Paralelizacija izračuna dubokih neuralnih mreža na grafičkim procesorima

Poleksić, Andrija

Undergraduate thesis / Završni rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka / Sveučilište u Rijeci**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:195:174813

Sveučilište u Rijeci – Odjel za informatiku

Jednopredmetni preddiplomski studij informatike

Andrija Poleksić

Paralelizacija izračuna dubokih neuralnih mreža na grafičkim procesorima

Završni rad

Mentor: dr. sc. Vedran Miletić

Rijeka, 23. rujna 2021.

Sažetak

U ovome radu opisana je primjena i prednosti korištenja paralelizma (paralelnog programiranja) kod rada sa dubokim neuralnim mrežama, nadalje DNN. Opisani su osnovni pojmovi iz područja dubokog učenja. Prikazana je usporedba učinkovitosti procesa treniranja modela u serijskoj izvedbi na CPU i paralelnoj izvedbi na heterogenom sustavu (CPU i GPU). Objašnjena su rješenja koja se primjenjuju u paralelizaciji treninga DNN te teoretska rješenja aktualnih znanstvenih radova.

Ključne riječi

Paralelizam, Duboke neuralne mreže, Heterogeni sustavi

Sadržaj

ODJEL ZA INFORMATIKU
SVEUČILIŠTE U RIJECI

Rijeka, 2. ožujka 2021.

Zadatak za završni rad

Pristupnik: Andrija Poleksić

Naziv završnog rada: Paralelizacija izračuna dubokih neuralnih mreža na grafičkim procesorima

Naziv završnog rada na eng. jeziku: Parallelization of deep neural network computation on the GPU

Sadržaj zadatka:

Duboke neuralne mreže su temelj modernih primjena strojnog učenja u brojnim domenama, od samovozećih automobila do računalne kemije. Zbog veličine problema koje obrađuju često se izvode paralelno na grafičkim procesorima i upravo su jedna od najčešćih primjena grafičkih procesora. Cilj rada je opisati primjenu paralelizma u radu s dubokim neuralnim mrežama te usporediti učinkovitost procesa treniranja modela u serijskoj izvedbi na osnovnom i paralelnoj izvedbi na grafičkom procesoru.

Mentor

V. pred. dr. sc. Vedran Miletić

Voditelj za završne radove

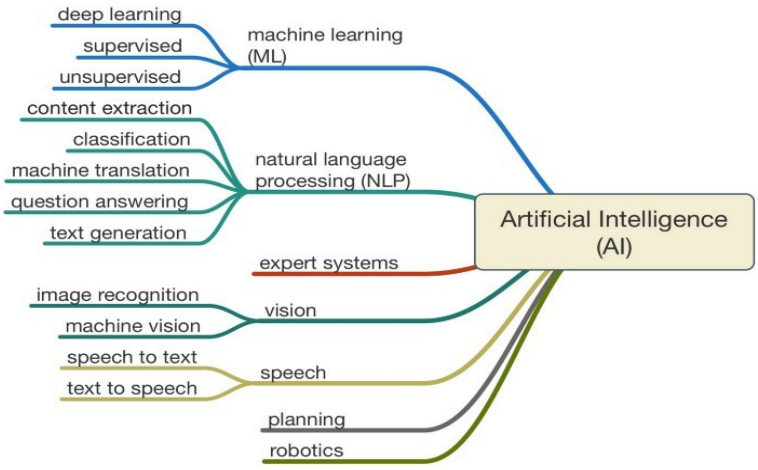
Doc. dr. sc. Miran Pobar

Zadatak preuzet: 2. ožujka 2021.

