Ubrzavanje kodiranja videa na dron uređajima informacijama o kretanju drona

Vitaliani, Robert

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1631

UBRZAVANJE KODIRANJA VIDEA NA DRON UREĐAJIMA INFORMACIJAMA O KRETANJU DRONA

Robert Vitaliani

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1631

UBRZAVANJE KODIRANJA VIDEA NA DRON UREĐAJIMA INFORMACIJAMA O KRETANJU DRONA

Robert Vitaliani

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1631

Pristupnik:	**Robert Vitaliani (0036541861)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	izv. prof. dr. sc. Daniel Hofman

Zadatak: **Ubrzavanje kodiranja videa na dron uređajima informacijama o kretanju drona**

Opis zadatka:

Kodiranje videa na uređajima kod kojih je bitna energetska efikasnost zahtjeva pažljivo izvođenje algoritama i traženje mjesta na kojima je moguće smanjiti količinu vremena provedenu na procesiranje određenih dijelova algoritma. Jedan od najzahtjevnijih dijelova kod kodiranja videa je procjena pokreta. Potrebno je proučiti algoritme za kodiranje videa. Potražiti implementacije H.265 algoritma za kodiranje videa i izabrati jednu za optimizaciju. Predložene implementacije su Bolt65 i Kvazaar. Proučiti mogućnosti ubrzavanja dijela za procjenu pokreta korištenjem informacija o kretanju drona. Implementirati dio kodera u FPGA-u i istestirati rad kodera na testnim podatcima. Usporediti rad optimizirane verzije i originalne verzije kodera. Napraviti dokumentaciju i objaviti programski kôd na repozitoriju Git.

Rok za predaju rada: 14. lipnja 2024.

Zahvaljujem se mentoru na punoj podršci, inspiraciji te za stečeno znanje i vještine koje su mi obogatile akademsko iskustvo i pripremile me za buduće izazove.

Sadržaj

1. Uvod

U današnjem tehnološkom i digitalnom dobu, potražnja za visokokvalitetnim video sadržajem stalno raste, što donosi izazove u području video kodiranja. Zbog toga, optimizacija procesa kodiranja videa postaje od ključne važnosti, naročito kada se koristi na uređajima s ograničenim resursima, poput dronova. Energetska efikasnost postaje ključni aspekt u razvoju algoritama za kodiranje videa namijenjenih ovim uređajima, što zahtijeva pažljivo proučavanje i unapređenje postojećih algoritama.

Cilj ovog završnog rada je ubrzati dio za procjenu pokreta (motion estimation) korištenjem informacija o kretanju drona. Kako bi se postigao ovaj cilj, potrebno je poznavanje algoritama za kodiranje videa, pri čemu je odabrana implementacija algoritma H.265. High Efficiency Video Coding (HEVC), poznatiji kao H.265, predstavlja standard za kompresiju videa koji nudi poboljšanu video kvalitetu i algoritamsku efikasnost u usporedbi s njegovim prethodnikom H.264 (AVC). Od postojećih H.265 implementacija, odabrani su Bolt65 i Kvazaar.

Bolt65 je softversko/hardversko rješenje za HEVC koje se razvija na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu. Kvazaar je video koder za najnoviji standard visokoučinkovitog kodiranja videozapisa (HEVC/H.265) koji se razvija od 2012. godine, a razvoj koordinira Ultra Video Group. Oba ova algoritma nastoje poboljšati kodiranje videozapisa u stvarnom vremenu te energetsku efikasnost tijekom korištenja.

Proučavanjem ovih algoritama i posebno dijela za procjenu pokreta, potrebno je uvesti male modifikacije kako bi se funkcionalnost implementirala na digitalno sklopovlje poput FPGA (Field-Programmable Gate Array). Implementacijom algoritma na digitalnom sklopovlju iskoristio bi se paralelizam i brzina koju FPGA nudi. Za implementaciju algoritma na FPGA, potrebno je poznavanje alata za sintezu i analizu digitalnog sklopovlja

kao što su Vitis i Vivado.

Za verifikaciju i validaciju sinteze algoritma, odabrana je FPGA razvojna pločica PYNQ-Z1 koju je dizajnirao Xilinx. PYNQ-Z1 pločica omogućuje dostupniji i lakši razvoj FPGA, posebno za aplikacije strojnog učenja, obrade podataka te obrade signala u stvarnom vremenu. PYNQ-Z1 nudi testiranje i razvoj implementiranog algoritma preko Jupyter Notebooka, u kojem se koristi programski jezik Python. Korištenjem Jupyter Notebooka, moguće je lako provjeriti i testirati optimizaciju algoritma te njegovo ubrzanje na testnim podacima, uspoređujući rezultate i performanse optimirane verzije s originalnom verzijom kodera.

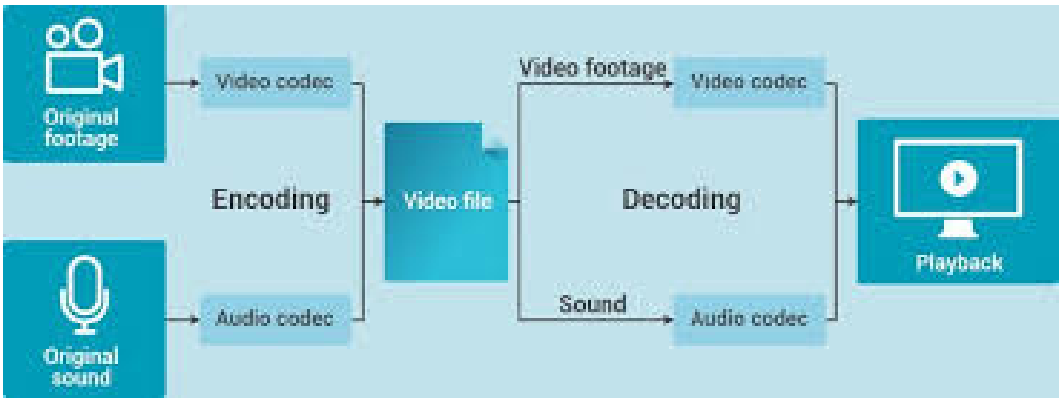
Završni rad se nadovezuje na prethodne radove u području video kodiranja i optimizacije, ali se ističe kroz primjenu specifičnog pristupa koji uključuje optimizaciju algoritama za procjenu pokreta uzimajući u obzir informacije o kretanju dron uređaja. Kroz literaturni pregled istaknut će se relevantni radovi koji su se bavili sličnim problemima, ističući jedinstvenost pristupa i doprinos ovom području istraživanja u ovom završnom radu.

2. Algoritmi za procjenu pokreta (motion estimation)

Ovaj završni rad bio je usmjeren na istraživanje algoritama za kodiranje videa s naglaskom na optimizaciju vremenske složenosti, posebno u dijelu procjene pokreta. Glavni algoritmi koji su proučavani uključuju H.265 (HEVC) algoritme za kodiranje videa kao što su implementacije Bolt65 i Kvazaar. Za razumijevanje ovih algoritama potrebno je poznavati rad samog video kodeka te postupaka kodiranja videa.

2.0.1. Video kodek

Video kodek je važna komponenta softvera i hardvera koja omogućuje kompresiju i dekompresiju digitalnog video sadržaja. Predstavlja ključnu ulogu u smanjenju veličine video datoteka uz očuvanje prihvatljive kvalitete, omogućujući učinkovito pohranjivanje, prijenos i tok video sadržaja. Video kodek se obično sastoji od dvije glavne komponente: kodera i dekodera. Koder komprimira video podatke, dok ih dekoder dekomprimira. Format komprimiranih podataka pridržava se standardnih formata video kodiranja, a proces kompresije je obično s gubitkom, što znači da se neke informacije iz izvornog videa gube. Posljedično, dekomprimirani video može imati lošiju kvalitetu u usporedbi s izvornim videom.



Slika 2.1. Kodiranje i dekodiranje videa [1]

Različiti čimbenici utječu na izvedbu i učinkovitost video kodeka, a neki od njih su: kvaliteta videa brzina prijenosa, složenost algoritama za kodiranja i dekodiranje, osjetljivost na gubitak podataka, nasumični pristup, kašnjenje...

Jedan značajan napredak u tehnologiji video kodeka je uvođenje H.265 (HEVC) standarda. H.265, razvijen 2013., nasljednik je H.264/MPEG-4 AVC i nudi poboljšanu učinkovitost kompresije. Postiže veće omjere kompresije u usporedbi s prethodnikom, što rezultira manjim veličinama datoteka bez značajnog gubitka kvalitete. H.265 posebno je koristan za video formate visoke razlučivosti kao što su 4K i 8K, kao i za platforme koje nude usluge za digitalnu distribuciju kao što su YouTube, Netflix i Vimeo. Također, osim H.264/AVC i H.265/HEVC algoritama za kodiranje videa, postoje i mnogi drugi algoritmi poput MPEG-2, VP9, AV1, MJPEG... U ovom završnom radu upravo je odabran algoritam H.265/HEVC, zato što koristi metode i algoritme za procjenu pokretu koje je potrebno optimizirati.

Slika 2.2. H.265 ili HEVC - High Efficiency video coding [2]

2.0.2. Implementacije H.265/HEVC algoritama

Proces kodiranja videa uključuje nekoliko faza od kojih su najvažnije: prostorna kompresija, vremenska kompresija te predikcija i kodiranje. Jedan od najzahtjevnijih dijelova kod kodiranja videa je procjena pokreta (motion estimation). Procjena pokreta pripada procesu vremenske kompresije. Za kodiranje videa su odabrane implementacije Bolt65 i Kvazaar, zato što koriste metode za procjenu pokreta koje je potrebno ubrzati. Stoga za početak recimo nešto o Bolt65 i Kvazaaru.

Bolt65 predstavlja integrirani softversko-hardverski sustav za HEVC kodiranje koji se razvija na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu. Glavni cilj Bolt65 je postizanje zahtjeva za izvršavanjem u stvarnom vremenu, što bi omogućilo kodiranje video sadržaja po potrebi. Bolt65 koristi sve raspoložive softverske i hardverske komponente sustava te se uspoređuje s referentnim HM HEVC softverom. Dobiveni rezultati pokazuju značajno ubrzanje u različitim konfiguracijama kodiranja uz uštedu na bitrateu u određenim situacijama, pri čemu se žrtvuje kvaliteta kako bi se zadovoljili zahtjevi stvarnog vremena.

Slika 2.3. Fakultet elektrotehnike i računarstva [3]

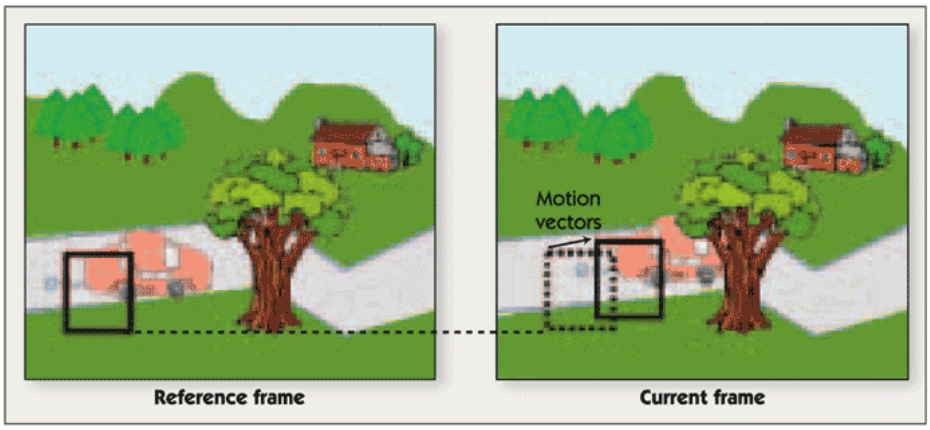
Kvazaar, otvoreni akademski video koder razvijen 2012. godine, predstavlja implementaciju HEVC standarda koju je razvio Ultra Video Group. Implementiran je u programskom jeziku C i optimiran koristeći SSE/AVX instrukcije. Kvazaar je dizajniran s ciljem postizanja visoke učinkovitosti kodiranja, jednostavne prenosivosti na različite platforme te podrške za kodiranje u stvarnom vremenu. Osim toga, Kvazaar uključuje sve potrebne alate za kodiranje glavnih profila HEVC standarda te omogućuje modularni pristup koji olakšava paralelizaciju na višejezgrenim procesorima te ubrzanje algoritama na hardveru.

Slika 2.4. Ultra Video Group - Kvazaar [4]

Budući da su Bolt65 i Kvazaar implementacije H.265/HEVC kodera, znači da koriste napredne algoritme za procjenu pokreta (motion estimation). Prije nego se prikaže sama implementacija algoritama za procjenu pokreta, potrebno je dati sažet i razumljiv opis ovog postupka u procesu video kodiranja.

Procjena pokreta (motion estimation)

Procjena pokreta u video kodiranju je tehnika koja se koristi za komprimiranje videozapisa tako da se samo promjene u pokretu između okvira zapisa zabilježe i kodiraju.



