Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 621.38
DOI: https://doi.org/10.24867/25BE28Pilipovic

RAZVOJ PYTHON VEZA, BIBLIOTEKE I APLIKACIJE ZA HARDVERSKI AKCELERATOR NEURONSKIH MREŽA

DEVELOPMENT OF PYTHON BINDINGS, LIBRARY AND APPLICATION FOR A HARDWARE CNN ACCELERATOR

Nebojša Pilipović, *Fakultet tehničkih nauka, Novi Sad*

Oblast – ELEKTROTEHNIKA I RAČUNARSTVO

Kratak sadržaj – *U ovom radu je izložen proces razvijanja softverskih slojeva i aplikacije za korištenje hardverskog akceleratora neuronskih mreža baziranog na modifikovanoj CoNNa arhitekturi. Modifikovana je postojeća C++ biblioteka, razvijen Python interfejs za istu, a zatim razvijena i sama aplikacija koja ima mogućnost izvršavanja više mreža odjednom, sa proizvoljnom raspodelom po jezgrima.*

Ključne reči: *embeded softver, CNN akcelerator, embedded biblioteka, multiprocesorski sistemi*

Abstract – *This paper presents a development process of the required software stack, and the application, for the control of a hardware neural network accelerator, based on a modified CoNNa architecture. The existing C++ library was modified, a Python interface to it was developed, and lastly the user application itself was developed, with the ability to run multiple networks concurrently, on an arbitrary number of available cores.*

Keywords: *embedded software, CNN accelerator, embedded library, multicore systems*

1. UVOD

Mašinsko učenje je trenutno jedno od najpopularnijih polja u IT akademiji i industriji. Koriste se za obradu i inteligentno prepoznavanje šablona u slikama, audio zapisima, analognim signalima i ostalim tipovima ulaza. Jedan od modela mašinskog učenja su konvolucione neuronske mreže (CNN). Ove mreže sadrže bar jedan sloj, gde se vrši konvolucija između kernela i njegovog receptivnog polja na ulazu. Ova operacija je računski zahtevna kada se radi o količini kao što je broj piksela na slici, odbiraka u zvučnom zapisu i slično.

Implementacije konvolucionih neuronskih mreža su široko dostupne u softverskom svetu, i mogu se koristiti za razne primene, uključujući i izvršavanje mreža za klasifikaciju iz ovog rada. Ali, zbog mogućnosti paralelnih konvolucija istovremeno, softverska implementacija će uvek biti daleko sporija od akceleratora kao što je CoNNa akcelerator.

Za primenu obrade slike u realnom vremenu (sigurnosne kamere, autonomna vozila i slično), uvek se preferira hardverska implementacija, jer softverska ne bi bila dovoljno brza, ili uopšte upotrebljiva za dati zadatak.

U ovom radu je realizovana Python aplikacija koja koristi dati hardverski akcelerator. Pre same aplikacije, razvijeni su svi potrebni međuslojevi od dostupne C++ biblioteke: C interfejs, Python interfejs, serijalizacija podataka za komunikaciju sa bibliotekom, Python klasa i ostale pomoćne konstrukcije koje su potrebne za ispravnu komunikaciju i rad korisničke aplikacije.

Krajnja aplikacija omogućava učitavanje više tipova mreža, i njihovu alokaciju na više jezgara akceleratora. Moguć je simultan rad više mreža odjednom.

Odrađene su vremenske analize za više vrsta implementacija aplikacije, zarad dobijanja maksimalne brzine obrade slike.

2. POSTOJEĆI SLOJEVI

Ovaj rad se bazira na prethodno razvijenom hardverskom akceleratoru za konvolucione neuronske mreže. Radi se o modifikovanoj verziji CoNNa akceleratora predstavljenom u radu [1].

Dostupan alat za rad sa njim koji je bio na raspolaganju jeste Linux drajver i upravljačka biblioteka napisana u jeziku C++. Celi sistem je namenjen da se izvršava na MPSoC platformi Xilinx Zynq ZCU102.

2.1. Komunikacija sa akceleratorom

Akcelerator je razvijen kao višejezgarni hardverski modul. Ovaj hardveski IP modul je sposoban da izvršava sve standardne slojeve koji se koriste u konvolucionim mrežama:

- Standardni konvolucioni sloj
- Depthwise separable slojevi
- Pooling sloj
- Fully-connected sloj i ostali

Koristeći kombinacije tih slojeva, ovaj akcelerator podržava popularne mreže kao što su ResNet, MobileNet, SqueezeNet, SSD, Inception i bilo koje druge proizvoljne mreže koje koriste podržane slojeve. U ovom radu, testirane su mreže MobileNet V1 sa SSD postprocessingom.

