Univerza v Ljubljani

Fakulteta za elektrotehniko

Domen Ipavec

Strojna izvedba konvolucijske nevronske mreže na programirljivem vezju

Magistrsko delo

Mentor: izr. prof. dr. Andrej Trost

Ljubljana, 2018

Vsebina

Seznam slik

Seznam tabel

Seznam uporabljenih kratic

FPGA	Programirljivo polje logičnih vrat (ang. *Field-Programmable Gate Array*)
HLS	Visoko nivojska sinteza (ang. *High Level Synthesis*)
AXI	Napreden razširljiv vmesnik (ang. *Advanced eXtensible Interface*)
DMA	Direktni dostop do pomnilnika (ang. *Direct Memory Access*)
DSP	Digitalno procesiranje signalov (ang. *Digital signal processing*)
FIFO	Prvi noter in prvi ven (ang. *First In First Out*)
LUT	Vpogledna tabela (ang. *Lookup Table*)
ASIC	Integrirano vezje za določen namen (ang. *Application-Specific Integrated Circuit*)
GEMM	Splošno matrično množenje (ang. *GEneral Matrix to Matrix Multiplication*)
YOLO	Algoritem pogledaš samo enkrat (ang. *You Only Look Once*)
CNN	Konvolucijske nevronske mreže (ang. *Convolutional Neural Networks*)
ReLU	Usmerjena linearna enota (ang. *Rectified Linear Unit*)
IoU	Presek čez unijo (ang. *Intersection over Union*)
AP	Povprečna natančnost (ang. *Average Precision*)
mAP	Srednja povprečna natančnost (ang. *mean Average Precision*)

Povzetek

YOLO je algoritem za prepoznavanje in določanje lokacije predmetov na slikah. Za to uporablja konvolucijske nevronske mreže. Je računsko precej zahteven, zato bi radi zanj izkoristili FPGA vezja. V magistrski nalogi je opisana implementacija algoritma v jeziku C++ z uporabo visoko nivojske sinteze. Primerjana sta hitrost algoritma na procesorju ARM in v FPGA vezju. Prav tako so raziskani učinki uporabe različne velikosti podatkovnih tipov za računanje. Z uporabo števil z nepremično decimalno vejico velikosti 24-bitov, ki so optimalna, dosežemo na majhnem FPGA vezju na ZedBoard-u, bistveno hitrejšo implementacijo kot na primerljivem procesorskem sistemu.

Ključne besede: Konvolucijske nevronske mreže, YOLO, FPGA, HLS

Abstract

YOLO is a system for detection and localization of objects in images using convolutional neural networks. It is computationally intensive, so we want to use FPGAs for better performance. This master's thesis describes the implementation of the algorithm in C++ using high level synthesis. The speed of the algorithm is compared between software implementation on ARM processor and the FPGA. The effects of using different data types and sizes on computation are also explored. Using 24 bit fixed point numbers, that are optimal, on the small FPGA chip available on ZedBoard, we can achieve significantly faster implementation than a comparable processor system.

Key words: Convolutional neural networks, YOLO, FPGA, HLS

1 Uvod

Na področju strojnega učenja se v zadnjem času vedno bolj pogosto uporabljajo nevronske mreže. Specifično na področju strojnega vida in prepoznavanju slik so najbolj pogoste konvolucijske nevronske mreže. Eden izmed takih algoritmov je algoritem YOLO [1], ki je namenjen prepoznavanju in lokalizaciji predmetov na slikah.

Večinoma taki algoritmi tečejo na močnih strežniških sistemih in grafičnih karticah. Vendar je za veliko aplikacij, zakasnitev ki jo predstavlja omrežje do strežnika lahko ovira. V takih primerih je potrebno lokalno izvajanje algoritma. Za to ponavadi potrebujemo močne procesorske sisteme.

V tej nalogi bomo raziskali primernost programirljivih logičnih vrat (FPGA) za implementacijo algoritma YOLO. FPGA vezje uporabljeno v tej nalogi je Xilinxov sistem na čipu Zynq-7000, ki vsebuje kombinacijo procesorja ARM in vezja FPGA. Uporabljena je preizkusna plošča ZedBoard. Za implementacijo bomo uporabili Xilinxov sistem za sintezo visoko nivojskega jezika C++ v FPGA vezje.

2 Konvolucijske nevronske mreže

Pri nadzorovanem strojnem učenju, se algoritem iz velike količine označenih primerov nauči (prilagodi vrednosti uteži), kako izgledajo željene izhodne vrednosti. Tako lahko dobro naučen algoritem na novih primerih izračuna uporabne rezultate. Konvolucijske nevronske mreže so ena najbolj pogosto uprabljenih metod za nadzorovano strojno učenje na področju prepoznavanja slik, saj ponujajo precej prednosti pred klasičnimi nevronskimi mrežami. V tem poglavju jih bom predstavil na splošno, v naslednjem pa bom predstavil specifičen algoritem, ki sem ga preizkusil na FPGA vezju.

2.1 Nevron

Glavni gradnik nevronskih mrež je posamezen nevron. Nevron si lahko predstavljamo kot matematično funkcijo, saj na podlagi vhodnih podatkov izračuna izhodno vrednost. Vhodni podatek za nevron je vektor vhodnih vrednosti x. Na podlagi vektorja uteži w, korekcije b in z aktivacijsko funcijo A, se izračuna izhodna vrednost y po enačbi (2.1).

$$y = A(w^T x + b) \tag{2.1}$$

Glede na območje željenih izhodnih vrednosti in položaja nevrona v nevronski mreži, lahko izbiramo med različnimi aktivacijskimi funkcijami:

- **Identiteta:** $f(x) = x$

  Omogoča poljubne izhodne vrednosti. Ni primerna za notranje plasti ne-

vronski mrež, saj se v primeru linearne aktivacijske funkcije posamezne plasti preprosto poenostavijo v eno linearno plast.

- **Sigmoidna funkcija:** $f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$
  izhodne vrednosti med 0 in 1. Pogosto se uporablja za klasifikacijo izhodnih vrednosti v dva razreda ali določanje verjetnosti. Je manj primerna za notranje plasti globokih nevronski mrež, saj odvod hitro postane zelo majhen, kar upočasni učenje.

- **Usmerjena linearna enota (ReLU):** $f(x) = max(0, x)$
  Izhodne vrednosti med 0 in ∞. Uporablja se za pozitivne izhodne vrednosti in notranje plasti globokih nevronskih mrež.

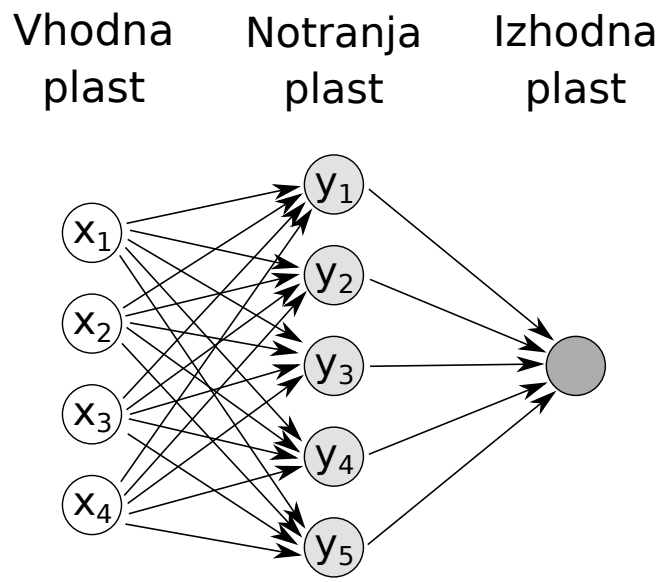
- **Prepustna ReLU:** $f(x) = max(0.01x, x)$
  Reši problem izginjanja odvoda za negativne vrednosti. Uporablja se predvsem za notranje plasti globokih nevronski mrež.

2.2 Nevronska mreža

Več nevronov združimo v plast nevronske mreže. Vsi nevroni v eni plasti delujejo na istih vhodnih vrednostih x z različnimi utežmi w_j. Uteži nevronov v posameznih plasteh lahko združimo v matriko uteži W, izhodne vrednosti v vektor y. Tako nevronsko plast lahko tako predstavimo s preprosto matrično enačbo (2.2).

$$y = W^T x + b \tag{2.2}$$

Preproste nevronske mreže vsebujejo eno notranjo plast nevronov. Izhodne vrednosti notranje plasti nevronov nato služijo kot vhodne vrednosti za izhodne nevrone kot kaže slika 2.1.

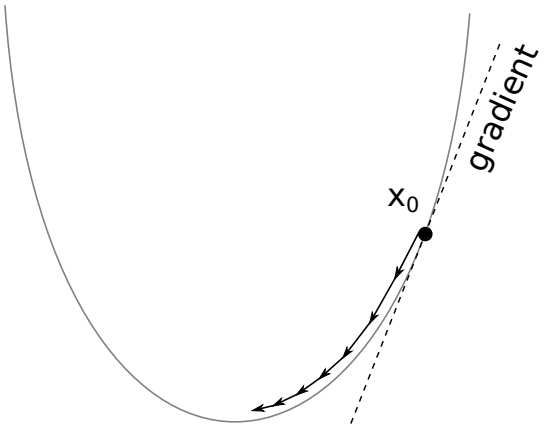


Slika 2.1: Preprosta nevronska mreža

2.3 Učenje nevronskih mrež

Za pravilno delovanje je potrebno algoritem najprej naučiti na označenih primerih. V tej nalogi na FPGA vezjih implementiram že naučen algoritem YOLO, za katerega so naučene uteži na voljo na spletu, zato bom postopek učenja predstavil le na kratko. Najprej nastavimo vse uteži na naključne vrednosti, izračunamo funkcijo napake na označenih primerih, nato pa za en korak gradientnega spusta s postopkom vzvratnega razširjanja popravimo uteži. Ponovno izračunamo funkcijo napake in naredimo določeno število ponovitev.

