje koordinatu (200, 216), ako je najbolje poklapanje određeno za blok sa koordinatama (205, 212) onda vektor pomjeraja za dati referentni blok izosi [5, -4]. Po istom principu se određuje vektor pomjeraja za svaki blok koji je dobijen dijeljenjem referentnog frejma na blokove, a kao rezultat cijelog procesa dobija se matrica vektora pomjeraja čiji je broj redova jednak broju redova blokova kada se referentni frejm podijeli na blokove fiksnih dimenzija, a broj kolona

samim tim jednak broju kolona blokova u referentnom frejmu nakon dijeljenja na blokove. Sa ograničenim prostorom pretrage iscrpna metoda je i dalje računarski zahtjevna metoda pa su predložene mnoge varijacije optimizacije pretrage, a u nstavku su obrađene pretraga u tri koraka i dijamantska metoda pretrage.

3.2.2 Metoda pretrage u tri koraka

Smatra se jednim od najranijih pokušaja da se implementira brza metoda za uparivanje blokova, a po prvi put se pojavljuje još sredinom osamdesetih godina prošloga vijeka [5]. Na Slici 3.5 prikazana je generalna ideja iza ovog algoritma.



Slika 3.5 – Procedura metode pretrage u tri koraka. Tačkama su označene lokacije pretrage tokom prve faze, troulgovi predstavljaju lokacije pretrage tokom druge faze pretrage, a pravougaonici lokacije pretrage treće faze. Strelicama je označen pravac napredovanja algoritma kroz faze [5]

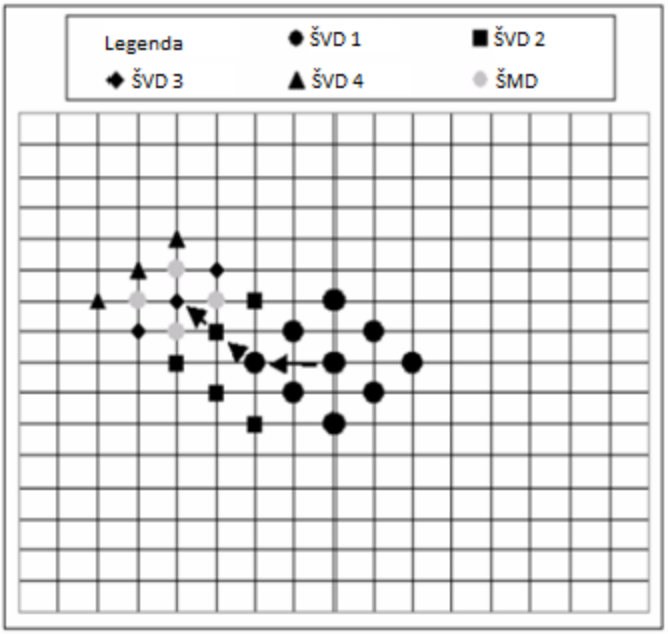
Pretraga bloka se ovaj put ne odvija unutar prozora pretrage, nego se u tri uzastopna koraka algoritma definišu fiksne lokacije na kojima se blokovi porede sa referentnim blokom sa trenutnog frejma. Pretraga na prostoru prethodnog frejma započinje na istoj lokaciji na kojoj se nalazi i referentni blok na trenutnom frejmu, tačnije iz centra lokacije referentnog bloka. Za izvšavanje algoritma bitan je parametar *step_size (S)* koji se na početku postavlja na vrijednost $S = 4$. Pored izračunavanja mjere nesličnosti za centralnu lokaciju, mjera nesličnosti se računa i za osam okružujućih blokova koji se od centralnog bloka nalaze na distancama – S i S po horizontalnoj i vertikalnoj osi tako da se ukupno vrši devet poređenja u prvoj fazi algoritma. Blok za koji je izračunata minimalna vrijednost mjere nesličnosti uzima se kao novi centralni blok i prelazi se u drugu fazu algoritma. U drugoj fazi algoritma vrijednost koraka se polovi i dobija vrijednost $S = 2$. Kao i kod prve faze algoritma, računa se mjera nesličnosti za referentni blok i za novi centralni blok sa svojih osam okružujućih blokova koji se kao i u prvoj fazi od centralnog bloka nalaze na distancama od – S i S po horizontalnoj i vertiklanoj osi. Kada se odredi blok sa minimalnom vrijednošću

mjere nesličnosti prelazi se na finalnu treću fazu koja je ista kao i prve dvije faze, samo što se vrijednost koraka pomjeraja polovi i uzima vrijednost $S = 1$. Blok koji je u trećoj fazi određen kao blok sa minimalnom mjerom nesličnosti u odnosu na referentni blok uzima se kao najbolje poklapanje, a vektor pomjeraja se računa za njegovu poziciju i za poziciju referentnog bloka.

Ova implementacija algoritma daje značajno poboljšanje u performansama u odnosu na iscrpnu metodu pretrage, a izvršava se čak i do devet puta brže [5]. U situaciji iscrpne metode pretrage sa proširenjem $p = 7$, da bi se pronašlo najbolje poklapanje za samo jedan blok sa referentnog frejma potrebno je izračunati vrijednosti mjere nesličnosti za 225 različitih blokova iz prozora pretrage sa prethodnog frejma. Sa druge strane, metoda pretrage u tri koraka svaki put fiksno radi dvadeset i sedam izračunavanja mjere nesličnosti prilikom pretrage jednog bloka. Za manu ovoga algoritma može se uzeti to što zbog ograničenog broja blokova nad kojima će se računati mjera nesličnosti ne pronalazi uvijek blok sa minimalnom mjerom nesličnosti. U slučaju detekcije pomjeraja objekata ova mana ne predstavlja veliki problem ali u slučaju primjene algoritma kao koraka komprimovanja video sadržaja (koja je prethodno navedena kao jedna od čestih primjena za algoritme estimacije pokreta) mogu se dobiti lošiji rezultati u odnsu na iscrpnu metodu pretrage. Poboljšanje ovog algoritma dato je u vidu takozvane nove pretrage u tri koraka (*eng. NTSS – New Three Step Search*) [5] koji je jedan od prvih široko rasprostranjenih brzih algoritama za uparivanje blokova ali koji neće biti obrađen u nastavku rada.

3.2.3 Dijamantska metoda pretrage

Dijamantska metoda pretrage jeste algoritam čiji je šablon pretrage sličan pretrazi kod metode pretrage u tri koraka, samo što se umjesto šablona pravougaonika koristi šablon dijamanta. Na Slici 3.6 prikazana je generalna ideja iza ovog algoritma.



Slika 3.6 – Procedura dijamantske metode pretrage. Na slici je crnim tačkama označen šablon velikog dijamanta, dok je sivim tačkama označen šablon malog dijamanta. Strelicama je označen pravac napredovanja algoritma kroz faze [5]

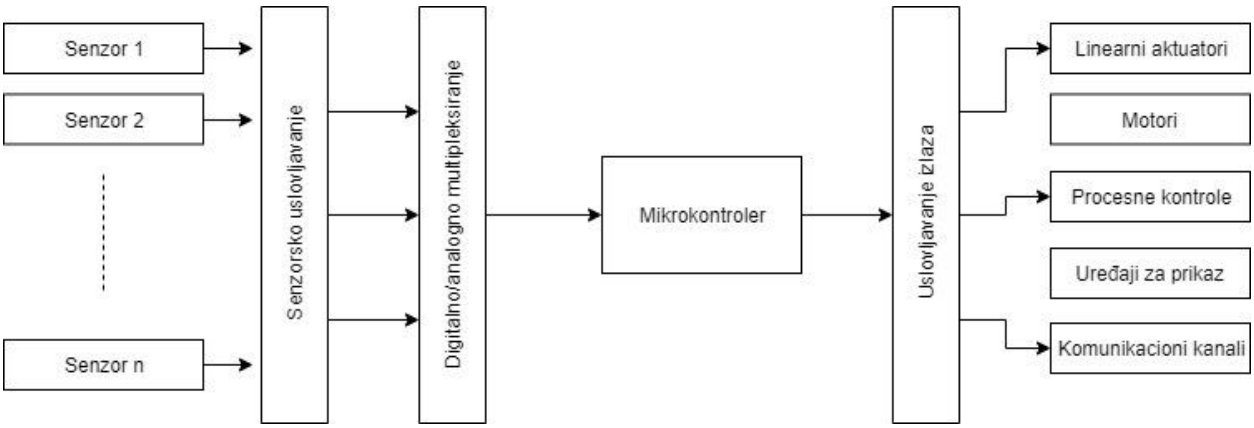
Kao i kod metode pretrage u tri koraka, ni ova metoda ne koristi prozor pretrage. Dijamantska metoda pretrage tokom pretraživanja koristi dva fiksna šablona pretrage, a to su šablon velikog dijamanta pretrage (*eng. LDSP – Large Diamond Search Pattern*) i šablon malog dijamanta pretrage (*eng. SDSP – Small Diamond Search Pattern*). Slično kao i kod pretrage u tri koraka, u prvoj fazi algoritma vrši se izračunavnaje mjere nesličnosti između referentnog bloka sa trenutnog frejma i 9 blokova sa prethodnog frejma koji se nalaze na lokacijama koje odgovaraju tačkama šablona velikog dijamanta pretrage. Lokacija bloka sa najboljim poklapanjem uzima se kao nova centralna lokacija i prelazi se u drugu fazu algoritma u kojoj se izračunava mjera nesličnosti između referentnog bloka sa trenutnog frejma i 5 blokova sa prethodnog frejma koji se nalaze na lokacijama koje sada odgovaraju tačkama šablona malog dijamanta pretrage. Ako se ispostavi da je minimalna vrijendost mjere nesličnosti izračunata za centralni blok dijamanta, algoritam se prestaje izvršavati i blok se uzima kao blok sa najboljim poklapanjem. U suprotnom, lokacija sa najboljim poklapanjem postaje nova centralna lokacija i ponavlja se druga faza algoritma. Algoritam uobičajeno nema ograničen broj koraka da bi mogao sa što većom preciznošću odrediti blok sa najboljim poklapanjem.

U situacijama kada se pozicija objekata u videu mijenja sporo iz frejma u frejm, algoritam daje dobre rezultate uz znatno manji broj koraka potrebnih da se odredi najbolje poklapanje u odnosu na iscrpnu metodu pretrage. Sa druge strane, ako se pozicija objekata u videu mijenja brzo algoritam mora da obradi znatno veći broj koraka da bi pronašao najbolje poklapanje pa se u takvim situacijama performanse algoritma degradiraju [5].

4. UGRAĐENI SISTEMI

Paralelno sa napretkom računarskih tehnologija, sve više se javljaju uređaji koji zavise od interno ugrađenih računarskih sistema (*eng. Embedded System*). Mogućnosti koje ti električni uređaji koji zavise od ugrađenih sistema posjeduju prevazilaze mogućnosti koje se ostvaruju implementacijom na isključivo hardverskom nivou. Iz tog razloga, ugrađeni sistemi se danas mogu naći u širokom opsegu namjenskih uređaja, od onih najmanjih kao što su električni tajmeri gdje se ugrađeni sistemi koriste za male količine procesiranja, pa sve do znatno kompleksnijih kao što su igračke konzole i mnogi drugi industrijski proizvodi.

Ugrađeni sistem nije standardan sistem kao što je personalni računar koji može da izvršava veliki broj različitih programa i obavlja još veći broj pozadinskih zadataka. Ugrađeni sistem uvijek se fokusira na izvršavanje jednog specifičnog programa ili zadatka za koji je isprogramiran. Da bi se ispunila ova potreba, uređaji koji u sebi sadrže ugrađene sisteme na njima imaju preinstaliran softver koji se ne mijenja često ali postoji mogućnost ažuriranja sa vremena na vrijeme [6]. Na Slici 4.1 prikazana je uproštena arhitektura ugrađenog sistema.



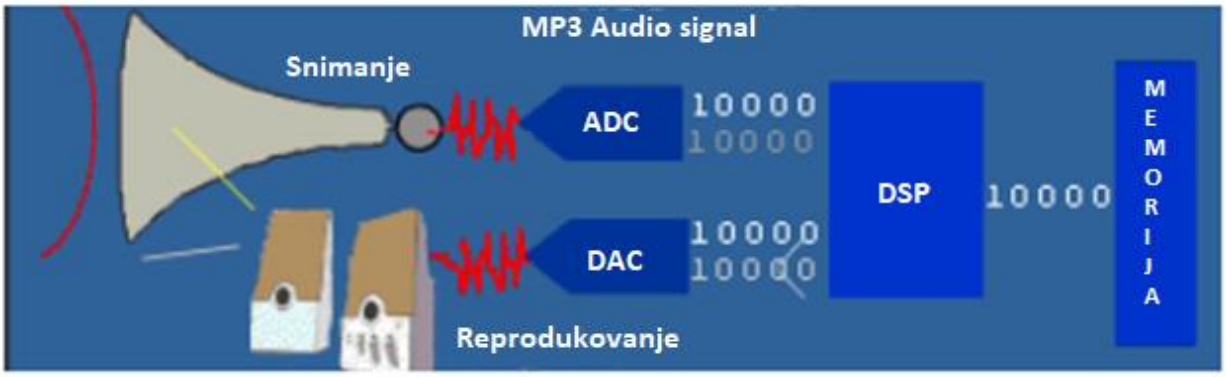
Slika 4.1 – Uproštena arhitektura ugrađenog sistema

Generalno, ugrađeni sistemi definišu se kao bilo koji računarski sistemi sadržani unutar nekog proizvoda ili sistema koji se ne klasifikuje kao računarski sistem [6].

Ugrađeni sistemi sastoje se od dvije glavne komponente, a to su: hardverska komponenta i softerska komponenta. Softver ugrađenog sistema klasično se naziva *firmware*. Za razliku od standardnog softvera na računarima koji se čuva na diskovima, *firmware* se čuva na samome čipu [6]. Hardverska komponenta je hardverska platforma na kojoj se softverska komponenta izvršava. Hardvarska komponenta može biti bazirana na mikroprocesoru i na mikrokontroleru. Uz to, hardverska komponenta uključuje i ostale elemente kao što su: memorija, interfejsi za ulaz i izlaz i slično.