Sa akceleratorom se komunicira AXI-Lite transakcijama, za njegovu konfiguraciju i pristup registrima. Za prenos samih podataka (slika), u pozadini se koristi AXI-Full interfejs. Najbitniji registri za kontrolu prikazani su u tabeli 1.

NAPOMENA:
Ovaj rad proistekao je iz master rada čiji mentor je bio dr Predrag Teodorović, red. prof.

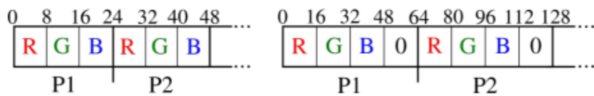
Tabela 1. Osnovni kontrolni registri

Adresa	Registar	Sadržaj	
		Bit	Namena
0x0000	Kontrolni registar	[0]	Reset
		[1]	Start, pokreće celu mrežu
		[31:2]	Ne koristi se
0x0004	Adresa opisa mreže	[31:0] - Adresa u memoriji gde se nalaze podaci o mreži	

Pored njih, postoje i statusni registri koji govore o stanju jezgra, postojanju greške, sadrže informacije o slojevima mreže i slično. Ukratko, za uspešno upravljanje jezgrom, potrebno je učitati podatke o mreži u memoriju, učitati sliku za obradu u memoriju, zatim u kontrolne registre upisati adresu podataka o mreži, i pokrenuti jedno ili više jezgara.

2.3. Format ulaznih podataka i predprocesiranje

Akcelerator, pored specifičnog formata podataka o slojevima mreže, zahteva i ulazne podatke u određenom formatu. U ovom radu će biti korištena MobileNet V1 mreža i SSD mreža, i tip ulaza je fotografija. Fotografija će inicijalno biti u formatu LxWxD, gde je dubina D = 3. Akcelerator zahteva izmenjen format predstave piksela u slici zavisno od mreže koju izvršava. MobileNet V1 mreža zahteva skaliranje vrednosti piksela na 16-bitnu vrednost. SSD mreža zahteva ulaze skalirane na opseg od 0 do 1, a zatim normalizovane u odnosu na skup podataka kojim je mreža trenirana.



Slika 1. *Očekivani format ulaza*

3. PYTHON VEZE ZA BIBLIOTEKU

Kao što je rečeno, unapred je bila dostupna biblioteka za rad sa akceleratorom. Ona definiše dve klase: *AiScaleController* i *SsdController*. Bilo je potrebno preneti funkcionalnost biblioteke u Python. Razvijene su Python veze za datu biblioteku (eng. bindings). Za ovo je korištena standardna Python biblioteka *ctypes*. Ona pruža mogućnost interakcije Python interpreter sa deljenim bibliotekama nastalim iz C koda [2].

3.1. C interfejs

Ctypes omogućava interakciju samo sa bibliotekama koje poštuju isti binarni intefejs (eng. Application Binary Interface - ABI) kao i C jezik. S druge strane, interakcija sa C++ bibliotekama nije moguća. Ovo je iz razloga što C++ nema jedinstven standardizovan ABI i načini povezivanja biblioteka mogu se razlikovati od kompajlera do kompajlera. Ovo znači da postojeća biblioteka za akcelerator *libaiscale*, koja je pisana u C++, neće moći odmah da se koristi kao takva. Rešenje koje je odabrano za ovo je pravljenje C interfejsa za C++ funkcije i metode. Na ovaj način biće korišteni isključivo C primitivni tipovi podataka na interfejsu, odabrane funkcije će biti povezane u C stilu, i *ctypes* neće imati problema da poziva te funkcije.

Interfejs je napravljen za bitne metode originalnih klasa, odnosno one važne za ispravanu konfiguraciju akceleratora. Ovo je podrazumevalo definiciju slobodnih funkcija u *extern "C"* bloku, koje za zadatak imaju da konvertuju C tip u C++ tip, i pozovu original metodu. Zaglavlja funkcija u ovom bloku ne smeju imati tipove koji nisu podržani u C jeziku, kao što je *vector* ili klasa. Na primeru koda na slici 2 prikazana je jedna slobodna funkcija koja predstavlja C interfejs za original metodu.

```
extern "C" {
  void * AiScaleController_params(AiScaleCnnData* data, int
   ↪ noOfCnns, int global_debug_level = 0){

    std::vector<AiScaleController::AiScaleCnnData> in_vec;
    for (int i = 0; i < noOfCnns; i++){
      AiScaleController::AiScaleCnnData data_c{data[i].weightfile,
       ↪ data[i].configfile};
      in_vec.push_back(data_c);
    }

    return new AiScaleController(in_vec, global_debug_level);
  }
}
```