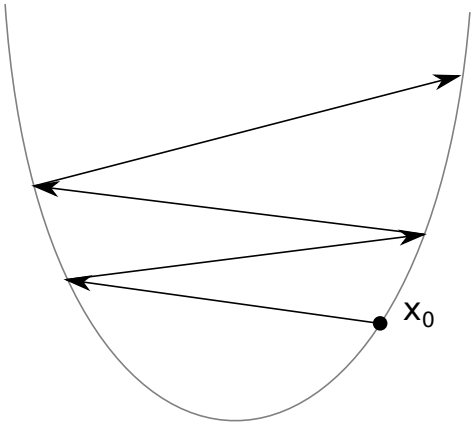
Gradientni spust je postopek računanja minimuma funkcije $F(x)$ več spremenljivk. Funkciji F se vrednost najhitreje zmanjšuje v nasprotni smeri gradienta ∇F. Gradientni spust začne v naključno izbrani začetni točki x_0 in se spušča v nasprotni smeri gradienta proti minimumu, po enačbi (2.3). Slika 2.2 prikazuje gradientni spust v dveh dimenzijah. Parameter α se imenuje hitrost učenja in ga lahko prilagajamo. Prevelika hitrost učenja lahko povzroči, da se x_n oddaljuje od minimuma (primer kaže slika 2.3). Premajhna pa lahko nepotrebno poveča



Slika 2.2: Gradientni spust

čas učenja. Ker algoritem vedno nadaljuje proti manjšim vrednostim, je lahko končni rezultat lokalni minimum. V praksi za nevronske mreže to ne predstavlja problema [2].

$$x_{n+1} = x_n - \alpha \nabla F(x_n) \tag{2.3}$$



Slika 2.3: Gradientni spust - oddaljevanje

Funkcija napake nam z eno vrednostjo pove, kako dober je razultat nevronske mreže v primerjavi s pravilnimi vrednostmi. Za to lahko uporabimo srednji kva-

dratni pogrešek, ki ga izračunamo po enačbi (2.4). y je rezultat nevronske mreže, $\hat{y}$ pa pravilna vrednost.

$$C = \sum (y - \hat{y})^2 \tag{2.4}$$

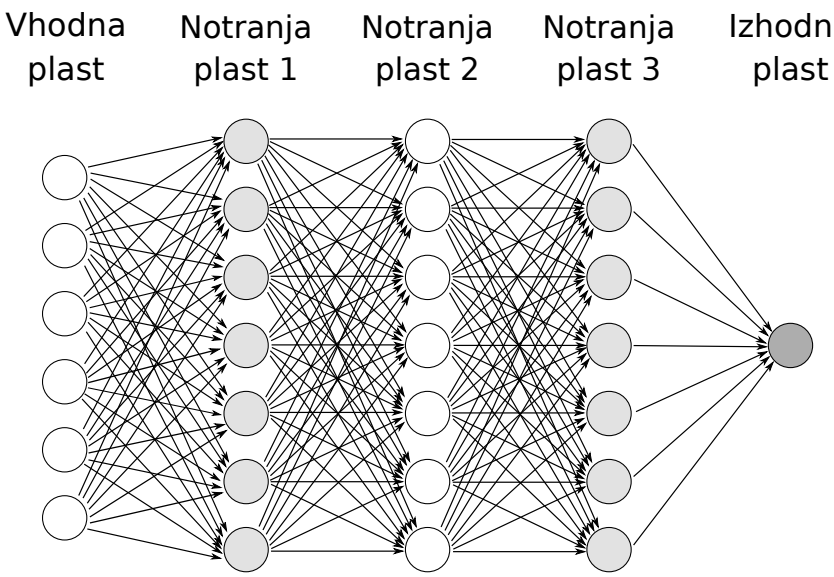
Vzvratno razširjanje je postopek računanja gradienta napake glede na funkcijo napake. Z njim lahko popravimo uteži v večplastnih nevronski mrežah glede na napako izhodnih vrednosti.

2.4 Globoke nevronske mreže

Globoke nevronske mreže vsebujejo več notranjih plasti nevronov. Primer prikazuje slika 2.4. Njihova glavna prednost je, da lahko izhodne vrednosti predstavijo kot bolj kompleksne funkcije vhodnih vrednosti. Imajo pa tudi več slabosti: so računsko bolj zahtevne in bolj so nagnjene k prevelikemu prilagajanju vhodnim podatkom - to se kaže v slabih rezultatih pri novih primerih vhodnih podatkov. Prevelikemu prilagajanju vhodnim podatkom se lahko izognemo z večjo količino označenih primerov za učenje in z uporabo regularizacije.

Za uporabo in predvsem učenje globokih nevronskih mrež, potrebujemo veliko količino vhodnih podatkov in močne računalnike. Oboje je v zadnjih nekaj letih postalo dovolj dostopno, da se je uporaba globokih nevronski mrež močno povečala.

Hierarhična zgradba globokih nevronskih mrež jim omogoča detekcijo postopno bolj kompleksnih oblik. Pri prepoznavanju slik, si lahko na primer predstavljamo, da prvi nivoji zaznajo preproste oblike kot so robovi, naslednji nivo bolj kompleksne oblike na primer oči in nos. Zadnji nivo pa prepozna celoten željen predmet, na primer obraz.

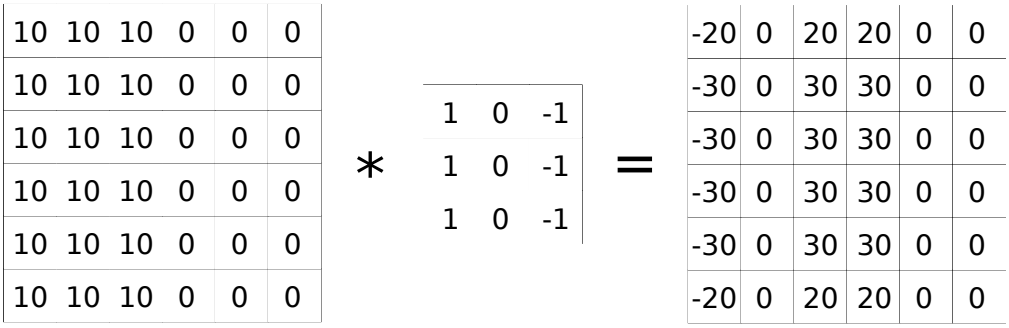


Slika 2.4: Globoka nevronska mreža

2.5 Konvolucijske nevronske mreže

Pri prepoznavanju slik potrebujemo za nevronske mreže ogromno število uteži, saj so za vsako piko na barvni sliki potrebne 3 vhodne vrednosti v prvi plasti. Za primer že pri barvni sliki majhne resolucije 100x100 ima prva plast 30000 uteži. Drugi problem je prostorska neodvisnost, če zaznavamo obraz nam je vseeno, ali je ta malo premaknjen levo ali desno. Klasične nevronske mreže prostorsko neodvisnost težko zagotovijo.

Oba problema lahko rešimo z uporabo konvolucijskih nevronskih mrež. Konvolucijske nevronske mreže iste uteži uporabijo na več vhodnih vrednostih. Uteži manjših dimenzij (na primer 3x3), premikamo čez vhodne podatke v dveh dimenzijah, in na vsaki koordinati seštejemo produkt uteži in pripadajoče vrednosti. Tako dobimo izhodne vrednosti iste dimenzije, glej sliko 2.5. Da ohranimo enako dimenzijo izhodnih podatkov, moramo vhodnim podatkom dodati rob ničel.



Slika 2.5: Konvolucija z robom ničel

Matematično lahko konvolucijo predstavimo z enačbo (2.5):

$$y_{ij} = \sum_{k=0}^{3} \sum_{l=0}^{3} w_{kl} \cdot x_{(i+k-1)(j+l-1)} \tag{2.5}$$

Uteži imajo isto globino kot vhodni podatki. Na primer za vhodne podatke dimenzije 100x100x5, bomo imeli uteži dimenzije 3x3x5. Tak sklop uteži v konvolucijskih nevronskih mrežah imenujemo filter. Vsak filter nam da izhodne podatke z globino 1. V vsaki plasti je ponavadi več filtrov. Njihovo število določa globino izhodnih vrednosti. Enačba (2.6) predstavlja konvolucijsko plast z več filtri:

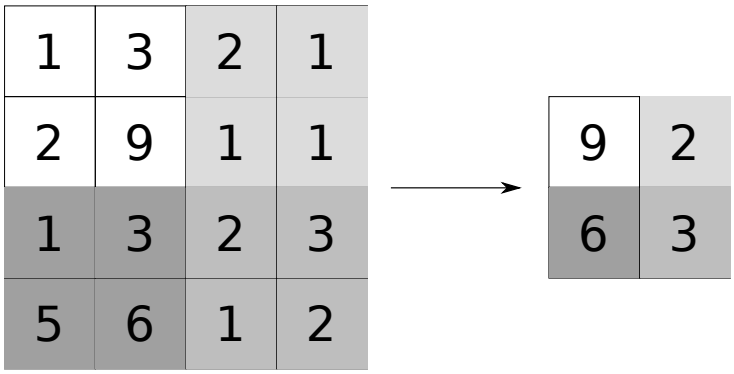
$$y_{kij} = \sum_{l=0}^{5} \sum_{m=0}^{3} \sum_{n=0}^{3} w_{lmn} \cdot x_{l(i+m-1)(j+n-1)} \tag{2.6}$$

Takoj lahko vidimo, da je pri konvolucijskih mrežah uteži precej manj. Za barvno sliko en filter potrebuje le 27 uteži. Ker iste uteži uporabimo čez celo sliko, to zagotavlja tudi prostorsko neodvisnost.

Po konvoluciji na izhodnih vrednostih tako kot pri klasičnih nevronskih mrežah uporabimo aktivacijsko funkcijo. Največkrat je to ReLU ali prepustna ReLU.

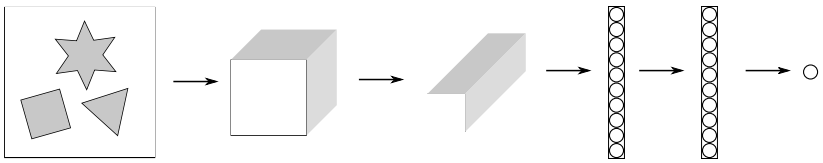
Ker pri konvoluciji ostajajo dimenzije enake, dodamo še združevalne plasti da razpolovimo dimenzije vhodnih podatkov. Te plasti delujejo na delčkih dimenzije 2x2. Lahko uporabimo maksimalno združevanje, kjer iz vsakega delčka vzamemo

največjo vrednost. Druga možnost pa je povprečno združevanje, kjer za izhod vzamemo povprečno vrednost. Slika 2.6 kaže primer maksimalnega združevanja.



