Ubrzavanje kodiranja videa na dron uređajima informacijama o kretanju drona

Kozina, Jan

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1630

UBRZAVANJE KODIRANJA VIDEA NA DRON UREĐAJIMA INFORMACIJAMA O KRETANJU DRONA

Jan Kozina

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1630

UBRZAVANJE KODIRANJA VIDEA NA DRON UREĐAJIMA INFORMACIJAMA O KRETANJU DRONA

Jan Kozina

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1630

Pristupnik:	**Jan Kozina (0036540203)**
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.

Zahvale mentoru izv. prof. dr. sc. Danielu Hofmanu na učenju, savjetima i pomoći u izradi ovog završnog rada

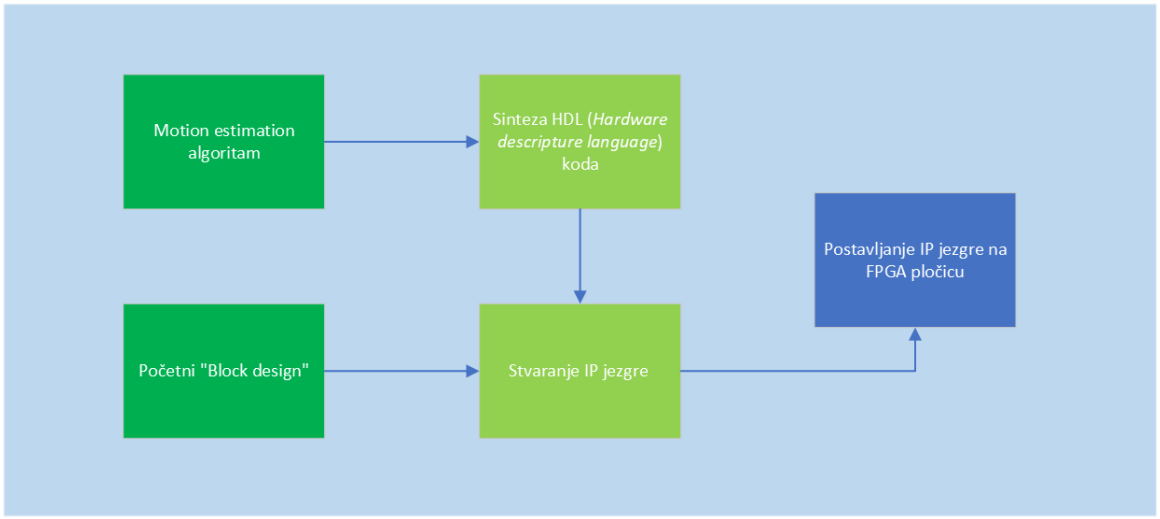
Uvod

Ubrzavanje kodiranja podataka u prijenosu i videa na dron uređajima jedan je od ključnih faktora za poboljšanje funkcionalnosti i performansi tih letjelica čime se postiže veća autonomnost u širokom spektru aplikacija. Između brojnih tehnika kojima se postiže navedeni cilj, jedna koja se istaknula kao vrlo primjenjiva jest procjena pokreta (engl. *Motion estimation*) gdje se koriste algoritmi kako bi se omogućilo precizno praćenje objekata i okoline u kojoj se nalazi uređaj. U prijenosu podataka (videozapis) u stvarnom vremenu procjenom kretanja može se postići kompresija čime se eliminira vremenska redundancija, dobije veća brzina prijenosa i propusnost kroz komunikacijski kanal. Implementacije H.265 algoritma za kodiranje videa koje koriste procjenu pokreta su Kvazaar („open-source“) i Bolt65 kao FER-ova inačica koji služe kao polazna točka za algoritme procjene pokreta.

S obzirom na to da je procjena pokreta, kao optimizacijska tehnika, vrlo koristan način ubrzavanja, potrebno je pronaći načine kako možemo postići poboljšanje performansi direktno na dron uređajima. Potencijalno rješenje je izvršavanje programskog koda na specijaliziranim sklopovskim platformama dizajniranim i namijenjenim za specifične ciljeve. FPGA (engl. *Field-programmable gate array*) platforma za razvoj sklopovlja omogućava iskorištavanje resursa sklopovlja implementiranog na FPGA pločicama za potrebe izvršavanja programskog koda. Glavne prednosti FPGA uključuju visoku fleksibilnost, visoke performanse, nisku potrošnju energije u usporedbi s drugim platformama i brzo vrijeme prototipiranja i razvoja. U ovom radu korištena je pločica PYNQ-Z1 proizvođača Xilinx.

Najvažniji korak razvoja za FPGA u mnogim implementacijama jest tehnologija HLS (engl. *High level synthesis*) koji je praktički postao norma u industriji digitalnog dizajna. To je postupak koji predstavlja sintezu jezika za opis sklopovlja iz nekog višeg programskog jezika nakon čega se jezik za opis sklopovlja direktno implementira na FPGA. Time se omogućuje veća fleksibilnost, bolja razumljivost i značajno smanjenje vremena razvoja.

Na slici (Slika 0.1) nalazi se generalni dijagram glavnih koraka kroz koje ovaj rad prolazi.

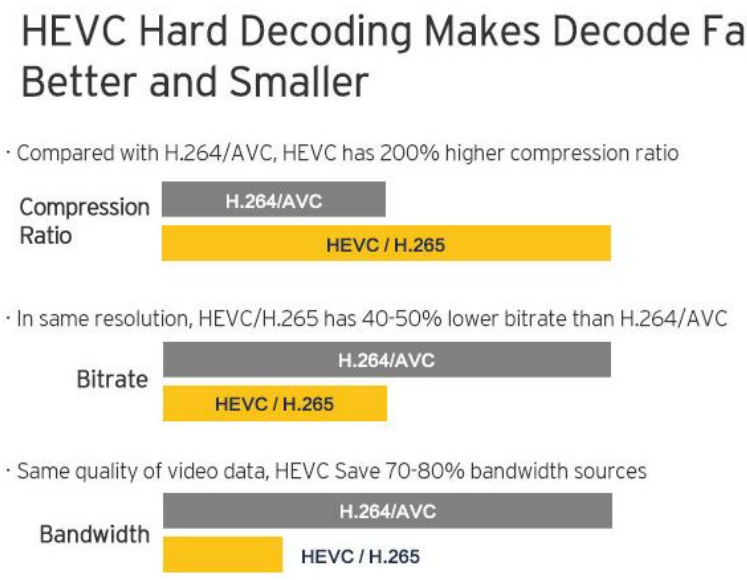


Slika 0.1 Dijagram koraka procesa ubrzavanja kodiranja

1. Video kodiranje

Video kodiranje je proces kompresije i dekompresije video zapisa kako bi se smanjila veličina datoteke, olakšao prijenos i skladištenje, a pritom zadržala što je moguće viša kvaliteta slike. Kompresija se postiže korištenjem različitih algoritama koji smanjuju redundanciju i nepotrebne informacije u video zapisu. Postoje dva glavna tipa kompresije: s gubicima i bez gubitaka. U praksi, kompresija s gubicima češće se koristi jer značajno smanjuje veličinu datoteka uz minimalan utjecaj na percepciju kvalitete slike.

H.265, ili *High Efficiency Video Coding* (HEVC), je napredni standard za video kodiranje razvijen kao nasljednik H.264/AVC. Cilj H.265 je da poboljša efikasnost kompresije video sadržaja, smanjujući „bitrate“ za do 50% u odnosu na svog prethodnika te koristeći bolje metode intra i inter predikcije koje smanjuju redundanciju (Slika 1.1).



Slika 1.1 Usporedba H.265 i H.265 standarda [1]

H.265 je široko prihvaćen u industriji zbog svojih prednosti. Koristi se u digitalnim televizijskim prijenosima, video konferencijama i mnogim drugim aplikacijama gdje je efikasnost prijenosa podataka ključna. Ovaj rad se prvenstveno dotiče metode procjene kretanja koje implementacije H.265 standarda koriste u procesu kodiranja.