Slika 2. *Primer C interfejsa za konstruktor*

Svaka slobodna intefejs-funkcija u *extern "C"* bloku ima argumente i povratne vrednosti koje ogledaju one originalnih metoda. Naravno, tipovi su konvertovani u C kompatibilne tipove, npr. vektor u niz, klasa u strukturu i slično. Ovakav konstruktor u memoriji inicijalizuje jedan objekat klase, i vraća njegovu adresu. Ona mora biti generičkog, *void*, tipa jer tip klase nije podržan ovde. Sve ostale metode imaju po jedan dodatni argument, koji služi da im se prosledi ova adresa, kako bi nad tim objektom u memoriji pozvali njegovu metodu. Na ovaj način sve slobodne funkcije u C interfejsu se ponašaju kao metode kojima je vlasnik jedinstveni objekat u memoriji stvoren konstruktorom sa slike 2.

3.2. Python omotač klasa

Nakon kompajliranja biblioteke sa interfejs funkcijama linkovanim u C stilu, moguće je koristiti deljenu biblioteku iz Python jezika. Napravljena je Python klasa *AiScaleController* (i *SsdController*) koja oponaša klasu iz originalne biblioteke. Ona ne redefiniše identične metode, nego poziva već postojeće iz deljene biblioteke koristeći funkcionalnosti *ctypes* biblioteke.

```
from ctypes import cdll
cdll.LoadLibrary("libaiscale.so")
#Bez argumenata
cdll.ssd_sanity_check()
#Sa argumentima
cdll.getConfigAddress.argtypes = [ctypes.c_uint32]
cdll.getConfigAddress.restype = [ctypes.c_uint32]
res = cdll.getConfigAddress(net)
```

Slika 3. *Učitavanja biblioteke i poziv funkcije*

Unutar konstruktora klase učitava se kompajlirana biblioteka u memoriju. Primer učitavanja biblioteke i pozivanja jednostavne funkcije iz nje prikazan je na slici 3. Ctypes upravljački objekat za učitanu biblioteku čuva se kao polje *AiScaleController* klase, kroz celi život objekta. Svaka metoda ove Python klase pristupa učitanoj biblioteci preko tog upravljačkog objekta, kako bi pozvala i izvršila originalnu implementaciju. Pored učitavanja biblioteke, u konstruktoru Python omotač-klase poziva se i konstruktor originalne klase iz učitane biblioteke.

On će vratiti generičku adresu originalnog objekta *AiScaleController* klase. Ovo je drugo trajno važno polje Python klase, jer predstavlja objekta vlasnika svih metoda koje budu pozivane. Na način sa slike 3, sve metode ove

Python klase pozivaju originalne metode izvorne C++ klase, uz potrebne međukonverzije tipova podataka.

3.3. Aplikacioni interfejs visokog nivoa

Konačni sloj, pre samog razvijanja korisničke aplikacije je aplikacioni interfejs (eng. Application Programming Interface - API) na nešto višem nivou apstrakcije. Dok se za korištenje *AiScaleController* i *SsdController* klasa zahtevala određena familijarnost sa interfejsom hardverskog sistema i načinom komunikacije, ovaj konačni nivo interfejsa definiše funkcije za upravljanje akceleratorom sa višeg nivoa apstrakcije. Ideja je da ovaj deo Python biblioteke nudi jednostavne pozive funkcija kao što su inicijalizacija akceleratora, jednostavan odabir tipa mreže, jednostavno procesiranje pojedinačnih okvira slike i slično - bez potrebe korisnika da ulazi u detalje o izlaznim baferima, rasporedu i formatu ulaza, rezultata, kontrolnih registara i slično.

Funkcije koje su u ovom sloju definisane predstavljaju bazu API-ja za razvijanje aplikacije i one su sledeće:

- *__init__(mem_off, core_off)* - Konstruktor koji stvara glavni objekat. Koristi memorijski i jezgarni ofset za pomeranje mreže u memoriji u slučaju da je početna regija već zauzeta drugom mrežom.
- *loadDesc(desc)* - Učitava podatke o mreži u memoriju, kao što je format IFM, format kernela i sl.
- *loadNets(nets)* - Učitava pojedinačne konfiguracije jezgara (težine kernela, tip sloja, i sl.) u svako jezgro koje se koristi.
- *processFrame(image)* - Procesira ulaznu numpy sliku: konfiguriše jezgra, kopira sliku u DRAM, pokreće sva potrebna jezgra transakcijama ka kontrolnim registrima, a zatim dobavlja rezultate, deserijalizuje ih i interpretira na dogovoren način.
- *softMax(array)* - nad 1000 članova radi probabilističku gradaciju, radi lakše interpretacije rezultata.
- *oneHot(array)* - Radi one hot kodovanje. Odbacuje sve osim najverovatnije klase.
- *findLabel(array)* - Vraća indeks elementa niza čija je vrednost 1.