Slika 2.5. Procjena pokreta - pronalaženje vektora gibanja [5]

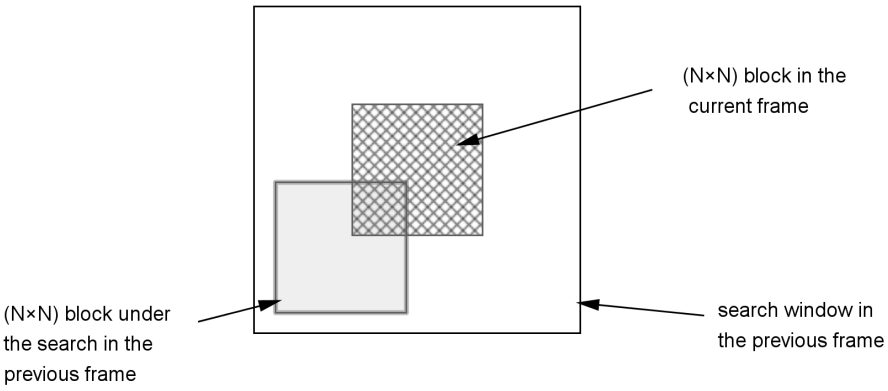
Proces započinje tako da se ulazna slika dijeli na male blokove piksela, obično veličine 8x8 ili 16x16 piksela. Zatim za svaki blok u trenutnom okviru (trenutnoj ulaznoj slici), sustav traži najbliži blok u referentnom okviru (prethodnoj ulaznoj slici) koji najbolje odgovara bloku u trenutnom okviru. Za svaki blok u trenutnom okviru, izračunava se razlika između bloka u trenutnom okviru i odgovarajućeg bloka u referentnom okviru. Nakon što se izračuna razlika, algoritam za procjenu pokreta koristi različite načine kako bi pronašao najbolje pomaknute blokove koji minimiziraju razliku. To se postiže pomicanjem blokova u različitim smjerovima i veličinama kako bi se pronašla najbolja podudarnost. Nakon što se odredi najbolji pomak za svaki blok (tj. kada se pronađu

najbolji vektori gibanja), ti pomaci se kodiraju i pohranjuju s informacijama o razlici između blokova. Na ovaj način se omogućuje da dekoder rekonstruira trenutni okvir s pomoću referentnog okvira i pomaka. Konačno, dekoder koristi informacije o pomaku i razlikama kako bi rekonstruirao originalni videozapis.

Ovaj proces omogućuje učinkovitu kompresiju videozapisa tako što se samo promjene u pokretu između okvira zabilježe i kodiraju, dok se statične dijelove slike ponavljaju ili predviđaju iz prethodnih okvira.

Algoritmi podudaranja blokova (Block-matching algorithms)

Unutar procesa procjene pokreta spomenuto je da nakon što se izračuna razlika, algoritam za procjenu pokreta koristi različite načine kako bi pronašao najbolje pomaknute blokove koji minimiziraju razliku. Ti različiti načini za pronalazak najboljih pomaknutih blokova koji minimiziraju razliku, upravo predstavljaju algoritme za podudaranje blokova.



Slika 2.6. Algoritam podudaranja blokova [6]

Algoritam podudaranja blokova (Block-Matching algoritam) koristi se za pronalaženje sličnih makroblokova u nizu video okvira kako bi se procijenilo kretanje. Osnovna ideja procjene pokreta je da se objekti i pozadina unutar okvira video sekvence pomiču na odgovarajuće pozicije u sljedećem okviru. Ova metoda pomaže otkriti vremensku redundanciju u video sekvenci, čime se povećava učinkovitost kompresije između okvira tako da se sadržaj makrobloka definira referencom na sličan makroblok.

Proces uključuje dijeljenje trenutnog okvira videa na makroblokove i usporedbu sva-

kog makrobloka s odgovarajućim blokom i susjednim blokovima u prethodnom okviru videa. Rezultat ove usporedbe je vektor koji opisuje kretanje makrobloka s jedne pozicije na drugu. Ovo kretanje, izračunato za sve makroblokove u okviru, predstavlja procijenjeno kretanje u okviru.

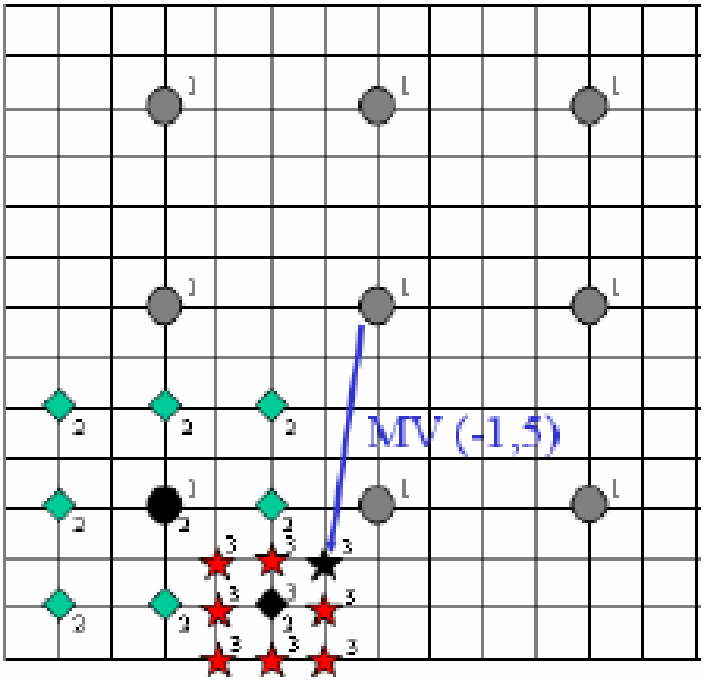
Područje pretraživanja za najbolju podudarnost makrobloka određuje se 'parametrom pretraživanja' (p), koji je broj piksela na sve četiri strane makrobloka u prethodnom okviru. Parametar pretraživanja mjeri količinu kretanja. Veća vrijednost p znači veće potencijalno kretanje i veću mogućnost pronalaženja dobrog podudaranja. Međutim, potpuna pretraga svih mogućih blokova je vrlo računalno zahtjevna. Tipični makroblokovi su veličine 16 piksela, a područje pretraživanja je često p = 7 piksela.

Neki od češće korištenih algoritama za podudaranje blokova su: Exhaustive Search, Optimized hierarchical block matching (OHBM), Three Step Search, Two Dimensional Logarithmic Search, New Three Step Search, Simple and Efficient Search, Four Step Search, Diamond Search te Adaptive Rood Pattern Search.

2.0.3. Bolt65

Bolt65 koristi više algoritama podudaranja blokova unutar algoritma za procjenu pokreta kao što su Three Step Search (Pretraga u tri koraka), Full Integer Area Search, Full Integer Search te Full Area Search. Za potrebe ovog rada odabran je algoritam Three Step Search (TSS), zato što je on najsličniji implementaciji algoritma Diamond Search koji se koristi i kod Kvazaarovog kodera.

Three Step Search (TSS) algoritam



Slika 2.7. Three Step Search - ilustracija rada algoritma [7]

Three Step Search je jedan od prvih brzih algoritama podudaranja blokova. Algoritam radi na sljedeći način:

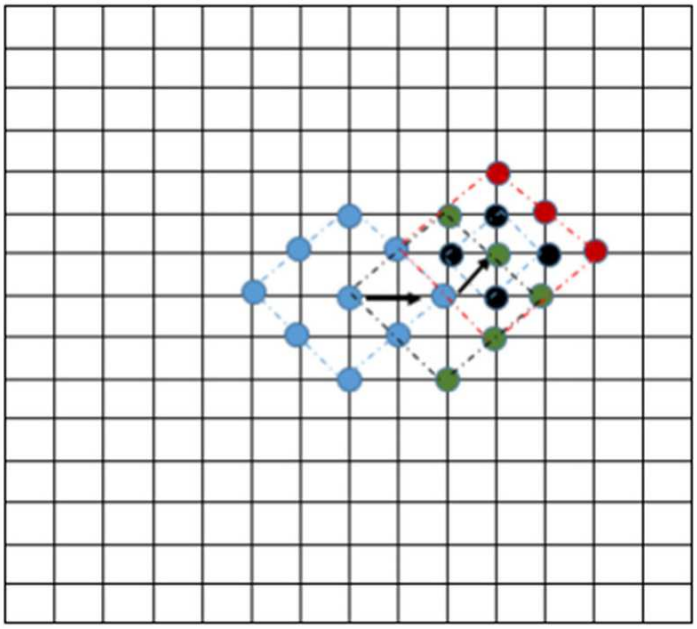
1. Počni s lokacijom pretraživanja u sredini.
2. Postavi korak S = 4 i parametar pretraživanja p = 7.
3. Pretraži 8 lokacija piksela +/- S oko lokacije (0,0) te samu lokaciju (0,0).
4. Odaberi među 9 pretraženih lokacija onu s minimalnom funkcijom troška.
5. Postavi novi izvor pretraživanja na prethodno odabranu lokaciju.
6. Postavi novu veličinu koraka na S = S/2.
7. Ponavljaj postupak pretraživanja sve dok S = 1.

Ciljna lokacija za S = 1 je ona s minimalnom funkcijom troška i makro blok na ovoj lokaciji predstavlja najbolje podudaranje. Implementacija algoritma Three Step Search u Bolt65 prati pseudokod ovog općenitog Three Step Search algoritma.

2.0.4. Kvazaar

Kvazaar koristi više algoritama podudaranja blokova unutar algoritma za procjenu pokreta kao što su Diamond Search (dijamantna pretraga), Hexagon Based Search, Test Zone Search te Full Search. Za potrebe ovog rada odabran je algoritam Diamond Search (DS), zato što je on najsličniji implementaciji algoritma Three Step Search koji se koristi i kod Bolt65 kodera.

Diamond Search (DS) algoritam



Slika 2.8. Diamond Search algoritam - ilustracija rada algoritma [8]

Diamond search (DS) algoritam ili algoritam dijamantne pretrage koristi uzorak dijamantnih točaka pretrage te ne postoji ograničenje broja koraka koje algoritam može poduzeti.

Za pretraživanje se koriste dvije različite vrste fiksnih uzoraka:

- LDSP (Large Diamond Search Pattern) - Veliki dijamantni uzorak pretraživanja
- SDSP (Small Diamond Search Pattern) - Mali dijamantni uzorak pretraživanja

Algoritam radi na sljedeći način:

LDSP:

1. Počni s lokacijom pretraživanja u sredini.

2. Postavi korak S = 2.

3. Pretraži 8 lokacija piksela (X, Y) tako da vrijedi (|X|+|Y|=S) oko lokacije (0,0) koristeći dijamantni uzorak točaka.

4. Odaberi među 9 pretraženih lokacija onu s minimalnom funkcijom troška.

5. Ako je minimalna težina pronađena u sredini prozora pretraživanja, prijeđi na korak SDSP.

6. Ako je minimalna težina pronađena na jednoj od 8 lokacija osim središnje, postavi novi izvor na tu lokaciju.

7. Ponovi LDSP.

SDSP:

1. Postavi novi izvor pretraživanja.

2. Postavi novu veličinu koraka na S = S/2 (tj. S = 1).

3. Ponovi postupak pretraživanja kako bi pronašla lokacija s najmanjom težinom.

4. Odaberi lokaciju s najmanjom težinom kao vektor gibanja.

Ovaj algoritam vrlo precizno pronalazi globalni minimum budući da uzorak pretraživanja nije ni prevelik ni premalen. Implementacija algoritma Diamond Search u Kvazaaru prati pseudokod ovog općenitog Diamond Search algoritma.

2.0.5. User Input Search algoritam

User Input Search (UIS) algoritam ili algoritam korisničkog unosa je proširenje Diamond Search algoritma u HEVC koderu Kvazaar čiji je cilj pojednostaviti procjenu kretanja uključivanjem korisničkog unosa kontrolera. Za razliku od Diamond Searcha, koji istražuje više blokova kandidata u uzorku pretraživanja u obliku dijamanta, User Input Search optimizira proces pretraživanja uzimajući u obzir samo jedan blok kandidata u smjeru kretanja drona. Na primjer, ako dron lebdi prema lijevo, onda bi imalo smisla

tražiti najsličnije blokove na lijevoj strani trenutnog bloka u prethodnom okviru. User Input Search rezultira s manje pretraživanja, što znači manje operacija u procesu kodiranja. Ova optimizacija održava kvalitetu videozapisa te povećava brzinu kodiranja (odnosno smanjuje latenciju).

Algoritam radi na sljedeći način:

1. Postavi lokaciju pretraživanja u središte dijamantnog uzorka i definiraj veličinu koraka pretraživanja.

2. Procijeni cijenu bloka kandidata u smjeru kretanja drona. Na primjer, ako se dron kreće prema gore, razmotri blok kandidata iznad središta dijamanta.

3. Izračunaj cijenu bloka kandidata s pomoću funkcije troškova kao što je zbroj apsolutnih razlika (SAD) ili srednja kvadratna pogreška (MSE).

4. Postavi novo središte dijamanta na blok kandidata uz najnižu cijenu.

5. Ponavljaj korake 2. - 4. dok se ne pronađe najbolje podudaranje u središtu dijamanta ili dok se ne ispuni kriterij zaustavljanja (ovaj kriterij može biti postizanje maksimalnog broja koraka pretraživanja).

Algoritam na kraju pohranjuje najbolji vektor gibanja pronađen tijekom procesa pretraživanja i vraća ga kao rezultat.

2.0.6. Funkcija za izračun minimalnog troška

Za izračun funkcije minimalnog troška Bolt65 i Kvazaar koriste sumu apsolutnih razlika ili na engleskom SAD (Sum of Absolute Differences) što predstavlja čestu metriku koja se koristi u obradi slika i videa za procjenu sličnosti između dva bloka piksela. SAD mjeri ukupne apsolutne razlike između odgovarajućih piksela trenutnog bloka i bloka kandidata. Formula za izračunavanje SAD-a između dva bloka A i B iste veličine (obično kvadratne) glasi:

$$\mathrm{SAD}(A, B) = \sum_{i=1}^{n} |A_i - B_i|$$

Gdje:

- A_i i B_i predstavljaju intenzitete odgovarajućih piksela u blokovima A i B respektivno.
- n je ukupan broj piksela u svakom bloku.