2. Procjena pokreta

Procjena pokreta kao tehnika ubrzavanja kodiranja predstavlja generiranje vektora pokreta (engl. *Motion vectors*) koji označavaju smjer i iznos pomaka između pojedinih elemenata uzastopnih sličica ili grupiranih dijelova sličica od kojih se sastoji videozapis. Postoji više metoda s kojima se taj postupak može primijeniti. Najčešće metode su:

- Uspoređivanje blokova (engl. *Block-matching method*) – pojedini elementi sličice grupiraju se u disjunktne blokove, najčešće pravokutne, nad kojima se jednostavno može primijeniti usporedba sličnosti
- Metode usporedbe gradijenata – analiziranjem gradijenata intenziteta slike utvrđuje se obrazac kretanja što zapravo predstavlja optički tok (engl. *Optical flow*)
- Fazna korelacija – koristi Fourierovu transformaciju (FFT) i svojstvo povezanosti prostorne i frekvencijske domene da bi se otkrio pokret

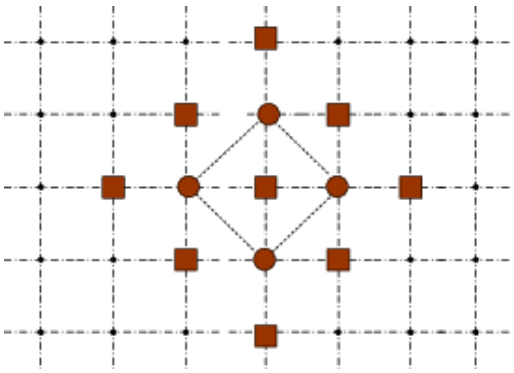
Svaka od navedenih metoda ima prednosti i nedostatke te upotreba ovisi o prirodi i specifičnosti zadatka. U ovom radu koristi se Block-matching (Slika 2.1) metoda s različitim algoritmima.. Rezultat metode su vektori pokreta koji pokazuju pomak bloka sličice ili pojedinog piksela između dvije uzastopne sličice koje ćemo razlikovati po vremenu. Sličica u vremenu T = t-1 se naziva referentna sličica (engl. *Reference frame*), a sličicu u vremenu T = t ćemo smatrati trenutnom sličicom (engl. *Current frame*). Sada razlikujemo dvije konvencije: slučaj u kojem vektori koji pokazuju od bloka u trenutnoj sličici prema bloku u referentnoj sličici, u tom slučaju vektori se zovu kompenzacijski te će se taj slučaj referirati kao „CTR – Current-to-reference“, i slučaj u kojem vektori pokazuju u suprotnom smjeru od prvog slučaja – od bloka u referentnoj sličici prema bloku u trenutnoj – „RTC – Reference-to-current“ . Broj vektora jednak je broju blokova sličice. Konceptualno, u slučaju CTR, vektor pokreta pokazuje smjer, i posljedično lokaciju, na kojoj se nalazi blok u referentnoj sličici koji je najsličniji bloku koji razmatramo u trenutnoj sličici. Time se dobije mogućnost konstruiranja trenutne sličice već kodiranim blokovima iz referentne sličice čime se ubrzava i smanjuje količina posla za svaku sličicu. Kvazaar[4] je *open-source* HEVC (H.265) video enkoder koji je razvila organizacija „Ultra Video Group“ na Tehničkom sveučilištu u Tampereu. Implementiran je u programskom jeziku C te ga karakteriziraju visoka učinkovitost i prilagodljivost, kao i mogućnost

pokretanja na različitim platformama. Algoritmi procjene pokreta koje koristi su „Diamond Search“, „Full Search“, „Test Zone Search“ i „Hexagon Based Search“. U ovom radu detaljnije će biti obrađen „Diamond Search“ algoritam za generiranje vektora pomaka.

Slika 2.1 Block-matching algoritam [5]

2.1. „Diamond search“ algoritam

Diamond search algoritam predstavlja tehniku pretraživanja polja korištenjem oblika dijamanta kao smjerove pretraživanja (Slika 2.2). Pritom je nitno naglasiti da se razlikuju dva oblika na slici: točke spojene linijama predstavljaju mali dijamant koji se koristi u postupku „Small diamond search pattern“ (SDSP) koji služi za detaljniju pretragu manjeg polja; točke vanjskog obruča koje nisu spojene linijama predstavljaju veliki dijamant koji se koristi u postupku „Large diamond search pattern“ (LDSP) koji služi za općenitiju pretragu s većim korakom čime može brže generirati vektor pokreta.



Slika 2.2 Pretraživanje oblikom dijamanta [2]

Algoritam je sljedeći [3]:

LDSP:

1. Početak pretrage postavite na središnju lokaciju.
2. Postavite veličinu koraka S = 2.
3. Pretražite 8 lokacija piksela (X,Y) tako da (|X|+|Y|=S) oko lokacije (0,0) koristeći uzorak pretrage u obliku dijamanta.
4. Odaberite među 9 pretraženih lokacija onu s najmanjom vrijednosti funkcije troška.
5. Ako se minimalna težina nalazi u središtu prozora pretrage, idite na korak SDSP.
6. Ako se minimalna težina nalazi na jednoj od 8 lokacija koje nisu središte, postavite novi izvor na tu lokaciju.
7. Ponovite LDSP.

SDSP:

1. Postavite novi izvor pretrage.
2. Postavite novu veličinu koraka kao S = S/2 (tj. S = 1).
3. Ponovite postupak pretrage kako biste pronašli lokaciju s najmanjom težinom.
4. Odaberite lokaciju s najmanjom težinom kao vektor gibanja.

Kvazaar HEVC enkoder [4], između ostalih, koristi ovaj algoritam te je njegova implementacija korištena kao polazna točka za funkciju čija je deklaracija sljedeća:

```
void diamond_search(int pic[n][n], int ref[n][n], int offset_x, int offset_y, int width, int height, uint32_t steps, int *best_cost, vector2d_t *best_mv, int refNum, int picNum)
```

Objašnjenje parametara:

- `int pic[N][N]:`
    - Dvodimenzionalni niz koji predstavlja trenutni blok slike koji se analizira.
- `int ref[N][N]:`
    - Dvodimenzionalni niz koji predstavlja referentni blok slike prema kojem se vrši pretraga.
- `int offset_x:`

- Horizontalni pomak u referentnom bloku odakle počinje pretraga.

- `int offset_y:`
  - Vertikalni pomak u referentnom bloku odakle počinje pretraga.

- `int width:`
  - Širina bloka slike koji se analizira.

- `int height:`
  - Visina bloka slike koji se analizira.

- `uint32_t steps:`
  - Maksimalan broj koraka pretrage koji se mogu izvršiti. Ograničava broj iteracija algoritma dijamantne pretrage.

- `int *best_cost:`
  - Pokazivač na varijablu u kojoj će biti pohranjena najmanja cijena izračunata troškovnom funkcijom (engl. Cost function) tokom pretrage. Troškovna funkcija predstavlja sličnost između trenutnog bloka i referentnog bloka.

- `vector2d_t *best_mv:`
  - Pokazivač na strukturu tipa vector2d_t koja će sadržati najbolji vektor pomaka pronađen tokom pretrage. vector2d_t sadrži dva člana: x i y, koji predstavljaju horizontalni i vertikalni pomak.

- `int refNum`:
  - Identifikator referentne sličice. Koristi se za identifikaciju u kontekstu više sličica u video nizu.

- `int picNum:`
  - Identifikator trenutne sličice. Koristi se za identifikaciju u kontekstu više sličica u video nizu.

Funkcija se primjenjuje na M sličica veličine N x N elemenata, što znači da se može postaviti proizvoljan broj sličica ovisno o videu. Sličice će biti podijeljene u blokove elemenata.

Važan parametar koji se treba razmotriti jest `steps` koji je pozitivan cijeli broj. Ovaj parametar direktno je vezan za preciznost i kompleksnost funkcije. Predstavlja maksimalan

broj iteracija kroz koje će algoritam proći za svaki blok pri čemu se u svakoj iteraciji veličina vektora pomaka povećava za 1. Drugim riječima, to je broj koraka koje jedan blok ima na raspolaganju za pomak. Koliko koraka će blok učiniti ovisi o funkciji troška. Deklaracija:

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

Funkcija troška (engl. *Cost function*) računa cijenu pomaka bloka za jedan korak, tj. povećanje vektora smjera za jedan. Poziva se za svaki smjer oblika dijamanta. U matematičkom smislu, radi se o računanju apsolutnih razlika (engl. SAD – *Sum of absolute differences*) vrijednosti elemenata slike, što je česta metoda u području digitalne obrade slike. SAD se računa prema izrazu (1).

$$SAD(X,Y) = \sum |(X_i - Y_i)| \quad (1)$$

Oblik s kojim ova funkcija radi je mali dijamant povezan linijama (Slika 2.2). Smjerovi su sljedeći:

- Prema gore – vektor pokreta: (x=0,y=1)
- Prema dolje - vektor pokreta: (x=0,y=-1)
- Ulijevo - vektor pokreta: (x=-1,y=0)
- Udesno – vektor pokreta (x=1,y=0)
- Centar - vektor pokreta: (x=0,y=0)

Pomak u svakom smjeru se jedanput izračuna u funkciji troška te je rezultat trošak kao brojčana vrijednost. Izabere se smjer koji je dao najmanju vrijednost jer pomak u tom smjeru daje blok najsličniji onome koji trenutno promatramo. Važno je napomenuti kako je moguć slučaj u kojem je najmanja vrijednost troška upravo za centar tj. bez pomaka te je moguće da funkcija ne nalazi jeftinije troškove pomaka za bilo koji smjer. U oba slučaju funkcija `diamond search` staje s pretragom jer smatra kako je nađen najsličniji blok.