4.1 Procesor za obradu digitalnih signala

Procesor za obradu digitalnih signala (*eng. DSP – Digital Signal Processor*) predstavlja specijalno prilagođenu vrstu procesora koji se u praksi najčešće koriste pri obradi audio i video signala, kao i signala promjene temerature ili vazdušnog pritiska, promjene pozicije u prostoru. Signali koji se obrađuju u realnom vremenu prije dolaska na DSP pretvoreni su u digitalnu formu uz pomoć analogno-digitalnog konvertora da bi DSP mogao da ih obradi. DSP je dizajniran sa ciljem da se na njemu ekstremno brzo izvršavaju operacije koje se često primjenjuju pri obradi digitalnih signala u koje se prije svega ubrajaju aritmetičke operacije sabiranja, oduzimanja, množenja i dijeljenja, a isto tako i logičke opetacije *i, ili, negacija.*



Slika 4.2 – Uprošten slučaj upotrebe DSP procesora [8]

Na Slici 4.2 dat je prikaz sistema koji predstavlja uprošten slučaj upotrebe DSP procesora. Sve počnije fazom snimanja sadržaja, u ovom slučaju to je audio sadržaj. Audio signal koji dolazi u ovaj sistem analognog je oblika pa je potrebno da se pretvori u digitalni oblik kako bi se mogao obraditi na DSP procesoru. Konverziju iz analognog u digitalni oblik obavlja analogno-digitalni konvertor (*eng. ADC – Analog-to-Digital Converter*). Nakon konverzije, digitalni signal pristiže na DSP procesor gdje se obrađuje i enkoduje u neki od formata prikladnih za čuvanje audio sadržaja, u ovom slučaju to je MP3. Kada je potrebno reprodukovati audio sadržaj iz memorije, enkodovani sadržaj se dekoduje na DSP procesoru prije nego što se takav u digitalnom obliku pošalje na digitalno-analogni konvertor (*eng. DAC – Digital-to-Analog Converter*) koji sadržaj pretvara u analogni oblik koji se dalje prosljeđuje uređaju za reprodukciju zvuka.

Primjene DSP procesora su raznolike, a može se susresti u sistemima za sigurnost, telefonske komunikacije, kućna kina, video i audio komprimovanje, itd. Signali se mogu komprimovati da bi se brže i efikasnije prenosili u realnom vremenu, signali se mogu nadograditi ili modifikovati da bi im se poboljšale određene karakterisitke koje utiču na to kako ih čovjek percipira. Iako se signali mogu obrađivati i u svom analognom obliku, procesiranje signala u njihovom digitalnom obliku nosi određene prednosti kao što je brzina [8].

DSP procesor sadrži nekoliko ključnih komponenata, a među njih spadaju:

- Programska memorija – momorija u kojoj se nalazi program koji se izvršava na DSP procesoru
- Memorija podataka – čuva informacije koje će biti obrađene programima koji se izvršavaju na DSP procesoru
- Računarska jedinica – izvršava operacije nad podacima iz memorije podataka
- Ulaz/Izlaz – zaduženi za komunikaciju sa ostalim komponentama sistema

Navedene komponente i njihove veze date su na Slici 4.3.

Slika 4.3 – Pojednostavljena šema kompomenti DSP procesora [8]

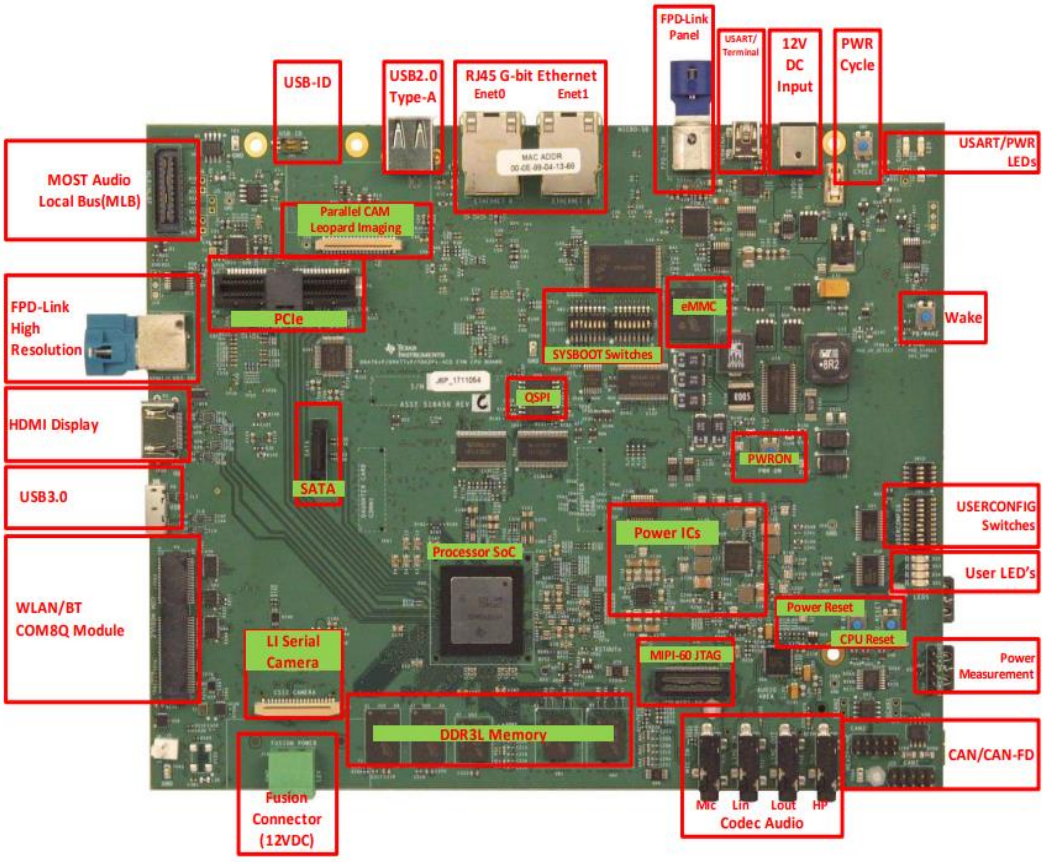
4.2 TDA2Px evaluacioni modul

TDA2Px evaluacioni modul (*eng. EVM – Evaluation Module*) predstavlja evaluacionu platformu dizajniranu da bi se ubrzao razvoj aplikacija baziranih na naprednim sistemima za pomoć vozačima (*eng. ADAS – Advanced Drvier Assist Systems*) i smanjilo vrijeme potrebno za uvođenje istih. TDA2Px EVM baziran je na TDA2Px *sistem na čipu* (*eng. SoC – System on Chip*) arhitekturi koja je heterogena i skalabilna po prirodi, a uključuje kombinaciju TI procesora koji mogu da rade sa podacima fiksnog formata kao i sa podacima u pokretnom zarezu. Procesori koji se nalaze na evaluacionoj platformi jesu:

- TMS320C66x procesor za obradu digitalnih signala (*eng. DSP – Digital Signal Processor*)
- Vision AccelerartionPac (EVE – Embedded Vision Engine) procesor
- ARM® Cortex®-A15 MP Core procesor
- Dvojezgarni Cortex®-M4 procesor

Pored navedenih procesora koji se nalaze na TDA2Px evaluacionoj platformi, ona uključuje i set korisnih periferija kao što su: nekolicina interfejsa za kamere (serijski i paralelni), CSI2 interfejs, CAN

interfejs kao i GigBEthernet AVB interfejs. Na Slici 4.4 prikazana je prednja strana ploče sa označenim interfejsima.



Slika 4.4 – TDA2Px evaluacioni modul sa označenim interfejsima [7]

TDA2Px EVM se u industiji najčešće koristi kao evaluaciona platforma za aplikacije u koje spadaju:

- Automobilske aplikacije za zabavu (*eng. Automotive Infotainment applications*)
- Automobilske aplikacije za vid (*eng. Automotive Vision Applications*)

U svrhe ovoga rada, TDA2Px EVM korištena je kao evaluaciona platforma za razvoj aplikacije za detekciju pomjeraja objekta u videu na DSP procesoru, a u nastavku rada kada se pomene izraz *evaluaciona platforma* odnosiće se na TDA2Px EVM evaluacionu platformu. Kao što je prethodno u poglavlju navedeno, jedan od procesora koji je dostupan na TDA2Px EVM platformi jeste TMS320C66x DSP procesor, a u poglavljima koja slijede dat je opšti pregled arhitekture i primjene DSP procesora, kao i specifičnog modela koji se nalazi na korištenoj evaluacionoj platformi. Od periferija dostupnih na ploči, korišten je MIPI CSI-2 interfejs za kameru na koji je spojena RDACM41-16 kamera, HDMI izlaz na koji se šalje obrađen video signal, UART interfejs preko kojeg se vršila komunikacija sa aplikacijom koja se izvršava na ploči. Evaluaciona platfroma spojena je na 12V DC napajanje.

4.3 TMS320C66x DSP procesor

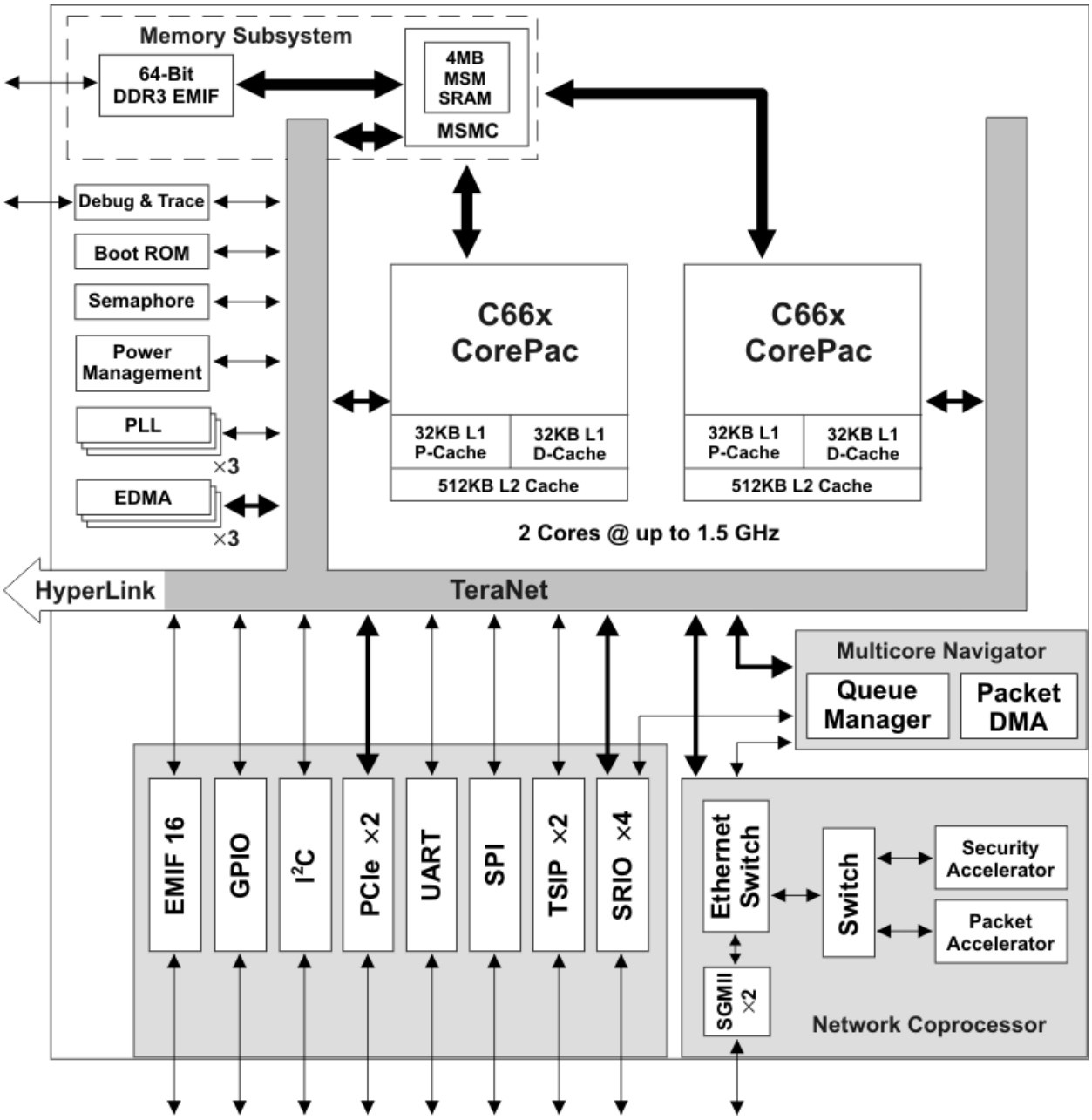
U prethodnom poglavlju bilo je riječi o evaluacionoj platformi na kojoj će da se izvršava implementacija praktičnog dijela ovog rada. Riječ je o TDA2Px evalucionom modulu na kojem se između ostalog nalazi i TMS320C66x DSP procesor. U nastavku rada kada se bude pominjao DSP procesor podrazumijevaće se da se radi o TMS320C66x procesoru ako drugačije ne bude naglašeno.

Texas Instrumentsov TMS320C66x procesor je višejezgarni DSP procesor koji omogućava dobre performanse u realnom vremenu, a najčešća primjena ovog procesora je u sistemima za naprednu analizu videa sistema za nadzor i sigurnost [9]. Iz tog razloga je odabran kao najprikladniji procesor za realizaciju aplikacije opisane ovim radom. Procesor karakterišu niska potrošnja i visoke performanse pa je idealan za sisteme kao što su: pametne IP kamere, digitalni video rekorderi (*eng. DVR – Digital Video Recorders*), umreženi video rekorderi (*eng. NVR – Networked Video Recorders*), serveri za video analitiku (*eng. Video analytics servers*).