Jednostavnije rečeno, SAD računa apsolutnu razliku između svakog para odgovarajućih piksela trenutnog bloka i bloka kandidata te zbraja te apsolutne razlike kako bi dobio ukupnu vrijednost SAD-a. Dakle, SAD pruža mjeru različitosti između blokova. Što je niža vrijednost SAD-a, to su blokovi sličniji u pogledu intenziteta piksela.

Osim SAD-a, postoje i druge funkcije za izračun minimalnog troška koje se isto koriste u algoritmima za procjenu pokreta. Jedna metoda je MSE (Mean Squared Error), koja se računa kao prosječna kvadratna razlika između vrijednosti piksela u trenutnom bloku i bloku kandidatu. Formula za izračunavanje MSE-a između dva bloka A i B iste veličine glasi:

$$\text{MSE}(A, B) = \frac{1}{n} \sum_{i=1}^{n} (A_i - B_i)^2$$

Gdje:

- A_i i B_i predstavljaju intenzitete odgovarajućih piksela u blokovima A i B respektivno.
- n je ukupan broj piksela u svakom bloku.

Niža vrijednost MSE-a znači da su blokovi sličniji u pogledu intenziteta piksela.

3. Postupak implementacije algoritama na hardveru i testiranje

U ovom radu predlaže se optimizacija algoritama za procjenu pokreta u kontekstu video kodiranja na dronovima. Specifično, predlaže se iskorištavanje informacija o kretanju drona kako bi se ubrzao i učinkovitije izveo dio algoritma koji se bavi procjenom pokreta. Pretpostavka je da se dron kreće po određenoj putanji te da se njegova pozicija i orijentacija mogu kontinuirano pratiti s pomoću navigacijskog sustava ili nekog drugog senzora. Te informacije o kretanju drona mogu pružiti korisne smjernice algoritmu za procjenu pokreta, umanjujući potrebu za opsežnim pretraživanjem i izračunima.

Dakle, predložena metoda uključuje sljedeće korake:

1. Proučavanje implementacija algoritama za procjenu pokreta kod Bolt65 i Kvazaara. Proučene implementacije su Three Step Search, Diamond Search te User Input Search.

2. Razmatranje mogućnosti ubrzavanja dijela za procjenu pokreta korištenjem informacija o kretanju drona te sukladno tome modifikacija postojećih algoritama za procjenu pokreta.

3. Pokretanje izvornih programskih implementacija algoritama za procjenu pokreta, testiranje rada dijela kodera te mjerenje potrebnog vremena za izvršavanje algoritma na testnim podacima.

4. S pomoću alata za sintezu digitalnog sklopovlja (HLS), kao što su Vitis i Vivado, implementirati dio kodera u FPGA-u. Za implementaciju je odabrana pločica PYNQ-Z1.

5. Kroz sučelje Jupyter bilježnice istestirati rad dijela kodera te izmjeriti potrebno vri-

jeme za izvršavanje algoritma na testnim podacima.

6. Na kraju usporediti rad optimirane verzije i originalne verzije kodera.

Implementacija algoritama na hardveru, poput FPGA, omogućuje paralelizaciju i ubrzanje računski zahtjevnih dijelova algoritma. Korištenjem jezika za sintezu hardvera (HLS), dijelovi algoritma se prevode u hardverske komponente te se tako omogućuje programerima i dizajnerima sklopovlja da opišu algoritam na višoj razini apstrakcije koristeći jezike kao što su C, C++ ili SystemC, koji se zatim automatski sintetiziraju u digitalne sklopove. Ovi sklopovi su optimizirani za paralelnu obradu, omogućujući izvođenje više operacija istovremeno. To značajno smanjuje ukupno vrijeme potrebno za procjenu pokreta, čineći proces učinkovitijim i bržim.

U nastavku rada detaljno će se analizirati implementacija i evaluacija performansi predložene metode. Očekuje se da će rezultati pokazati značajna poboljšanja u brzini izvođenja i energetskoj učinkovitosti, što je ključno za primjenu video kodiranja na bespilotnim letjelicama.

4. Programska izvedba i vanjske biblioteke

U ovom poglavlju opisuju se implementacije algoritama za procjenu pokreta. Prvo će se pokazati njihova originalna implementacija u programskom jeziku C te objasniti princip njihova rada. Nakon toga objasnit će se postupak High-level synthesis (HLS) s pomoću programa Vitis i Vivado. Na kraju će se pokazati implementacija optimiziranih algoritama na pločici PYNQ-Z1 i pristup sučelju pločice preko Jupyter bilježnice.

4.0.1. Bolt65 - Three Step Search (TSS)

Jedan od algoritama za procjenu pokreta koju Bolt65 koristi u svojoj implementaciji je Three Step Search. Izvorni C++ kod je prebačen u C kod te su napravljene manje modifikacije Bolt65 Three Step Search algoritma kako bi bilo lakše sintetizirat sam kod. Korištene vanjske biblioteke za potrebe ovog programa su: `<stdio.h>`, `<stdlib.h>`, `<math.h>` te `<limits.h>`.

Funkcija za izračun minimalnog troška kod TSS-a - Suma apsolutnih razlika (SAD)

Bolt65 koristi funkciju za izračun minimalnog troška kako bi odredio najbolji vektor gibanja između dva bloka slike. Funkcija koristi Sumu apsolutnih razlika (SAD) kao mjeru troška, gdje manja vrijednost SAD-a znači bolju podudarnost između blokova piksela. Deklaracija funkcije za izračun SAD-a je sljedeća:

```
int SAD(unsigned char *block_1, unsigned char *block_2, int puWidth,
int puHeight);
```

Funkcija `SAD` je definirana kao `int` funkcija koja ima četiri parametra: `block_1` i `block_2`, koji predstavljaju blokove piksela koje uspoređujemo, te `puWidth` i `puHeight`,

koji predstavljaju širinu i visinu bloka.

Dakle, funkcija `SAD` računa minimalni trošak vektora gibanja koristeći sumu apsolutnih razlika između blokova piksela u trenutnom bloku i referentnom bloku. Trošak se koristi za odabir najboljeg vektora gibanja unutar TSS algoritma. Ako je izračunati trošak manji od dosad najmanjeg troška, ažurira se najbolji trošak i vektor gibanja, te funkcija vraća `true`. Inače, ako nije pronađen bolji vektor gibanja, funkcija vraća `false`.

Funkcija za procjenu pokreta - Three Step Search (TSS)

Three Step Search algoritam provodi pretragu u tri koraka pretražujući uzorak od 8 lokacija oko središta točke pretraživanja koji se kreće kroz okvir slike. Deklaracija funkcije za procjenu pokreta Three Step Search algoritma izgleda ovako:

```
void ThreeStepSearch(unsigned char *referenceFrame, unsigned char *block,
int puWidth, int puHeight, int startingIndex, int areaSize, int *deltaX,
int *deltaY, int width, int height, int *chosenBmcValue);
```

Funkcija Three Step Search definirana je kao `void` funkcija koja ima sljedeće parametre:

- `referenceFrame`: Referentni okvir slike.
- `block`: Trenutni blok slike.
- `puWidth` i `puHeight`: Širina i visina bloka.
- `startingIndex`: Početni indeks bloka unutar okvira slike.
- `areaSize`: Veličina područja pretrage.
- `deltaX` i `deltaY`: Komponente vektora pomaka koje se izračunavaju.
- `width` i `height`: Širina i visina okvira slike.
- `chosenBmcValue`: Najbolja vrijednost podudaranja blokova.

Funkcija Three Step Search započinje inicijalizacijom varijabli komponenata vektora pomaka (`deltaX` i `deltaY`) te najbolje vrijednosti podudaranja (`bestBmcValue`). Zatim

se područje pretrage dijeli na tri koraka kako bi se postupno smanjila veličina pretrage i fokusirala na najperspektivnije regije. Za svaki korak, funkcija uspoređuje trenutni blok s referentnim blokom na različitim položajima unutar okvira slike. Koristi funkciju SAD za izračunavanje vrijednosti minimalnog troška kako bi se odredio najbolji vektor gibanja između dva bloka slike. Nakon toga odabire položaj s najmanjom vrijednosti kao najbolji. Nakon svakog koraka, funkcija prilagođava veličinu koraka i početni indeks za sljedeći korak na temelju najboljeg podudaranja pronađenog u trenutnom koraku pretrage. Kada je pretraga završena, funkcija izračunava komponente vektora gibanja (`deltaX` i `deltaY`) na temelju razlike između početnog indeksa najboljeg podudaranja i originalnog početnog indeksa.

4.0.2. Kvazaar - Diamond Search (DS)

Jedan od algoritama za procjenu pokreta koju Kvazaar koristi u svojoj implementaciji je Diamond Search. Napravljene su manje modifikacije Kvazaarovog Diamond Search algoritma kako bi bilo lakše sintetizirat sam izvorni C kod. Korištene vanjske biblioteke za potrebe ovog programa su: `<stdio.h>`, `<stdbool.h>`, `<stdlib.h>`, `<stdint.h>` te `<limits.h>`.

Funkcija za izračun minimalnog troška kod DS-a - Sum of Absolute Differences (SAD)

Kvazaar koristi funkciju za izračun minimalnog troška kako bi odredio najbolji vektor gibanja između dva bloka slike. Funkcija koristi sumu apsolutnih razlika (SAD) kao mjeru troška, gdje manja vrijednost SAD-a znači bolju podudarnost između blokova piksela. Deklaracija funkcije za izračun minimalnog troška vektora gibanja izgleda ovako:

```
static bool check_mv_cost(int pic[64][64], int ref[64][64],
int offset_x, int offset_y, int width, int height, int x, int y,
int *best_cost, vector2d_t *best_mv);
```

Funkcija `check_mv_cost` definirana je kao `static bool` funkcija koja ima sljedeće parametre:

- `pic[64][64]`: Trenutni okvir slike.

- `ref[64][64]`: Referentni okvir slike.
- `offset_x` i `offset_y`: Pomaci za trenutni blok unutar okvira.
- `width` i `height`: Širina i visina bloka.
- `x` i `y`: Koordinate trenutnog vektora gibanja.
- `*best_cost`: Pokazivač na dosad najmanji trošak.
- `*best_mv`: Pokazivač na dosad najbolji vektor gibanja.

Dakle, funkcija `check_mv_cost` računa minimalni trošak gibanja koristeći sumu apsolutnih razlika između blokova piksela u trenutnom bloku i referentnom bloku. Ako je izračunati trošak manji od dosad najmanjeg troška, ažurira se najbolji trošak i vektor gibanja te funkcija vraća `true`. Inače, ako nije pronađen bolji vektor gibanja funkcija vraća `false`.

Funkcija za procjenu pokreta - Diamond Search (DS)

Diamond Search provodi pretragu za procjenu pokreta koristeći uzorak u obliku dijamanta. Tako se može efikasno pronaći najbolji vektor gibanja između blokova okvira. Deklaracija Diamond Search funkcije za procjenu pokreta izgleda ovako:

```
void diamond_search(int ref[64][64], int pic[64][64], int offset_x,
int offset_y, int width, int height, int steps, int *best_cost,
vector2d_t *best_mv);
```

Funkcija `diamond_search` definirana je kao `void` funkcija koja ima sljedeće parametre:

- `ref[64][64]`: Referentni okvir slike.
- `pic[64][64]`: Trenutni okvir slike.
- `offset_x` i `offset_y`: Pomaci za trenutni blok unutar okvira.
- `width` i `height`: Širina i visina bloka.
- `steps`: Broj koraka za pretragu.

- `*best_cost`: Pokazivač na dosad najmanji trošak.
- `*best_mv`: Pokazivač na dosad najbolji vektor gibanja.

Funkcija `diamond_search` započinje definiranjem dijamantnog uzorka i inicijalizacijom varijabli za minimalni trošak, trenutni vektor gibanja i najbolji indeks. Dijamantni uzorak sastoji se od pet točaka: središnje točke i četiri točke u kardinalnim smjerovima (desno, gore, lijevo, dolje). Nakon postavljanja početnih vrijednosti pomaka za trenutni blok unutar okvira slike, funkcija prolazi kroz svaki 8x8 blok slike. Za svaki blok, inicijalizira trenutni vektor gibanja dijeljenjem vrijednosti pomaka s 4 radi prilagodbe preciznosti pretrage te postavlja najbolji indeks (`best_index`) na središnju točku uzorka. Zatim prolazi kroz sve točke dijamantnog uzorka, pozivajući funkciju `check_mv_cost` za svaku točku kako bi se izračunao trošak za trenutnu poziciju i ažurirao najbolji trošak i vektor gibanja ako se pronađe bolje podudaranje.

Ako je najbolje podudaranje u središnjoj točki, funkcija ponovo postavlja najbolji vektor gibanja i trošak te prelazi na sljedeći blok. U suprotnom, središte pretrage premješta se na novu najbolju točku. Funkcija nastavlja pretraživati u smjeru nižih troškova dok ne pronađe bolje podudaranje ili dok ne iscrpi dopušteni broj koraka. U svakoj iteraciji provjerava četiri susjedne točke (isključujući onu iz koje je došla) te ažurira vektor gibanja i smjer pretrage prema novom najboljem podudaranju. Nakon završetka pretrage za trenutni blok, funkcija ponovo postavlja vektor gibanja, trošak i broj koraka na početne vrijednosti za sljedeći blok. Funkcija završava nakon što je procijenila sve blokove unutar okvira slike.

4.0.3. User Input Search (UIS)

User Input Search algoritam je nastao modifikacijom Kvazaarovog Diamond Search algoritma. Diamond Search nastoji pronaći najbolji vektor gibanja korištenjem dijamantnog uzorka kojim provjerava sve smjerove, dok User Input Search (iako radi na sličan način kao Diamond Search) pronalazi najbolji vektor gibanja korištenjem dijamantnog uzorka kojim provjerava samo smjer u kojem se dron pomaknuo. Napravljene su manje modifikacije User Input Search algoritma kako bi bilo lakše sintetizirat sam izvorni C kod. Korištene vanjske biblioteke za potrebe ovog programa su: `<stdio.h>`, `<stdbool.h>`, `<stdlib.h>`, `<stdint.h>` te `<limits.h>`.