Naposlijetku, generiran je vektor pokreta sa svojom veličinom iz koje možemo vidjeti koliko koraka je pojedini blok učinio (Slika 2.3). Pomak vektora računa se u višekratnicima broja 4 jer se time uračunava granuliranost detaljnija od jednog elementa slike tj. jednog piksela. Konkretno, moguće je raspoznavanje do četvrtine piksela (0.25 piksela). Pogledamo li vektor za blok (4,2) u sličici 2 možemo vidjeti kako vektor pokazuje pomak od 4 koraka u smjeru pozitivne x-osi (16/4), te pomak od jednog koraka u smjeru negativne y-osi (-4/4). Dakle taj blok se pomaknuo ukupno 5 koraka od mogućeg maksimalnog broja u varijabli `steps` kako bi našao najsličniji blok.

Slika 2.3 Ispis pokretanja algoritma `diamond search`

2.2. User Input Search algoritam

Ovaj algoritam predstavlja proširenje standardnog algoritma pretraživanja te, iako će ovdje o njemu biti govoreno u kontekstu „diamond search“ algoritma, može ga se primijeniti i na ostale. Kod upravljanja dron uređajima pretpostavljamo da korisnik daljinskim upravljanjem kontrolira kretanje drona na način da mu zadaje naredbe smjera po želji. Na video koji predstavlja pogled dron uređaja primjenjuju se algoritmi procjene pokreta no možemo dodatno iskoristiti korisnikove naredbe kako bismo postigli još bržu i efikasniju kompresiju.

Deklaracija funkcije je sljedeća:

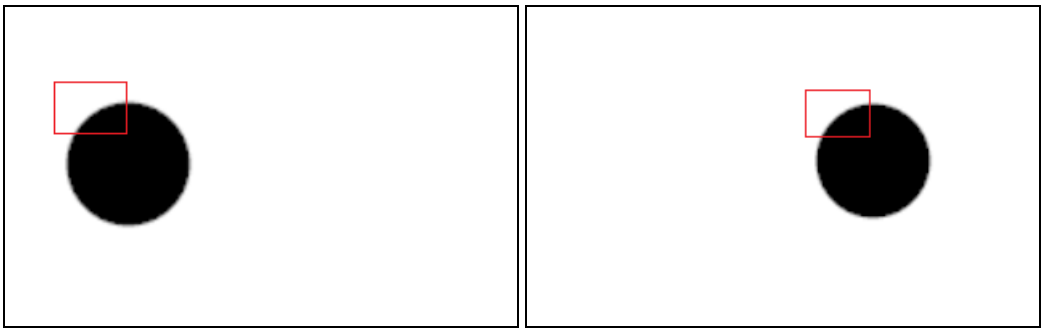
```
void user_input_search(int ref[N][N], int pic[N][N], int offset_x, int offset_y, int width, int height, int steps, int *best_cost, vector2d_t *best_mv, int refNum,int picNum,int direction)
```

Promjena u odnosu na `diamond search` funkciju jest dodatak parametra `direction` koji predstavlja korisnikovu kontrolu, tj. unos. Ovisno o broju koji se nalazi u tom parametru mogući smjerovi su:

- Pomak prema gore – 1
- Pomak prema dolje – 2
- Pomak udesno – 3
- Pomak ulijevo – 4
- Okret ulijevo – 5
- Okret udesno – 6
- Poniranje ulijevo – 7
- Poniranje udesno – 8

Kod je podešen na način da koristi matrice vjerojatnosti sa pripadajućim skalarima za svaki smjer koji utječu na generiranje vektora smjera, no u stvarnoj aplikaciji smjer bi bio direktno spojen na korisnikove kontrole.

Ovime se postiže dodatna brzina kompresije jer algoritam provjerava potencijalno povećanje iznosa vektora pomaka samo u smjeru u kojem se slika pomiče. Primjerice, pomičemo li dron ulijevo, slika se pomiče udesno te očekujemo novi sadržaj na lijevoj strani sličice, a stari sadržaj potiskuje se na desnu stranu, što znači da će se, koristimo li CTR konvenciju, blokovi u trenutnoj sličici trebati pomaknuti udesno kako bi pronašli slične blokove među starim sadržajem (Slika 2.4). Time se eliminira nepotrebno provjeravanje svih 5 smjerova u slučaju oblika malog dijamanta.



Slika 2.4 Uzastopne sličice pomaka kugle i najsličniji blokovi

2.3. Bolt65 enkoder

Druga implementacija HEVC algoritma za kodiranje videa je Bolt65. To je FER-ova vlastita implementacija koja je još u procesu razvoja, ali su već ugrađeni neki algoritmi procjene pokreta, konkretno:

- „Full Area Search“
- „Three Step Search“ i
- „Full Integer Area Search“.

U nastavku razmotrit će se „Three Step Search“.

2.3.1. Three Step Search algoritam

Three Step Search jedan je od najranijih implementacija algoritma usporedbe blokova. Kao što njegovo ime kaže, ovaj algoritam obavlja pretraživanje polja ili prostora u tri koraka iterativno smanjujući veličinu oblika koji se pretražuje.

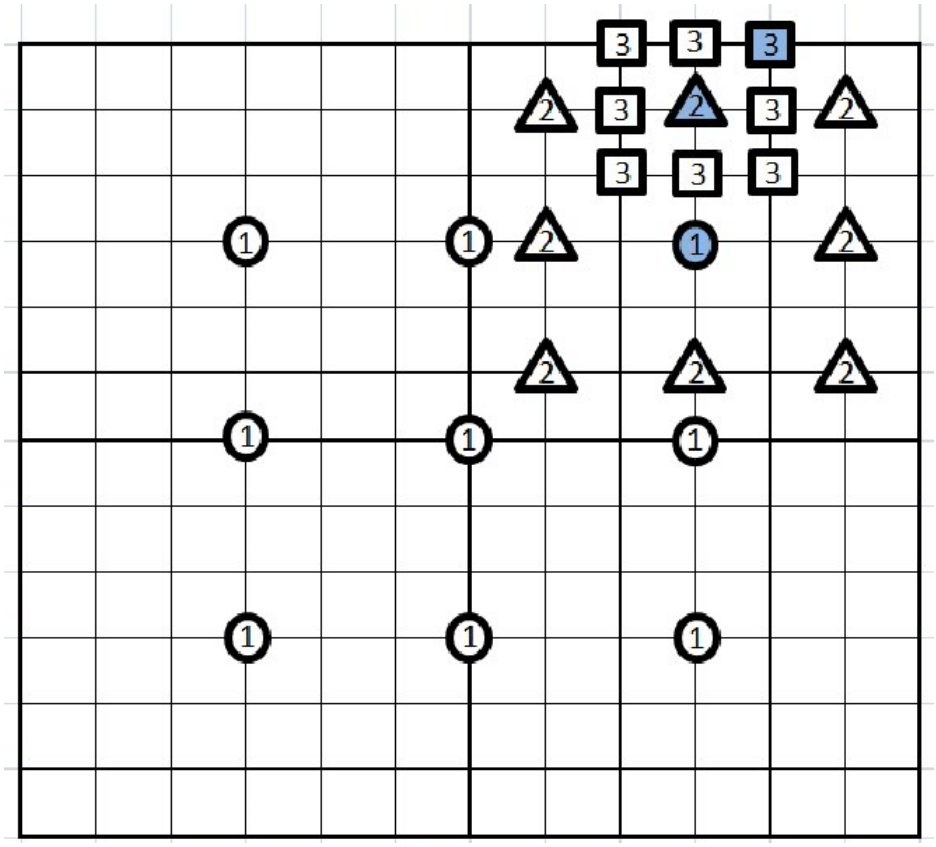
Algoritam [6]:

- Počni s pretragom na središnjoj lokaciji
- Postavi korak S = 4 i parametar pretrage p = 7.
- Pretraži 8 lokacija +/- S piksela oko točke (0,0) i lokaciju (0,0).
- Odaberi među 9 pretraženih lokacija onu s minimalnom funkcijom troška.