Dimenzije procesora iznose 21x21*mm* uz visinu koja iznosi samo 2.99*mm* i omogućavaju da se sistemi bazirani na ovom procesoru pakuju u minimalne dimenzije, kao i minimalne težine sistema. Na Slici 4.5 dat je funkcionalni blok dijagram TMS320C66x procesora.

Najznačajnije karakteristike procesora uključuju:

- Dva TMS320C66x DSP CorePac podsistema koji redom rade na brzinama od 1.00Ghz i 1.25 GHz
    - Procesori posjeduju 32KB L1P keš memorije, 32KB L1D keš memorije i po 512KB L2 keš memorije po jezgru procesora
- Multicore Navigator koji je zadužen za kontrolu i kretanje paketa kroz sistem
- TeraNet Switch Fabric zaduženu za brzu komunikaciju između CorePac podsistema sa podsistemima za memoriju, brzine do 2Tb
- 64-Bit DDR3 memorijski interfejs prema 8 gigabajtnom adresabilnom memorijskom prostoru
- 16-Bit interfejs za eksternu memoriju (*eng. EMIF – External Memory Interface*)
- UART interfejs
- I2C interfejs
- 16 GPIO pinova
- SPI interfejs
- 16 64bit tajmera
- 3 On-Chip kontrolna sistema fazno-zaključanih petlji (*eng. PLL – Phase-locked loop*)



Slika 4.5 – Funkcionalni blok dijagram TMS320C66x procesora [9]

5. IMPLEMENTACIJA ALGORITMA DETEKCIJE POMJERAJA OBJEKTA U VIDEU

Za svrhe realizacije praktičnog dijela ovog rada korišten je Processor SDK-Vision (Vision SDK) set alata razvijen od strane Texas Instruments, a predstavlja set alata za razvoj softvera (*eng. SDK – Software Development Kit*) koji će se izvršavati na TDAx procesorskoj arhitekturi. Ovaj softverski frejmvork dozvoljava korisnicima da kreiraju različite aplikacije koje će da se koriste u naprednim automobilskim sistemima sa različitim tokovima podataka koji uključuju: radarske signale, video signale, preprocesiranje video sadržaja, obradu videa raznim algoritmima za analizu videa, kao i prikazivnje videa na izlaz nekog sistema. Processor SDK-Vision baziran je na C programskom jeziku. Vision SDK je baziran na frejmvorku po imenu *Links and Chains,* a korisnički interfejs za pristup (*eng. API – Application Programming Interface*) naziva se *Links API.*

Neki od glavnih ciljeva ovog alata jesu:

- Obezbjeđivanje konzistentih intefejsa za razvoj aplikacija da bi se kreirali novi sistemi tokova podataka koje je naknadno lako izmijeniti
- Omogućavanje brzog razvoja prototipa algoritama na DSP/EVE procesorima i kreiranje tokova podataka u kontekstu cijelog sistema
- Omogućavanje optimizacije i kontrole parametara kao što su potrošnja, efikasnost, performanse i opterećenje sistema
- Mogućnost da korisnik razvije algoritam van frejmvorka i da koristi samo interfejse prema SDK modulima koji su njemu potrebni

5.1 "Links and Chains" arhitektura

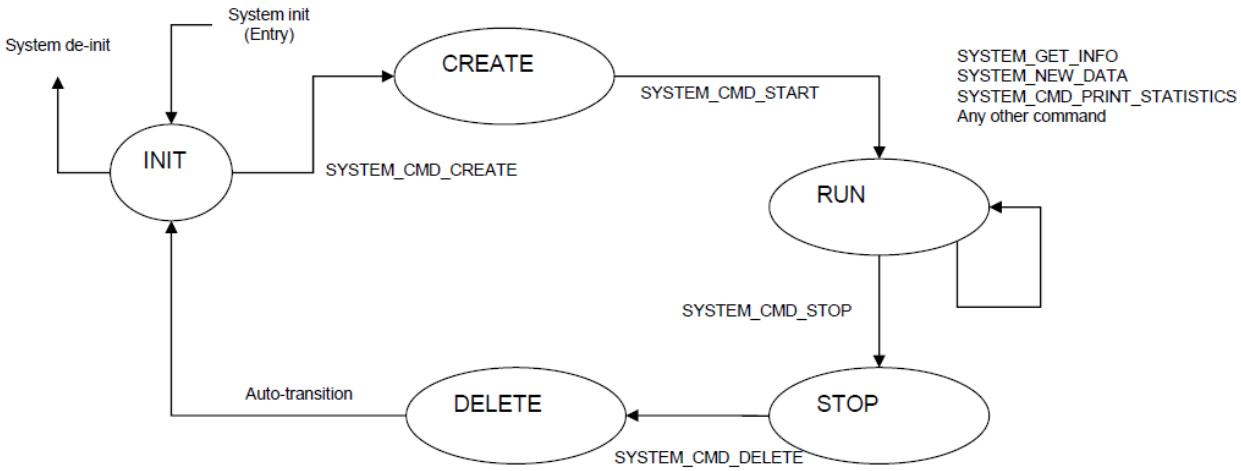
Prethodno je pomenuto da je Processor SDK-Vision baziran na *Links and Chains* frejmvorku, a da korisnik mogućnostima frejmvorka pistupa preko interfejsa koji se naziva *Links API.*

Link predstavlja osnovni gradivni element cijele arhitekture koja stoji iza *Links and Chains* frejmvorka. Link se može posmatrati kao jedan od više koraka u procesiranju podataka koji prolaze kroz sistem za obradu video sadržaja. Svaki link u sistemu zadužen je da obavlja neku specifičnu funkcionalnost. Jedan link se koristi za dohvatanje video signala sa kamere, neki drugi link se može koristiti za analizu i obradu dohvaćenog video signala, dok se neki treći link koristi da bi se taj obrađen video signal prikazao na ekranu. Svaki od linkova posjeduje takozvanu kutiju za poruke (*eng. message box*) koji mu služi da komunicira sa ostalm linkovima u sistemu. Svaki link se izvršava na posebnoj procesorskoj niti pa se tako na sistemima koji posjeduju više procesora/procesorskih jezgri linkovi mogu izvršavati paralelno jedan sa drugim. Link implementira poseban interfejs koji omogućava drugim linkovima da sa njima direktno razmjenjuju video sadržaj na bazi frejmova ili na bazi bajtovskog toka podataka. Implementacija algoritma obrađenog ovim radom podrazumijevaće da se radi o razmijeni sadržaja na bazi frejmova.

Link API dozvoljava korisnicima da kreiraju nove linkove, kontrolišu njihove parametre, kao i da ih povezuju sa drugim linkovima. Veza između više linkova naziva se lanac. Lanac se kreira na procesoru koji je određen kao domaćinski (*eng. HOST*) procesor, a u slučaju evaluacione platforme i DSP procesora

koji se koriste u svrhe ovog rada to je Cortex-M4 procesor. Kontrolni kod piše se na jednom procesoru, a interno, *Link API* koristi mehanizme međuprocesorske komunikacije (*eng. IPC – Inter-process communication*) da bi se ostvarila komunikacija između linkova koji se izvršavaju na različitim procesorima, tako da korisnik ne mora da brine o detaljima koji se tiču komunikacije linkova na niskom nivou.

Link se tokom svog životnog ciklusa može da nađe u nekoliko stanja kao što se može vidjeti na Slici 5.1.



Slika 5.1 – Dijagram stanja životnog ciklusa linka [10]

Početno stanje linka jeste INIT stanje u kojem se nalazi nakon inicijalizacije sistema. Informacija o tome da link treba da pređe iz jednog stanja u drugo komunicira se komandama. Komande kojima link prelazi u naredno stanje ciklusa jesu:

- SYSTEM_CMD_CREATE – faza kreiranja linka u kojoj se poziva *create* funkcija linka gdje se vrši inicijalna konfiguracija parametara linka
- SYSTEM_CMD_START – faza izvršavnja linka u kojoj se poziva *start* funkcija linka koja obrađuje logiku implementiranu datim linkom. Dok se nalazi u fazi izvršavanja link može da prima i neke druge komande dostupne u frejmvorku a to su:
    - SYSTEM_GET_INFO – komanda zadužena za dohvatanje trenutnih informacija o linku
    - SYSTEM_NEW_DATA – komanda koja obavještava sistem da link može da primi nove podatke na obradu
    - SYSTEM_CMD_PRINT_STATISTICS – komanda koja obavještava sistem da je zatražen pregled trenutne statistike izvršavanja sistema
- SYSTEM_CMD_STOP – faza zaustavljanja linka u kojoj se poziva *stop* funkcija linka koja zaustavlja izvršavanje logike
- SYSTEM_CMD_DELETE – faza brisanja linka u kojoj se poziva *delete* funkcija koja je najčešće zadužena za oslobađanje memorije koju je zauzimao link kao i oslobađanje memorije koja je bila

rezervisana za pomoćne strukture korištene u fazi izvršavanja linka. Nakon što je link obrisan automatski se vrši i deinicijalizacija linka iz sistema

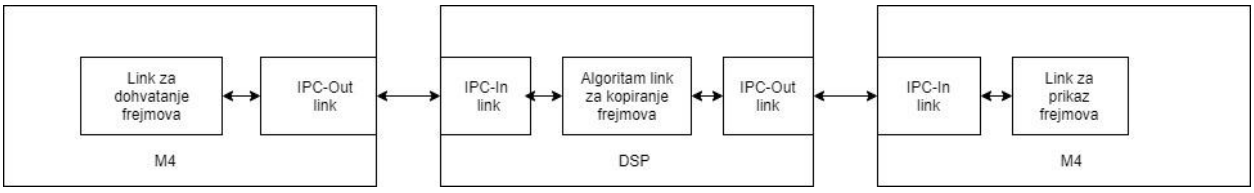
Svaki od linkova u sistemu posjeduje jednu ili više ulaznih konekcija, a isto tako posjeduje i jednu ili više izlaznih konekcija. Svaka konekcija, bila da je ulazna ili izlazna, u sebi sadrži jedan ili više logičkih kanala koji su zapravo baferi preko kojih se razmjenjuju podaci između linkova. Isto tako, ako link ima više ulaznih ili izlaznih konekcija, svaka od tih konekcija može da ima različit broj kanala. Numerisanje kanala počinje indeksom nula. Svaki kanal je povezan sa baferom podataka, a taj bafer može biti različitih tipova pa se njime mogu prenositi video frejmovi, bitski tokovi podataka, korisnički definisani podaci (*eng. Meta Data*), pa čak i kompozicije video frejmova i korisnih podataka.

Da bi se dva linka povezala i da bi se kreirala struktura lanca, potrebno je da se povežu izlazna konekcija prethodnog linka sa ulaznom konekcijom narednog linka. Ove veze moraju da poštuju određena pravila:

- Ulazna konekcija narednog linka mora da posjeduje onoliko kanala koliko posjeduje izlazna konekcija prethodnog linka, u suprotnom konekcija neće biti moguća
- Vrsta bafera za dati kanal ulazne konekcije narednog linka mora odgovarati vrsti bafera odgovarajućeg kanala izlazne konekcije prethodnog linka
- Prethodni link i naredni link moraju da se izvršavaju na istom procesoru, a izuzetak predstavljaju IPC OUT i IPC IN linkovi koji su zaduženi za komunikaciju likova koji su na različitim procesorima

5.2 Slučaj upotrebe sistema

Processor SDK-Vision daje mogućnost korisniku da na jednostavan način specifikuje kako će mu izgledati struktura aplikacije, koji linkovi će biti iskorišteni i kako će oni međusobno biti povezani. Način na koji se to specifikuje jeste upotrebom slučaja upotrebe (*eng. Use Case*). Slučaj upotrebe zapravo predstavlja isprojektovanu šemu linkova i međusobnih veza linkova unutar aplikacije. Svaki od linkova unutar Processor SDK frejmvorka posjeduje jedinstveno ime, koje se koristi tokom specifikovanja slučaja upotrebe kada se taj link koristi. Primjer jednog jednostavnog slučaja upotrebe dat je na Slici 5.2.



Slika 5.2 – Primjer slučaja upotrebe [11]

Primjerom je prikazan sistem koji na početku lanca sadrži Capture Link za dohvatanje video sadržaja (izvršava se na Cortex-M4 procesoru), koji komunicira sa linkom za kopiranje frejmova (izvršava se na DSP procesoru), koji dalje komunicira sa linkom za prikazivanje videa na ekranu (izvršava se na Cortex-M4 procesoru). Na Slici 5.2 takođe se mogu primijetiti IPC-OUT i IPC-IN linkovi koji su zaduženi za razmjenu podataka na različitim procesorima (slanje sa Cortex-M4 procesora na DSP procesor i

obrnuto). Slučaj upotrebe na osnovu kojeg frejmvork korisniku generiše potrebne fajlove definiše se u tekstualnom fajlu, a formatira se tako što se jedan iza drugog navode likovi koji se nalaze u lancu, a između njih se stavlja strelica formirana kao "->". Uz naziv linka u zagradama je moguće specifikovati na kojem procesoru treba da se izvršava dati link. Moguće je granati lanac tako da jedan link ima dvije izlazne konekcije gdje se svaka veže za po jedan novi link. Takvo granje se specifikuje u novom redu tekstualnog fajla gdje se kao prvi link navodi link koji se granao, te se na njega nadovezuje druga grana lanca koja je specifikovana slučajem upotrebe. IPC-IN i IPC-OUT linkove nije potrebno eksplicitno specifikovati. Tako na primjer slučaj upotrebe definisan na Slici 5.2 specifikujemo na sljedeći način:

```
Capture -> Alg_FrameCopy (DSP1_0) -> Display
```
(5.1)

Na primjeru koji je prikazan Slikom 5.2 korištena je posebna vrsta linkova koji se nazivaju algoritam linkovi (*eng. Algorithm Link*) i koji imaju ulogu da obrađuju video nekim određenim algoritmom, a u slučaju ovog primjera radi se o algoritmu kopiranja frejmova koji ulazni frejm kopira u izlazni bafer. Da bi se bilo koji eksterni algoritam koristio unutar Processor SDK frejmvorka potrebno je implementirati algoritam link koji će se ponašati kao interfejs za taj algoritam.