Slika 2.6: Maksimalno združevanje

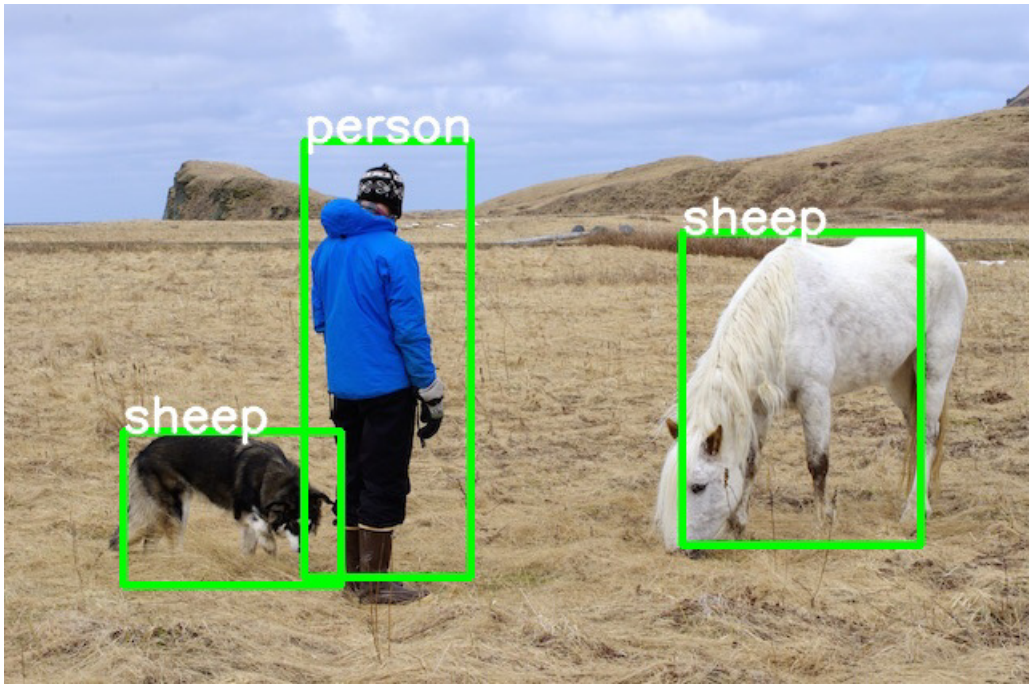
Konvolucijske nevronske mreže vsebujejo več konvolucijskih in združevalnih plasti. Na koncu pa ponavadi še nekaj polno povezanih plasti klasičnih nevronskih mrež, ki izračunajo izhodno vrednost. Slika 2.7 prikazuje primer take konvolucijske mreže.



Slika 2.7: Konvolucijska nevronska mreža

3 Algoritem YOLO

YOLO [1] je algoritem za detekcijo, prepoznavanje in lokalizacijo predmetov na slikah. Implementacija uporablja konvolucijske nevronske mreže. Prednost algoritma YOLO pred njegovimi predhodniki (npr. R-CNN) je, da uporablja samo en korak. Vhod v nevronsko mrežo je slika, izhod pa razred in lokacija predmetov. Slika 3.1 prikazuje primer delovanja algoritma YOLO s pravilno in tudi nepravilno prepoznanimi objekti.



Slika 3.1: Primer delovanja algoritma YOLO [1]

V tej nalogi bomo na vezjih FPGA implementirali algoritem Tiny YOLOv2.

To je manjša različica algoritma namenjena sistemom z omejeno procesorsko močjo.

3.1 Prepoznavanje objektov in določanje pozicije

Za detekcijo predmetov algoritem YOLO določi verjetnost, da je predmet na sliki: p.

Za prepoznavanje predmetov ima algoritem določeno število razredov, v katere razvrsti predmet. Za vsak razred i izračuna verjetnost, da predmet spada v ta razred c_i.

Pozicijo predmetov na sliki lahko določimo z metodo drsečega okna. Sliko razdelimo v mrežo oken. Za vsako okno poženemo detekcijo predmetov. Tako lahko določimo v katerem oknu se nahaja predmet.

Ker se pozicija in dimenzije predmetov ponavadi ne ujemajo z okni detekcije, si želimo boljše natančnosti. Zato algoritem YOLO izračuna tudi koordinate predmeta v oknu b_x in b_y ter dimenzije predmeta b_h in b_w.

Te vrednosti sestavimo v izhodni vektor y. Ker je lahko v posameznem oknu

več predmetov, se te vrednosti v izhodnem vektorju ponovijo večkrat:

$$y = \begin{bmatrix} p \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ \vdots \\ p \\ b_x \\ b_y \\ b_h \\ b_w \\ c_1 \\ c_2 \\ \vdots \end{bmatrix} \tag{3.1}$$

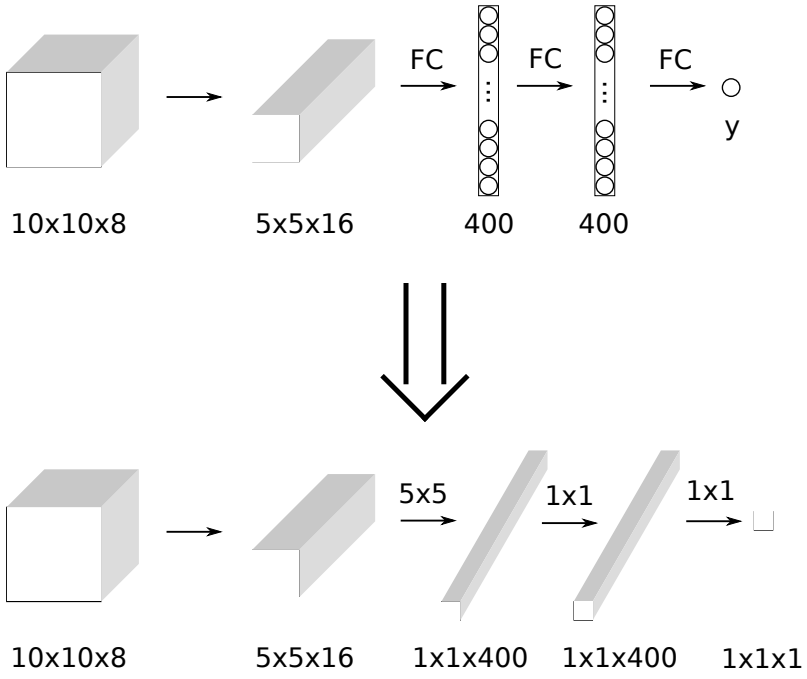
Algoritem YOLO lahko za vsako okno zazna več predmetov, ki se lahko med seboj tudi prekrivajo. Zato je treba izhodne podatke najprej obdelati. Najprej odstranimo vse predmete, ki imajo verjetnost p manjšo od izbranega praga - na primer 0.6. Nato pa izvedemo naslednji algoritem:

1. Izberemo predmet z največjo verjetnostjo p in ga izpišemo
2. Odstranimo vse predmete, ki se prekrivajo z izbranim
3. Če imamo še kakšen predmet, gremo na korak 1

3.2 Uporaba konvolucije za polno povezane plasti in drseče okno

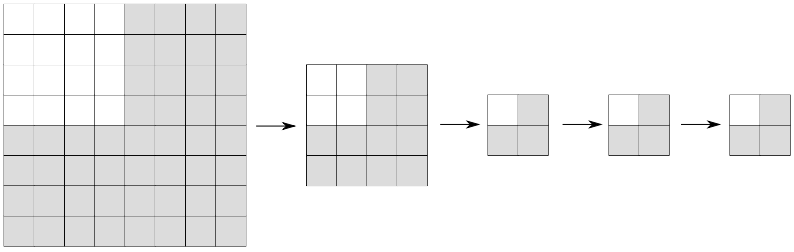
Izvajanje algoritma YOLO za vsako okno posebej, bi bilo kompleksno in računsko zahtevno. Vendar lahko za preprostejšo implementacijo drsečega okna dobro izkoristimo konvolucijo.

Najprej moramo vse polno povezane plasti spremeniti v konvolucijske plasti. Ugotovimo lahko, da je polno povezana plast klasičnih konvolucijskih mrež enakovredna konvolucijski plasti, ki ima dimenzije uteži enake dimenzijam vhodnih podatkov. Na primer za vhodne podatke dimenzije 5x5x16 vzamemo uteži dimenzije 5x5x16 in ne dodamo roba ničel. Tako dobimo izhod dimenzije 1x1. Primer take spremembe kaže slika 3.2.



Slika 3.2: Sprememba polno povezanih plasti v konvolucijo

Ko so vse plasti konvolucijske, lahko izhodni vektor preprosto povečamo v 2x2 matriko, kar ustreza 2x2 mreži drsečega okna na vhodni sliki. To kaže slika 3.3.



Slika 3.3: Povečanje izhodne matrike za implementacijo drsečega okna

Ker uteži pri tem postopku ostanejo enake, lahko s to metodo preprosto spreminjamo računsko zahtevnost modela in natančnost, brez da bi morali model ponovno naučiti. Če zmanjšamo velikost izhodne matrike s tem zmanjšamo računsko zahtevnost, vendar hkrati tudi zmanjšamo natančnost zaznave predmetov.

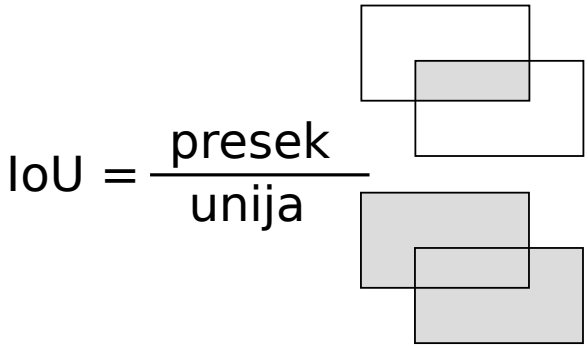
To je še posebej dobra lastnost za implementacijo na vezjih FPGA, saj lahko ob pomanjkanju sredstev na FPGA vezju preprosto zmanjšamo velikost algoritma.