- Postavi novi izvor pretrage na odabranu lokaciju.
- Postavi novi korak kao S = S/2.
- Ponovi postupak pretrage dok S ne postane 1.

Rezultirajuća lokacija za S=1 je ona s minimalnom funkcijom troška, a blok na toj lokaciji je najbolje podudaranje.

Na slici (Slika 2.5) može se vidjeti vizualni prikaz rada ovog algoritma. Početni korak je 3 te se u svakoj iteraciji smanjuje za 1. Početna točka je centar sustava te se za sve točke označene brojem 1 izračuna funkcija troška. U ovom slučaju najmanji trošak daje lokacija (3,3) pa se novi ta lokacija postavlja kao novi izvor pretrage. U novom koraku ponovno se provjerava 9 lokacija (8 + centar) označenih brojem 2, ali je korak postavljen na 2. Odabire se lokacija s najmanjim troškom i postavlja novi izvor pretrage. U zadnjoj iteraciji korak je postavljen na 1 i lokacija s najmanjim troškom označava kraj algoritma.



Slika 2.5 Algoritam „Three Step Search“ [7]

Funkcija koja implementira ovaj algoritam ima sljedeću deklaraciju:

```
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)
```

Objašnjenje parametara:

- `unsigned char *referenceFrame`
    - Pokazivač na referentni okvir koji se koristi kao referenca za procjenu pokreta
- `unsigned char *block`
    - Pokazivač na trenutni blok u okviru za koji se vrši procjena pokreta
- `int puWidth`
    - Parametar specificira širinu predikcijske jedinice tj. bloka
- `int puHeight`
    - Parametar specificira visinu predikcijske jedinice tj. bloka
- `int startingIndex`
    - početni indeks za pretraživanje u algoritmu procjene pokreta
- `int areaSize`
    - parametar određuje veličinu područja pretraživanja oko bloka
- `int *deltaX`
    - Pokazivač na varijablu u kojoj će biti pohranjen horizontalni pomak vektora pokreta
- `int *deltaY`
    - Pokazivač na varijablu u kojoj će biti pohranjen vertikalni pomak vektora pokreta
- `int width`
    - Širina bloka.
- `int height`
    - Visina bloka.

- `int *chosenBmcValue`

    - pokazivač na varijablu u kojoj će biti pohranjena odabrana vrijednost „block-matching“ kriterija (BMC). BMC zapravo predstavlja SAD mjeru za procjenu sličnosti između blokova tijekom procjene pokreta.

Funkcija je slična `Diamond Search` funkciji, ali koristi drugačiji algoritam pretraživanja. Razlike u parametrima su `areaSize` koji je zamijenio varijablu `steps` s obzirom da je broj koraka u ovom algoritmu fiksan, te parametar `startingIndex` koji omogućava varijabilni početak pretrage. BMC je zamijenio `best_cost` no koncept je ostao isti.

3. FPGA implementacija algoritama

Sljedeći korak u procesu odnosi se na FPGA implementaciju programskog koda algoritama pretraživanja polja. Konkretno, potrebno je provesti postupak HLS (Slika 1) kako bismo iz izvornog koda u jeziku C dobili sintetizirani kod u jeziku za opis sklopovlja (HDL – *Hardware descripture language*). Sintetizirani kod će predstavljati IP (engl. *Intellectual property*) koji će se integrirati u cjelokupni FPGA dizajn. Kako bi se cijeli proces proveo ispravno potrebno je znati s kojim alatima se izvodi pojedini korak. Njihov redoslijed je sljedeći:

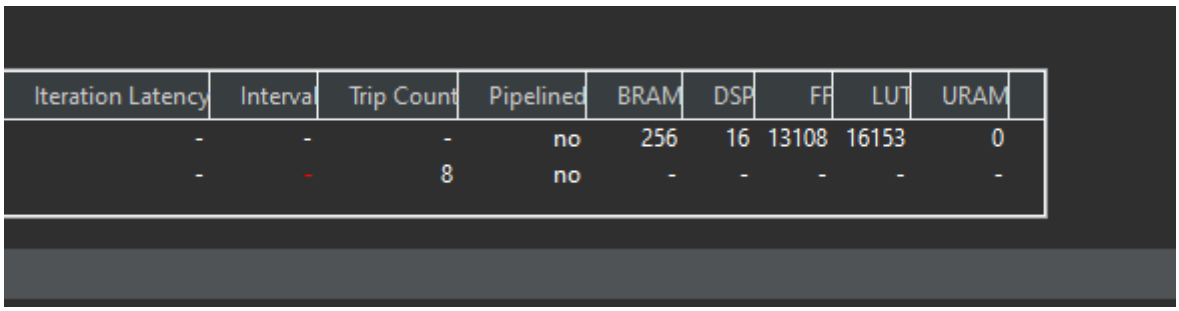
- Vitis HLS
- Razvojno okruženje Xilinx Vivado
- PYNQ Z1 FPGA pločica
- Jupyter Notebook interaktivno okruženje

3.1. Vitis HLS

Vitis HLS[9] je alat razvijen od strane Xilinxa, koji omogućava dizajnerima da razviju hardverske akceleratore koristeći programske jezike višeg nivoa kao što su C, C++ i OpenCL. Umjesto pisanja kompleksnog i često ljudima teško čitljivog koda u jeziku za opis sklopovlja (HDL), dizajneri mogu koristiti Vitis HLS kako bi automatizirali proces prevođenja visokog nivoa apstrakcije koda u efikasni HDL kod. Alat se koristi u raznim industrijama za ubrzavanje algoritama u aplikacijama kao što su obrada signala, kodiranje i dekodiranje videa, strojno učenje, financijska analiza i drugi zahtjevni računalni zadaci.

Alat u HLS procesu nudi 4 opcije: simulaciju, sintezu, kosimulaciju i „export“ RTL (engl. *Register Transfer Level*) dizajna. Korak simulacije odnosi se na pokretanje izvornog C koda iz čega alat generira izvještaj o provedenom postupku. Taj korak ne smije sadržavati greške jer daljnja sinteza neće uspjeti. Sljedeća opcija jest najvažniji korak – sinteza. U njemu se programski kod u jeziku C prevodi u HDL jezik pri čemu postoje dvije mogućnosti tog jezika: Verilog i VHDL. U ovom zadatku koristio se VHDL zbog

jednostavnosti. Ovaj korak također rezultira izvještajem u kojem se može vidjeti konkretna procjena resursa FPGA pločice za implementaciju koda (Slika 3.1).



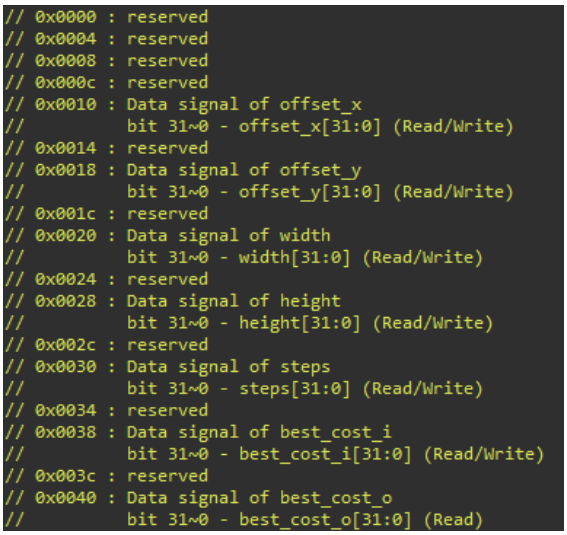
Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
-	-	-	no	256	16	13108	16153	0
-	-	8	no	-	-	-	-	-

Slika 3.1 Resursi potrebni za implementaciju koda

U slučaju da pločica nema dovoljno fizičkih resursa tj. CLB blokova (engl. *Configurable logic blocks*) sinteza neće uspjeti i generirat će se izvještaj o pogrešci.

Treći korak je kosimulacija koja nije nužna za izvedbu sinteze i RTL dizajna. U ovom koraku paralelno se pokreću C kod i sintetizirani HDL kod pri čemu se uspoređuju razlike između njih. Zadnji korak je kreiranje RTL dizajna i izvoz na disk za korištenje u drugim alatima. RTL je potreban kako bi pločica znala kako organizirati registre, logička vrata i druge strukture u implementaciji.