Funkcija za izračun minimalnog troška kod UIS-a - Sum of Absolute Differences (SAD)

User Input Search algoritam za procjenu pokreta koristi funkciju za izračun minimalnog troška kako bi odredio najbolji vektor gibanja između dva bloka slike. Funkcija koristi sumu apsolutnih razlika (SAD) kao mjeru troška, gdje manja vrijednost SAD-a znači bolju podudarnost između blokova piksela. Deklaracija funkcije za izračun minimalnog troška vektora gibanja izgleda i radi identično kao funkcija za izračun minimalnog troška kod Diamond Search algoritma.

Funkcija za procjenu pokreta - User Input Search (UIS)

User Input Search nastoji usavršiti procjenu pokreta na temelju korisničkog unosa, omogućujući ciljanije pretraživanje prostora okvira. Deklaracija User Input Search funkcije za procjenu pokreta izgleda ovako:

```
void user_input_search(int ref[64][64], int pic[64][64], int offset_x,
int offset_y, int width, int height, int steps, int *best_cost,
vector2d_t *best_mv, int direction);
```

Funkcija `user_input_search` definirana je kao void funkcija koja ima sljedeće parametre:

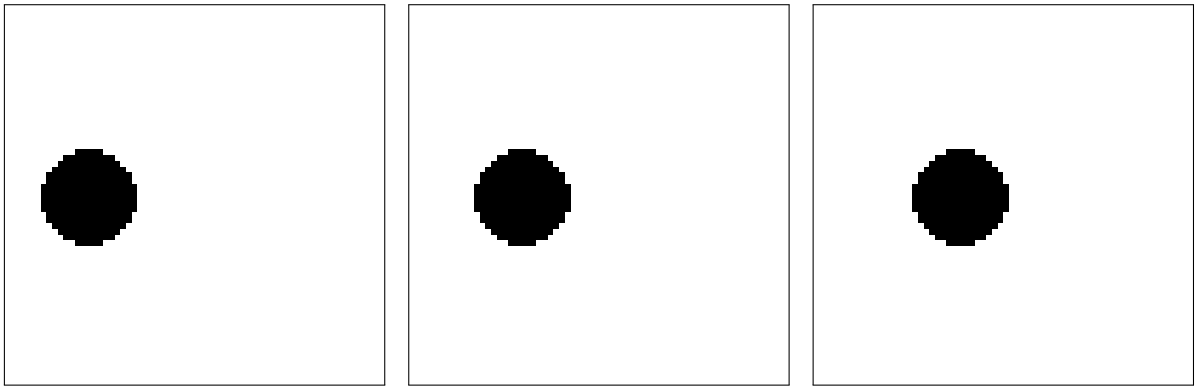
- `ref[64][64]`: Referentni okvir slike.
- `pic[64][64]`: Trenutni okvir slike.
- `offset_x` i `offset_y`: Pomaci za trenutni blok unutar okvira.
- `width` i `height`: Širina i visina bloka.
- `steps`: Broj koraka za pretragu.
- `*best_cost`: Pokazivač na dosad najmanji trošak.
- `*best_mv`: Pokazivač na dosad najbolji vektor gibanja.
- `direction`: Smjer korisničkog unosa.

Funkcija `user_input_search` na početku inicijalizira različite parametre poput matrice vjerojatnosti, koordinata bloka te faktora raspona pretrage. Ti parametri igraju važnu ulogu u određivanju strategije za procjenu pokreta. Na primjer, matrica vjerojatnosti može voditi pretragu prema područjima s većom vjerojatnošću registriranog pokreta. Nakon inicijalizacije, funkcija kreće u proces procjene pokreta putem dijamantnog uzorka pretrage. Ovaj uzorak omogućuje istraživanje potencijalnih vektora gibanja iterativno. Tijekom istraživanja, funkcija pronalazi najbolji vektor gibanja na temelju smjera koji pruža korisnički unos te se tako osigurava da je proces pretrage prilagođen specifičnim zahtjevima ili ograničenjima koje je postavio korisnik. Tako na primjer, ako se dron pomakne prema gore, Diamond Search bi izračunao trošak za sva četiri kandidata u svim smjerovima, dok bi User Input Search samo izračunao trošak za gornjeg kandidata. Također, funkcija dinamički prilagođava svoje parametre pretrage (poput raspona pretrage i koraka) kako bi kontinuirano ubrzavala proces pretrage.

Funkcija procjenjuje kvalitetu vektora gibanja usporedbom bloka u trenutnom okviru s odgovarajućim blokom u referentnom okviru. Ova procjena temelji se na funkciji troška, a funkcija iterativno ažurira vektor gibanja kako bi minimizirala ovaj trošak. Nakon završetka pretrage za određeni blok, funkcija vraća najbolji pronađeni vektor gibanja s pripadajućim troškom.

5. Testiranje brzine i performanse ME algoritama u C-u

U ovom poglavlju pokazat će se rezultati mjerenja svih triju algoritama implementiranih u programskom jeziku C. Mjerenje vremena izvršavanja izvedeno je u integriranom razvojnom okruženju Visual Studio Code koristeći C biblioteku `time.h`. Testiranja su provedena kroz 200 iteracija za svaki algoritam na uzorcima od 10 slikovnih okvira (frameova) koji su veličine 64x64 piksela. Uzorci predstavljaju crni krug na bijeloj pozadini koja se miče malo po malo s lijeve strane slikovnog okvira prema desnoj strani slikovnog okvira. Algoritam se mogao testirati i na većim slikovnim okvirima, no radi jednostavnosti i lakše implementacije algoritma na FPGA uzeti su dosta manji slikovni okviri, no što je to uobičajeno.

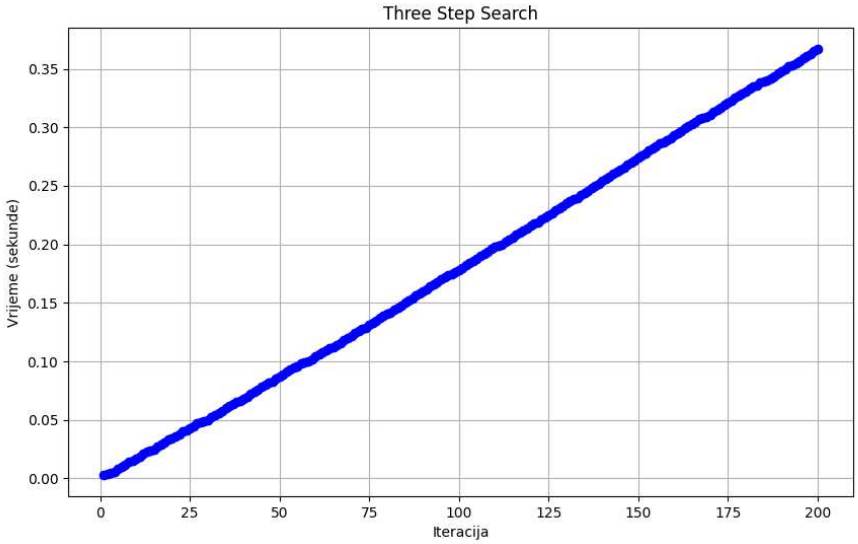


Slika 5.1. Primjer tri slikovna okvira s kojima su testirani ME algoritmi

Također (radi lakše vizualizacije), rezultati algoritama će biti prikazani u obliku grafova na kojima će se moći vidjeti dobivene izmjerene brzine te usporedba njihovih performansi. Graf je napravljen u programskom jeziku Python koristeći Matplotlib biblioteku. Svi testovi su provedeni na 64-bitnom Windows osobnom računalu s 32 GB RAM-a, AMD Ryzen 9 7900X 12-Core 4.70 GHz procesorom i AMD Radeon RX 7900 XT grafičkom karticom.

5.0.1. Eksperimentalni rezultati za Three Step Search

Bolt65 koristi algoritam Three Step Search za procjenu pokreta. Rezultati brzine i performanse ovog algoritma kroz 200 iteracija prikazani su na slici u nastavku:

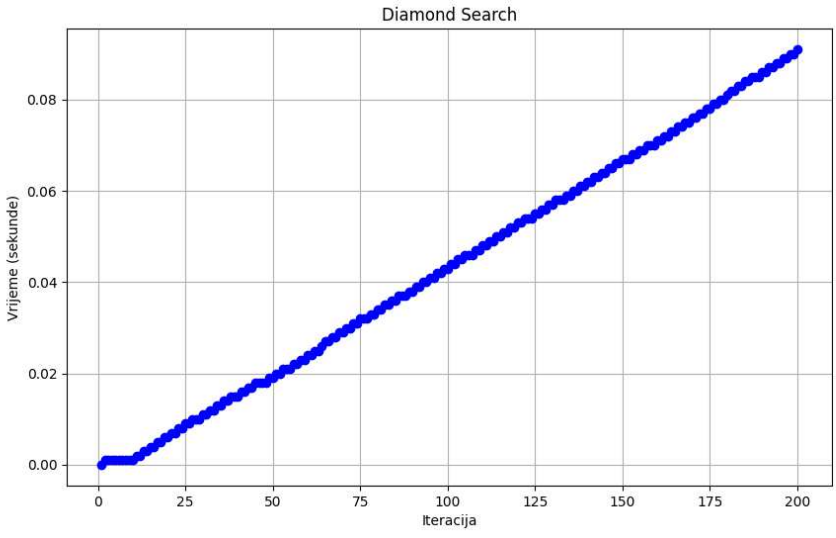


Slika 5.2. Three Step Search - mjerenje brzine i performanse u C-u

Vrijeme trajanja algoritma kroz 200 iteracija iznosi: 0.367000 sekundi.

5.0.2. Eksperimentalni rezultati za Diamond Search

Kvazaar koristi algoritam Diamond Search za procjenu pokreta. Rezultati brzine i performanse ovog algoritma kroz 200 iteracija prikazani su na slici u nastavku:

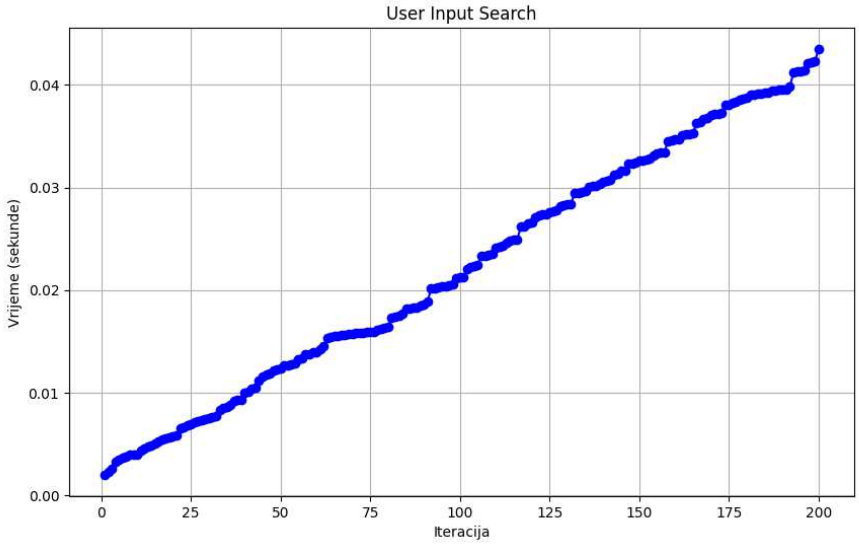


Slika 5.3. Diamond Search - mjerenje brzine i performans u C-u

Vrijeme trajanja algoritma kroz 200 iteracija iznosi: 0.091000 sekundi.

5.0.3. Eksperimentalni rezultati za User Input Search

User Input Search je algoritam za procjenu pokreta koji je napravljen na temelju Kvazaarovog Diamond Search algoritma. Rezultati brzine i performanse ovog algoritma kroz 200 iteracija prikazani su na slici u nastavku:

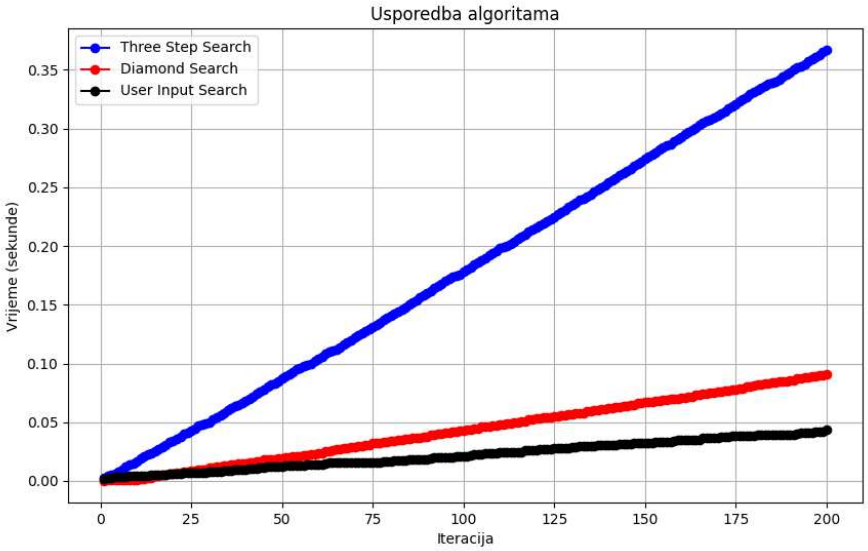


Slika 5.4. User Input Search - mjerenje brzine i performanse u C-u

Vrijeme trajanja algoritma kroz 200 iteracija iznosi: 0.043500 sekundi.