3.3 Merjenje natančnosti algoritma

Ovrednotenje rezultatov pri algoritmih za detekcijo in lokalizacijo predmetov na slikah ni preprosto, saj moramo ovrednotiti dve različni stvari. Kako dobro algoritem zazna predmete in kako dobro določi njihovo pozicijo. Poleg tega moramo ovrednotiti še določanje razreda predmeta. Radi bi uporabili mero, ki nam z eno številko pove, kako dober je algoritem.

Presek čez unijo (IoU) je mera, ki določi kako dobro je bila določena pozicija predmeta. Je razmerje med površino preseka zaznanega območja predmeta in pravilnega območja ter njuno unijo. Primer kaže slika 3.4. Pravilno zaznane predmete določimo tako, da si izberemo mejo, nad katero mora biti mera IoU - ponavadi 0.5 ali več.

Natančnost je razmerje med pravilnimi zaznavami predmeta (pravilno pozi-



Slika 3.4: Presek čez unijo

tivni) in celotnim številom predmetov, ki jih je algoritem zaznal, (vsi pozitivni). Če je natančnost 1, so vsi zaznani predmeti pravilni.

Priklic je razmerje med pravilnimi zaznavami predmeta (pravilno pozitivni) in celotnim številom predmetov v testnem naboru podatkov. Če je priklic 1, so zaznani vsi predmeti v naboru podatkov.

Priklic in natančnost sta v obratnem razmerju. Če povečamo prag verjetnosti p za zaznavo predmeta, se natančnost poveča (zaznamo manj predmetov, od tega več pravilnih), priklic pa zmanjša (zaznamo manj predmetov, pravilnih pa ostane enako). Slika 3.5 prikazuje natančnost v odvisnosti od priklica pri različnih pragih za verjetnost.

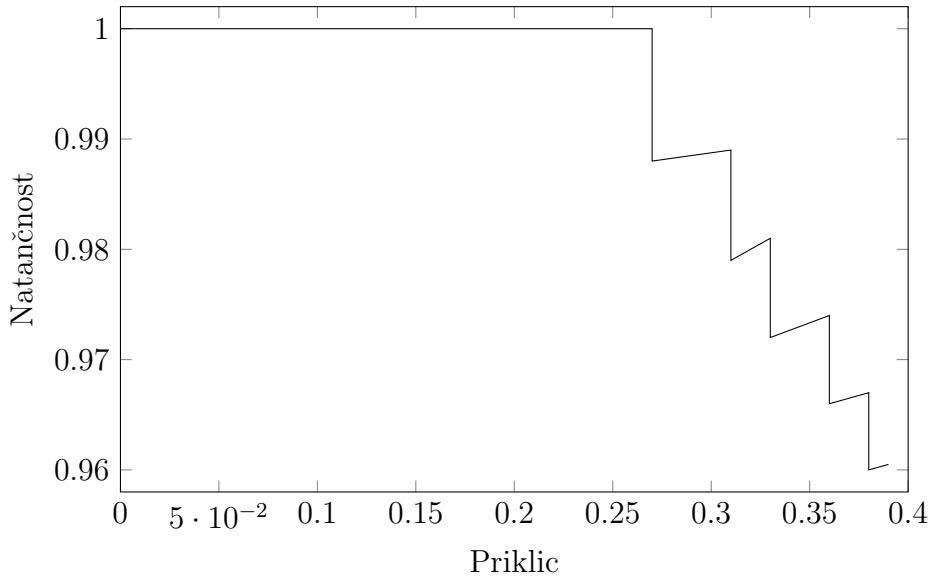
Povprečna natančnost (AP) je povprečje interpolirane vrednosti natančnosti $N_{interp}(p)$ pri 11 vrednostih priklica: $p_i = [0, 0.1, 0.2, ..., 1.0]$:

$$AP = \frac{1}{11} \sum_{p_i} N_{interp}(p_i) \tag{3.2}$$

Interpolirana vrednost natančnosti pri vrednosti priklica je definirana kot največja izmerjena vrednost natančnosti $N_(p)$ pri priklicu večjem ali enakem tej vrednost priklica:

$$N_{interp}(p) = \max_{\tilde{p}:\tilde{p} \geq p} N(\tilde{p}) \tag{3.3}$$

Srednja povprečna natančnost (mAP) [3] je izračunana kot srednja vrednost povprečnih natančnosti za vse razrede predmetov. Nekatere novejše verzije te



Slika 3.5: Natančnost v odvisnosti od priklica za letala pri algoritmu YOLO

mere pa upoštevajo še različne meje IoU in v mAP povprečijo rezultate pri teh različnih mejah.

4 Implementacija na vezju FPGA

Pri izbiranju načina implementacije algoritma na vezju FPGA sem izbiral med strojno opisnim jezikom VHDL in uporabo visoko nivojske sinteze iz jezika C. Predvsem zaradi lažje uporabe decimalnih podatkovnih tipov - števil z plavajočo vejico in števil z nepremično vejico sem se odločil za visoko nivojsko sintezo.

Vsa koda je na voljo na GitHub-u: https://github.com/matematik7/cnn-magistrska [9].

Visoko nivojska sinteza (HLS) [5] omogoča implementacijo kompleksnejših algoritmov napisanih v C/C++ jezikih na vezjih FPGA. Uporabimo lahko večino funkcij jezika C, razen tistih ki potrebujejo interakcijo z operacijskim sistemom - na primer dinamični pomnilnik in operacije z datotekami. Za računanje s plavajočo vejico lahko preprosto uporabimo tip float, za števila z nepremično vejico pa HLS ponuja razred ap_fixed. Kako se algoritem pretvori v strojno implementacijo, lahko v HLS-u dodatno kontroliramo z direktivami pragma.

Primer preprostega algoritma v HLS:

```
1 void primer(float a, float b, float &c) {
2 #pragma HLS INTERFACE s_axilite port=return bundle=CTRL_BUS
3 #pragma HLS INTERFACE s_axilite port=a bundle=CTRL_BUS
4 #pragma HLS INTERFACE s_axilite port=b bundle=CTRL_BUS
5 #pragma HLS INTERFACE s_axilite port=c bundle=CTRL_BUS
6         c = a*b;
7 }
```

V vrstici 1 definiramo funkcijo, ki jo želimo implementirati v vezju FPGA. Vrstice 2-5 vsebujejo direktive pragma, ki določajo da so vhodi in izhodi funkcije na voljo

preko AXI vmesnika. Vrstica 6 pa vsebuje dejansko operacijo - množenje dveh števil s plavajočo vejico.

Še ena prednost visoko nivojske sinteze je preprosta simulacija. Ker je algoritem napisan v jeziku C, lahko v njem napišemo tudi simulacijo. Simulacija v okolju HLS je preprosto funkcija main. Ta algoritmu poda testne podatke in jih nato preveri. Tak program lahko nato poženemo na računalniku in preverimo pravilnost algoritma še pred sintezo.

Primer simulacije za preprost primer algoritma:

```
1 int main() {
2         float result;
3         primer(1.1, 2.2, result);
4         if (result != 2.42) {
5                 std::cerr << "Wrong result: " << result << std::endl;
6                 return 1;
7         }
8         return 0;
9 }
```

V 3. vrstici pokličemo funkcijo algoritma s testnimi vrednostmi. V vrstici 4 preverimo pravilnost rezultata. Če je rezultat pravilen funkcija main vrne 0, kar okolju HLS pove, da je simulacija končala brez napake. Drugače pa vrne 1, kar nakazuje napako v simulaciji.

4.1 Predstavitev sorodnega dela

V zadnjih 3 letih je bilo veliko raziskav usmerjenih v strojno pospeševanje konvolucijskih mrež tako v industriji kot v znanosti. Večina pospeševalnikov uporablja grafične kartice, vendar z uporabo namenskih vezij lahko dosežemo višje hitrosti [10]. Vezja FPGA ponujajo višje hitrosti in so še vedno prilagodljiva, podjetja Microsoft in Amazon jih ponujajo v svojih podatkovnih centrih. Podjetje Google pa ponuja namenske čipe ASIC, ki ponujajo najvišje hitrosti vendar imajo neprilagodljivo arhitekturo [11].

Ena od možnosti implementacije je z uporabo splošnih matričnih množenj

(GEMM) [12]. Ti omogočajo preprostejšo posplošitev različnih arhitektur nevronskih mrež na isto strojno izvedbo. Vendar pa z uvedbo dodatnih vhodov lahko privedejo do neučinkovite implementacije.

Konvolucijski algoritem lahko optimiziramo z Winograd transformacijo [13], hitro Fourierovo transformacijo [14] in drugimi metodami. Implementacijo algoritma na vezju lahko pospešimo z odvijanjem zank [15], cevovodi [16], uporabo optimalnih podatkovnih tipov [17] in odstranjevanjem nepotrebnih uteži [18]. Popularni pa postajajo tudi algoritmi, ki so posebej namenjeni implementaciji na vezjih FPGA, na primer binarne nevronske mreže [19].

4.2 Komponente FPGA vezij

FPGA vezja so sestavljena iz različnih komponent iz katerih sestavimo vezje, ki izvaja naš algoritem. Na vsakem FPGA čipu je navoljo le končno število vsake komponente, kar predstavlja glavno omejitev pri sintezi algoritma. Več kot imamo na voljo komponent na čipu, bolj lahko optimiziramo algoritem.

Glavne komponente so:

- vpogledne tabele (LUT): z njimi predstavimo poljubna logična vrata, to je glavna komponenta za izvajanje preprostih operacij.
- Flip Flop (FF): preproste spominske celice, ki omogočajo implementacijo registrov, ki so potrebni za kratkotrajno pomnenje podatkov med posameznimi operacijami z vpoglednimi tabelami.
- DSP blok: namenski blok za računske operacije, vsebuje množilnik in seštevalnik.
- Pomnilniški blok (BRAM): blok za shranjevanje podatkov, posamezen blok vsebuje 18 kilobitov.

Tabela 4.1 prikazuje število posameznih FPGA komponent, ki so navoljo na

nekaj FPGA vezjih družine Zynq. Na preizkusni plošči ZedBoard, uporabljeni v tej magistrski nalogi, je FPGA vezje Zynq Z-7020.