Nakon provedenog procesa, generirat će se datoteka s nastavkom „…hw.h“ koja sadrži mapu registara. U toj mapi definirane su memorijske lokacije koje služe kao posrednici koji se koriste za kontrolu i komunikaciju s hardverskim akceleratorom (IP-om dobivenim HLS postupkom) implementiranim na FPGA. Ova mapa registara je ključna za softver koji radi na glavnom procesoru (kao što je ugrađeni CPU u SoC-u) kako bi mogao komunicirati s prilagođenim hardverskim akceleratorom. Komunikacija može predstavljati prijenos podataka, direktnu kontrolu IP jezgre ili generalno provjeravanje statusa komponenti (engl. „Monitoring“). Prikaz dijela mape je na slici (Slika 3.2).

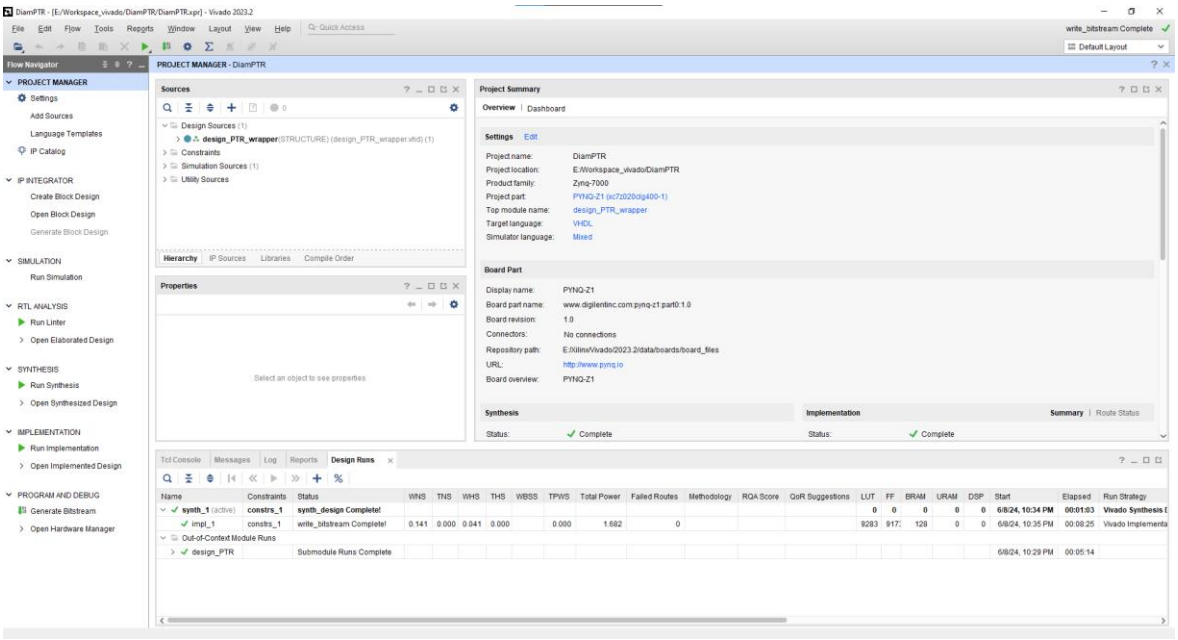


Slika 3.2 Mapa registara za FPGA logiku

ARM procesor koji se nalazi na FPGA pločici koristit će registre kako bi predavao vrijednosti potrebne za FPGA logiku da odsimulira funkciju. Rezultati funkcije će biti vrijednosti registara koji su mapirani na vektore koje računamo i želimo dobiti.

3.2. Vivado

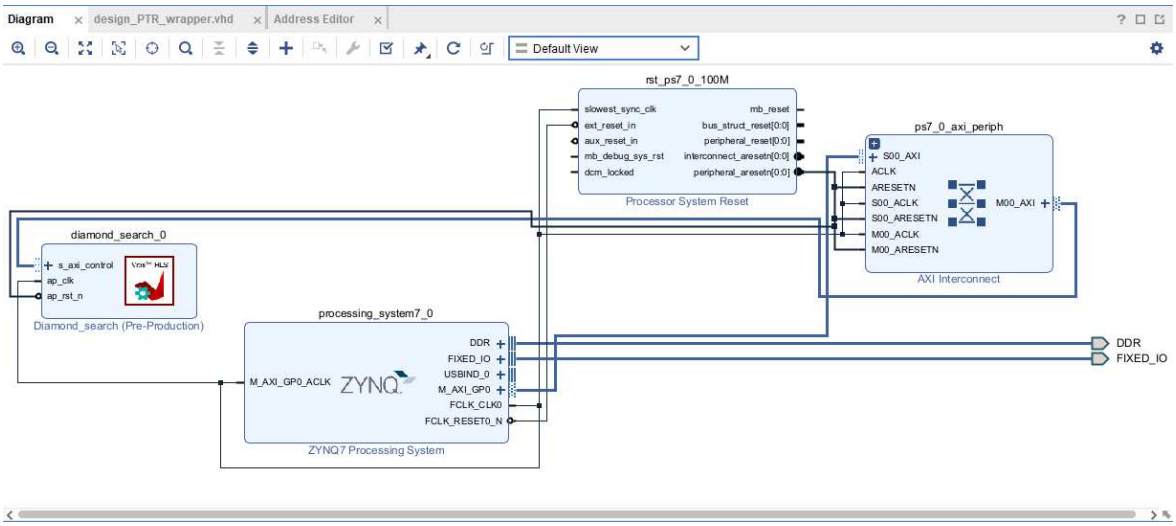
Vivado je integrirano razvojno okruženje (IDE) također razvijeno od strane tvrtke Xilinx za dizajn i implementaciju digitalnih sustava na njihovim FPGA i SoC uređajima. Vivado je nasljednik starijeg Xilinx ISE alata, koji je obuhvaćao i HLS postupak kao cjelinu prije nego je nastao Vitis kao zaseban alat. Glavne prednosti alata su visoka razina optimizacije za smanjenje potrošnje FPGA resursa, IP integrator koji služi za povezivanje jezgara i modula olakšavajući razvoj složenih sustava, i alati za pregledavanje mreže i postavljenih ruta u sustavu. Grafičko sučelje može se vidjeti na slici (Slika 3.3)



Slika 3.3 Prikaz Vivado grafičkog sučelja

Proces započinje kreiranjem početnog blok dizajna koji će sadržavati sve IP jezgre kao zasebne komponente koje se mogu međusobno spajati vezama (Slika 3.4). Te veze mogu predstavljati sučelja za interakciju dvaju blokova, signale, primjerice za takt kojim sinkroniziramo rad različitih blokova ili za reset, ili mogu biti sabirnice za komunikaciju i prijenos podataka između komponenti. Prva i glavna IP jezgra koja se mora dodati u dizajn je jezgra procesorskog sustava ZNYQ 7 Processing System (PS7) koja uključuje ARM Cortex-A9 procesor i pripadajuću periferiju. Taj procesor nalazi se na pločici PYNQ Z1 i odgovoran je za izvođenje programskog koda izvan FPGA logike. Sljedeća jezgra (blok koja se mora dodati jest upravo ona koju dobijemo iz HLS postupka. Ovisno o algoritmu to će biti Diamond_search, User_input_search ili Three_step_search.

Dodatni blokovi koji su potrebni za ispravan rad su blok za resetiranje PS7 i blok koji se zove „ps7_0_axi_periph“. Taj blok je služi kao poveznica između procesorskog sustava PS7 i različitih AXI (engl. *Advanced Extensible Interface*) periferija koji se nalaze u dijelu programabilne logike (PL) FPGA. Dakle, cjelokupna komunikacija se usmjeruje i prosljeđuje kroz taj blok.