5.0.4. Usporedba svih triju algoritama

Sljedeći graf prikazuje usporedbu izmjerenih brzina i performansi svih triju algoritama kroz 200 iteracija u programskom jeziku C. Iz grafa je moguće vidjeti da je algoritam User Input Search najbrži. Iz ovoga se može zaključiti da je dio za procjenu pokreta ubrzan korištenjem informacija o kretanju drona. Ovime je potvrđeno da se postiže ubrzavanje kodiranja videa na dron uređajima informacijama o kretanju drona, što znači da je modifikacija Diamond Search funkcije (iz koje je nastala User Input Search funkcija) puno pogodnija za korištenje kod uređaja kod kojih energetska efikasnost predstavlja veliki značaj. Drugi algoritam po brzini je algoritam Diamond Search te na kraju dolazi najsporiji algoritam Three Step Search.



Slika 5.5. Usporedba svih triju algoritama u C-u

Nakon izmjerenih brzina i performansi izvorni programskih implementacija, u sljedećem poglavlju će se pokazati implementacija prikazanih algoritama na FPGA te mjerenje brzina i performansi u Jupyter bilježnici koristeći pločicu PYNQ-Z1.

6. Testiranje brzine i performanse ME algoritama na FPGA

U ovom poglavlju pokazat će se dobiveni eksperimentalni podaci svih triju algoritama kroz 200 iteracija na uzorcima od istih 10 slikovnih okvira koji su se koristili u prošlom poglavlju. Kako bi se ME algoritmi mogli testirati na FPGA, potrebno je bilo koristiti alate za sintezu digitalnog sklopovlja (HLS - High-Level Synthesis) od kojih su izabrani AMD-ovi alati Vitis i Vivado.

6.0.1. Vitis

Prvi korak je bio napraviti HLS sintezu C programskog koda sva tri ME algoritma u Vitisu te je zbog toga potrebno ponešto i reći o samom softverskom alatu.

Slika 6.1. Vitis - High-level synthesis [9]

Vitis je sveobuhvatna razvojna platforma koju je razvila tvrtka Xilinx. Dizajnirana je za ubrzavanje aplikacija koristeći Xilinxove FPGA (Field-Programmable Gate Array) i ACAP (Adaptive Compute Acceleration Platform) uređaje. Vitis omogućuje korisnicima da razvijaju, implementiraju i optimiziraju visokoučinkovite aplikacije koristeći jezike visokog nivoa kao što su C, C++ i Python, kao i koristeći RTL (Register Transfer Level) dizajn.

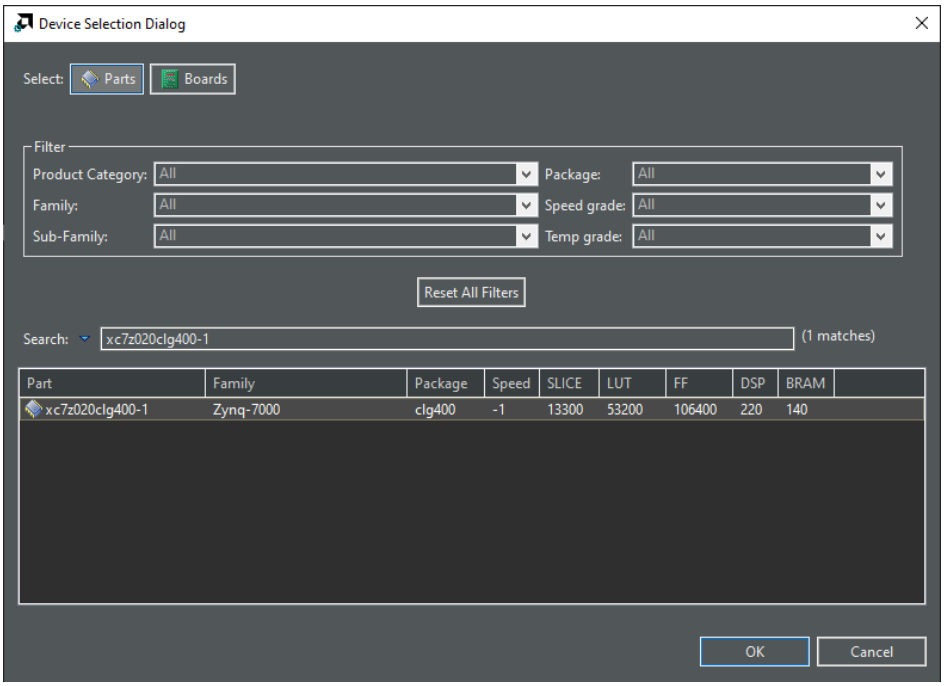
Vitis je softverski alat potreban za optimizaciju i ubrzanje C koda koji omogućuje

iskorištenje snage FPGA uređaja za povećanje performansi i efikasnosti aplikacija. Koristeći HLS sintezu, može se automatski prevesti C kod u optimizirani hardverski dizajn, što može značajno skratiti vrijeme razvoja i povećati kvalitetu konačnog rješenja.

Kako bi se započeo proces sinteze za sva tri ME algoritma potrebno je napraviti novi projekt u Vitisu. Unutar projekta potrebno je uključiti modificirane izvorne C kodove u kojima se ne nalazi main funkcija, nego samo funkcija (ili funkcije) koje se želi sintetizirati. Također, prije same sinteze, potrebno je dodati i određen broj direktiva (engl. pragma) u izvorni C kod.

Pragme predstavljaju direktive prevoditelja koje se koriste za pružanje dodatnih informacija HLS alatu. Koriste se za vođenje i optimizaciju procesa sinteze, omogućujući dizajnerima da kontroliraju različite aspekte implementacije hardvera izravno iz koda visoke razine. Dakle, pragme pomažu u poboljšanju performansi, korištenju resursa te ispunjavanju ograničenja dizajna. U ovom radu su korištene samo pragme sučelja (INTERFACE pragma) koje definiraju kako bi parametri funkcije trebali biti povezani kada se implementiraju na sklopovlju.

Za svaki ME algoritam potrebno je napraviti novi projekt u kojem se izabere funkcija ME algoritma kao top funkcija. Nakon toga potrebno je odabrati odgovarajući dio (engl. part) na kojem se planira implementirati RTL (Register Transfer Level) dizajn. RTL dizajn se koristi za dizajniranje integriranih krugova poput ASIC-a i FPGA-a. Sva tri ME algoritma su bila implementirana na PYNQ-Z1 pločici koja ima oznaku `xc7z020clg400-1` unutar Vitisa. Iz donje slike je vidljivo da pločica PYNQ-Z1 raspolaže s 53200 LUTova, 106400 bistabila (Flip-flop), 140 BRAMova te 220 DSP jedinica. SLICE pokazuje broj CLBova (Configurable Logic Block) na pločici, a svaki CLB se kod ove pločice sastoji od 4 LUT-a i 8 bistabila. LUT (Look-Up Table) predstavlja mali blok memorije koji implementira neku kombinacijsku funkciju, a BRAM (Block RAM) predstavlja memorijski blok koji se koristi za pohranu puno podataka. BRAM ima puno više prostora za pohranu od LUT-a koji koristi distribuirani RAM puno manjeg kapaciteta. Također, postoje i DSP jedinice koje se koriste za digitalnu obradu signala. Nakon odabira komponente potrebno je završiti proces izrade projekta.

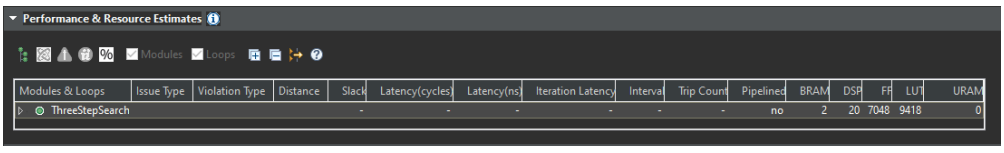


Slika 6.2. Odabir komponente u Vitisu

Ako je projekt ispravno postavljen, potrebno je pokrenuti proces sinteze C koda. Procesom sinteze se izvorni C kod (opis visoke razine) pretvara u HDL (Hardware Description Language) oblik poput VHDL-a ili Verilog-a (opis niske razine). Nakon procesa sinteze može se pogledati sažeti izvještaj u kojem se prikazuju procjene učinkovitosti i iskorištenih resursa sintetiziranog algoritma. Također se dobije header file u kojem su prikazane adrese te pripadni registri u koje će se upisivati pripadni parametri pojedinog algoritma. Dakle, popis registara sadrži popis kontrolnih i podatkovnih signala koji se mogu koristiti kako bi se podaci upisivali u ME funkciju te kako bi se i na kraju krajeva pročitali iz registara i rezultati te funkcije. U nastavku će biti prikazana tablica procjene učinkovitosti i iskorištenih resursa sintetiziranog algoritma za sva tri algoritma.

Three Step Search - izvještaj sinteze

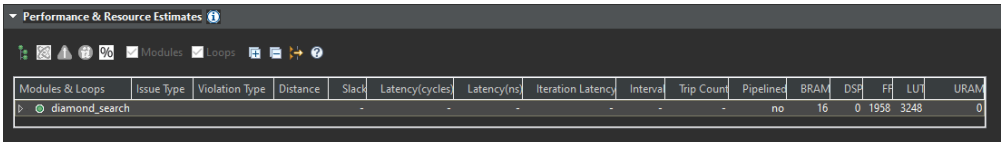
Za algoritam Three Step Search iskorišteno je 2 BRAM-a, 20 DSPova, 7048 bistabila i 9418 LUTova.



Slika 6.3. Three Step Search - procjena performanse i resursa

Diamond Search - izvještaj sinteze

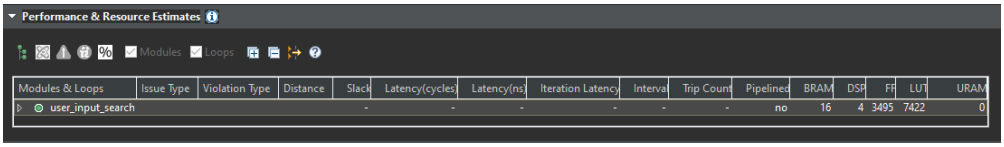
Za algoritam Diamond Search iskorišteno je 16 BRAM-a, 0 DSPova, 1958 bistabila i 3248 LUTova.



Slika 6.4. Diamond Search - procjena performanse i resursa

User Input Search - izvještaj sinteze

Za algoritam User Input Search iskorišteno je 16 BRAM-a, 4 DSPova, 3495 bistabila i 7422 LUTova.



Slika 6.5. User Input Search - procjena performanse i resursa

Nakon C sinteze izvodi se implementacija. Potrebno je izvesti RTL (engl. Export RTL) dizajn u obliku Vivado IP-a (engl. Intellectual Property). Na ovaj način se generira IP jezgra iz RTL dizajna koja će se koristiti u Vivado projektima. Time se omogućuje integracija generiranog RTL dizajna s drugim IP-ovima na modularan način koji se može ponovno koristiti.

6.0.2. Vivado

Drugi korak je bio napraviti implementaciju IP-a u Vivadu. Kao i za Vitis slijedi kratki opis programskog alata.

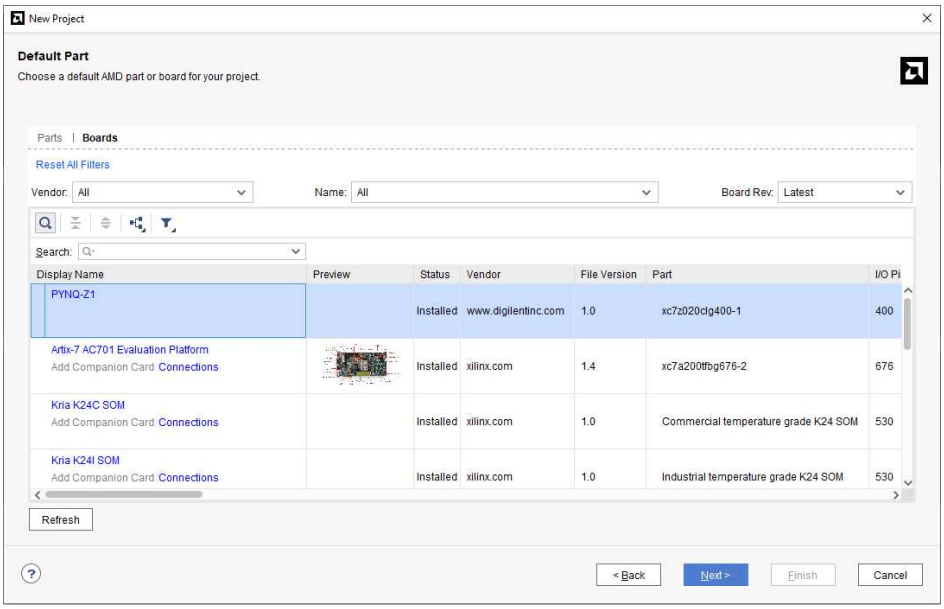
Slika 6.6. Vivado - High-level synthesis [10]

Vivado je sveobuhvatna razvojna platforma koju je razvila tvrtka Xilinx. Dizajnirana je za razvoj i verifikaciju digitalnih sustava koristeći Xilinxove FPGA (Field-Programmable

Gate Array) uređaje. Vivado omogućuje korisnicima da razvijaju, implementiraju i optimiziraju visokoučinkovite digitalne sustave koristeći jezike visokog nivoa kao što su VHDL (VHSIC Hardware Description Language) i Verilog, kao i koristeći HLS (High-Level Synthesis) za prevođenje C, C++ i SystemC koda u RTL (Register Transfer Level) dizajn.