Tabela 4.1: Število komponent na voljo v FPGA čipu

Komponenta	Z-7007S	Z-7020	Z-7100
LUT	14 400	53 200	277 400
FF	28 800	106 400	554 800
DSP	66	220	2020
BRAM	100	280	1510

4.3 Implementacije algoritma YOLO in simulacija

Prva implementacija algoritma v C++ jeziku je dokaj preprosta. Za konvolucijo potrebujemo 6 nivojev zank: 3 za vse dimenzije vhodnih podatkov in 3 za dimenzije uteži. Operacije pa so preprosta množenja in seštevanja. Paziti je potrebno le na nekaj robnih pogojev, kjer je potrebno vzeti rob ničel. Konvoluciji sledi še algoritem maksimalnega združevanja. Ti dve operaciji ponovimo za vse plasti v algoritmu YOLO, in imamo delujoč algoritem.

Pravilnost implementacije sem preizkusil s primerjavo rezultatov z obstoječo implementacijo v jeziku Python. Vhodne in izhodne podatke ter uteži iz programa v Pythonu sem shranil v tekstovne datoteke, ki jih nato preberem v HLS simulaciji. V simulaciji najprej preberemo vhodne podatke in uteži, jih posredujemo implementaciji algoritma, nato pa primerjamo izhod z pravilnim izhodom iz Python programa.

Na porabo pomnilnika in drugih komponent FPGA vezij, precej vpliva uporabljen podatkovni tip za reprezentacijo števil. V programski implementaciji algoritma so uporabljena števila s plavajočo vejico, katerih operacije so v večini procesorjev dobro podprte. V FPGA vezjih pa so bolj učinkovita števila z nepremično vejico. V simulaciji sem najprej preveril delovanje različnih podatkovnih tipov. Tabela 4.2 prikazuje absolutne vrednosti razlik izhodnih vrednosti, upo-

rabo pomnilniških in DSP blokov v FPGA vezju in čas računanja v ciklih ure. Vidimo lahko, da plavajoča vejica porabi več množilnikov (DSP) v FPGA vezju in ima večjo zakasnitev, nepremična vejica pa ima slabšo natančnost. Odločil sem se za implementacijo z uporabo nepremične vejice velikosti 24 bitov, ker sem ocenil da je takšna napaka še sprejemljiva. Vendar pa je ocenjevanje primernosti podatkovnih tipov na podlagi razlike vrednosti zelo težko, zato sem v zadnjem delu te naloge za primerjavo podatkovnih tipov uporabil še mero mAP (glej 5.3).

Tabela 4.2: Primerjava podatkovnih tipov s simulacijo

Podatkovni tip	Razlika vrednosti	BRAM	DSP	Zakasnitev
plavajoča vejica 32 bitov	1e-7	8	5	51647168
plavajoča vejica 16 bitov	5e-4	4	4	56342171
nepremična vejica 16 bitov	5e-3	4	1	18782147
nepremična vejica 24 bitov	6e-4	6	1	23477150

Preprost algoritem sicer deluje pravilno, vendar se ne sintetizira v uporabno vezje, saj uporablja mnogo preveč FPGA pomnilnika. Pomnilnik na voljo v FPGA vezjih je precej omejen, algoritem YOLO pa potrebuje veliko vhodnih podatkov in uteži. Na srečo algoritem ne potrebuje vseh podatkov hkrati, tako lahko rešimo problem pomanjkanja pomnilnika z uporabo tokov podatkov.

4.4 Tokovi podatkov

Vivado HLS ponuja dobro podporo za tokove podatkov z uporabo razreda hls::stream [8]. Razred ponuja metodi za branje (read) in pisanje (write). Notranji tokovi so v FPGA vezju implemetirani z FIFO vrstami. Vhodne in izhodne tokove pa lahko priključimo direktno na AXI Stream tokovni vmesnik.

HLS ponuja tudi pragma direktivo DATAFLOW. Ta funkciji določi, da se vse podfunkcije programa izvajajo vzporedno, podatki pa med njimi tečejo po tokovih. To je primerno za aplikacije, v katerih nad podatki izvajamo več korakov procesiranja.

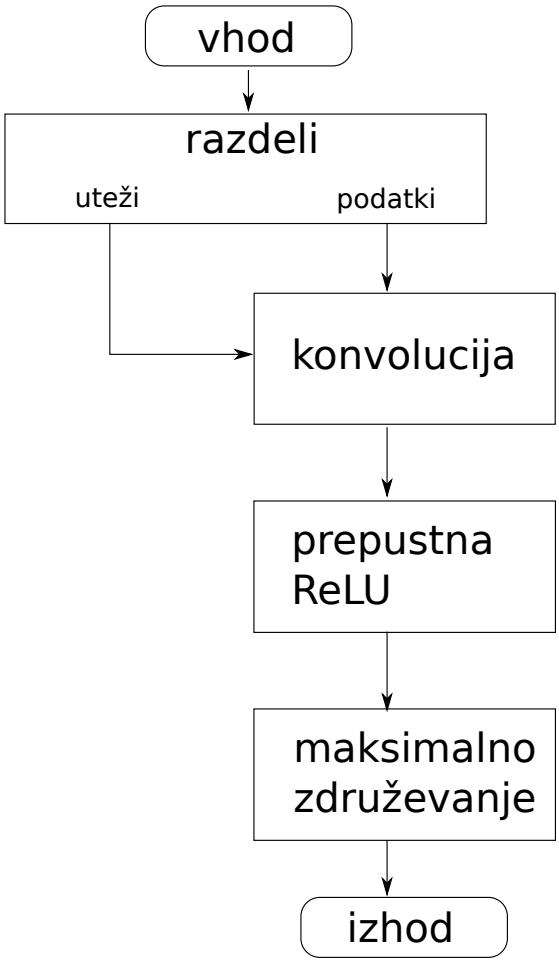
Primer tokovne implementacije v HLS:

```
1  void primer(hls::stream<float> &vhod, hls::stream<float> &izhod) {
2  #pragma HLS INTERFACE axis register both port=izhod
3  #pragma HLS INTERFACE axis register both port=vhod
4
5  #pragma HLS DATAFLOW
6
7          hls::stream<float> tok("tok");
8  #pragma HLS STREAM variable=tok depth=128 dim=1
9
10         korak1(vhod, tok);
11         korak2(tok, izhod);
12 }
13
14 void korak1(hls::stream<float> &vhod, hls::stream<float> &izhod) {
15         for (int i = 0; i < 100; i++) {
16                 izhod.write(vhod.read()*5);
17         }
18 }
19
20 void korak2(hls::stream<float> &vhod, hls::stream<float> &izhod) {
21         for (int i = 0; i < 100; i++) {
22                 izhod.write(vhod.read()+5);
23         }
24 }
```

V 1. vrstici definiramo funkcijo, ki bo implementirana na FPGA vezju. Vhodne in izhodne podatke določimo kot hls::stream tokove. V vrsticah 2 in 3 določimo, da naj bosta vhodni in izhodni tok povezana na AXI vmesnik. V 5. vrstici določimo, da je algoritem DATAFLOW, kar pomeni, da se funkciji korak1 in korak2 izvajata vzporedno in podatki med njima tečejo po tokovih. V 7 in 8 vrstici definiramo tok podatkov, ki ga uporabimo za pretok med obema korakoma. V direktivi pragma določimo globino FIFO vrste, ki je uporabljena za implementacijo toka. Korak 1 pomnoži vrednosti vhodnega toka s 5 in jih zapiše v izhodni tok. Korak 2 pa vrednostim prišteje 5.

Pri implementaciji algoritma YOLO, uporabimo DATAFLOW za vzporedno implementacijo konvolucije in maksimalnega združevanja. Shemo korakov in tokov za algoritem YOLO prikazuje slika 4.1. Vhodni tok najprej razdelimo na

uteži in podatke. Uteži shranimo v pomnilniku FPGA vezja, podatke pa obdelujemo sproti. Konvoluciji sledita še prepustna ReLU in maksimalno združevanje, nato pa podatki končajo v izhodnem toku.



Slika 4.1: Koraki algoritma YOLO s tokovi podatkov

4.5 Ponovna uporaba podatkov

Pri konvoluciji imamo dva vhodna tokova - vhodne podatke in uteži. Ne moremo narediti algoritma, ki bi oba tokova uporabljal zaporedno, brez shranjevanja podatkov v pomnilnik. Potrebno si je enega shraniti v FPGA pomnilnik. Najbolje

bi bilo shraniti tiste podatke, ki jih bomo uporabili največkrat in so dovolj majhni za FPGA pomnilnik. Tabela 4.2 prikazuje velikosti in število uporabe vhodnih podatkov in uteži za algoritem YOLO za vse plasti algoritma.

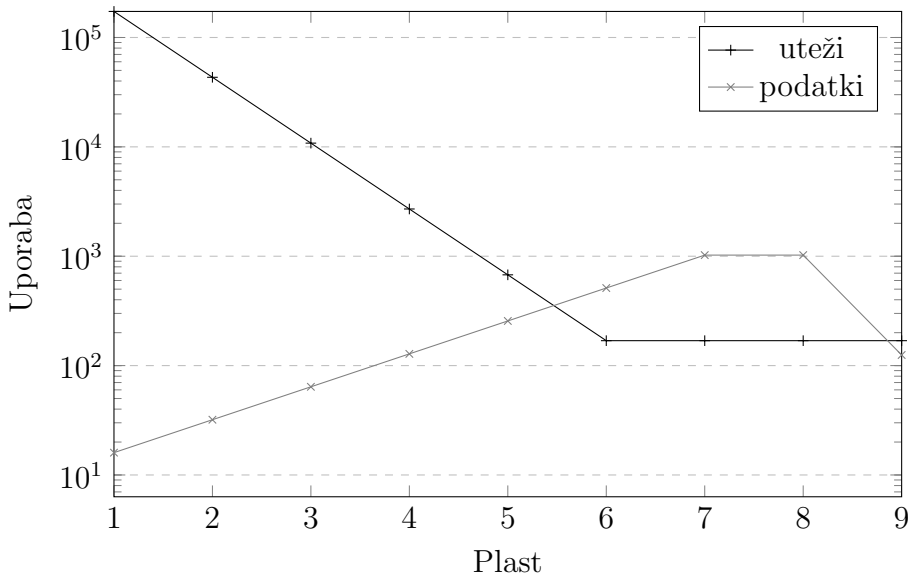
Tabela 4.3: Velikost in uporaba vhodnih podatkov in uteži za algoritem YOLO

Velikost	Globina	#Filtrov	#Uteži	#Podatkov	Uporaba uteži	Uporaba podatkov
416	3	16	432	519168	173056	16
208	16	32	4608	692224	43264	32
104	32	64	18432	346112	10816	64
52	64	128	73728	173056	2704	128
26	128	256	294912	86528	676	256
13	256	512	1179648	43264	169	512
13	512	1024	4718592	86528	169	1024
13	1024	1024	9437184	173056	169	1024
13	1024	125	128000	173056	169	125

Iz tabele lahko razberemo, da je za prve 4 plasti podatkov več od uteži in za prvih 5 plasti je število uporabe uteži večje od uporabe podatkov. Slika 4.2 prikazuje graf ponovne uporabe uteži in podatkov za vse plasti. Če količine podatkov in uteži primerjamo s pomnilnikom na voljo v FPGA vezju ugotovimo tudi, da je v prvih plasteh podatkov preveč za pomnilnik, v zadnjih plasteh pa je preveč uteži.