Slika 3.4 Blok dizajn sustava u Vivado-u

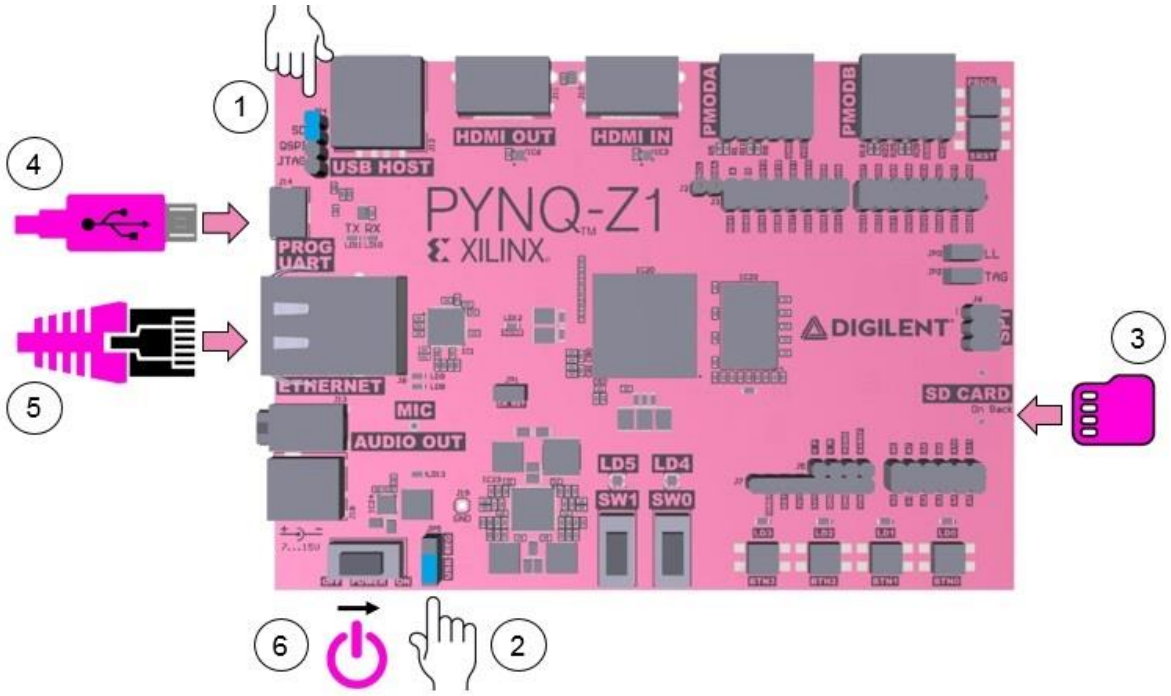
Nakon što je sustav dizajniran i međusobno ispravno povezan potrebno je generirati „bitstream“ opcijom koju nudi Vivado. „Bitstream“ predstavlja binarnu datoteku koja sadrži konfiguracijske informacije koje FPGA pločica treba pročitati kako bi znala podesiti interne resurse (logičke ćelije, međupovezanost, I/O pinovi itd.) da bi se izvršila željena funkcionalnost. Svaki blok dizajn rezultirat će različitim „bitstream-om“. Taj „bitstream“ se postavlja na pločicu te je potreban za Jupyter bilježnicu u koju će biti uključen.

Posljednji korak u ovom dijelu procesa je spremanje blok dizajna u .Tcl (engl. *Tool Command Language*) skriptu koja se također mora postaviti na FPGA pločicu.

3.3. PYNQ Z1 pločica

Sklopovlje na bazi FPGA arhitekture koje se koristi za implementaciju HDL programskog koda jest PYNQ Z1. To je razvojna pločica koju je dizajnirala tvrtka Xilinx u suradnji s tvrtkom Digilent. Ona je dio PYNQ (engl. *Python Productivity for Zynq*) ekosustava, koji omogućuje programerima da kroz programski jezik Python iskoriste Xilinx Zynq SoC funkcionalnost (engl. *System-on-Chip*) i MPSoC (engl. *Multiprocessor SoCs*) bez potrebe za detaljnim poznavanjem dizajna FPGA-a. Programabilni logički sklopovi koriste se kao hardverske biblioteke i programiraju putem njihovih API-ja (Aplikacijsko programsko sučelje) na konceptualno isti način na koji se uvoze i programiraju softverske biblioteke.

Na slici (Slika 3.5) nalazi se dizajn i način pripreme pločice za rad.

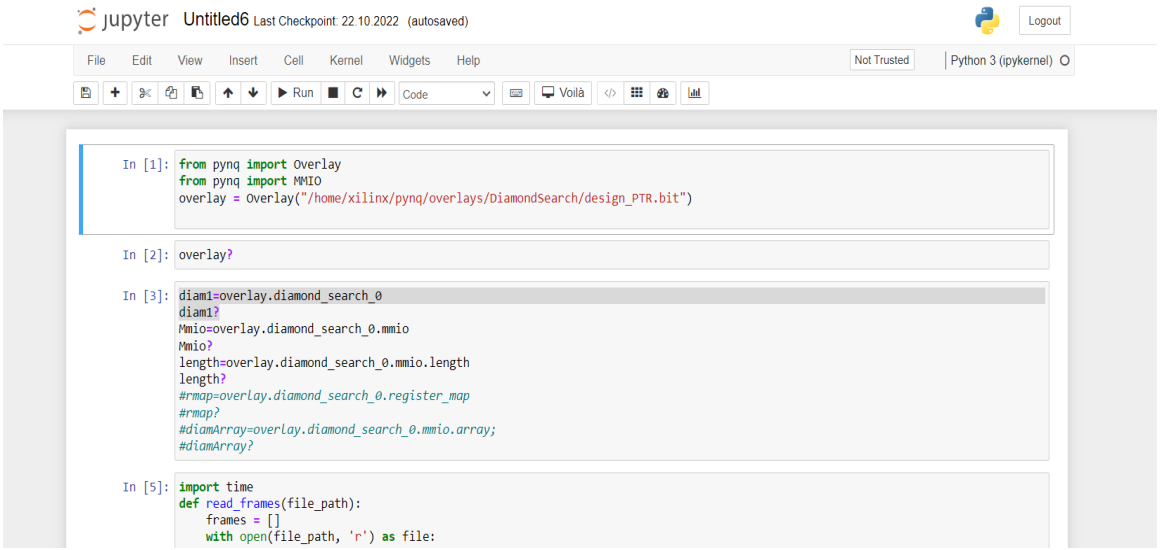


Slika 3.5 PYNQ Z1 FPGA pločica[8]

3.4. Jupyter Notebook

Posljednje okruženje potrebno za ostvarivanje cilja je Jupyter Notebook *open-source* alat. To je interaktivno okruženje za programiranje koje se često koristi za analizu podataka, i vizualizaciju te, iako podržava različite više programske jezike, najčešće se koristi u kombinaciji sa Python-om. Time se omogućuje pristup Python bibliotekama kao što su „NumPy“, „Matplotlib“ za matematičke operacije te „TensorFlow“ i „PyTorch“ za duboko učenje. Jupyter bilježnice organiziraju izvršavanje koda u ćelijama čime se omogućava eksperimentiranje sa pojedinim dijelovima koda i dodatna interaktivnost.

Jupyter Notebook će služiti kao sučelje putem kojeg će se odvijati komunikacija s registrima definiranim u mapi koju je generirao Vitis. Prikaz jedne bilježnice može se vidjeti na slici (Slika 3.6)



Slika 3.6 Grafičko sučelje Jupyter bilježnice

Kako bismo omogućili korištenje registara potrebno je uključiti binarnu datoteku (bitstream) generiranu u Vivadu kao shemu (engl. *Overlay*). Na taj način pločica može simulirati funkcionalnost koja je od nje tražena. Shema u sebi sadrži sve IP blokove, a nama je potreban diamond_search_0 jer on predstavlja algoritam koji testiramo. Nakon uključivanja, potrebno je vidjeti parove varijabli i memorijskih lokacija u registarskoj mapi iz Vitisa te se na te lokacije upisuju vrijednosti parametara jedne od funkcija procjene pokreta. Dio koda može se vidjeti na slici (Slika 3.7)

Vrijednosti koje su povezane sa registrima predstavljaju dio programa koji se pokreće na programskoj logici (PL) tj. FPGA dijelu pločice. Jupyter bilježnice se nalaze u memoriji PYNQ pločice pa se izvršavaju na procesorskom sustavu (PS) što je ustvari ARM Cortex A9 koji obavlja funkcije kao svaki opće namjenski CPU.

```
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)
ref=frame_1d[i]
pic=frame_1d[i+1]
h=0
```

Slika 3.7 Upisivanje vrijednosti u registre

4. Mjerenje vremenskih rezultata u C-u

Kako bi mogli usporediti vremena izvođenja potrebno je izmjeriti vremenske rezultate originalnog C izvornog koda svih algoritama. S obzirom na to da želimo vidjeti isključivo vrijeme procesorskog izvođenja programa, izvorni kodovi bit će modificirani na način da će svi ispisi vektora i drugih vrijednosti varijabli biti isključeni. Time osiguravamo maksimalnu brzinu i preciznije rezultate.