Vivado je potreban kako bi se optimizirao i ubrzao digitalni dizajn, omogućujući iskorištenje snage FPGA uređaja za povećanje performansi i efikasnosti sustava. Koristeći HLS sintezu, može se automatski prevesti C kod u optimizirani hardverski dizajn, što može značajno skratiti vrijeme razvoja i povećati kvalitetu konačnog rješenja. Vivado također pruža alate za analizu performansi, verifikaciju dizajna i generiranje konačnih konfiguracijskih datoteka za FPGA uređaje, čime omogućuje sveobuhvatan razvojni tijek za digitalne sustave.

Kako bi se započeo proces implementacije IP jezgre koja je dobivena u Vitisu potrebno je napraviti novi projekt u Vivadu. Pri odabiru tipa projekta potrebno je odabrati RTL Project. Pri odabiru pločice potrebno je odabrati pločicu PYNQ-Z1. Pri odabiru pločice moguće je vidjeti dodatne specifikacije i komponente od kojih je pločica građena. Nakon odabira pločice potrebno je završiti proces izrade projekta.



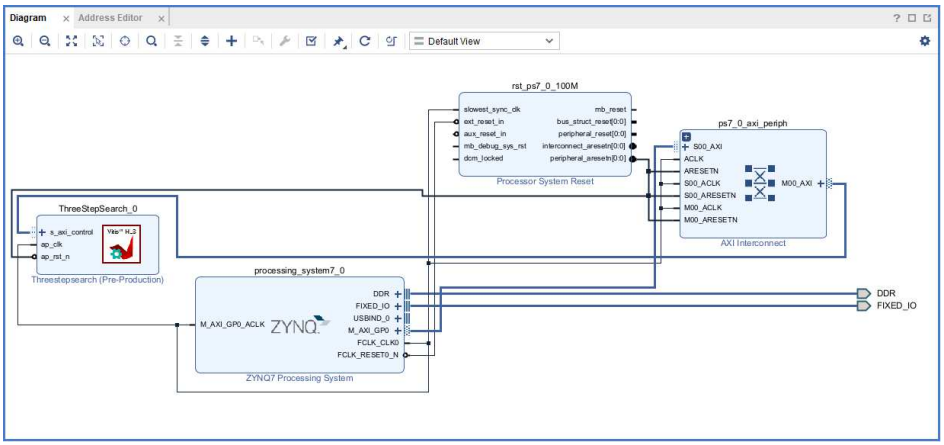
Slika 6.7. Odabir komponente u Vivadu

Ako je projekt ispravno postavljen, potrebno je započeti proces implementacije IP jez-

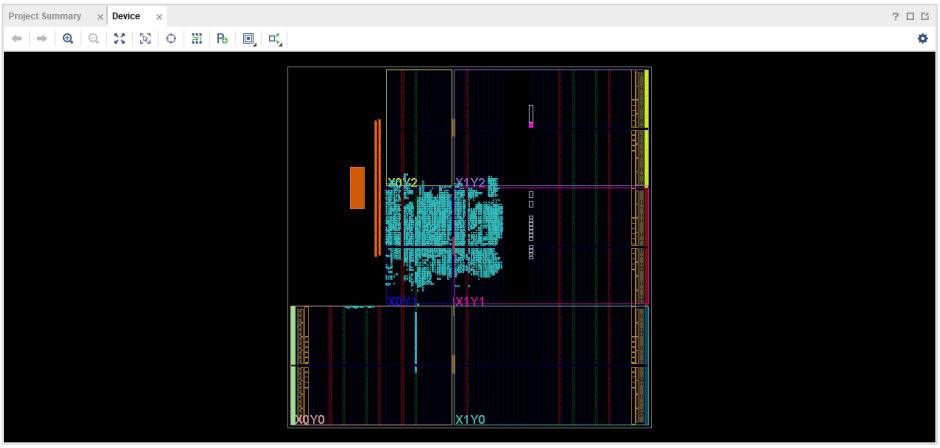
gre u FPGA dizajn. Prvo je potrebno napraviti blok dizajn (engl. Block Design) unutar izbornika IP INTEGRATOR. Zatim je unutar dijagrama blok dizajna potrebno dodati IP - ZYNQ Processing System. ZYNQ Processing System omogućuje konfiguraciju i integraciju ARM Cortex-A9 sustava s FPGA logikom. Potrebno je odabrati opciju Run Block Automation kako bi se automatski postavio ZYNQ Processing System. Zatim je potrebno unutar izbornika Tools odabrati Settings te u IP repozitorij (engl. IP Repository) projekta dodati IP koji je napravljen u Vitisu. Dodani IP potrebno je dodati u blok dizajn na isti način kako se bio dodao ZYNQ Processing System. Nakon dodavanje potrebno je odabrati opciju Run Connection Automation koja će povezati dodani IP sa ZYNQ Processing System IP-om. Nakon povezivanja potrebno je napraviti HDL omotač (engl. Create HDL Wrapper) za napravljeni blok dizajn. Zatim je potrebno unutar izbornika PROGRAM AND DEBUG odabrati Generate Bitstream. Kada se završi generiranje bitstreama može se pogledati izgled implementiranog dizajna na samoj pločici PYNQ-Z1. Na kraju je još potrebno unutar izbornika File odabrati opciju Export te unutar nje Export Block Design. Ovim će se izvesti tcl datoteka koju će biti potrebno prebaciti na pločicu, zato što će nju čitati PYNQ biblioteke te tako interpretirati blok dizajn koji je izveden. Osim tcl datoteke potrebno je pripremiti i bitstream generiranu wrapper bit datoteku i hwh datoteku te ih preimenovati da sve tri datoteke imaju isto ime. U nastavku su prikazani dobiveni blok dizajnovi te izgled implementiranih dizajnova na samoj pločici za sva tri algoritma posebno.

Three Step Search - Blok dijagram te prikaz implementacije na hardveru

Dobiveni dijagram blok dizajna te prikaz njegove implementacije na hardveru za ME algoritam Three Step Search.



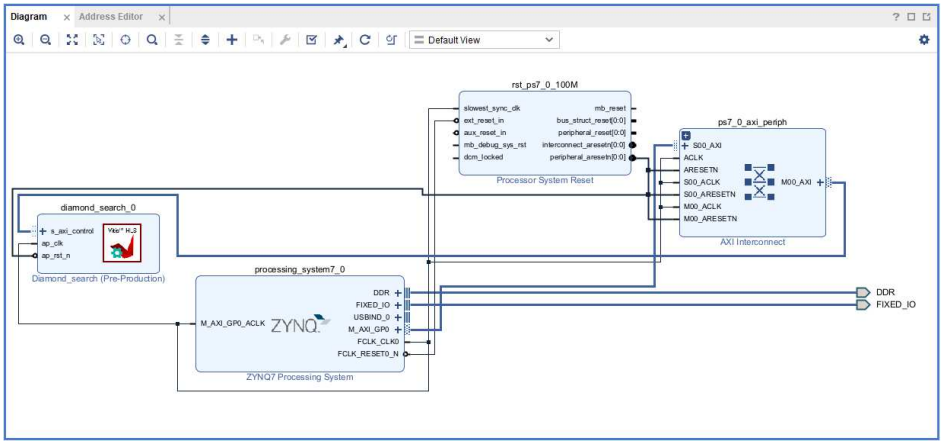
Slika 6.8. Dijagram blok dizajna za Three Step Search



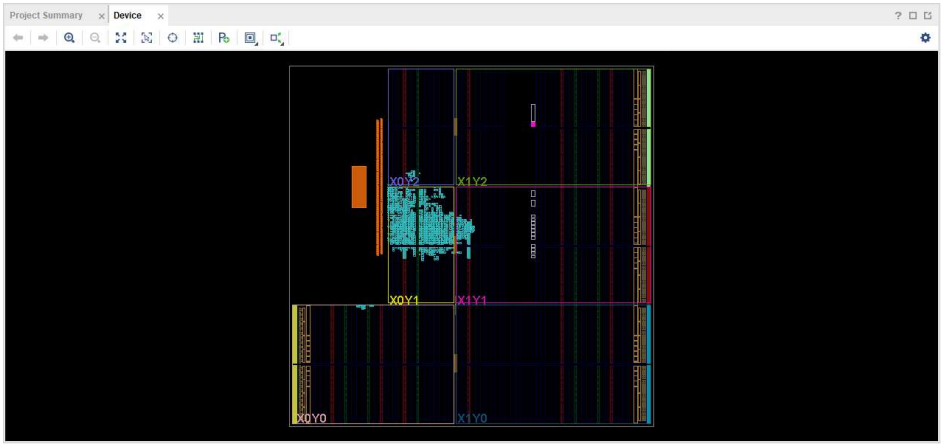
Slika 6.9. Implementacija dizajna za Three Step Search

Diamond Search - Blok dijagram te prikaz implementacije na hardveru

Dobiveni dijagram blok dizajna te prikaz njegove implementacije na hardveru za ME algoritam Diamond Search.



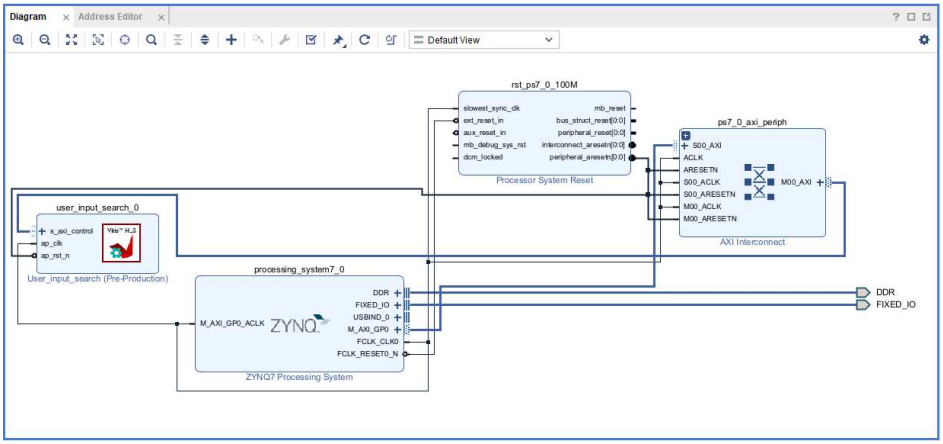
Slika 6.10. Dijagram blok dizajna za Diamond Search



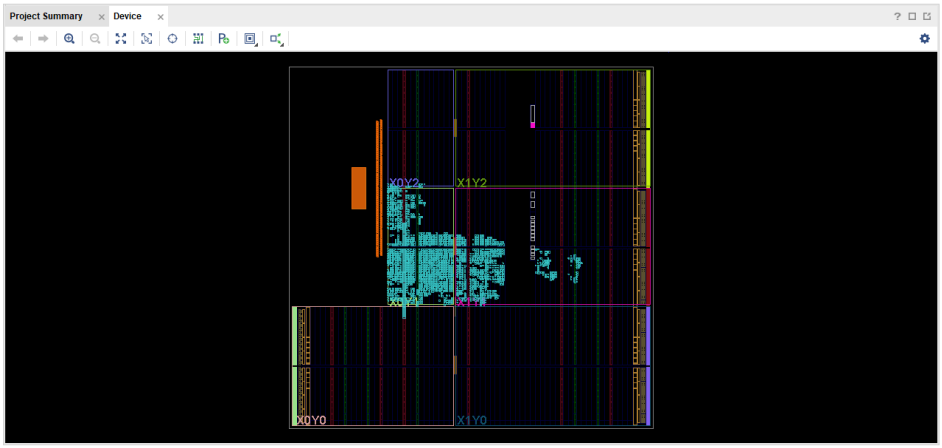
Slika 6.11. Implementacija dizajna za Diamond Search

User Input Search - Blok dijagram te prikaz implementacije na hardveru

Dobiveni dijagram blok dizajna te prikaz njegove implementacije na hardveru za ME algoritam User Input Search.



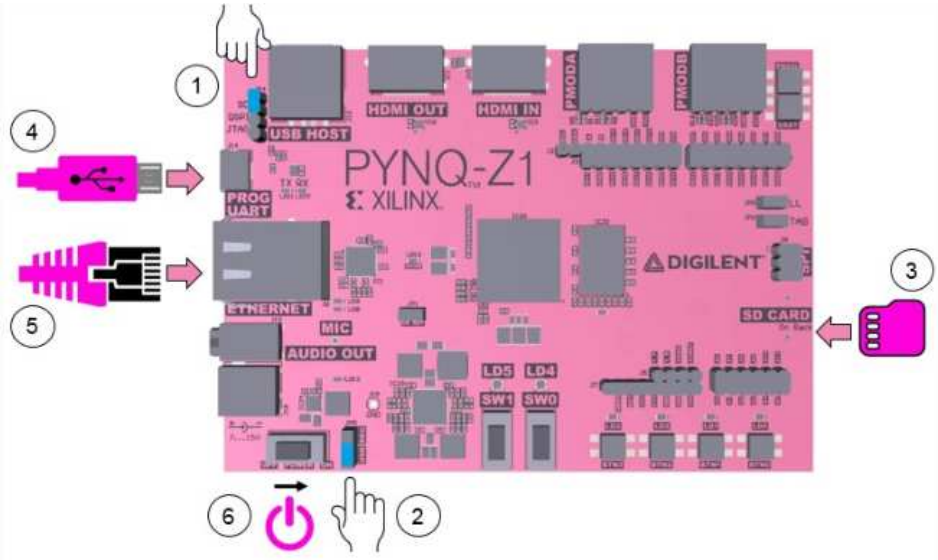
Slika 6.12. Dijagram blok dizajna za User Input Search



Slika 6.13. Implementacija dizajna za User Input Search

6.0.3. Pločica PYNQ-Z1

PYNQ-Z1 pločica je razvojna platforma temeljena na Xilinx Zynq-7000 SoC (System on Chip) koja je dizajnirana za korištenje s radnim okvirom PYNQ (Python Productivity for Zynq). Pločica kombinira dvojezgreni procesorski sustav ARM Cortex-A9 s Xilinx FPGA što je čini prikladnom za različite primjene, uključujući ugrađene sustave, digitalnu obradu signala, IoT (Internet of Things) i strojno učenje. Raznolike mogućnosti povezivanja poput Etherneta, USB, HDMI i raznih drugih priključaka za proširenje, čine ga svestranim za povezivanje s drugim uređajima i sustavima.