Zato je potrebno algoritem razdeliti na dva dela. Za prve plasti bomo uporabili algoritem, ki v pomnilnik shrani uteži in računa na toku vhodnih podatkov. Za zadnje 4 plasti pa algoritem, ki shrani v pomnilnik vhodne podatke in računa na toku uteži. Implementacija je v obeh primerih podobna, razlikuje se le v razvrstitvi izhodnih podatkov. V prvi implementaciji so izhodni podatki razvrščeni po izhodnih koordinatah, v drugi pa si sledijo izhodni podatki za vsak filter posebej. Posledica tega je, da je potrebno ob prehodu iz ene na drugo implementacijo, podatke pravilno razvrstiti. Taka implementacija nam omogoča, da zelo dobro izkoristimo pomnilnik, ki je navoljo v vezju FPGA.

Pri shranjenih utežeh v pomnilniku FPGA, algoritem procesira tok vhodnih



Slika 4.2: Uporaba podatkov za uteži in podatke

podatkov in nanj množi ustrezne uteži. Za vsako koordinato preberemo niz vhodnih podatkov, in izračunamo niz izhodnih podatkov za filtre, katerih uteži imamo v pomnilniku.

Pri shranjenih vhodnih podatkih v pomnilniku, pa algoritem procesira tok uteži, ki jih množi z ustreznimi vhodnimi podatki. Uteži so razvrščene po filtrih. Ko preberemo vse uteži filtra, dobimo niz izhodnih podatkov za ta filter.

4.6 Omejitve strojne implementacije

Seveda nobena implementacija ne gre brez problemov. Tukaj bom predstavil nekaj stvari, na katere sem naletel med implementacijo algoritma YOLO.

HLS vsako funkcijo sintetizira v vezju samo enkrat. To lahko privede do nepričakovanih rezultatov ali celo do obtičanja programa na mrtvi točki, če eno funkcijo v HLS programu uporabimo večkrat. Tega v sekvenčni programski simulaciji algoritma ne opazimo. Rešitev je več kopij funkcije v kodi, ali pa uporaba

predlog (template) funkcij z različnimi parametri. Primer:

```
1  void primer(hls::stream<float> &vhod, hls::stream<float> &izhod) {
2  #pragma HLS INTERFACE axis register both port=izhod
3  #pragma HLS INTERFACE axis register both port=vhod
4
5  #pragma HLS DATAFLOW
6
7          hls::stream<float> tok("tok");
8  #pragma HLS STREAM variable=tok depth=128 dim=1
9
10         korak<1>(vhod, tok);
11         korak<2>(tok, izhod);
12 }
13
14 template <int ID>
15 void korak(hls::stream<float> &vhod, hls::stream<float> &izhod) {
16         for (int i = 0; i < 100; i++) {
17                 izhod.write(vhod.read()*5);
18         }
19 }
```

V vrstici 14 definiramo predlogo funkcije s parametrom ID. Ko funkcijo uporabimo v vrsticah 10 in 11 z različnim ID-jem, dosežemo da HLS sintetizira dve različici funkcije.

Da algoritem spravimo na FPGA vezje, je potrebna čim bolj optimalna uporaba pomnilniških modulov na FPGA. Oba načina računanja konvolucije opisana v prejšnjem razdelku, morata uporabljati isti pomnilnik za shranjevanje vhodni podatkov ali uteži. Za implementacijo tega uporabimo razrede, ki so na voljo v jeziku C++. Pomnilnik definiramo kot lastnost razreda, obe implementaciji pa kot funkcije na razredu.

V 8. plasti algoritma je vhodnih podatkov 13x13x1024, kar je ravno preveč za pomnilnik na voljo v FPGA vezju. Problem sem rešil tako, da sem zmanjšal velikost vhodne slike tako, da so vhodni podatki v tej plasti 11x11x1024, za kar je pomnilnika dovolj. Zmanjšanje nam preprosto omogoča narava algoritma YOLO, kot je opisano v poglavju 3.2.

Pomnilnik je v FPGA vezjih razdeljen v bloke velikosti 16 kilobitov. V pro-

gramu pa ga definiramo kot večdimenzionalno polje. Velikost vsake dimenzije je pomembna, da lahko HLS polje optimalno razporedi v bloke pomnilnika v FPGA. 2048 24-bitnih vrednosti ravno zapolni 3 dele pomnilnika. Največja količina podatkov, ki jo moramo shraniti v pomnilnik, so vhodni podatki velikosti 11x11x1024. Eno od dimenzij 11 razdelimo v 2x6 in polje definiramo velikosti 2048x66. V programu preračunamo ustrezne indekse, da se prilagodijo tej velikosti.

Tudi za shranjevanje uteži je v pomnilniku premalo prostora. Pomnilnik zadostuje za uteži za 64 filtrov. Zato moramo plasti 4 in 5, ki vsebujejta več filtrov in se računata z metodo shranjevanja uteži v pomnilnik, računati v več korakih. V pomnilnik shranimo uteži za prvih 64 filtrov in izračunamo rezultate na toku vhodnih podatkov. Nato pa shranimo naslednje uteži in ponovno pošljemo vhodne podatke. Izhodni podatki so pri tem načinu računanja zbrani v skupinah po 64 filtrov. Če ne želimo razvrščati podatkov v procesorju po končanem računanju, lahko uporabimo DMA funkcijo razpršitve in združevanja opisano v naslednjem razdelku.

4.7 Komunikacija s procesorjem in DMA

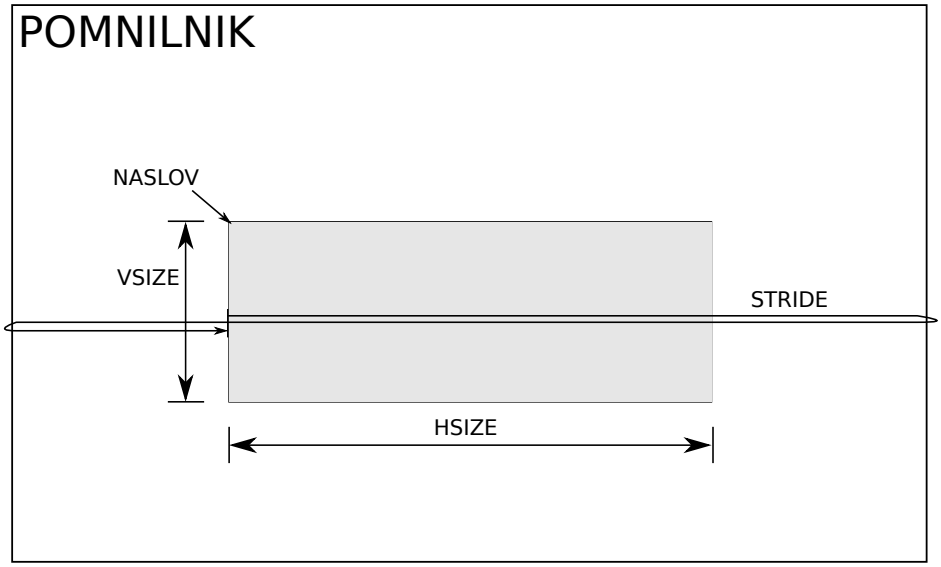
FPGA vezje z ARM procesorjem v sistemu Zynq komunicira preko vodila AXI. To vodilo ponuja več načinov komunikacije. AXI-Lite se uporablja za posamezne vrednosti, v našem primeru so to nastavitve velikosti vhodnih podatkov, število filtrov in podobno. AXI-Stream pa je primeren za pošiljanje večje količine podatkov iz sistemskega pomnilnika v tok v FPGA vezju.

Direktni dostop do pomnilnika (DMA) [7] nam omogoča neposredno pošiljanje sistemskega pomnilnika na zunanjem vodilu v tok na AXI vodilu. To uporabimo za pošiljanje vhodnih podatkov in uteži v FPGA brez obremenjevanja procesorja.

DMA lahko deluje v dveh načinih: preprostem načinu in načinu razpršitve in združevanja. V preprostem načinu določimo začetni naslov in velikost podatkov,

in DMA jih pošlje na AXI vodilo po vrsti. V večini primerov je ta način dovolj, za bolj kompleksne primere pa potrebujemo bolj kompleksen način razpršitve in združevanja.

V načinu razpršitve in združevanja, lahko določimo več območij podatkov z naslovi in velikostmi. DMA pošlje na vodilo podatke iz vseh območij enega za drugim. Prav tako lahko določimo horizontalno velikost (HSIZE), korak (STRIDE) in vertikalno velikost (VSIZE). DMA bo prenesel VSIZE vrstic, vsako velikosti HSIZE, med njimi pa bo izpustil STRIDE-HSIZE bytov. Slika 4.3 prikazuje shemo razporeditve podatkov v pomnilniku z omenjenimi nastavitvami. Ker DMA deluje z podatki velikosti 32-bitov, je pomembno da so velikosti HSIZE, STRIDE in VSIZE večkratniki števila 4, drugače DMA ne bo deloval pravilno.