Specifikacije računala na kojem se mjerenja obavljaju su:

- CPU – AMD Ryzen 5 1600 sa taktom od 3.2 GHz
- GPU – Nvidia Geforce GTX 1060
- RAM – 16 Gb DDR4 sa frekvencijom od 3200 MHz

4.1. Vrijeme izvođenja „Diamond Search“ funkcije

`Diamond_search` funkcija iz poglavlja 2.1 bit će prva testirana funkcija. Veličina sličice će biti 64 x 64 elemenata slike koji su predstavljeni jednim cijelim brojem između 0 i 255. Taj broj će označavati monokromatski intenzitet svakog elementa. Veličina bloka je 8 x 8 elemenata što znači da će se za svaku sličicu izračunati 64 vektora pomaka. Ukupan broj sličica M će biti 10, a broj koraka `steps` koji predstavlja maksimalan broj iteracija kroz koje algoritam prolazi će biti 8. Program će se pokrenuti 10 puta kako bismo dobili statističke vrijednosti. Nakon prikupljanja podataka koristimo formulu (2) za računanje srednje vrijednosti skupa podataka te formule (3) i (4) za računanje varijance i standardne devijacije.

$$\mu = \sum (x_i) / N \tag{2}$$

$$s^2 = \sum (x_i - \mu)^2 / (N-1) \tag{3}$$

$$\sigma = \sqrt{s^2} \qquad (4)$$

Tablica 4.1 Vrijednosti rezultata `diamond_search` funkcije

Iteracija	Vrijeme u sekundama [s]
I=0	0.001172
I=1	0.001207
I=2	0.001156
I=3	0.001138
I=4	0.001138
I=5	0.001171
I=6	0.001138
I=7	0.001142
I=8	0.001139
I=9	0.001139

Srednja vrijednost izračunata izrazom (2) je:

$$\mu = 0.001154 \text{ [s]}$$

Varijanca skupa podataka izračunata izrazom (3) je:

$$s^2 = 5.32e\text{-}10 \text{ [s]}$$

Standardna devijacija izračunata izrazom (4) je:

$$\sigma = 2.31e\text{-}5 \text{ [s]}$$

4.2. Vrijeme izvođenja „User Input Search“ funkcije

Sljedeća funkcija podvrgnuta mjerenju vremena je `user_input_search` iz poglavlja 2.2. Parametri su isti: veličina sličice će biti 64 x 64 elemenata slike koji su predstavljeni jednim cijelim brojem između 0 i 255. Veličina bloka je 8 x 8 elemenata što znači da će se za svaku sličicu izračunati 64 vektora pomaka. Ukupan broj sličica M će biti 10, a broj koraka 8. Program će se pokrenuti 10 puta kako bismo dobili statističke vrijednosti. Nakon prikupljanja podataka koristimo formulu (2) za računanje srednje vrijednosti skupa podataka te formule (3) i (4) za računanje varijance i standardne devijacije.

Tablica 4.2 Vrijednosti rezultata `user_input_search` funkcije

Iteracija	Vrijeme u sekundama [s]
I=0	0.000463
I=1	0.000538
I=2	0.000462
I=3	0.000460
I=4	0.000483
I=5	0.000461
I=6	0.000461
I=7	0.000477
I=8	0.000460
I=9	0.000462

Srednja vrijednost izračunata izrazom (2) je:

$$\mu = 0.0004727 \text{ [s]}$$

Varijanca skupa podataka izračunata izrazom (3) je:

$$s^2 = 5.90e\text{-}10 \text{ [s]}$$

Standardna devijacija izračunata izrazom (4) je:

$$\sigma = 2.43e\text{-}5 \text{ [s]}$$

Iz rezultata je vidljivo kako je vrijeme izvođenja ove funkcije manje u usporedbi sa izvođenjem funkcije `diamond_search`. Prisjetimo li se načina na koji algoritam radi, logično je zaključiti kako je to ispravno. Funkcija `user_input_search` pretražuje polje oblikom dijamanta, ali umjesto pretraživanja svih četiriju smjerova kao `diamond_search`, koristi se informacija o smjeru pomicanja slike u parametru `direction` što omogućuje algoritmu da pretraži samo jedan smjer. Time se postiže značajna ušteda vremena što se vidi u testnim podacima.

4.3. Vrijeme izvođenja „Three step Search“ funkcije

Posljednja funkcija za koju je potrebno izmjeriti vrijeme je `Three_Step_Search` iz poglavlja 2.3.1. Parametri predani funkciji će biti isti kao i za prijašnje funkcije: veličina slike 64x64 te veličina bloka 8x8. Algoritam ne prima broj koraka kao parametar, a `areaSize` je postavljen na 4. Program će se pokrenuti 10 puta.

Tablica 4.3 Vrijednosti rezultata `Three_Step_Search` funkcije

Iteracija	Vrijeme u sekundama [s]
I=0	0.004095
I=1	0.003960
I=2	0.003820
I=3	0.003812
I=4	0.003924

I=5	0.003762
I=6	0.003930
I=7	0.003978
I=8	0.003731
I=9	0.003718

Srednja vrijednost izračunata izrazom (2) je:

$$\mu = 0.003873 \text{ [s]}$$

Varijanca skupa podataka izračunata izrazom (3) je:

$$s^2 = 1.52e\text{-}8 \text{ [s]}$$

Standardna devijacija izračunata izrazom (4) je:

$$\sigma = 0.00012327386 \text{ [s]}$$

5. Mjerenje vremenskih rezultata implementacije na FPGA

Posljednji korak u procesu je mjerenje vremenskih rezultata IP jezgre na FPGA pločici. Mjerenje se odvija u Jupyter bilježnici na način da se zapamti početno vrijeme u trenutku prije upisivanja u registar posljednje vrijednosti potrebne za izvršavanje funkcije i završno vrijeme u trenutku nakon čitanja vrijednosti rezultata. Mjerenje se provodi za sve 3 funkcije.

5.1. Diamond_search IP jezgra

Parametri koji se upisuju u registre (na memorijske lokacije) su identični kao u C verziji funkcije kako bi osigurali podjednake uvjete izvođenja. To uključuje i broj sličica (10) i jednake elemente sličica. Program će se pokrenuti 10 puta te će se nakon toga izračunati srednja vrijednost (2), varijanca (3) i standardna devijacija (4).

Tablica 5.1 Vrijednosti rezultata `diamond_search` IP jezgre

Iteracija	Vrijeme u sekundama [s]
I=0	0.001043
I=1	0.000993
I=2	0.000978
I=3	0.000948
I=4	0.000963
I=5	0.001055
I=6	0.001001

I=7	0.001052
I=8	0.001021
I=9	0.000961

Srednja vrijednost izračunata izrazom (2) je:

$$\mu = 0.0010015 \text{ [s]}$$

Varijanca skupa podataka izračunata izrazom (3) je:

$$s^2 = 1.57e\text{-}9 \text{ [s]}$$

Standardna devijacija izračunata izrazom (4) je:

$$\sigma = 3.96e\text{-}5 \text{ [s]}$$

Ako usporedimo rezultate s rezultatima iz C funkcije dobijemo:

0.001043 - 0.001172 = - 0.000129
0.000993 - 0.001207 = - 0.000214
0.000978 - 0.001156 = - 0.000178
0.000948 - 0.001138 = - 0.000190
0.000963 - 0.001138 = - 0.000175
0.001055 - 0.001171 = - 0.000116
0.001001 - 0.001138 = - 0.000137
0.001052 - 0.001142 = - 0.000090
0.001021 - 0.001139 = - 0.000118
0.000961 - 0.001139 = - 0.000178

Prosječno ubrzanje iznosi 0.0001525 [s].

5.2. User_input_search IP jezgra

Sljedeća IP jezgra koju testiramo je User_input_search. Parametri koje predajemo funkciji su identični kao u C verziji funkcije. Parametar `direction` postavlja se na 1 kako bi simulirali jedan smjer. Kao i u prošlom primjeru koristi se 10 sličica te se program ujedno ponavlja 10 puta.

Tablica 5.2 Vrijednosti rezultata `user_input_search` IP jezgre

Iteracija	Vrijeme u sekundama [s]
I=0	0.001150
I=1	0.001048
I=2	0.001069
I=3	0.001110
I=4	0.001075
I=5	0.001067
I=6	0.001115
I=7	0.001143
I=8	0.001069
I=9	0.001138

Srednja vrijednost izračunata izrazom (2) je:

$$\mu = 0.0010984 \text{ [s]}$$

Varijanca skupa podataka izračunata izrazom (3) je:

$$s^2 = 1.38e\text{-}9 \text{ [s]}$$

Standardna devijacija izračunata izrazom (4) je:

$$\sigma = 3.72e\text{-}5\ [s]$$

Ako usporedimo rezultate sa rezultatima iz C funkcije dobijemo:

0.001150 - 0.000463 = 0.000687
0.001048 - 0.000538 = 0.000510
0.001069 - 0.000462 = 0.000607
0.001110 - 0.000460 = 0.000650
0.001075 - 0.000483 = 0.000592
0.001067 - 0.000461 = 0.000606
0.001115 - 0.000461 = 0.000654
0.001143 - 0.000477 = 0.000666
0.001069 - 0.000460 = 0.000609
0.001138 - 0.000462 = 0.000676

Prosječno ubrzanje iznosi -0.0006257 [s] što zapravo znači da IP jezgra radi sporije od izvornog koda.