4. KORISNIČKA APLIKACIJA

Korisnička aplikacija koristi najviši sloj - Python biblioteku koja definiše klasu *AiScaleAPI*. Aplikacija je razvijena na više načina, radi ispitivanja performansi. Razvijen je i vrlo jednostavan grafički meni za izbor mreža i alokaciju po jezgrima.

4.1. Multithread verzija

Prva verzija aplikacije, razvijena je na Linux Ubuntu 20.04 operativnom sistemu, na četvorojezgarnom Cortex-A53 procesoru na ZCU102 platformi. Ona je pravljena da se izvršava na više niti, koristeći Python multithreading. Raspodela poslova po softverskim nitima je sledeća:

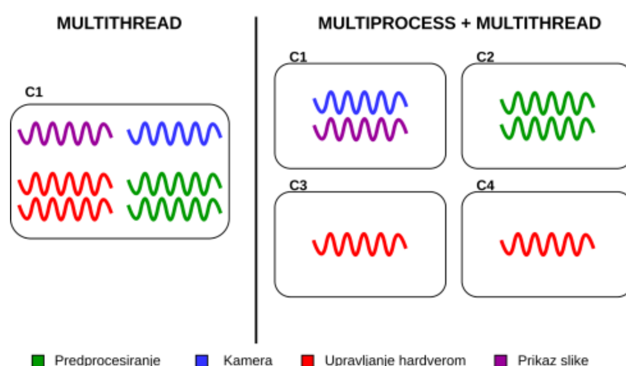
- Dobavljanje slike sa kamere je smešteno u zasebnu nit. Ovo ima smisla jer se ovde često čeka na resurs (kameru), što bi gušilo ostatak koda.
- Predprocesiranje je smešteno u zasebnu nit. Ovo je nešto CPU-intenzivniji zadatak i, ponovo, ima smisla odvojiti ga od niti koji čekaju na resurs. Ova stavka potencijalno podrazumeva dve niti, u slučaju kada se izvršavaju dve mreže istovremeno: njihov tip predprocesiranja nije isti, i potrebna je po jedna nit za svaku mrežu.
- Kontrola akceleratora je smeštena u zasebnu nit. Telo ove niti podrazumeva korištenje prethodno stvorene klase *AiScaleAPI* za prosleđivanje predprocesirane slike hardveru, i čekanje na rezultat. Ova nit je resursno intenzivna jer većinu vremena čeka na obradu hardvera. I ovde takođe je potrebno stvoriti dve niti za ovu namenu, u slučaju izvršavanja dve simultane mreže: jedan *AiScaleAPI* objekat po mreži.
- Prikaz slike u grafičkom okruženju je smešten u zasebnu nit. Iz pogleda modularizacije funkcionalnosti, ovo je takođe smešteno u zasebnu nit.

Sve niti komuniciraju putem LIFO redova u vidu Python klase *Deque*. Ovo je optimalno jer njihove definicije *read* i *write* metoda dozvoljavaju nitima da "spavaju" i ustupe mesto drugima za to vreme.

4.2. Multiprocesna verzija

Standardan Python interpreter CPython ima ograničenje u domenu niti. Postojanje tzv. GIL-a (eng. Global interpreter lock) onemogućava raspodelu niti na više fizičkih jezgara, nego sve niti se izvršavaju na jednom jezgru, unutar jednog procesa [3]. Ovo se može zaobići kreiranjem više nezavisnih procesa, i delegiranjem zadataka svakom od njih, gde će se svaki proces izvršavati na zasebnom jezgru procesora. Ovo potencijalno može ubrzati rad aplikacije, ako u njoj ima dosta CPU-intenzivnih zadataka.

Procesor Cortex-A52 ima četiri fizička jezgra, pa su u ovoj verziji sva četiri iskorištena za rad aplikacije. Ovo olakšava standardna Python biblioteka *multiprocessing*, koja pojednostavljuje stvaranje više procesa, i što je najvažnije, interprocesnu komunikaciju, koja inače nije jednostavna zbog odvojenog memorijskog prostora zasebnih procesa. Modifikovana je prva verzija da koristi ovaj način izvršavanja, i performanse su se u nekim delovima povećale. Na slici 4 prikazan je način raspodele poslova po nitima i jezgrima (C1-C4) za obe verzije aplikacije.