5.3 Koraci implementacije

Prije kreiranja slučaja upotrebe za sistem koji će da detektuje pomjeraj objekta u videu implementiran je algoritam detekcije pomjeraja objekta u videu baziran na estimaciji pokreta uparivanjem blokova. Implementirane su tri varijacije algoritma o kojima je bilo riječi u Poglavlju 3.2. U implementirane metode spadaju: iscrpna metoda pretrage, metoda pretrage u tri koraka i dijamantska metoda pretrage.

Zajedničko za tri navedene implementacije jeste dimenzija makro blokova koji se koriste za metodu estimacije uparivanjem blokova, a njihova dimenzija predstavljena je konstantom *MACRO_BLOCK_DIM* koja je postavljena na vrijednost 16, a vrijednost koja se dodaje na dimenzije makro bloka na svaku od četiri strane da bi se dobila dimenzija prozora pretrage predstavljena je konstantom *SEARCH_WIDNDOW_P* koja iznosi 7. Samim tim, vrijednost dimenzije prozora pretrage iznosi 30. Interfejs prema algoritmu omogućen je jedinstvenom funkcijom koja kao parametre uzima širinu i visinu frejma, trenutni frejm, prethodni frejm, metodu pretrage algoritma, adresu prostora na koju će biti smješten rezultat algoritma, kao i *fleg* kojim se specifikuje da li će se koristiti *threshold* optimizacija o kojoj će više biti rečeno u Glavi 6. Deklaracija funkcije data je izrazom (5.2).

```
Int8*** calculateMotionVectorMatrix(Int32 video_w, Int32 video_h,
UInt8* currentFrame, UInt8* prevFrame,AlgorithmLink_MotionDetection_Mode
mode, Int32 treshold_optimization, Int8*** res_mvm)
```
(5.2)

Funkcija je implementirana tako da se parametri propagiraju u nove pozive funkcija u zavisnosti od izabranog metoda pretrage algoritma. Izračunavanje mjere nesličnosti za sva tri metoda pretrage algoritma realziovano je preko računanja sume apsolutnih razlika iz prostog razloga što je za izračunavanje

potrebno koristiti samo operacije sabiranja i oduzimanja koje se nešto brže izračunavaju u odnosu na operaciju množenja koja je potrebna da bi se izračunala suma kvadrata razlika, koja se nameće kao alternativa odabranoj metodi. Kompletne implementacije navedene tri varijacije algoritma date su na CD-u koji je priložen uz rad, te se rad u nastavku glave neće fokusirati na detalje implementacija. Više riječi o navedene tri implementirane varijacije biće u Glavi 6 gdje je pažnja posvećena metodama za optimizaciju algoritma.

Naredni korak jeste da se kreira algoritam link koji će da se koristi kao interfejs za implementirani algoritam. Da bi se implementirao algoritam link potrebno je da se implementiraju dva njegova dijela a to su kostur algoritam linka (*eng. Algorithm Link Skeletal*) i *plug-in* funkcija linka. Processor SDK omogućava korisniku da se kostur linka i svi potrebni fajlovi generišu automatski, a u generisanim fajlovima se definišu funkcije za kreiranje, kontrolisanje, pokretanje, izvršavanje, zaustavljanje i brisanje linka. Ako korisnik ima potrebu da unutar linka koristi neke korisnički definisane strukture ili enumeracije, potrebno je da ih doda unutar predefinisanih struktura algoritam linka koje su definisane unutar kostura algoritam linka. Za potrebe izvršavanja algoritma struktura algoritam linka za detekciju pomjeraja objekta u videu definisana je na sljedeći način:

```
typedef struct
{
    UInt32 frameWidth;
    UInt32 frameHeight;
    UInt8* previousFrame;
    Int8*** motionVectorMatrix;
    UInt8 firstFrame;
    AlgorithmLink_MotionDetection_Mode mode;
    AlgorithmLink_MotionDetection_DrawMode drawMode;
} Alg_MotionDetection_Obj;
```

(5.3)

gdje su enkapulirane vrijednosti koje su od značaja za vrijeme izvršavanja algoritma. Vrijednosti uključuju širinu i visinu frejma, pokazivač na niz bajta koji predstavljaju reprezentaciju prethodnog frejma koji je potreban da bi se izračunala matrica vektora pomjeraja, samu matricu vektora pomjeraja u kojoj se čuva rezultat jedne iteracije algoritma da bi se taj rezultat kasnije mogao koristiti pri vizuelizaciji detekcije pomejraja objekta, promjenljivu koja daje informaciju o tome da li se radi o čitanju prvog frejma sa ulaza, način rada algoritma kojim se navodi koja od tri implementacije algoritma treba da se koristi i na kraju način iscrtavanja. Promjenljive *mode* i *drawMode* su enumeracije koje sadrže moguće izbore za način izvršavanja i način iscrtavanja. Nakon kreiranja i modifikovanja kostura algoritam linka, implementirane su *plug-in* funkcije. Lista funkcija koje su implementiraje uključuje:

- AlgorithmLink_AlgPluginCreate - zadužena za kreiranje instance algoritma
- AlgorithmLink_AlgPluginProcess - zadužena za obradu novih podataka, interno će pozivati funkciju algoritma
- AlgorithmLink_AlgPluginControl - zadužena za kontrolisanje parametara algoritma za vrijeme izvršavanja

- AlgorithmLink_AlgPluginStop - zadužena za obavljanje procesa koji su predviđeni da se obave na kraju izvršavanja algoritma
- AlgorithmLink_AlgPluginDelete - zadužena za brisanje instance algoritma na kraju izvršavanja

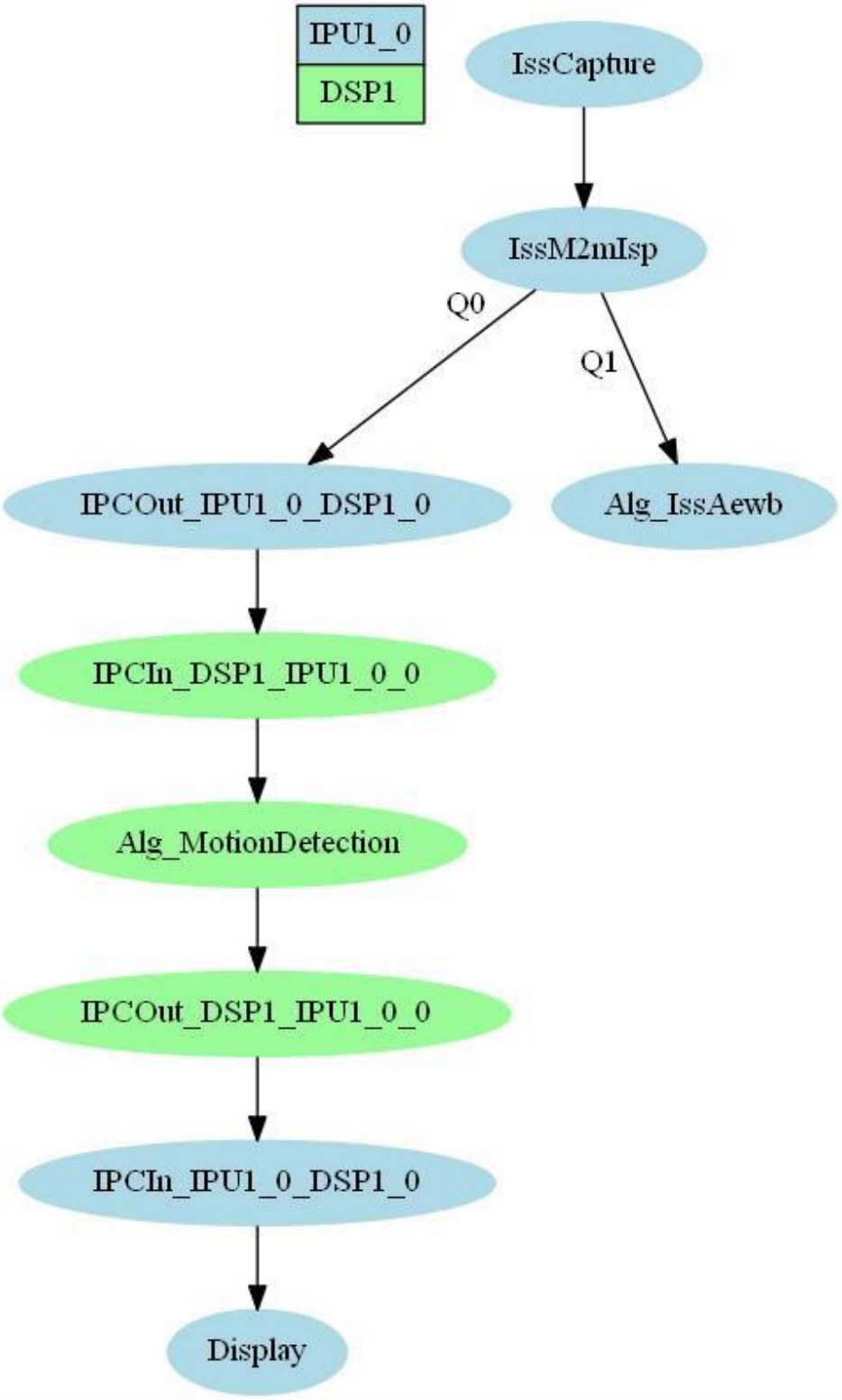
Nakon kreiranja algoritam linka, naredni korak jeste da se kreira slučaj upotrebe za projektovani sistem. Slučaj upotrebe sistema za detekciju pomjeraja objekta u videu definisan je na sljedeći način:

```
UseCase: chains_MotionDetection
IssCapture -> IssM2mIsp -> Alg_MotionDetection(DSP1) -> Display
IssM2mIsp -> Alg_IssAewb
```
(5.4)

Sistem predstavljen izrazom (5.4) definisan je kao lanac sljedećih linkova:

- IssCapture – link zadužen za dohvatanje toka frejmova
- IssM2mlsp – link zadužen za prilagođavanje videa na odgovarajući format boje (YCrBr420 u slučaju ovog sistema)
- Alg_MotionDetection(DSP1) – implementirani algoritam link koji vrši obradu dobijenog videa algoritmom za detekciju pomjeraja
- Alg_IssAewb – algoritam link koji je zadužen za brzu korekciju osvjetljenja videa koji dolazi sa kamere u situacijama kad se osvjetljenje kadra naglo promijeni

Na osnovu izraza (5.4) generisan je slučaj upotrebe za koji je izgenerisana i šema koja je data na Slici 5.3.



Slika 5.3 – Šema slučaja upotrebe za sistem detekcije pomjeraja objekta u videu

6. OPTIMIZACIJA ALGORITMA DETEKCIJE POMJERAJA

Implementacija algoritma detekcije pomjeraja objekta u videu u početku je realizovana i testirana na računarskoj platformi čije performanse daleko prevazilaze performanse evaluacionog modula koji je ciljana platforma ove implementacije. Iako je početna implementacija pokazala zadovoljavajuće rezultate na hardverski znatno snažnijoj arhitekturi, nakon kratkog prilagođavanja i testiranja implementacije na ciljanoj evaluacionoj platformi bilo je jasno da početna implementacija algoritma nije prilagođena za arhitekturu koja se bazira na DSP procesoru i da će zahtijevati dodatne modifikacije i optimizacije da bi zadovoljila određene zahtijeve, a to je da se algoritam izvršava u šezdeset frejmova po sekundi za video visoke rezolucije (*eng. FHD – Full HD*). U nastavku glave dat je pregled optimizacija koje su primijenjene na algoritam da bi se dostigli zadovoljavajući rezultati, a u Glavi 7 dat je detaljan pregled rezultata nakon testiranja algoritma prvo bez optimizacija, pa nakon toga sa primijenjenim optimizacijama.

6.1 Optimizacije na nivou algoritma

6.1.1 Prag poređenja blokova

Kao što je u Glavi 5 naglašeno, algoritam detekcije pomjeraja objekta u videu implementiran je sa ciljem da radi na videu visoke rezolucije širine 1920 i visine 1080 piksela. Dimenzija blokova na koje se trenutni i prethodni frejm dijele tokom izvršavanja algoritma iznosi 16 piskela tako da se samo jedan frejm dijeli na 120 blokova po horizontali i 67 blokova po vertikali, a to ukupno izlazi na 8.040 blokova. Da bi se jedna iteracija algoritma izvršila potrebno je da se za svaki od blokova izvrši pretraga za blokom sa najboljim poklapanjem, a za proširenje prozora pretrage koje iznosi 7 piskela sa svake strane bloka, ako se radi o iscrpnoj metodi pretrage, potrebno je da se blok trenutnog frejma uporedi sa ukupno 225 blokova prethodnog frejma. Ako se sve to uzme u obzir ukupno je potrebno izvršiti tačno 1.809.000 poređenja u samo jednoj iteraciji algoritma, a da bi se zadovoljili uslovi koji su postavljeni potrebno je izvršiti bar 60 iteracija algoritma u jednoj sekundi.