5.3. Three_step_search IP jezgra

Mjerenje vremena za posljednju jezgru ne može se odrediti na isti način kao za prethodne dvije. Slika (Slika 5.1) prikazuje razlog. Ovaj algoritam računa blokove za usporedbu izvan funkcije što znači da se vrijeme mora računati i zbrojiti za sve blokove. To znači više pokretanja štoperice i više uzastopnih čitanja i pisanja u registre. Vremena su mjerena s obzirom na te okolnosti.

```
start_cpu_time = time.process_time()
diam1.write(0x30,offsetY * 64 + offsetX) #starting index
diam1.read(0x44)
diam1.read(0x54)
end_cpu_time = time.process_time()
```

Slika 5.1

Koristi se 10 sličica veličine 64x64 elemenata sa blokovima od 8x8 elemenata. Vremena su sljedeća:

Tablica 5.3 Vrijednosti rezultata `Three_step_search` IP jezgre

Iteracija	Vrijeme u sekundama [s]
I=0	0.053105
I=1	0.054223
I=2	0.054742
I=3	0.053708
I=4	0.055003
I=5	0.052139
I=6	0.056256
I=7	0.053823
I=8	0.053823
I=9	0.055410

Srednja vrijednost izračunata izrazom (2) je:

$$\mu = 0.0542232 \text{ [s]}$$

Varijanca skupa podataka izračunata izrazom (3) je:

$$s^2 = 1.40e\text{-}6 \text{ [s]}$$

Standardna devijacija izračunata izrazom (4) je:

$$\sigma = 0.00118312 \text{ [s]}$$

Ako usporedimo rezultate sa rezultatima iz C funkcije dobijemo:

$$0.053105 - 0.004095 = 0.049010$$

$$0.054223 - 0.003960 = 0.050263$$

$$0.054742 - 0.003820 = 0.050922$$

0.053708 - 0.003812 = 0.049896

0.055003 - 0.003924 = 0.051079

0.052139 - 0.003762 = 0.048377

0.056256 - 0.003930 = 0.052326

0.053823 - 0.003978 = 0.049845

0.053823 - 0.003731 = 0.050092

0.055410 - 0.003718 = 0.051692

Prosječno ubrzanje iznosi - 0.0501502 [s] što zapravo znači da IP jezgra radi sporije od izvornog koda, no uzimamo u obzir ograničenje navedeno ranije u poglavlju.

5.4. Evaluacija rezultata

Kao što je vidljivo iz mjerenja, kod `diamond_search` funkcije postiže se ubrzanje od otprilike 0.1 milisekunde, a kod `user_input_search` IP jezgra obavlja rad sporije nego izvorna C funkcija za približno 0.6 milisekundi. Važno je napomenuti kako HLS sintezator samostalno odlučuje kako postaviti mape i rute u dizajnu te kako prevesti programski kod. Česti su slučajevi kada sintezator ne može odvrtiti (engl. *unroll*) ili izravnati (engl. *flatten*) petlju, što je potrebno zbog linearne prirode HDL jezika i paralelizma, pa se mora prilagođavati na način da se troši više od jednog ciklusa za jedan atom koda. Još jedan slučaj koji utječe na performanse jest kada sintezator ne može uspostaviti protočni tok (engl. *pipeline*) koji bi omogućio izvođenje više dijelova naredbi istodobno; takav slučaj se događa kada postoje petlje sa unaprijed nepoznatim brojem iteracija ili kada se događaju greške poput memorijske ovisnosti (engl. *memory dependency*) koje nastaju kada se u jednom procesorskom ciklusu pristupa memoriji više od 2 puta, a sabirnica dozvoljava samo 2 pristupa za čitanje i pisanje. To će iziskivati trošenje još jednog ciklusa što povećava vrijeme. Svi ovi slučajevi narušavaju optimizaciju što neposredno utječe na performanse i vrijeme izvođenja.

Mjerenje vremena kod `Three_step_search` IP jezgre pokazuje sporije izvođenje algoritma u odnosu na izvorni kod zbog drugačijeg načina čitanja registara i računanja blokova izvan funkcije za razliku od `diamond_search` algoritma.

Zaključak

Brzine kodiranja videa dosegle su vrlo visoke vrijednosti zahvaljujući algoritmima procjene pokreta. Izvorni kodovi u programskom jeziku C izvršavaju se u vrlo kratkim intervalima mjerenim u milisekundama što iskazuje kvalitetu performansi algoritama. Implementacija algoritama na FPGA platformi iziskuje korištenje posebnih programskih alata i interaktivnih okruženja kao što su Vitis i Vivado koji omogućavaju rad sa sklopovljem bez direktnog poznavanja jezika za opis sklopovlja ili same platforme. Implementacija može doprinijeti ubrzanju kod nekih algoritama, s napomenom kako je mjerenje točnih vremenskih intervala zahtjevno zbog prirode PYNQ okruženja i same pločice. Programska logika na pločici neposredno je povezana sa RISC (engl. *Reduced instruction set computer*) procesorskim sustavom koji će neizbježno usporavati sveukupno izvođenje IP jezgre. S tim na umu, implementacija algoritama na FPGA platformama sigurno olakšava korištenje programske funkcionalnosti u ugradbenim sustavima gdje nemamo luksuz korištenja kompleksnijeg i snažnijeg sklopovlja. Uostalom, na snažnijem sklopovlju koristili bi se izvorni kodovi HEVC kodera.

Literatura

[1] https://en.sdmctech.com/news/industry-knowledge_1803.html

[2] Myoung-Seo Kim, Cheong Ghil Kim, Cheong Ghil Kim, *A Slope-Based Search Technique for Block Motion Estimation* (2004)

[3] https://en.wikipedia.org/wiki/Block-matching_algorithm#Diamond_Search

[4] https://github.com/ultravideo/kvazaar

[5] https://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/AV0405/ZAMPOGLU/Hierarchicalestimation.html

[6] https://en.wikipedia.org/wiki/Block-matching_algorithm#Three_Step_Search

[7] C. S. Chee, A. B. Jambek, R. Hussin, *Review of energy efficient block-matching motion estimation algorithms for wireless video sensor networks*

[8] https://pynq.readthedocs.io/en/v2.2.1/getting_started/

[9] https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html

Sažetak

Ubrzavanje kodiranja videa na dron uređajima informacijama o kretanju drona.

Vremensko izvođenje kodiranja videa potrebno je smanjiti na najmanje moguće vrijednosti. To se postiže korištenjem algoritama procjene pokreta (engl. *Motion estimation*) gdje se pokušava smanjiti vremenska redundancija elemenata slike. Algoritmi procjene pokreta predstavljaju pretraživanje slike različitim oblicima: „Diamond Search“ u obliku dijamanta i „Three Step Search“ u osam smjerova u 2D koordinatnom sustavu slike. Implementacije kodera koje koriste ove algoritme su *open-source* koder Kvazaar te Bolt65. Također potrebno je implementirati algoritme na FPGA platformu u obliku pločice PYNQ Z1 kako bi se koristili u ugradbenim sustavima. Za taj zadatak koriste se posebni programski alati te, naposlijetku, usporedbom rezultata dolazimo do zaključaka.

H.265, kodiranje videa, FPGA, HLS, procjena pokreta, „Diamond Search“, Kvazaar, Bolt65

Summary

Accelerating video encoding on drone devices using information about the drone's movement.

The time required for video encoding needs to be reduced to the minimum possible values. This is achieved by using motion estimation algorithms, which aim to reduce the temporal redundancy of image elements. Motion estimation algorithms involve searching the image in different patterns: "Diamond Search" in a diamond shape and "Three Step Search" in eight directions in the 2D coordinate system of the image. Encoder implementations that use these algorithms include the open-source encoders Kvazaar and Bolt65. It is also necessary to implement these algorithms on the FPGA platform in the form of the PYNQ Z1 board to be used in embedded systems. Special software tools are used for this task, and finally, comparing the results will give us conclusions.

H.265, video encoding, FPGA, HLS, motion estimation, „Diamond Search“, Kvazaar, Bolt65