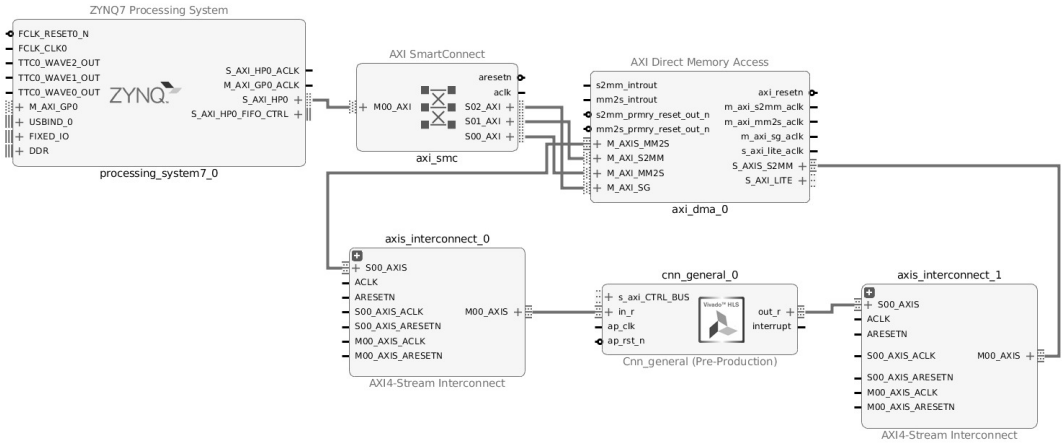


Slika 4.3: Podatki v pomnilniku za DMA

Za algoritem YOLO potrebujemo dva DMA kanala. Enega za pošiljanje vhodnih podatkov in uteži in drugega za prejemanje izhodnih podatkov. Možnost določitve več območij podatkov lahko izkoristimo za pošiljanje vhodnih podatkov in uteži z enim DMA ukazom. Način razpršitve in združevanja pa izkoristimo za pravilno razvrščanje izhodnih podatkov pri računanju konvolucije po skupinah 64

filtrov, ki je potrebna za plasti 4 in 5.

Ker v implementaciji algoritma uporabljam 24-bitne podatke, DMA pa deluje z 32-bitnimi podatki, potrebujemo med obema tokovoma pretvornik. Xilinx ponuja blok axis_interconnect, ki omogoča to funkcionalnost. Slika 4.4 kaže blokovno povezavo HLS z bloka preko axis_interconnect na DMA v programu Xilinx Vivado.



Slika 4.4: Vivado blokovna povezava

4.8 Optimizacije

Že sekvenčno računanje algoritma YOLO v FPGA izvedbi je hitrejše od programskega računanja na procesorju ARM. Vendar pa lahko vzporedno naravo FPGA vezij izkoristimo še bolje z nekaj direktivami pragma.

Odvijanje zank dosežemo z ukazom UNROLL. Ukaz določi HLS-u, da zanko sintetizira v vzporedno vezje. Ker se vse iteracije zanke na vezju izvajajo hkrati, je pomembno da v odvitih zankah hkrati dostopamo samo do mest pomnilnika, ki so na različnih blokih FPGA pomnilnika. V našem primeru imamo polje velikosti 2048x66, ki se nahaja na 66 ločenih blokih FPGA pomnilnika. Zato v odvitih zankah lahko dostopamo do različnih vrednosti samo po drugi dimenziji tega

polja.

Odvite zanke močno pospešijo izvajanje algoritma - za faktor dolžine zanke. Vendar pa odvijanje ni vedno možno. Potrebujemo pravilno razporejene podatke v pomnilniku, da lahko do njih dostopamo vzporedno. Druga omejitev pa je razpoložljiva velikost vezja FPGA, saj odvite zanke potrebujejo za faktor dolžine zanke več sredstev.

Druga pomembna optimizacija pa so cevovodi. Zanke, ki jih ne moremo odviti, lahko spremenimo v cevovod s pragma direktivo PIPELINE. Prednost cevovodov pred sekvenčnimi zankami je, da se nova iteracija začne v vsakem ciklu ure.

V optimiziranem algoritmu YOLO so odvite zanke po eni dimenziji. Pri implementaciji s shranjenimi utežmi je to ena dimenzija uteži (3), pri implementaciji s shranjenimi vhodnimi podatki pa ena dimenzija vhodnih podatkov (11). Vse druge zanke so optimizirane s cevovodi. Boljše optimizacije pri dani velikosti vezja FPGA nisem uspel doseči, saj je vsako dodatno odvijanje zank privedlo do pomanjkanja sredstev na FPGA. Porabo sredstev pri različnih optimizacijah kaže tabela 4.4.

Tabela 4.4: Poraba sredstev na FPGA pri različnih optimizacijah

Optimizacija	BRAM [%]	DSP [%]	FF [%]	LUT [%]
Neoptimizirano	89	6	8	39
Cevovod	90	8	6	43
1D odvito	88	14	7	71
2D odvito	96	64	20	136

4.9 Preizkus na razvojni plošči

Na razvojni plošči ZedBoard sem preizkusil dve implementaciji, programska implementacija je tekla na vgrajenem procesorju ARM pri frekvenci 667 MHz, strojna implementacija pa na FPGA vezju pri frekvenci 100 MHz. Obe implementaciji na koncu primerjata rezultate s pravilnimi vrednostmi izračunanimi v

Python programu.

Vhodni podatki za algoritem so slika velikosti 3x352x352 8-bitnih vrednosti in približno 16 milijonov 24-bitnih uteži. Izhodni podatek pa je matrika velikosti 11x11x125 24-bitnih vrednosti. Tako vhodne podatke kot pravilne izhodne podatke izračunane v Python programu sem na Zynq čip prenesel v času programiranja v obliki statičnih polj v C programu. Skupna velikost podatkov je okoli 50 MB, kar ne predstavlja problema za SDRAM pomnilnik na voljo na razvojni plošči ZedBoard.

Strojna implementacija za izvajanje potrebuje 13 krogov izvajanja na FPGA vezju. Tabela 4.5 prikazuje število krogov potrebnih za posamezno plast in način izvajanja v FPGA vezju - s shranjevanjem uteži ali s shranjevanjem podatkov v FPGA pomnilniških blokih.

Tabela 4.5: Krogi strojne implementacije

Številka plasti CNN	Shranjevanje	Število krogov
1	uteži	1
2	uteži	1
3	uteži	1
4	uteži	2
5	uteži	4
6	podatki	1
7	podatki	1
8	podatki	1
9	podatki	1

Za meritev časa izvajanja sem uporabil blok AXI Timer, ki je na voljo v Xilinx Vivado programu. Ta blok doda 64-bitni časovnik, ki teče pri 100 MHz, na AXI vodilo. Za merjenje časa izvajanja iz procesorja Zynq preberemo vrednost časovnika pred začetkom izvajanja algoritma in jo primerjamo z vrednostjo časovnika po končanem izvajanju algoritma.

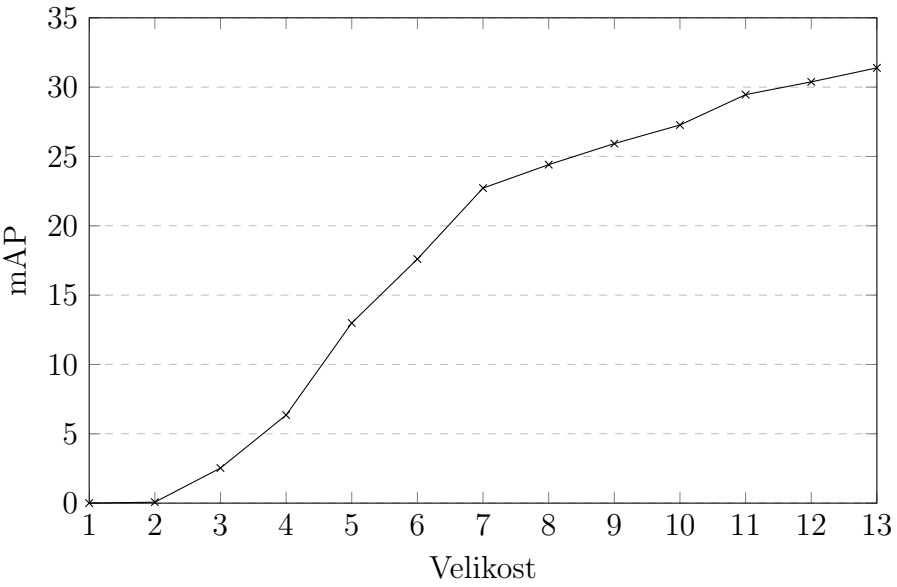
5 Rezultati

Za meritve natančnosti z uporabo mere mAP je potrebno večje število označenih slik. Za ta namen sem uporabil bazo slik VOC2007 [20], ki vsebuje 4952 označenih slik.

5.1 Primerjanje natančnosti glede na velikost algoritma

Kot sem omenil pri opisu implementacije, sem moral zmanjšati velikost algoritma YOLO, da se je ta prilegal FPGA vezju. Količina pomnilnika na vezju potrebna za shranjevanje podatkov za zadnje plasti, je sorazmerna kvadratu velikosti izhodne matrike algoritma. Da bi ugotovil kako spreminjanje velikosti algoritma YOLO upliva na njegovo natančnost, sem izmeril natančnost algoritma z mero mAP v odvisnosti od velikosti algoritma.

Slika 5.1 prikazuje natančnost algoritma z mero mAP v odvisnosti od velikosti izhodne matrike pri algoritmu YOLO. Vidimo, da z zmanjšanjem velikosti iz originalne velikosti algoritma YOLO 13x13 na implementacijo v FPGA vezju velikosti 11x11, ne izgubimo zelo veliko natančnosti. Prav tako lahko vidimo, da bi lahko velikost zmanjšali še do približno 7x7, če smo pripravljeni žrtvovati nekaj natančnosti. Pri še manjših velikostih pa začne natančnost hitro padati, tako da velikosti izhodne matrike manjše od 7x7 verjetno niso smiselne.



Slika 5.1: Natančnost algoritma YOLO v odvisnosti od velikosti

5.2 Primerjava hitrosti strojne in programske izvedbe

Končno implementacijo algoritma na vezju FPGA s frekvenco 100 MHz sem primerjal s programsko implementacijo na procesorju ARM pri frekvenci 667 MHz. Tabela 5.1 prikazuje rezultate časovnih meritev za celoten algoritem YOLO na testni sliki.