Slika 6.14. Podešavanje PYNQ-Z1 pločice

Prije nego što se krene raditi s pločicom potrebno ju je podesiti kao u nastavku:

1. Postaviti kratkospojnik JP4 / Boot u SD položaj postavljanjem kratkospojnika preko gornja dva pina JP4 kao što je prikazano na slici. (Ovo postavlja pločicu za pokretanje s Micro-SD kartice.)

2. Za napajanje PYNQ-Z1 s mikro USB kabela, postaviti kratkospojnik JP5 / Power u USB položaj. (Također se može napajati pločica iz vanjskog regulatora snage od 12 V postavljanjem kratkospojnika na REG.)

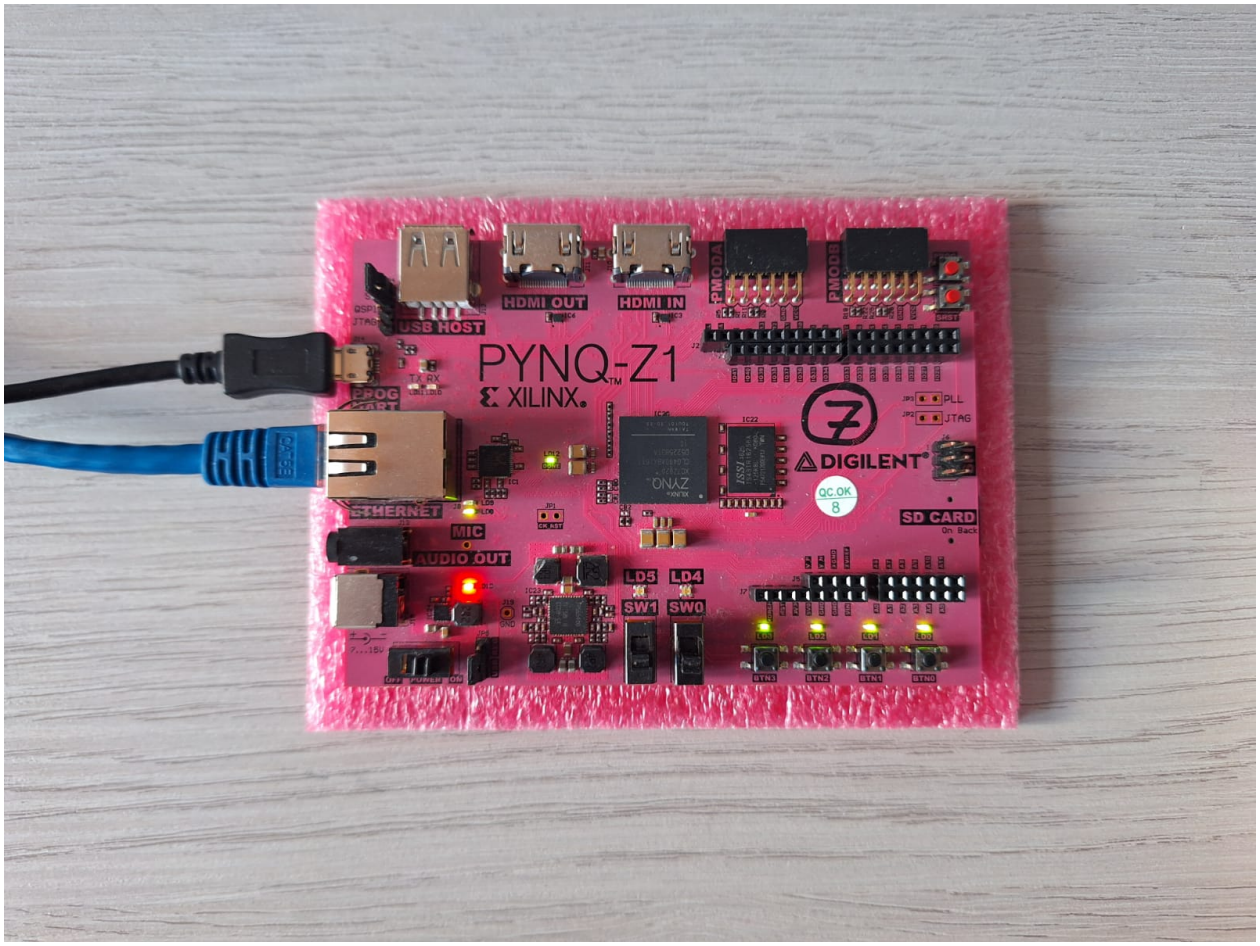
3. Umetnuti Micro SD karticu s unaprijed učitanim PYNQ-Z1 image-om u utor za Micro SD karticu ispod pločice.

4. Spojiti USB kabel na svoj PC/laptop i na PROG - UART / J14 MicroUSB priključak

na pločici kako bi se pločica mogla napajati.

5. Spojiti PC/laptop i pločicu s Ethernet kabelom kako bi se dobio pristup pločici preko mreže.

6. Uključiti PYNQ-Z1 i pričekati da se pokrene sustav. (Treba pričekati dok se ne upale 4 LED diode koje se nalaze u donjem desnom dijelu pločice.)



Slika 6.15. Izgled ispravno podešene pločice

Nakon što je pločica pokrenuta potrebno joj je pristupiti putem mreže preko PC/laptopa na koji je pločica spojena. Na kraju je još potrebno dodati tri datoteke s nastavcima tcl, bit i hwh (koje su dobivene u Vivadu) u mapu overlays za svaki od tri algoritama posebno. Nakon ovoga je moguće pristupiti Jupyter bilježnicama preko internet preglednika.

Potrebno je u internet preglednik upisati pynq:9090 te lozinku xilinx u kojoj će se otvoriti mapa gdje se mogu stvarati nove Jupyter bilježnice. Za svaki algoritam je potrebno napraviti novu Jupyter bilježnicu te učitati overlay s pomoću bit datoteke. Na ovaj način će Jupyter (Python) interpretirati napravljeni blok dizajn te naći IP i asocirati

driver (upravljački program) s njim. Taj driver će omotati MMIO uređaj na pločici kako bi se mogli upisivati i čitati podaci s IP-a. U nastavku će se prikazati učitavanje overlaya, nekih testnih podataka te izmjerena vremena izvršavanja za svaki algoritam posebno. Na kraju će se prikazati usporedba brzine izvršavanja svih triju algoritama.

6.0.4. Eksperimentalni rezultati za Three Step Search

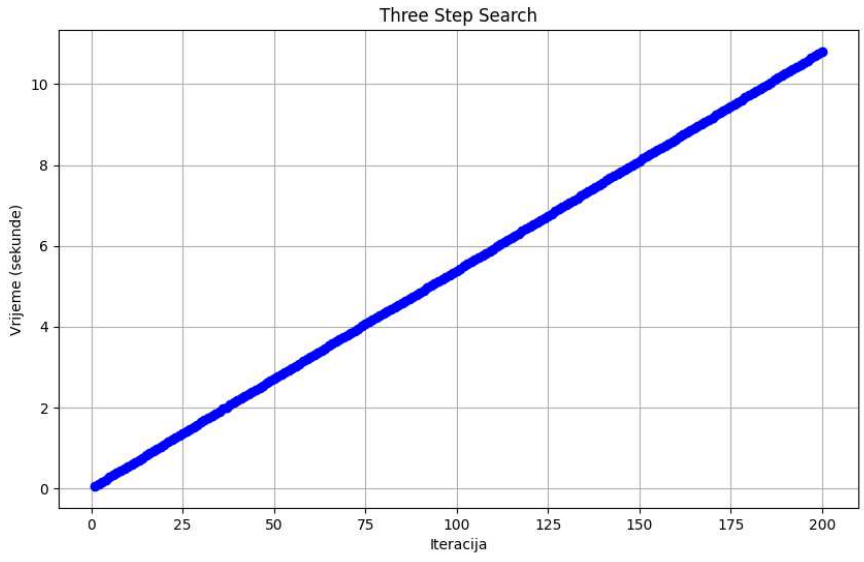
```
In [14]: from pynq import Overlay
         from pynq import MMIO
         overlay = Overlay("/home/xilinx/pynq/overlays/TSS/TSS.bit")

In [3]:  overlay?

In [15]: diam1=overlay.ThreeStepSearch_0
         diam1?
```

Slika 6.16. Učitavanje overlaya za Three Step Search

```
diam1.write(0x10,referenceFrame[0][0])
diam1.write(0x20,8) #pu_width
diam1.write(0x28,8) #pu_height
diam1.write(0x38,4) #areaSize
diam1.write(0x60,64) #height
diam1.write(0x68,64) #width
diam1.write(0x40,0)
diam1.write(0x50,0)
```

Slika 6.17. Primjer upisivanja vrijednosti u registre (parametre) funkcije

Rezultati brzine i performanse Three Step Search algoritma kroz 200 iteracija prikazani su na slici u nastavku:



Slika 6.18. Three Step Search - mjerenje brzine i performanse u Jupyteru

Vrijeme trajanja algoritma kroz 200 iteracija iznosi: 10.805764 sekundi.

6.0.5. Eksperimentalni rezultati za Diamond Search

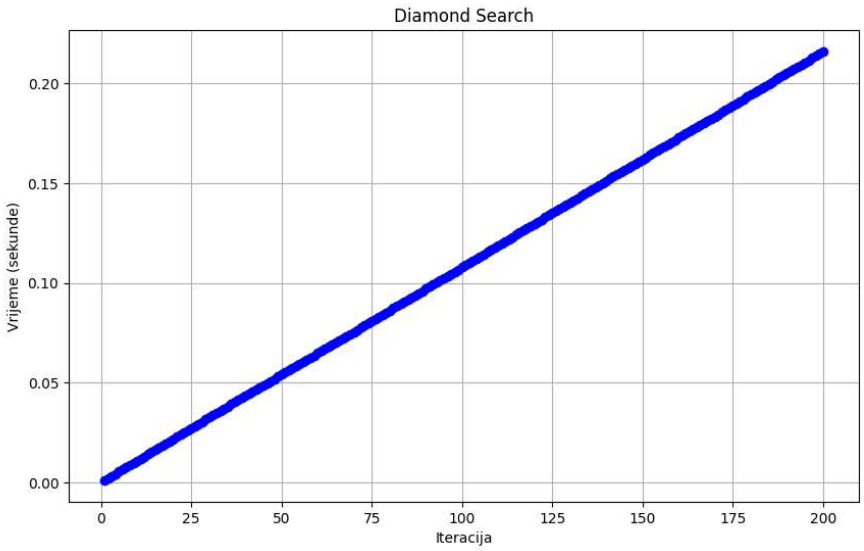
```
In [1]: from pynq import Overlay
        from pynq import MMIO
        overlay = Overlay("/home/xilinx/pynq/overlays/DiamondSearch/design_16320.bit")

In [2]: overlay?

In [2]: diam1=overlay.diamond_search_0
        diam1?
```

Slika 6.19. Učitavanje overlaya za Diamond Search

```
diam1.write(0x0010,0)
diam1.write(0x0018,0)
diam1.write(0x0020,8)
diam1.write(0x0028,8)
diam1.write(0x0030,7)
diam1.write(0x0038,16320)
diam1.write(0x00048,0)
diam1.write(0x0004c,0)
```

Slika 6.20. Primjer upisivanja vrijednosti u registre (parametre) funkcije

Rezultati brzine i performanse Diamond Search algoritma kroz 200 iteracija prikazani su na slici u nastavku:



Slika 6.21. Diamond Search - mjerenje brzine i performanse u Jupyteru

Vrijeme trajanja algoritma kroz 200 iteracija iznosi: 0.216047 sekundi.

```
In [1]: from pynq import Overlay
        from pynq import MMIO
        overlay = Overlay("/home/xilinx/pynq/overlays/DiamondSearch/design_UIS_4096.bit")

In [2]: overlay?

In [3]: diam1=overlay.user_input_search_0
        diam1?
```

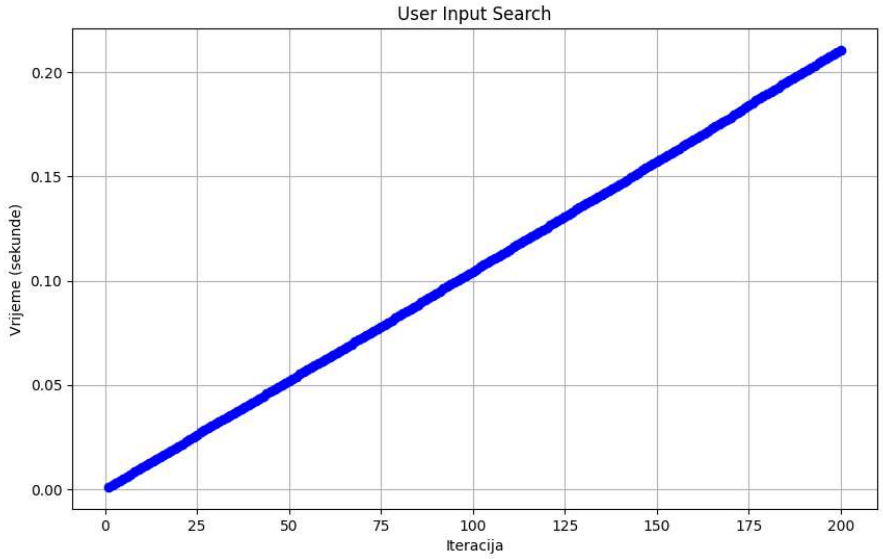
Slika 6.22. Učitavanje overlaya za User Input Search

```
diam1.write(0x0010,0)
diam1.write(0x0018,0)
diam1.write(0x0020,8)
diam1.write(0x0028,8)
diam1.write(0x0030,7)
diam1.write(0x0038,16320)
diam1.write(0x00048,0)
diam1.write(0x0004c,0)
```