Tokom izvršavanja jedne iteracije algorima za slučaj kada se radi o statičnoj pozadini frejma u kojem se pomjera jedan ili više objekata, može se primijetiti da većina blokova trenutnog frejma zadržava istu poziciju i na prethodnom frejmu videa, a radi se o blokovima koji obuhvataju pozadinu koja je statična. Izvršavanje algoritma nad blokovima koji ne mijenjaju svoju trenutnu poziciju u odnosu na prethodni frejm postaje redudantno jer će rezultat vektora pomjeraja za takve blokove uvijek iznositi $[0, 0]^T$. Iz tog razloga se uvodi optimizacija praga poređenja blokova (*eng. Threshold Optimization*) koja se primjenjuje prije izvršavanja algoritma pretrage bloka sa najboljim poklapanjem, a koja provjerava kolika je razlika između bloka sa trenutnog frejma i njemu korespondirajućeg bloka sa prethodnog frejma. Ako je razlika blokova manja od praga poređenja blokova *block_similarity_thresh,* može se pretpostaviti da se radi o istom bloku koji se nije pomjerio sa svoje originalne lokacije i da nije potrebno vršiti pretragu za dati blok, nego se kao rezultat pretrage automatski uzima vektor pomjeraja [0, 0] koji označava da se blok nije pomjerio. U suprotnom, ako razlika blokova iznosi više nego što je definisano pragom *block_similarity_thresh* onda se

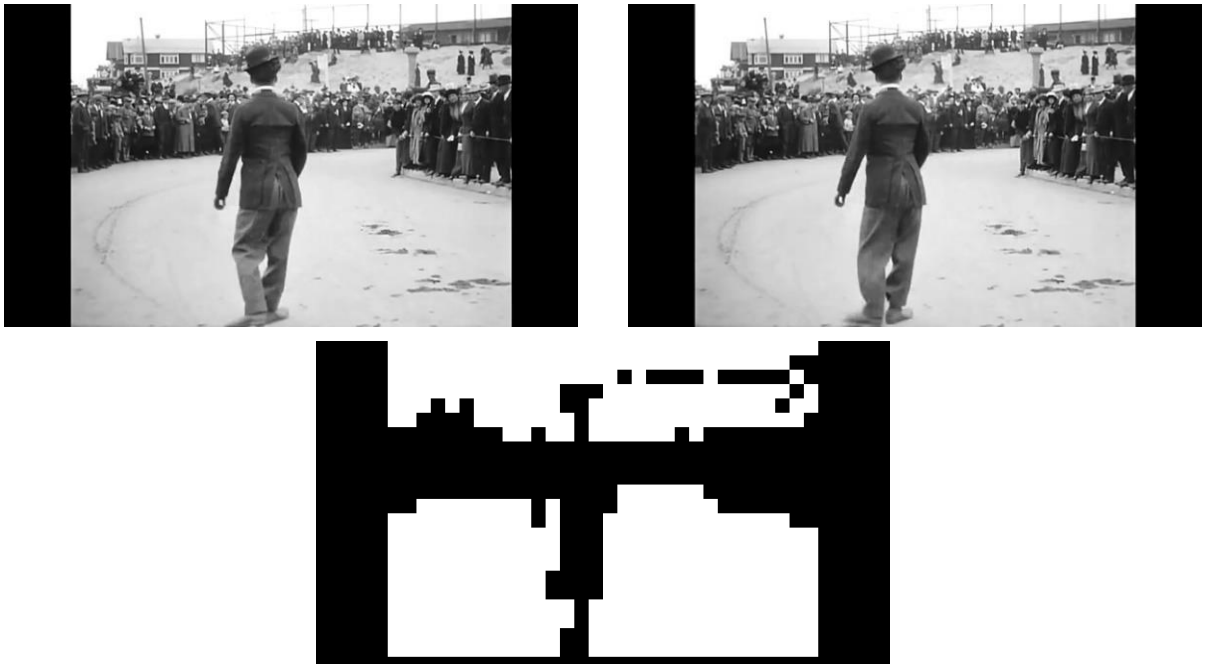
algoritam pretrage izvršava za dati blok trenutnog frejma. Kao metod izračunavnaja razlike blokova koristi se usrednjena suma apsolutnih razlika piskela na odgovarajućim blokovima a definiše se kao:

$$D = \frac{1}{dim^2} \sum_{(x,y)\in B} |b_1(x,y) - b_2(x,y)| \tag{6.1}$$

gdje *dim* predstavlja dimenziju bloka, a b_1 i b_2 predstavljaju blokove trenutnog frejma i prethodnog frejma uporedo. Na osnovu izraza (6.1) može se definisati i izraz kojim se definišu dva slučaja optimizacije praga poređenja kao:

$$\begin{aligned} &D > block\_similarity\_thresh, \text{ izvršiti pretragu} \\ &D < block\_similarity\_thresh, \text{ ne izvršiti pretragu} \end{aligned} \tag{6.2}$$

Na Slici 6.1 predstavljena je binarna reprezentacija podjele frejma na blokove gdje su bijelom bojom predstavljeni blokovi za koje nije izvršena pretraga, a crnom bojom predstavljeni blokovi za koje jeste izvršena pretraga nakon optimizacije algoritma pragom poređenja blokova. Sa slike se može vidjeti da se za veliki broj blokova pretraga neće odvijati, a time će se drastično smanjiti ukupan broj pretrage koje će se izvršiti tokom jedne iteracije algoritma. U Prilogu 1 nalazi se izvorni kod funkcije koja se odnosi na optimizaciju pragom poređenja blokova.



Slika 6.1 – Trenutni frejm, prethodni frejm i binarna reprezentacija frejma podijeljenog na blokove u zavisnosti od toga da li je vršena pretraga nakon optimizacije pragom poređenja blokova

Manje vrijednosti praga znače da će se tolerisati manje razlike frejmova i da će detekcija pomjeraja iz tog razloga biti preciznija, dok se kod većih vrijednosti praga tolerišu veće razlike blokova pa se iz toga razloga dobijaju bolje performanse ali uz smanjenu preciznost detekcije pomjeraja objekata u videu.

6.1.2 Ograničenje dubine dijamantske pretrage

U Poglavlju 3.2.3 definisana je dijamantska metoda pretrage gdje je opisana kao metoda koja se izvršava rekurzivno dok se ne pronađe blok prethodnog frejma koji je najsličniji bloku sa trenutnog frejma koji se pretražuje. U situacijama kada se radi o malim pomjerajima objekata u videu blokovi sa najboljim poklapanjem sa trenutnim blokom pretrage nalaze se najčešće na istoj lokaciji na prethodnom frejmu ili u neposrednoj blizini do koje se dolazi u nekoliko iteracija algoritma. U slučaju kada se radi o većim pomjerajima objekata u videu, broj poređenja blokova raste i time degradira performanse. Ograničavanje dubine dijamantske pretrage predstavlja kompromis kojim se ograničava degradiranje performansi uz cijenu gubitka preciznosti detekcije pomjeraja objekta u videu.

Kako svaka osim prve rekurzivne iteracije vrši ukupno pet poređenja blokova (prva iteracija vrši 9 poređenja) vrijednost ograničenja dubine dijamantske pretrage postavljena je na 25 čime su u najgorem slučaju performanse dijamantske pretrage i dalje dva puta bolje od prosječnih performansi iscrpne pretrage koja za jedan blok vrši 225 pretraga. Sa druge strane mana ove optimizacije jeste to što se u ograničenom broju rekurzivnih koraka može pretražiti ograničen broj blokova među kojima se potencijalno blok koji se traži ne nalazi, ali to je cijena koja se mora platiti da bi se algoritam izvršavao zadovoljavajućom brzinom na ciljanoj platformi.

6.1.3 Minimizacija broja alociranja memorije

Tokom prvobitne implementacije algoritma koji se izvršavao na računarskoj platformi, trajanje alokacije memorije nije predstavljao značajen problem za performanse algoritma, ali se nakon prilagođavanja i pokretanja programa na ciljanoj platformi pokazalo da trajanje alokacije memorije primjetno degradira performanse. Prvobitna implementacija algoritma je tokom svake iteracije algoritma alocirala novi prostor za prethodni frejm, kao i novi prostor za smještanje rezultata algoritma detekcije pomejraja. Kod je optimizovan na način da su se u sturkturu algoritam linka (Poglavlje 5.3) dodale reference na prostor koji je samo jednom tokom inicijalizacije algoritam linka alociran za prethodni frejm i rezultat pretrage, a svaki sljedeći put vrijednosti prethodnog frejma i rezultata pisane su na te iste lokacije. Pisanje vrijednosti prethodnog frejma prvobitno se vršilo pozivom metode standardne biblioteke *memcpy()* koja je kopirala bafer trenutnog frejma u bafer prethodnog frejma. Zbog veličine memorijskog bloka koji se kopira, ova operacija je takođe predstavljala prepreku kod dostizanja željenih performanski. Problem je otklonjen optimizacijom kopiranja pomoću modula za direktni pristup memoriji (*eng. DMA – Direct Memory Acccess*) koja je obrađena u narednom poglavlju koje se odnosi na optimizacije algoritma na nivou Processor SDK frejmvorka.

6.1.4 Odmotavanje petlji

Odmotavanje petlji (*eng. Loop unrolling or loop unwinding*) jeste metoda transformacije petlji koja se koristi za optimizaciju brzine izvršavanja programa po cijenu malo veće veličine binarnih fajlova, kompromis poznat kao kompromis prostora i vremena (*eng. space-time tradeoff*) [12]. Cilj odmotavanja petlji jeste da se poveća brzina izvršavanja programa tako što se smanjuje broj ili se potpuno eliminišu

instrukcije koje kontrolišu petlje, kao što su aritmetika pokazivača za kretanje kroz petlju i testovi koji se vrše nakon svake iteracije petlje. Da bi se eliminisao ovaj procesni višak, petlje se mogu pisati tako što se umjesto jednog izvršavanja operacija u petlji izvrši više istih nezavisnih operacija tokom jedne iteracije petlje. Odmotavanje petlji postaje još efikasnije kada se u obzir uzme količina podataka koje procesor kešira kada pročita određen memorijski prostor. Primjer odmotavanja petlji dat je izrazom (6.3).

```
for(int i = 0; i < 16; i++){
    array[i] += 10;
}

for(int i = 0; i < 16; i += 4){
    array[i] +=10;
    array[i + 1] +=10;
    array[i + 2] +=10;
    array[i + 3] +=10;
}
```
(6.3)

Prvi blok koda na primjeru (6.3) jeste kod bez primijenjene optimizacije odmotavanja petlji gdje se operacija povećavanja članova niza odvija jedna po iteraciji petlje. Drugi blok koda je optimizovan djelemičnim odmotavanjem petlje gdje se sada uzastopno odvijaju po četiri operacije uvećavanja članova niza. Drugi blok koda je još bolje optimizovan ako uzmemo u obzir da su nakon čitanja i-tog člana niza iz memorije keširana i naredna tri pa se operacije povećavanja odvijaju znatno brže. Odmotavanje petlji ima efekta samo ako su operacije koje će se odvijati tokom jedne iteracije petlje međusobno nezavisne jer ih u tom slučaju procesor može izvršavati paralelno.

6.2 Optimizacije na nivou Processor SDK frejmvorka

Iako su poboljšale sveukupne performanse programa, optimizacije na nivou algoritma i samog programskog jezika C nisu bile dovoljne da bi se postigle zadovoljavjauće performanse algoritma. Srećom, Texas Instrumesnts (skraćeno TI) je u svojoj implementaciji C jezika uveo mnoge načine da se performanse programa poboljšaju, počevši od optimizacija koje nudi samo kompajler, pa do optimizacija preko takozvanih intrinzika, pragmi, specijalnih ključnih riječi i sličnih mehanizama.

Kompajler koji je TI razvio u svrhe razvijanja programa prilagođenih za platforme sa DSP procesorom jeste *cl6x* kompajler. Moć ovog kompajlera ogleda se u mnogobrojnim opcijama koje je moguće navesti prilikom kompajliranja programa i koje kontrolišu način rada kompajlera. Među mnogobrojnim opcijama dostupnim korisniku koji koristi kompajler dostupne su i opcije za optimizaciju programa. Neke od značajnijih opcija za optimizaciju date su u Tabeli 6.1.

Tabela 6.1 – Optimizacione opcije kompajlera

Opcija	Alias	Opis
--opt_level=0	*-O0*	koristi se za optimizaciju iskorištenja registara alocira varijable na registre obavlja rotacije petlji ako je moguće izbacuje neiskorišten kod proširuje pozive funcijama koje su naznačene kao *inline*
--opt_level=1	*-O1*	koristi O0 optimizacije i optimizuje lokalno obavlja propagaciju lokalnih kopija i konstanti odbacuje nekorištene dodjele
--opt_level=2	*-O2 ili –O*	koristi O1 optimizacije i optimizuje globalno, ovo je podrazumijevani nivo optimizacije obavlja optimizacije petlji odbacuje globalne nesikorištene dodjele, obavlja automatsko odmotavanje petlji
--opt_level=3	*-O3*	koristi O2 optimizacije i optimizuje navedeni fajl odbacuje sve funkcije koje se nikada ne pozivaju pojednostavljuje funkcije koje imaju povratne slučajeve koji se nikada ne izvršavaju automatski dodaje *inline* ključnu riječ za male funkcije
--auto_inline=[size]	*-oi*	postavlja veličinu automatskog pretvaranja funkcije u *inline* ako se koristi –O3 optimizacija
--single_inline		pretvara funkcije u *inline* ako su samo jednom pozvane
--opt_for_speed[=n]	*-mf*	kontroliše odnos brzine i memorijskog zauzeća u opsegu od 0 do 5, podrazumijevano je 4
-gen_opt_info=1	*-on1*	nakon kompajliranja se kreira informacioni fajl o izvršenim optimizacijama koji je koristan prilikom analize efekta korištenih optimizacija

U nastavku poglavlja dat je pregled korištenih optimizacija na nivou Processor SDK frejmvorka koje uključuju: upotrebu intrinzika, DMA kopiranje, upotrebu programskih pragma direktiva, poravnanje memorijskih adresa uz pomoć *_nassert()* poziva, pisanje *inline* funkcija i upotrebu ključne riječi *restrict*.

6.2.1 *Upotreba intrinzika*

Svojstvene funkcije (*eng. Intrinsic functions*) ili intrinzici (kako će biti oslovaljavani u nastavku rada) jesu funkcije koje su ugrađene u kompajler. Ako se funkcija smatra intrinzikom, izvorni kod te funkcije posmatra se kao *inline* i ugraćuje se kompajlerski na mjesto gdje je taj intrinzik pozvan. To omogućava znatno efikasnije izvršavanje programa jer se odbacuje mogući *overhead* prilikom poziva standardne funkcije i dozvoljava da se visoko efikanse mašinske instrukcije emituju kroz poziv datog intrinzika. Intrinzik je često brži i od *inline* asemblera jer kompajler koji nudi određene intrinzike zna unaprijed kako se oni ponašaju pa je optimizovan za takve operacije [13]. C6000 (*cl6x*) kompajler podržava veliki broj intrinzika, a u Tabeli 6.2 dat je pregled intrinzika korištenih tokom optimizacije algoritma za detekciju pomjeraja objekta u videu. U Prilogu 2 dat je isječak koda koji obuhvata korištenje navedenih intrinzika.