Tabela 5.1: Primerjava hitrosti FPGA in Zynq implementacije

Implementacija	Čas [s]
Programska implementacija na ARM-u	437
Neoptimizirana implementacija na FPGA	53
Delno optimizirana implementacija na FPGA	24
Najbolje optimizirana implementacija na FPGA	15
Implementacija na računalniku	< 1

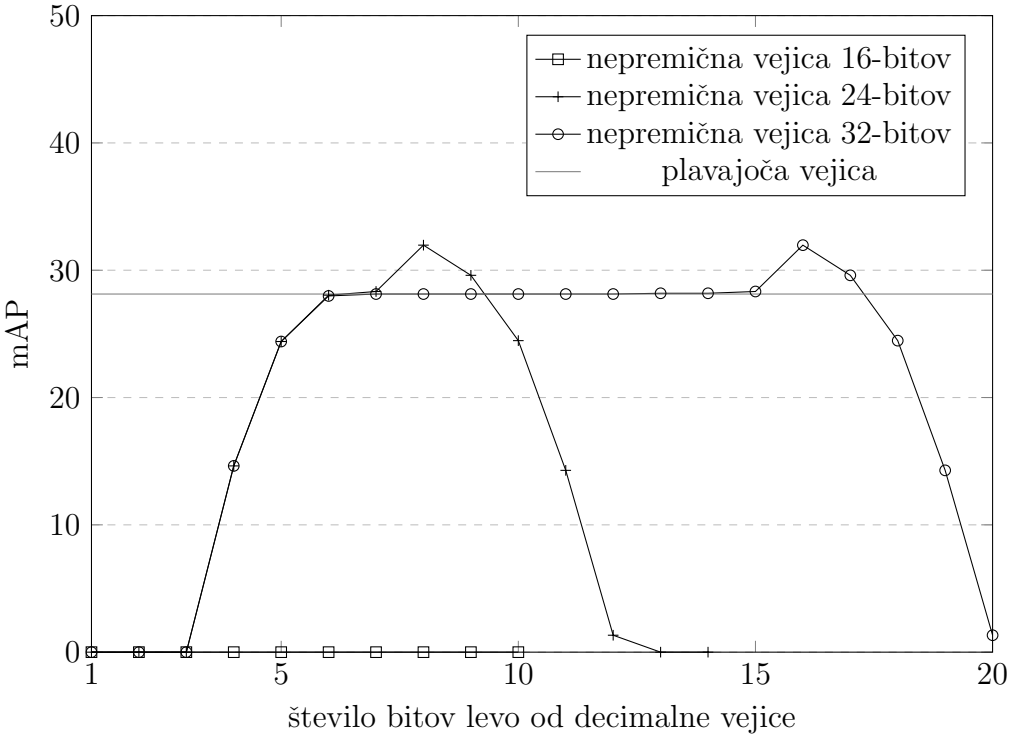
Vidimo lahko, da je že neoptimizirana implementacija na FPGA vezju precej hitrejša od programske implementacije na počasnem procesorju kot je ARM v našem sistemu Zynq, ki deluje pri frekvenci 667 MHz. Z optimizacijskimi

ukazi UNROLL in PIPELINE pa dosežemo še nekajkrat hitrejšo rešitev, preden zadanemo ob omejitve velikosti FPGA vezja na preizkusni plošči ZedBoard. Implementacija algoritma v vezju FPGA predstavlja precejšnjo pohitritev glede na programsko implementacijo na istem čipu, še vedno pa je počasnejša od implementacije na modernih grafičnih karticah.

5.3 Primerjava natančnosti z različnimi podatkovnimi tipi

Za implementacijo algoritma na FPGA vezju sem uporabil števila z nepremično vejico velikosti 24-bitov. Za ta sem s kratkim poskusom ocenil, da predstavljajo dobro razmerje med natančnostjo in porabo FPGA komponent. Ker pa je bila to le površna ocena, sem želel na koncu to predpostavko preveriti. V HLS simulaciji sem za 80 testnih slik izmeril natančnost z mero mAP pri različnih podatkovnih tipih. Slika 5.2 prikazuje natančnost mAP algoritma YOLO v odvisnosti od števila bitov levo od decimalne vejice pri številih z nepremično vejico velikosti 16, 24 in 32 bitov ter številih s plavajočo vejico.

Vidimo, da je nepremična vejica velikosti 24-bitov res najboljša izbira, saj z njo lahko dosežemo enako natančnost mAP kot s plavajočo vejico z najmanjšim številom bitov. Velikost 16-bitov nima zadostne natančnosti, natančnost je vedno 0. Nepremična vejica velikosti 32-bitov pa dosega isto natančnost kot plavajoča vejica pri velikem območju izbire števila bitov levo od decimalne vejice, kar nakazuje da večja natančnost ne prinaša nobene koristi.



Slika 5.2: Natančnost v odvisnosti od števila bitov levo od decimalne vejice

6 Zaključek

V nalogi smo ugotovili, da je z izbiro primernega podatkovnega tipa in optimizacijo tokov podatkov mogoča učinkovita implementacija konvolucijskih nevronskih mrež v FPGA vezjih. Implementacija je kompleksna in zahteva dobro poznavanje strukture algoritma, omogoča pa hitrejše izvajanje algoritma na procesorsko omejenih sistemih.

Za uspešno sintezo algoritma na FPGA vezje na preizkusni plošči ZedBoard, smo morali zmanjšati velikost in natančnost algoritma. To FPGA vezje ima relativno malo komponent, zato bi z uporabo večjega čipa lahko dosegli implementacijo celotnega algoritma. V večjem FPGA vezju bi bila možna tudi nadaljna optimizacija algoritma s popolnoma odvitimi zankami, kar bi lahko hitrost izvajanja dvignilo do hitrosti primerljivih z implementacijami na modernih računalnikih.

Nekatera podjetja na spletu ponujajo tako strežnike z zelo močnimi FPGA vezji in strežnike z modernimi grafičnimi karticami. Za nadaljno raziskavo bi bila zanimiva cenovna in hitrostna primerjava implementacije konvolucijskih algoritmov v teh dveh okoljih.

V povezavi s kamero, bi bila implementacija algoritma v FPGA vezjih primerna za avtonomne robotske sisteme, ki nimajo močnih računalnikov na katerih bi lahko drugače poganjali tak algoritem.

7 Literatura

[1] Redmon, Joseph and Farhadi, Ali, "YOLO9000: Better, Faster, Stronger", *arXiv preprint arXiv:1612.08242*, 2016.

[2] A. Choromanska, M. Henaff, M. Mathieu, G. Arous, Y. LeCun, "The Loss Surfaces of Multilayer Networks", *arXiv:1412.0233*, 2015.

[3] Timothy C Arlen, "Understanding the mAP Evaluation Metric for Object Detection", 2018 [Online]. Dosegljivo: https://medium.com/@timothycarlen/understanding-the-map-evaluation-metric-for-object-detection-a07fe6962cf3. [Dostopano: 13.8.2018]

[4] Andrew Ng, "Deep Learning Specialization", 2018 [Online]. Dosegljivo: https://www.deeplearning.ai/. [Dostopano: 12.7.2018]

[5] Xilinx, "Vivado Design Suite User Guide, High-Level Synthesis", 2018 [Online]. Dosegljivo: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_4/ug902-vivado-high-level-synthesis.pdf. [Dostopano: 15.7.2018]

[6] R. Kastner, J. Matai in S. Neuendorffer, "Parallel Programming for FPGAs", *arXiv:1805.03648v1*, 2018.

[7] Xilinx, "AXI DMA v7.1", 2018 [Online]. Dosegljivo: https://www.xilinx.com/support/documentation/

ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf. [Dostopano: 18.7.2018]

[8] Xilinx, "AXI4-Stream Infrastructure IP Suite v 2.2", 2018 [Online]. Dosegljivo: https://www.xilinx.com/support/documentation/ip_documentation/axis_infrastructure_ip_suite/v1_1/pg085-axi4stream-infrastructure.pdf. [Dostopano: 20.7.2018]

[9] D. Ipavec, "Strojna izvedba konvolucijske nevronske mreže na programirljivem vezju - koda", 2018 [Online]. Dosegljivo: https://github.com/matematik7/cnn-magistrska

[10] K. Abdelouahab, M. Pelcat, J. Serot in F. Berry, "Accelerating CNN inference on FPGAs: A Survey", 2018 [Online]. Dosegljivo: https://hal.archives-ouvertes.fr/hal-01695375/file/hal-accelerating-cnn.pdf. [Dostopano: 3.9.2018]

[11] Xilinx, "A MACHINE LEARNING APPLICATION LANDSCAPE AND APPROPRIATE HARDWARE ALTERNATIVES", 2017 [Online]. Dosegljivo: https://www.xilinx.com/support/documentation/backgrounders/Machine-Learning-Application-Landscape.pdf. [Dostopano: 3.9.2018]

[12] J. Zhang in J. Li, "Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network", *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA*, 2017.

[13] S. Winograd, "Arithmetic complexity of computations", volume 33, 1980.

[14] J. H. Ko, B. A. Mudassar, T. Na in S. Mukhopadhyay, "Design of an EnergyEcient Accelerator for Training of Convolutional Neural Networks using Frequency-Domain Computation", *Proceedings of the Annual Conference on Design Automation - DAC '17*, 2017.

[15] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao in J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks", *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '15*, 2015.

[16] J. B. Dennis in D. P. Misunas, "A Preliminary Architecture for a Basic Data-ow Processor", *Proceedings of the International Symposium on Computer Architecture - ISCA '75*, 1975.

[17] S. Anwar, K. Hwang in W. Sung, "Fixed point optimization of deep convolutional neural networks for object recognition", *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing - ICASSP '15*, 2015.

[18] B. Liu, M. Wang, H. Foroosh, M. Tappen in M. Pensky, "Sparse Convolutional Neural Networks", *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition - CVPR '15*, 2015.

[19] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv in Y. Bengio, "Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1", *arXiv e-print, 2*, 2016.

[20] M. Everingham in J. Winn, "The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Development Kit", 2007 [Online]. Dosegljivo: https://pjreddie.com/media/files/VOC2007_doc.pdf. [Dostopano: 3.8.2018]