Slika 6.23. Primjer upisivanja vrijednosti u registre (parametre) funkcije

6.0.6. Eksperimentalni rezultati za User Input Search

Rezultati brzine i performanse User Input Search algoritma kroz 200 iteracija prikazani su na slici u nastavku:

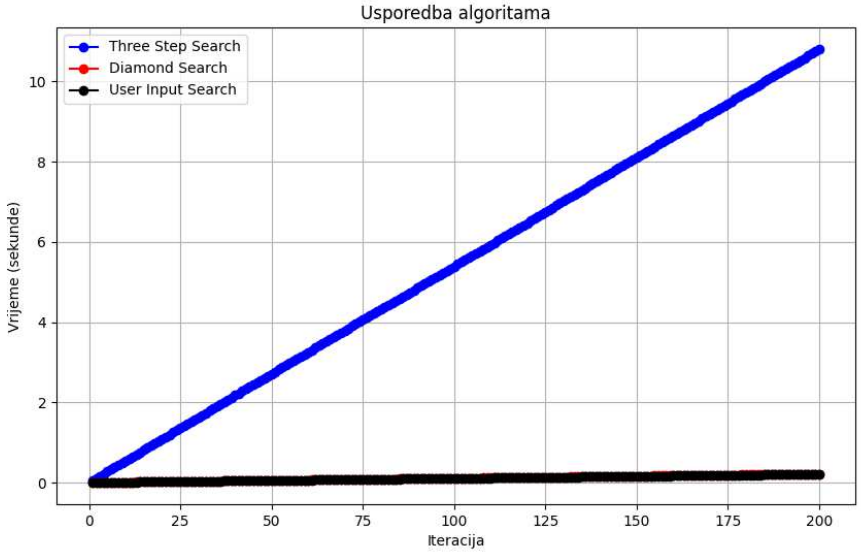


Slika 6.24. User Input Search - mjerenje brzine i performanse u Jupyteru

Vrijeme trajanja algoritma kroz 200 iteracija iznosi: 0.210484 sekundi.

6.0.7. Usporedba svih triju algoritama

Sljedeći graf prikazuje usporedbu izmjerenih brzina i performansi svih triju algoritama kroz 200 iteracija u Jupyter Notebooku.



Slika 6.25. Usporedba svih triju algoritama u Jupyteru

Iz grafa je moguće vidjeti da su algoritmi User Input Search i Diamond Search tu negdje po brzini, ali je na temelju mjerenja ipak User Input Search nešto malo brži od Diamond Searcha. Algoritam Three Step Search je dosta sporiji od njih. Također, treba napomenuti da su ovi rezultati testirani u okolini Jupyter bilježnice što znači da će i očekivano vrijeme izvršavanja biti puno sporije i lošije, nego kod onoga izmjerenog u programskom jeziku C. Naime, programski jezik C je jezik niske razine koji koristi kompajler i jako se brzo izvršava, dok je Python jezik visoke razine koji koristi interpreter i jako se sporo izvršava. Osim toga, na sporost izvršavanja programskog koda u Jupyteru, također utječe i činjenica da je Jupyter sučelje koje se otvara unutar preglednika te se spaja na poslužitelj. Na temelju izmjerenih vremena može se zaključiti da su se za sva tri algoritma dobili poprilično dobri rezultati, pogotovo za algoritme Diamond Search i User Input Search. To nam je dobar indikator da je akceleracija C programskog koda na FPGA uspješna uzevši u obzir sve navedene okolnosti pri mjerenju vremena u Jupyter bilježnici. Iz ovoga se može zaključiti da je dio za procjenu pokreta ubrzan korištenjem informacija o kretanju drona te je ovime potvrđeno da se postiže ubrzavanje kodiranja videa na dron uređajima informacijama o kretanju drona, što znači da je modifikacija

Diamond Search funkcije (iz koje je nastala User Input Search funkcija) pogodnija za korištenje. Iako ubrzanje na FPGA za User Input Search nije puno brže od Diamond Searcha, svako ubrzanje bi se trebalo uvažiti kada su u pitanju uređaji kod kojih energetska efikasnost predstavlja veliki značaj.

7. Zaključak

Ovim završnim radom istražena su različita softverska i hardverska rješenja za optimizaciju algoritama kodiranja videa s fokusom na energetsku učinkovitost i ubrzanje procesa procjene pokreta. Evaluacija implementacija H.265 algoritama poput Bolt65 i Kvazaar pokazala je njihovu učinkovitost u realnom vremenu, a implementacija optimiziranih algoritama na FPGA platformi uz korištenje alata Vitis i Vivado omogućila je njihovo ubrzanje pa tako onda i poboljšanje njihove energetske efikasnosti. Ključni doprinos rada leži u integraciji informacija o kretanju drona u proces procjene pokreta, što je rezultiralo boljim performansama kodiranja videa za bespilotne letjelice. Nadalje, metodološki pristupi poput modifikacije algoritama za procjenu pokreta kako bi se dobilo ubrzanje, pokazali su se uspješnima pri implementaciji na sklopovlju. Rezultati imaju praktične implikacije u industriji bespilotnih letjelica i šire područje video tehnologije, naglašavajući potrebu za efikasnijim algoritmima u situacijama gdje je energetska učinkovitost ključna. Buduće istraživanje može se usmjeriti na daljnju optimizaciju algoritama, istraživanje novih tehnologija poput umjetne inteligencije za još bolju obradu videa te razvoj novih generacija FPGA čipova koji mogu poboljšati brzinu i energetsku učinkovitost. Ovo istraživanje stoga postavlja temelje za daljnji razvoj u području video kodiranja i primjene na uređajima s ograničenim resursima.

Literatura

[1] C. S. Network, “Video coder and decoder”, https://www.cctvsg.net/what-are-video-codecs/.

[2] Softvelum, “H.265”, https://softvelum.com/nimble/hevc/.

[3] FER, “Fakultet elektrotehnike i racunarstva”, https://hr.m.wikipedia.org/wiki/Datoteka:FER_logo.jpg.

[4] U. V. Group, “Ultra video group”, https://ultravideo.fi/kvazaar.html.

[5] I. Berkeley Design Technology, “Car motion estimation”, https://www.bdti.com/InsideDSP/2007/08/15/Bdti.

[6] Wikipedia, “Block-matching algorithm”, https://en.wikipedia.org/wiki/Block-matching_algorithm.

[7] A. Samet, N. Litayem, W. Zouch, M. A. B. Ayed, i N. Masmoudi, “Three step search”, https://www.researchgate.net/publication/242156024_New_Horizontal_Diamond_Search_Motion_Estimation_Algorithm_For_H264AVC, 2015.

[8] D. Kerfa i M. F. Belbachir, “Diamond search”, https://www.researchgate.net/publication/270397625_Star_diamond_an_efficient_algorithm_for_fast_block_matching_motion_estimation_in_H264AVC_video_codec, 2015.

[9] B. Steinmann, “Vitis”, https://imperix.com/doc/help/xilinx-vitis-hls, 2021.

[10] “Vivado”, https://maker-hub.georgefox.edu/wiki/Xilinx_Vivado.

[11] I. E. G. Richardson, *Video Codec Design: Developing Image and Video Compression Systems*, 2002.

[12] “High efficiency video coding (hevc) family, h.265”, https://www.loc.gov/preservation/digital/formats/fdd/fdd000530.shtml.

[13] Wikipedia, “Video coding format”, https://en.wikipedia.org/wiki/Video_coding_format.

[14] I. Piljić, L. Dragić, A. Duspara, M. Čobrnić, H. Mlinarić, i M. Kovač, “Bolt65 - performance-optimized hevc hw/sw suite for just-in-time video processing”, 2019. https://doi.org/10.23919/MIPRO.2019.8756825

[15] R. Li, B. Zeng, i M. Liou, “A new three-step search algorithm for block motion estimation”, 1994. https://doi.org/10.1109/76.313138

[16] U. V. Group, “Kvazaar github”, https://github.com/ultravideo/kvazaar.

[17] J. Benjak i D. Hofman, “User input search”, 2023. https://doi.org/10.23919/MIPRO57284.2023.10159960

[18] I. E. G. Richardson, *H.264 and MPEG-4 Video Compression: Video Coding for Next-generation Multimedia*, 2003.

[19] H. Pishro-Nik, “Introduction to probability, statistics, and random processes”, https://github.com/casrou/ProbToPdf/blob/master/Introduction%20to%20Probability%2C%20Statistics%2C%20and%20Random%20Processes%20-%20Hossein%20Pishro-Nik.pdf, 2014.

[20] AMD, “Differences between vivado and vitis development flows”, https://docs.amd.com/r/en-US/ug1393-vitis-application-acceleration/Using-Alveo-Accelerator-Cards, 2024.

Sažetak

Ubrzavanje kodiranja videa na dron uređajima informacijama o kretanju drona

Robert Vitaliani

Danas postoji mnogo načina i algoritama za kodiranje videa, no malo je onih koji se usredotočuju na ubrzanje dijelova algoritma i metode kojima se nastoji poboljšati energetska efikasnost. Na dron uređajima je potrebno pažljivo pristupiti tom problemu, jer takvi uređaji zahtijevaju kodiranje videa s dobrom energetskom učinkovitošću, što se postiže pažljivim izvođenjem algoritma i pronalaženjem mjesta na kojima je moguće smanjiti količinu vremena provedenu na procesiranje određenih dijelova algoritma. Posebnu pažnju potrebno je obratiti na procjenu pokreta prilikom kretanja drona, koja predstavlja jedan od najzahtjevnijih dijelova kod kodiranja videa. Kako bi se pristupilo ovom problemu, bilo je potrebno proučiti algoritme za kodiranje videa, posebno algoritam H.265. Odabrane implementacije H.265 algoritma za kodiranje videa su Bolt65 i Kvazaar. Korištenjem ovih implementacija i njihovom manjom modifikacijom, postiže se bolja energetska učinkovitost i optimizacija algoritma za procjenu pokreta. Za uspješnu implementaciju algoritma i verifikaciju bolje energetske učinkovitosti, potrebno je bilo dio kodera implementirati u FPGA-u. Implementacija optimiranog algoritma izvedena je na FPGA koristeći alate za sintezu digitalnog sklopovlja - HLS (High-Level Synthesis). Korišteni alati prilikom HLS-a su AMD-ovi alati Vitis i Vivado koji se koriste za sintezu i analizu algoritma te SoC (System on Chip) razvoj. Ova tehnologija omogućava hardversku akceleraciju specifičnih dijelova algoritma, što može značajno smanjiti vrijeme obrade i potrošnju energije. Na ovaj način moguće je testirati rad kodera na testnim podacima te usporediti rad originalne i optimirane verzije kodera. Testiranje

implementacije algoritma provedeno je na PYNQ-Z1 pločici koristeći Jupyter Notebook (što omogućava jednostavno interaktivno programiranje i testiranje na FPGA platformi) pri čemu se može vidjeti ubrzanje i bolja učinkovitost algoritma. Uvijek je potrebno težiti optimizaciji i smanjenju korištenih resursa kad god je to moguće, pogotovo kada je u pitanju smanjenje energetske potrošnje odnosno povećanje energetske učinkovitosti.

Ključne riječi: Procjena pokreta; H.265/HEVC; Video koder; Bolt65; Kvazaar; Vitis; Vivado; PYNQ-Z1

Abstract

Acceleration of video encoding on drone devices using drone movement information

Robert Vitaliani

Today, there are many ways and algorithms for video encoding, but there are few that focus on speeding up parts of the algorithm and methods that try to improve energy efficiency. On drone devices, it is necessary to approach this problem carefully, because such devices require video encoding with good energy efficiency, which is achieved by carefully executing the algorithm and finding places where it is possible to reduce the amount of time spent on processing certain parts of the algorithm. Special attention should be paid to the estimation of movement during the movement of the drone, which is one of the most demanding parts of video encoding. In order to approach this problem, it was necessary to study video encoding algorithms, especially the H.265 algorithm. Selected implementations of the H.265 video encoding algorithm are Bolt65 and Kvazaar. By using these implementations and their minor modification, better energy efficiency and optimization of the motion estimation algorithm is achieved. For the successful implementation of the algorithm and the verification of better energy efficiency, it was necessary to implement part of the encoder in the FPGA. The implementation of the optimized algorithm was carried out on the FPGA using tools for the synthesis of digital circuits - HLS (High-Level Synthesis). The tools used during HLS are AMD's Vitis and Vivado tools, which are used for algorithm synthesis, analysis and SoC (System on Chip) development. This technology enables hardware acceleration of specific parts of the algorithm, which can significantly reduce processing time and energy consumption. In this way, it is possible to test the work of the coder on test data and compare the work of

the original and optimized version of the coder. Testing of the implementation of the algorithm was carried out on the PYNQ-Z1 board using Jupyter Notebook (which enables simple interactive programming and testing on the FPGA platform), where acceleration and better efficiency of the algorithm can be seen. It is always necessary to strive for optimization and reduction of used resources whenever possible, especially when it comes to reducing energy consumption or increasing energy efficiency.

Keywords: Motion estimation; H.265/HEVC; Video encoder; Bolt65; Kvazaar; Vitis; Vivado; PYNQ-Z1