Tabela 6.2 – Lista primijenjenih intrinzika

Intrinzik	Opis
_amem4_const	omogućava poravnata čitanja dužine 4 bajta iz memorije. Pokazivač u ovom slučaju mora da bude poravnat na adresu djeljivu sa četiri. Ovaj intirnizk je korišten da bi se zajedno mogla učitati i paralelno obrađivati po četiri piksela (njihove luminantne komponente) tokom procesa izračunavnaja mjerne nesličnosti blokova.
_subabs4	omogućava izračunvanje vrijednosti apsolutne razlike za svaki par pakovanih osmobitnih vrijednosti nad četvorobajtnom vrijednošću čija se adresa prosljeđuje kao parametar intrinzika. Ovaj intrinzik je korišten da bi se izračunale apsolutne razlike prethodno učitanih nizova od po četiri piskela za trenutni i prethodi frejm, u procesu određivanja mjere nesličnosti blokova.
_unpkhu4	omogućava raspakivanje dvije neoznačene osmobitne vrijednosti gornjeg dijela četvorobajtnog podatka u jedan neoznačeni šesnaestobitni pakovani podatak. Korišteno da bi se rezultati osmobitnih operacija kombinovali u šesneastobitne podatke.
_unpklu4	omogućava raspakivanje dvije neoznačene osmobitne vrijednosti donjeg dijela četvorobajtnog podatka u jedan neoznačeni šesnaestobitni pakovani podatak. Korišteno da bi se rezultati osmobitnih operacija kombinovali u šesneastobitne podatke.
_add2	omogućava operaciju dodavanja gornje i donje polovine podatka koji je prosljeđen kao prvi argument na gornju i donju polovinu podatka koji je prosljeđen kao drugi argument i vraća rezultat kao novu vrijednost. Koristi se da bi se izračunala ukupna suma šesnaestobitnih vrijednosti koje su sadržale rezultate osmobitnih operacija absolutne razlike.
_nassert	detaljnije objašnjen u Poglavlju 6.2.4

6.2.2 *DMA kopiranje*

U Poglavlju 6.1.3 koje govori o minimizaciji učestalosti alociranja memorije pomenut je sličan problem, a to je kopiranje sadržaja memorije u slučaju kada se kopira sadržaj trenutnog frejma u bafer koji je zadužen da reprezentuje prethodni frejm. Prvobitna implementacija algoritma kopiranje sadržaja memorije obavljala je preko standarnde funkcije *memcpy* koja je za kopiranje frejma dimenzija videa visoke rezolucije djelimično degradirala performanse toka programa, jer ovaj način kopiranja memorije opterećuje procesor.

Kontorler direktrnog pristupa memoriji (*eng. DMA – Direct Memory Access*) sa druge strane omogućava transfer podataka između regiona u memoriji bez intervencije procesora. DMA kontorler dozvoljava kretanje podataka iz i do interne memorije, internih periferija ili eksternih uređaja u pozadini rada procesora, nezavisno od njega [14]. DMA kontroler dostupan na TMS320C66x procesoru posjeduje četiri kanala i tako dozvoljava da se četiri različita konteksta odvijaju istovremeno.

DMA kontroler TMS320C66x procesora posjeduje sljedeće značajne karakteristike:

- Pozadinski rad – DMA kontorler radi nezavisno od procesora
- Visok protok podataka – Elementi se prenose brzinama koje odgovaraju taktu procesora
- Četiri kanala – DMA kontorler može da prati kontekst četiri nezavisna transfera podataka
- Puni 32-bitni adresni opseg – DMA kontroler može da pristupi bilo kojoj regiji memorijske mape
- Generisanje prekida (*eng. interrupt*) – Na kraju prenosa svakog frejma moguće je procesoru poslati prekid

Iz razloga što posjeduje ove karakteristike, prvenstveno što ima mogućnost da radi nezavisno od procesora, DMA je iskorišten da se izvrši kopiranje trenutnog frejma u prethodni frejm. Za ovu optimizaciju iskorišten je specijalizovan DMA kontorler koji je dostupan na TMS320C66x procesoru pod nazivom unaprijeđeni DMA (*eng. EDMA – Enhanced DMA*) koji je zadužen za transfer podataka između L2 keš memorije i periferija uređaja i omogućava korisnički programiranje transfere podataka. U Prilogu 3 dat je isječak koda u kojem je prikazan proces konfiguracije EDMA transfer strukture i pokretanje procesa transfera podataka pomoću EDMA kontrolera.

6.2.3 *Upotreba pragma direktiva*

U opštem slučaju, pragma direktive predstavljaju metod specifikovan C standardom kojim se kompajleru prosljeđuju dodatne informacije, pored onoga što je definisano samim jezikom. Pragma direktive nagovještavaju kompajleru kako da tretira neki dio koda, bilo to funkcija, neki objekat ili neka posebna sekcija koda. C6000 (cl6x) kompajler podržava značajan broj pragma direktiva a od značaja za ovaj rad jeste *MUST_ITERATE* direktiva. Sintaksa date direktive data je izrazom (6.4).

```
#pragma MUST_ITERATE(min, max, multiple);
```
(6.4)

Implementacija i optimizacija algoritma detekcije objekta u videu baziranoj na estimaciji pokreta bazirana je na velikom broju iteracija kroz mnogobrojne petlje pa se dosta vremena utroši na mašinske instrukcije koje rade sa petljama. U nekim slučajevima, da bi kompajler uspješno odmotao petlju, potrebno

mu je obezbijediti minimalan i maksimalan broj iteracija koje se mogu desiti tokom izvršavanja *for* petlje. Iz tog razloga *MUST_ITERATE* direktiva kao parametre uzima minimalan i maksimalan broj iteracija, kao i parametar *multiple* koji kompajleru govori sa kojim brojem je broj prolaza djeljiv. Svi parametri su opcioni i potrebno je biti siguran u tačnost vrijednosti koje šaljemo za date parametre, jer je u suprotnom ishod programa nepredvidiv. Ako minimalna vrijednost nije specifikovana koristiće se nula, a ako maksimalna vrijednost nije specifikovana koristiće se najveća moguća cjelobrojna vrijednost.

U Prilogu 2 može se vidjeti primjena *MUST_ITERATE* direktive koja za parametre minimalnog i maksimalnog broja iteracija uzima 16, što kompajleru garantuje da će se petlja sigurno izvršiti tačno 16 puta.

6.2.4 Poravnjanje memorijskih adresa uz _nassert

Da bi kompajler što bolje mogao optimizovati izvršavanje petlji i operacija koje se odvijaju u petljama, potrebno mu je obezbijediti što više informacija koje on onda može iskoristiti da sa sigurnošću izvrši neke optimizacije koje inače ne bi mogao sam zaključiti. Kao što se sa *MUST_ITERATE* kompajleru govori sa koliko iteracija i podataka će da radi neka petlja, tako se sa intrinzikom *_nassert* kompajleru daje do znanja da su podaci poravnati na neki specifičan način. Izrazom (6.5) se kompajleru daje do znanja da je adresa na koju pokazuje *ptr* poravnata po osmobajtnoj granici.

```
_nassert( (int) ptr % 8 == 0 );
```
(6.5)

Poravnjanje memorijskog prostora je od značaja jer kompajler može da iskoristi tu informaciju da proizvede kod koji izvršava određenu operaciju na više podataka istovremeno (*eng. SIMD – Single Instruction Multiple Data*). Bitno je naglasiti da *_nassert* samo nagovještava kompajleru da se radi o nekoj vrsti poravnanja, a korisnik je dužan da obezbijedi poravnanje memorije. Ova optimizacija korištena je tokom izračunavanja mjere nesličnosti blokova da bi se pokušalo nametnuti izvršavanje jedne operacije nad više uzastopnih osmobitnih podataka odjednom. Primjer primjene optimizacije dat je u Prilogu 2.

6.2.5 Pisanje inline funkcija

Razvoj kompleksnih sistema sa velikim brojem funkcionalnosti zahtjeva pisanje modularnog koda i time povlači pisanje velikog broja funkcija specijalizovane namjene koje se interno pozivaju iz drugih opštijih funkcija. U sistemima čiji procesi nisu striktno ograničeni vremenom izvrašavanja pisanje takvog koda je prihvatljivo. Ali kada se radi o sistemima gdje je vrijeme izvršavanja procesa ograničeno, česti pozivi funkcija mogu značajno usporiti sistem zog takovzvanog *overhead*a koji se generiše iz razloga što svaki poziv funkcije sa sobom povlači cijelu proceduru kreiranja pozivnog steka, prenosa parametara u poziv funkcije, kao i skidanja funkcije sa pozvinog steka i skidanje povratnih parametara sa steka.

Rješenje za ovaj problem jeste *inline* ključna riječ koja se navodi ispred naziva funkcije pri definisanju iste. Ključna riječ *inline* govori kompajleru da kod pozivane funkcije direktno ugradi na mjesto odakle se funkcija poziva i na taj način izbjegne *overhead* koji sa sobom nosi stvarni poziv funkcije. Preporuka je da se *inline* koristi za jednostavne funkcije koje se često pozivaju. Kompromis na koji se

pristaje kada se koristi *inline* ključna riječ jeste dobitak na brzini izvršavanja po cijeni veće dužine generisanog koda od strane kompajlera. Bitno je naglasiti da, kao i sve ostale specijalizovane ključne riječi, *inline* samo nagovještava kompajleru da bi trebao da ugradi kod pozvane funkcije na mjesto odakle je pozvana, a krajnji ishod zavisi od kompajlera i zaključka koji on donosi nakon detaljne analize koda.

Primjer primjene navedene *inline* optimizacije dat je u Prilogu 2 gdje je funkcija *calculateSumOfAbsoluteDifferences* proglašena kao *inline* iz razloga što obavlja jednostavne aritmetičke operacije i interno ne vrši pozive prema drugim funkcijama (pozivaju se intrinzici koji su takođe *inline*).

6.2.6 *Upotreba restrict ključne riječi*

Da bi se dodatno pomoglo kompajleru da predvidi memorijske zavisnosti moguće je označiti neki pokazivač, referencu ili niz ključnom riječi *restrict*. Na ovaj način se kompajleru daje garancija da se nekom memorijskom prostoru iz određenog programskog opsega može prsitupiti samo preko reference označene sa *restrict*. Ovo pomaže kompajleru jer se lakše definiše *alijasovanje* informacija. Primjer *restrict* optimizacije dat je izrazom (6.6) kojim se kompajleru govori da funkcija *func* nikada neće biti pozvana sa pokazivačima *arr1* i *arr2* čiji se objekti na koje oni pokazuju poklapaju u memoriji.

```
void func(restrict int* arr1, restrict int* arr2){
   //tijelo funkcije func
}
```
(6.6)

Na ovaj način se kompajleru garantuje da pisanje na jednu lokaciju neće uticati na čitanje druge lokacije i obrnuto. Primjer upotrebe *restrict* optimizacije dat je u Prilogu 2 gdje je naznačeno da se parametri funkcije koji pokazuju na početak blokova u memoriji neće preklapati tokom izvršavanja funkcije.

7. TESTIRANJE I ANALIZA REZULTATA TESTIRANJA

Tokom faze testiranja algoritma detekcije pomjeraja u videu testirane su tri implementirane varijacije pretrage za uparivanje blokova : iscrpna pretraga, pretraga u tri koraka i dijamantska pretraga. Testiranje je obavljeno za tri različita nivoa optimizacija koja su navedena ispod, a u zagradama su date skraćenice koje će u nastavku glave biti korištene kao zamjena za pune nazive navedena tri nivoa:

1. Nema optimizacije (NO)
2. Optimizacije na nivou algoritma (OA)
3. Optimizacije na nivou algoritma + optimizacije na nivou frejmvorka (OA + OF)

Testovi su izvršeni za dva različita parametra, a skraćenice koje će u nastavku glave biti korištene kao zamjena za puni naziv parametara date su u zagradama:

1. Trajanje izvršavanja jedne iteracije algoritma (DOA)
2. Broj frejmova po sekundi koje sistem obrađuje u realnom vremenu (FPS)

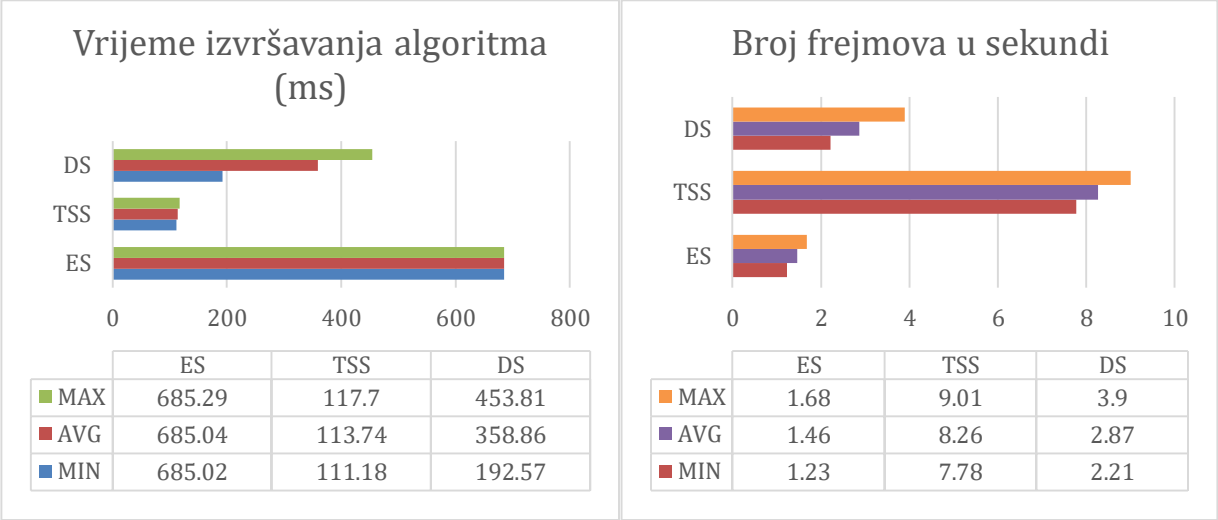
Takođe, metode pretrage će se u nastavku voditi pod skraćenim oblikom kao:

1. Iscrpna pretraga – ES, skrećeno od Exhaustive Seach
2. Pretraga u tri koraka – TSS, skraćeno od Three Step Search
3. Dijamantska pretraga – DS, skraćeno od Diamond Search

Ukupno je izvršeno po 25 testova za trajanje izvršavanja jedne iteracije algoritma za svaku od kombinacija metoda pretrage i nivoa optimizacije, a svaki test se bazirao na usrednjavanju 50 uzastopnih trajanja izvršavanja jedne iteracije algoritma. Takođe, ukupno je izvršeno po 25 testova za broj frejmova u sekuni koje sistem stigne da obradi za svaku od kombinacija metoda pretrage i nivoa optimizacije, a broj frejmova po sekundi je dobijen je na osnovu generisanja statistike performansi sistema koje nudi Processor SDK frejmvork i bazira se na posljednijh 5 sekudni rada sistema. Testovi su vršeni u nasumičnim trenucima pod različitim nivoima opterećenja algoritma (od sitnih pomjeraja pred kamerom, sve do naglih i čestih pomjeraja na različitim udaljenostima od kamere). U nastavku je dat pregled rezultata testova uz kratku analizu svakog od njih. Na CD-u priloženom uz rad nalazi se dokument u kojem se nalaze kompletni rezultati svih mjerenja izvršenih tokom faze testiranja. U Prilogu 4 dat je odsječak koda korišten za izračunavanje prosječnog vremena izvršavanja jedne iteracije algoritma. Potrebno je naglasiti da je evaluacioni model ploče konfigurisan da sa modula kamere dohvata video u nešto više od 56 frejmova po sekundi te se time u testovima nisu mogi postići bolji rezultati.

7.1 Bez optimizacije

Kao što je u uvodu Glave 6 naglašeno, nakon kratkog prilagođavanja algoritma TMS320C66x procesorskoj arhitekturi, implementacija algoritma uspješno je pokrenuta na ciljanom sistemu. Sa druge strane, performanse sistema bez dodatne optimizacije nisu se pokazale kao zadovoljavajuće, što se može zaključiti sa grafika predstavljenog Slikom 7.1.



	ES	TSS	DS
MAX	685.29	117.7	453.81
AVG	685.04	113.74	358.86
MIN	685.02	111.18	192.57

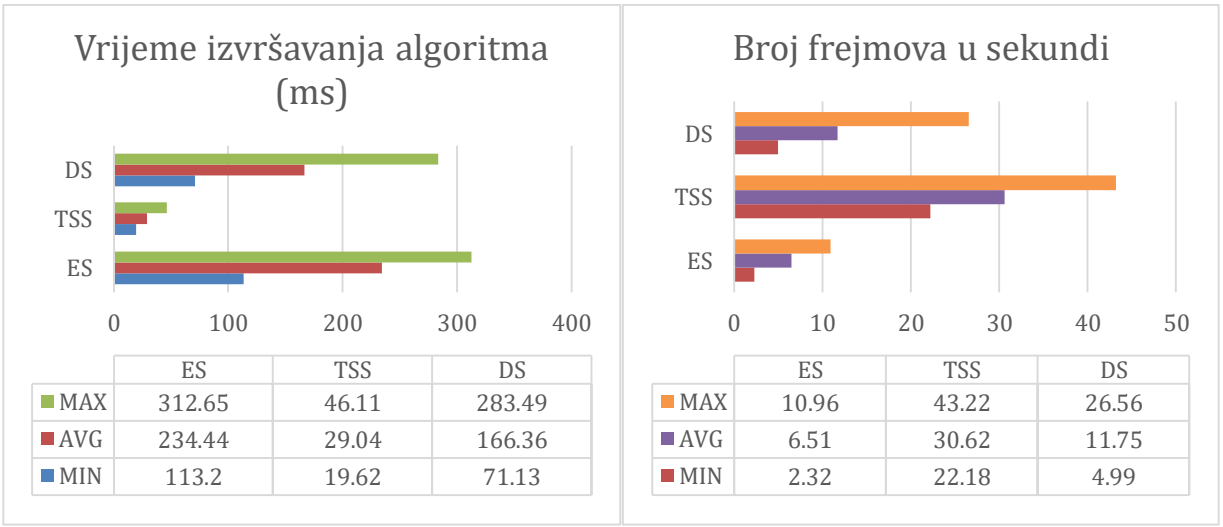
	ES	TSS	DS
MAX	1.68	9.01	3.9
AVG	1.46	8.26	2.87
MIN	1.23	7.78	2.21

Slika 7.1 – Performanse sistema bez optimizacija

Iako se TSS pokazao kao najbolja metoda pretrage, sa prosječnim brojem frejmova u sekudi od samo 8.26, razultati su još uvijek daleko od zadovoljavajućih. Očekivano, iscrpna pretraga se pokazala kao najgora implementacija bez optimizacija zbog njenog po prirodi velikog broja poređenja koje obavlja da bi pronašla najbolje poklapanje blokova.

7.2 Optimizacije na nivou algoritma

Na Slici 7.2 dati su rezultati testiranja nakon što su na algoritam primijenjene optimizacije na nivou algoritma.



	ES	TSS	DS
MAX	312.65	46.11	283.49
AVG	234.44	29.04	166.36
MIN	113.2	19.62	71.13

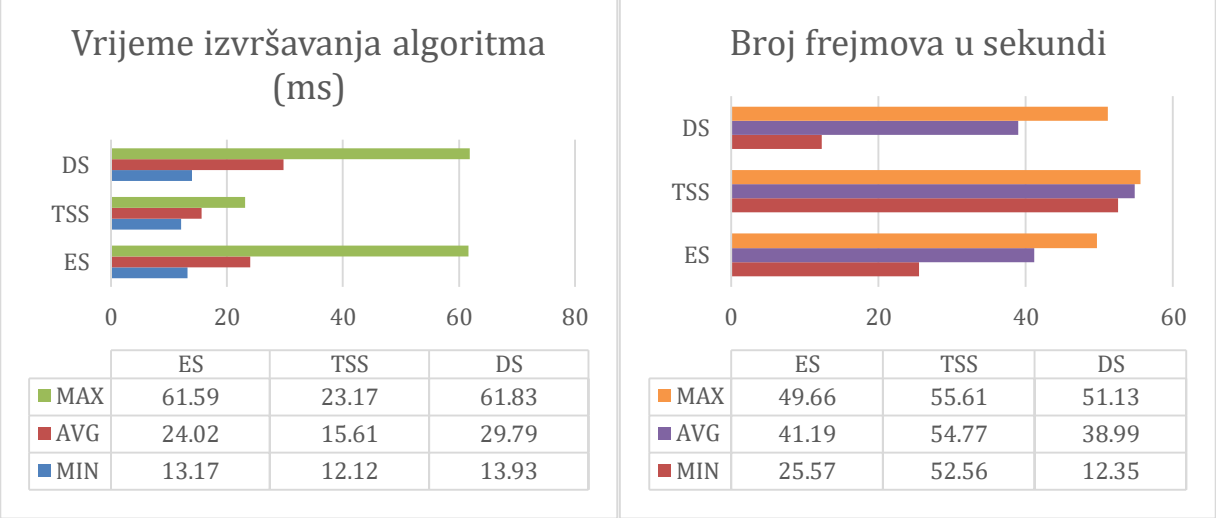
	ES	TSS	DS
MAX	10.96	43.22	26.56
AVG	6.51	30.62	11.75
MIN	2.32	22.18	4.99

Slika 7.2 – Performanse sistema uz optimizacije na nivou algoritma

Nakon primijenjenih optimizacija na nivou algoritma, performanse sistema znatno su se popravile. Najveći uticaj na nagli skok performansi sistema imala je optimizacija uvođenja praga poređenja, koja je eliminisala znatan broj bespotrebnih poređenja i omogućila da se TSS implementacija algoritma izvršava u prosjeku od preko 30 frejmova po sekundi što je već u tom trenutku predstavljalo porast od 370.82% u odnosu na TSS implementaciju bez optimizacije. Diferencijalno manji, ali procentualno veći porast performansi vidi se i kod ostala dva implementirana metoda pretrage gdje performanse ES metoda rastu za 445.9%, dok su performanse DS metoda unaprijeđene za cijelih 409.4%.

7.3 Optimizacije na nivou algoritma i optimizacije na nivou frejmvorka

Performanse sistema značajno se popravljaju uvođenjem optimizacija na nivou frejmvorka, a zajedno se optimizacijama na nivou algoritma dostižu očekivanja postavljena na početku izrade ovog rada. Slika 7.3 daje pregled performansi sistema nakon kombinovanja pomenutih optimizacija.

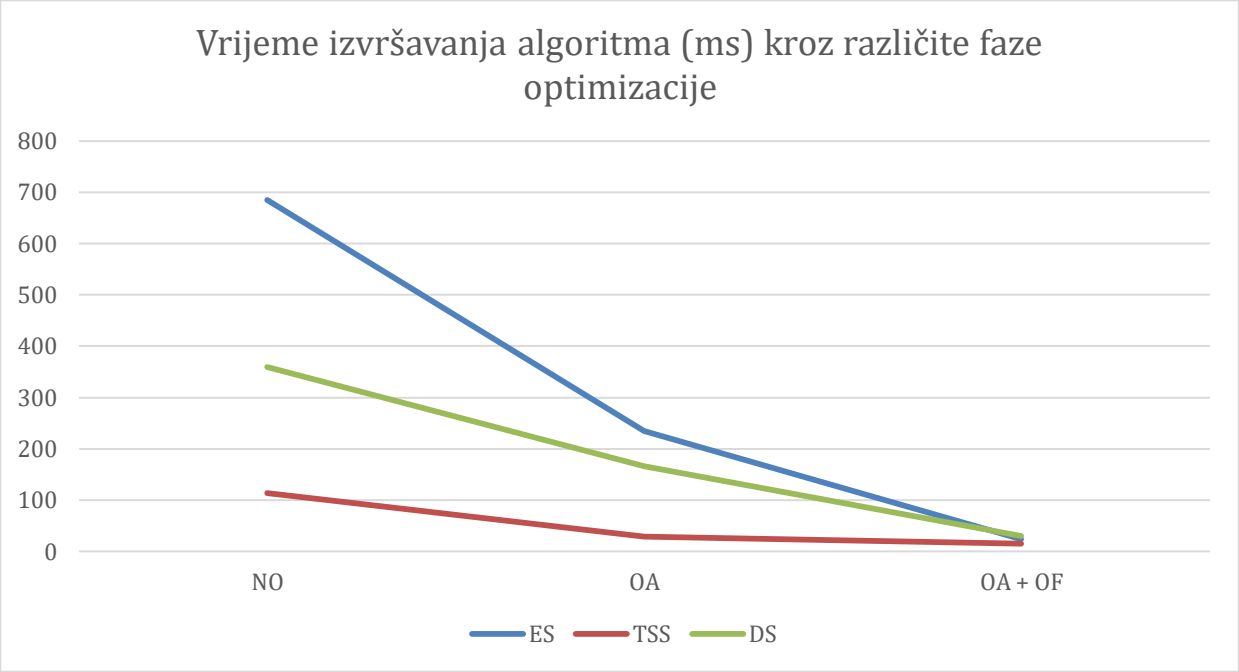


	ES	TSS	DS
MAX	61.59	23.17	61.83
AVG	24.02	15.61	29.79
MIN	13.17	12.12	13.93

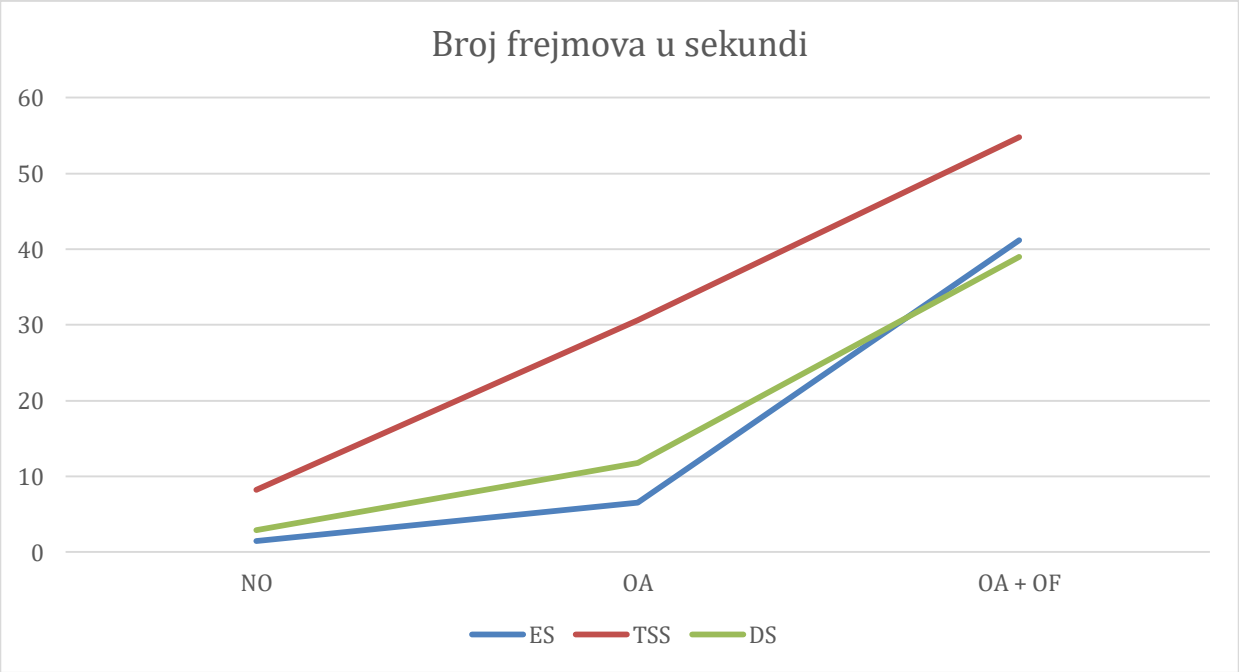
	ES	TSS	DS
MAX	49.66	55.61	51.13
AVG	41.19	54.77	38.99
MIN	25.57	52.56	12.35

Slika 7.3 – Performanse sistema uz optimizacije na nivou algoritma kombinovane sa optimizacijama na nivou frejmvorka

Nakon svih primijenjenih optimizacija performanse sistema postigle su zadovoljavajuće razultate na testovima, a ES metod koji se bez optimizacija izvršavao na nešto manje od dva frejma po sekundi sada postiže zavidne rezulate gdje broj frejmova u prosjeku iznosi 41,19 po sekundi. TSS implementacija pokazala se kao najkonzistentnija i u prosjeku najbolja, dok se DS implementacija pokazala kao nekonzistentna i čiji se broj frejmova po sekudni i nakon svih optimizacija spuštao na nešto više od 12 u najgorem slučaju. Tokom uvođenja optimizacija primjećeno je da je najbolji porast u performansama došao nakon uvođenja DMA kopiranja koje je vrijeme kopiranja svelo sa oko 3,5ms na vrijednost blizu 0,1ms. Na Slikama 7.4 i 7.5 data je vizuelizacija poboljšanja performansi svake od tri implementacije kroz različite faze procesa optimizacije.