Slika 4. *Raspodela poslova i niti po jezgrima*

Kod multiprocesorske varijante, naiveći napredak se dobio smeštanjem rutina za predprocesiranje ulaza u posebno jezgro, iz razloga što su to računski intenzivne rutine sa malo ili bez čekanja na resurse.

5. REZULTATI I ZAKLJUČAK

Korisnička aplikacija je razvijena, sa jednostavnim menijem gde se može specificirati koja mreža se izvršava na koliko hardverskih jezgara. Ovo je u potpunosti funkcionalno. Teoretski, izvršavanje mreže na više jezgara umesto jednog bi trebalo višestruko puta ubrzati rad. U krajnjoj verziji ovo nije u potpunosti tako, iz razloga što je uočeno da znatan deo vremena na softverskoj strani se potroši na predprocesiranje. Brzine rada takođe zavise od tipa mreže koji se izvršava: razlike u hardverskom delu su manje primetne, ali razlike u vrsti predprocesiranja mogu uvesti znatne razlike u brzinama. Naravno, veličine ulaznih slika takođe drastično igraju ulogu u ovome (224x224 za MobileNet - brže, 300x300 za SSD - nešto sporije). Izvršavanje obe mreže simultano je moguće, ali će imati različite FPS (eng. Frames per second).

Procenjivanje vremenskih karakteristika konačne implementacije urađeno je i primitivnim merenjem vremena koristeći standardnu Python biblioteku *time*, ali i stranim multiprocesnim profajlerima kao što je *viztracer*. Kombinacija tip mreže - broj jezgara ima puno, i u tabeli 2 će biti prikazane osnovne karakteristične konfiguracije. Osim hardverskih vremena, vremena u tabeli su okvirna i variraju zavisno od sporednih opterećenja operativnog sistema. Ovo može biti izraženo na skromnijim CPU jezgrima, gde je svaka nit dragocena.

Tabela 2. Hardverska i softverska vremena izvršavanja

MobN/SSD Jezgara -->	**1/0**	**0/1**	**2/0**	**0/2**	**4/0**	**2/2**
[HW] MobN / SSD Total	29.6 / -	- / 32.3	15.1 / -	- / 16.6	7.8 / -	15 / 16.5
[HW] Samo SSD	-	0.53	-	0.53	-	0.53
[SW] Postproces	5 / -	- / 0.2	5 / -	- / 0.2	5 / -	5 / 0.2
[SW+HW] MobN / SSD Total	46 / -	- / 46	26.6 / -	- / 25	27 / -	27 / 26
[SW] Predproces MobN / SSD	20 / -	- / 50	20 / -	- / 50	20 / -	35 / 70

* Vremena su u milisekundama

Dostignuta je propusna moć celog sistema od oko 18 FPS za MobileNet mrežu, i par FPS niže za SSD mrežu (veća slika, nešto komplikovanije predprocesiranje). Ovo je tako čak i pri multiprocesnoj realizaciji softverske strane. Subjektivno, slika izgleda dovoljno tečno, jasno se razaznaju oblici i mreža kao što je SSD izgleda responzivno, i precizno nalazi okvire objekata. Na slici 5 može se videti prikaz grafičkog izlaza aplikacije dok obe mreže rade simulatno.

Za narednu verziju potrebno je razmotriti načine dalje optimizacije trenutnih konfiguracija - jedna od ideja jeste razvoj jezgra za predprocesiranje u hardveru, kako bi se Cortex-A52 rasteretio tog dela posla. Dalje, u budućnosti je potrebno dodati i ostale kompatibilne mreže za obradu slike (mreže za segmentaciju slike i slično), i prilagoditi aplikaciju njihovom načinu izvršavanja.

Slika 5. *Demonstracija simultanog rada obe mreže*

6. LITERATURA

[1] R. Struharik, B. Vukobratović, A. Erdeljan, i D. Rakanović, „Conna–compressed cnn hardware accelerator“, u 2018 21st Euromicro Conference on Digital System Design (DSD), IEEE, 2018, str. 365–372.

[2] „ctypes — A foreign function library for Python“, Python documentation. https://docs.python.org/3/library/ctypes.html (pristupljeno 30. avgust 2023.).

[3] F. Doglio, Mastering Python High Performance. Packt Publishing, 2015.

Kratka biografija:

Nebojša Pilipović rođen je u Doboju 1998. god. Bečelor rad na Fakultetu tehničkih nauka iz oblasti Elektrotehnike i računarstva – Embedded sistemi i algoritmi odbranio je 2021.god. kontakt: nb.pilipovic@gmail.com