Slika 7.4 – Vrijeme izvršavanja algoritma kroz različite faze optimizacije



Slika 7.5 – Broj frejmova u sekundi rada sistema kroz različite faze optimizacije

8. ZAKLJUČAK

Oblast računarskog vida svakodnevno podliježe inovativnim idejama, radovima i implementacijama o kojima je čovjek prije nešto manje od pedeset godina mogao samo razmišljati, teoretisati i pitati se kada će tako nešto biti moguće, ako ikad. Kontinualan razvoj i rast industrije hardvera i procesne moći računara uopšte otvorio je nove puteve razvoja i implementacije rješenja koja se danas smatraju ključnima u polju računarskog vida, bili to raznoliki metodi detekcije objekata, oblika, lica ili pomjeraja, ili čak metodi većeg nivoa komplesnosti kao što je ekstrakcija trodimenzionalnog modela objekata na osnovu fotografija istih. Sve prethodno navedeno naravno ne bi bilo moguće realizovati bez računarske moći koja je danas dostupna.

Cilj ovog rada jeste da se suprostavi gore navedenoj tvrdnji i pokaže da se uz detaljno planiranje i implementaciju popraćenu adekvatnom optimizacijom sistema mogu postići zavidni rezultati i performanse čak i na sistemima ograničene računarske snage. Da bi se ova tvrdnja pokazala kao tačna, na sistemu TMS320C66x procesorske arhitekture implementiran je sistem za detekciju pomjeraja objekta u videu baziran na metodi estimacije pokreta uparivanjem blokova. Prvobitna implementacija koja nije bila prilagođena radu na ciljanoj platformi pokazala se loše po pitanju performansi pa je naredni korak bio da se sistem optimizuje za evaluacionu platformu sa ciljem izvršavnja u visokom broju frejmova po sekundi, za video visoke rezolucije koji u sistem dolazi u realnom vremenu.

Sistem je prvobitno optimizovan na nivou algoritma gdje je primarni cilj predstavljao detekciju i odbacivanje redudantnih koraka i izračunavanja u algoritmu. Uvedena optimizacija praga poređenja blokova pokazala se kao adekvatna optimizacija koja je znatno unaprijedila performanse sistema po cijeni zanemarljivo manje preciznosti detekcije pomjeraja u slučaju kada su pomjeraji mali i čak i za čovjeka teško uočljivi. Takođe je smanjen broj memorijskih alokacija, a gdje god je to bilo moguće, petlje su zamijenjene ponavljanjem niza nezavisnih instrukcija koje se optimizovane izvršavaju u paraleli. Rezultati prve grupe optimizacija pokazali su se zadovoljavajućim i postali su dobra podloga za dalje optimizacije, na nivou razvojnog frejmvorka i arhitekture. Druga grupa optimizacija koje uključuju intrinzike, DMA kopiranje, kao i određen broj sugestija kompajleru da se određeni dio koda može optimizovati, podigla je performanse sistema na očekivan nivo i time je završen proces optimizacije.

Rad može biti koristan kao referenca za mikro i makro optimizacije sličnih sistema koji se razvijaju za sličnu ili istu procesorsku arhitekturu. Iako je faza optimizacije zaustavljena nakon što je sistem pokazao zadovoljavajuće rezultate, prostor za dodatni napredak sistema postoji u vidu mnogobrojnih optimizacija koje nisu obrađene ovim radom, a među kojima se kao najznačajnije ističu utilizacija višestrukih procesorskih jezgri (rad se bazirao na utilizaciji samo jednog DSP jezgra) i izvršavanje algoritma na videu smanjene rezolucije (*eng. downscaling*).

PRILOG

Prilog 1

```
#define BLOCK_SIMILARITY_TRESH 12


Int32 blockDidMoveGray(Int32 frame_w, Int32 frame_h, Int32 block_row,
Int32 block_col, UInt8* current_frame, UInt8* prev_frame) {
    Int32 k;
    Int32 temp_difference = 0;


    #pragma MUST_ITERATE(16,16)
    for (k = 0; k < MACRO_BLOCK_DIM; k++) {
        UInt8* restrict currentMacroBlockStart = &current_frame[(block_row *
        MACRO_BLOCK_DIM + k) * frame_w + (block_col)* MACRO_BLOCK_DIM];
        UInt8* restrict prevMacroBlockStart = &prev_frame[(block_row *
        MACRO_BLOCK_DIM + k) * frame_w + (block_col)* MACRO_BLOCK_DIM];


        temp_difference +=
        calculateSumOfAbsoluteDifferences(currentMacroBlockStart,
        prevMacroBlockStart);
    }


    temp_difference /= MACRO_BLOCK_DIM * MACRO_BLOCK_DIM;


    return temp_difference > BLOCK_SIMILARITY_TRESH;
}
```

Prilog 2

```
inline Int32 calculateSumOfAbsoluteDifferences(UInt8* restrict
currentMacroBlockStart, UInt8* restrict prevMacroBlockStart){

    _nassert((int)currentMacroBlockStart % 4 == 0);
    _nassert((int)prevMacroBlockStart % 4 == 0);

    UInt32 currentMacroBlockPart_1 = _amem4_const(currentMacroBlockStart);
    UInt32 currentMacroBlockPart_2 =
    _amem4_const(currentMacroBlockStart + 4);
    UInt32 currentMacroBlockPart_3 =
    _amem4_const(currentMacroBlockStart + 8);
    UInt32 currentMacroBlockPart_4 =
    _amem4_const(currentMacroBlockStart + 12);

    UInt32 prevMacroBlockPart_1 = _amem4_const(prevMacroBlockStart);
    UInt32 prevMacroBlockPart_2 = _amem4_const(prevMacroBlockStart + 4);
    UInt32 prevMacroBlockPart_3 = _amem4_const(prevMacroBlockStart + 8);
    UInt32 prevMacroBlockPart_4 = _amem4_const(prevMacroBlockStart + 12);
    UInt32 subabsPart_1 = _subabs4(currentMacroBlockPart_1,
    prevMacroBlockPart_1);
    UInt32 subabsPart_2 = _subabs4(currentMacroBlockPart_2,
    prevMacroBlockPart_2);
    UInt32 subabsPart_3 = _subabs4(currentMacroBlockPart_3,
    prevMacroBlockPart_3);
    UInt32 subabsPart_4 = _subabs4(currentMacroBlockPart_4,
    prevMacroBlockPart_4);

    UInt32 addPartH_12 = _add2(_unpkhu4(subabsPart_1),
    _unpkhu4(subabsPart_2));
    UInt32 addPartL_12 = _add2(_unpklu4(subabsPart_1),
    _unpklu4(subabsPart_2));
    UInt32 addPartH_34 = _add2(_unpkhu4(subabsPart_3),
    _unpkhu4(subabsPart_4));
    UInt32 addPartL_34 = _add2(_unpklu4(subabsPart_3),
    _unpklu4(subabsPart_4));

    UInt32 tempRes = _add2(_add2(addPartH_12, addPartL_12),
    _add2(addPartH_34, addPartL_34));

    UInt32 total = ((tempRes & 0xFFFF0000U) >> 16) + (tempRes & 0x0000FFFFU);
    return total;
}
```

Prilog 3

```
//setting the options bit flags for EDMA transfer configuration
opt  =
        (EDMA3_CCRL_OPT_ITCCHEN_ENABLE << EDMA3_CCRL_OPT_ITCCHEN_SHIFT)
           |
        ((pAlgHandle->tccId << EDMA3_CCRL_OPT_TCC_SHIFT)
                    & EDMA3_CCRL_OPT_TCC_MASK
         )
           |
        (EDMA3_CCRL_OPT_SYNCDIM_ABSYNC << EDMA3_CCRL_OPT_SYNCDIM_SHIFT)
           |
        (EDMA3_CCRL_OPT_TCINTEN_ENABLE
                    << EDMA3_CCRL_OPT_TCINTEN_SHIFT);

  //setting all the necessary parameters required for EDMA data transfer
  pAlgHandle->pParamSet->destAddr   = (UInt32)prevFrame;
  pAlgHandle->pParamSet->srcAddr    = (UInt32)currentFrame;
  pAlgHandle->pParamSet->srcBIdx    = inPitch[0];
  pAlgHandle->pParamSet->destBIdx   = inPitch[0];
  pAlgHandle->pParamSet->srcCIdx    = 0;
  pAlgHandle->pParamSet->destCIdx   = 0;
  pAlgHandle->pParamSet->aCnt       = lineSizeInBytes;
  pAlgHandle->pParamSet->bCnt       = height;
  pAlgHandle->pParamSet->cCnt       = 1;
  pAlgHandle->pParamSet->bCntReload = height;
  pAlgHandle->pParamSet->opt        = opt;
  pAlgHandle->pParamSet->linkAddr   = 0xFFFF;

  //checking the EMDA transfer code complete flag
  EDMA3_DRV_checkAndClearTcc(pAlgHandle->hEdma,
                             pAlgHandle->tccId,
                             &tccStatus);
  EDMA3_DRV_clearErrorBits (pAlgHandle->hEdma,pAlgHandle->edmaChId);

  //starting the EDMA data transfer
  EDMA3_DRV_enableTransfer (pAlgHandle->hEdma,pAlgHandle->edmaChId,
                                          EDMA3_DRV_TRIG_MODE_MANUAL);

  //waiting in the background for the transfer to end and clearing TCC
  EDMA3_DRV_waitAndClearTcc(pAlgHandle->hEdma,pAlgHandle->tccId);
```

Prilog 4

```
UInt32 start;
UInt32 end;
static UInt32 iterationCounter = 0;
static UInt32 totalTime = 0;

if(!pAlgHandle->motionDetectionObj.firstFrame){
      start = Utils_getCurTimeInUsec();
      calculateMotionVectorMatrix(width, height, currentFrame, prevFrame,
      mode, BLOCK_SIMILARITY_OPTIMIZATION_ON, motionVectorMatrix);
      end = Utils_getCurTimeInUsec();
      totalTime += end - start;
      if(++iterationCounter == 50){
          iterationCounter = 0;
          totalTime /= 50;
          Vps_printf("Average time in 50 iterations : %u\n", totalTime);
          totalTime = 0;
      }
}else pAlgHandle->motionDetectionObj.firstFrame = FALSE;
```

LITERATURA

[1] R. Szeliski, *Computer Vision: Algorithms and Applications*. Springer, New York, 2011.

[2] V. Risojević, *Multimedijalni sistemi*, Univerzitet u Banjoj Luci, Elektrotehnički fakultet, 2018.

[3] S. Alavi, *Comparation of Some Motion Detection Methods in cases of Single and Multiple Moving Objects*, National Institute of Technology Silchar, 2012.

[4] B. Pesquet-Popescu, M. Cagnazzo, F. Dufaux, *Motion Estimation Techniques*, TELECOM ParisTech, 2015.

[5] A. Barjatya, *Block Matching Algorithms for Motion Estimation*, DIP 6620 Spring 2004 Final Project Paper, 2004.

[6] Electronic Notes, "*Underestanding Embedded Systems*", https://www.electronics-notes.com/articles/digital-embedded-processing/embedded-systems/basics-primer.php, posjećeno: 10.10.2019.

[7] Texas Instruments, *TDA2PxACD CPU EVM Board User's Guide (Rev. A)*, SPRUIC0A, Novembar 2018.

[8] Analog, "*A Beginner's Guide to Digital Signal Processing (DSP)*", https://www.analog.com/en/design-center/landing-pages/001/beginners-guide-to-dsp.html, posjećeno: 14.10.2019.

[9] Texas Instruments, *TMS320C66x high-performance multicore DSPs for video surveillance*, 2012.

[10] Texas Instruments, *Vision SDK Links Framework – Deep Dive*, 16. Mart 2015.

[11] Texas Instruments, *Vision SDK (v03.xx) Development Guide*, 2017.

[12] Wikipedia, "*Loop unrolling*", https://en.wikipedia.org/wiki/Loop_unrolling, posjećeno: 21.10.2019.

[13] C. Robertson, "*Compiler intrinsics*", https://docs.microsoft.com/en-us/cpp/intrinsics/compiler-intrinsics?view=vs-2019, 9. Februar 2019, posjećeno 21.10.2019

[14] Texas Instruments, *TMS320C6000 Peripehrals Reference Guide*, Februar 2001.

[15] Texas Instruments, *TMS320C6000 Optimizing Compiler v8.2.x – User's Guide*, Maj 2017.