(potpis pristupnika)

1. Uvod

Pojam umjetne inteligencije prvi spominje John McCarthy 1956. prilikom održavanja prve znanstvene konferencije [1] na temu inteligentnih računala. Iako sama ideja o uređajima koji mogu razmišljati kao ljudi postoji stoljećima unazad, kao prekretnica u razvoju umjetne inteligencije, odnosno kao njen začetak spominjemo matematičara Alana Turinga i njegov rad „Computing Machinery and Intelligence“ [2]. U radu Turing opisuje postupak provjere inteligencije računala nazvan „Imitation game“, danas popularno zvan Turingov test. Trenutno ni jedan uređaj ne prolazi test, odnosno ne postoji računalo (uređaj) koji može simulirati ljudsku inteligenciju, ali su ostvareni značajni napreci u izvođenju specifičnih zadataka, kao na primjer: igranje društvenih igara, prepoznavanje oblika, prevođenje teksta ili (autonomno) upravljanje vozilom. U radu ćemo se upoznati s granom umjetne inteligencije zvanom strojno učenje (eng. *machine learning*), točnije, jednom od vrsta strojnog učenja, dubokim učenjem (eng. *deep learning*). Za jasniju sliku o podjeli unutar pojma umjetne inteligencije pogledajte [3] [4] [5] te sliku (Slika 1).



Slika 1: Podjela umjetne inteligencije[3]

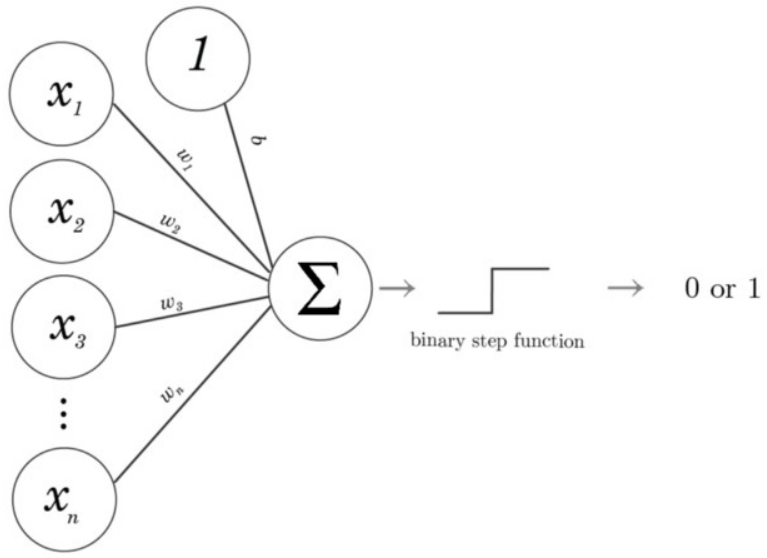
Strojno učenje definiramo kao područje koje se fokusira na imitaciju učenja blisku ljudima, odnosno na postepeno poboljšanje preciznosti izvođenja neke radnje korištenjem određenih algoritama na nekom ulazu podataka. Popularnu i sveopće prihvaćenu definiciju nalazimo u knjizi „Machine learning“ [6] autora Toma M. Mitchella: „A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E“.

U drugom poglavlju opisani su osnovni koncepti i rad neuralnih mreža te je razjašnjena podjela na duboke i plitke neuralne mreže. Poglavlje nakon (3. poglavlje) progovara o paralelnom programiranju i platformama (i bibliotekama) koje to omogućuju. Završno s upoznavanjem s bibliotekama CUDA platforme i njenim alternativama započinje 4. poglavlje. Unutar navedenog obrađena je paralelizacija DNN kao svojevrsna sinteza prethodna dva poglavlja. Paralelizacija DNN podijeljena je na lokalni trening te na složeniji distribuirani. Četvrto poglavlje zaokruženo je s aktualnim rješenjima paralelizacije DNN. Primjer rada DNN na heterogenom sustavu, te korišteni softver i hardver opisani su u 5. poglavlju. U sklopu 5. poglavlja prikazani su rezultati koji potkrepljuju sam zaključak (6. poglavlje).

2. Neuralne mreže

Neuralne mreže kao vrsta strojnog učenja idu korak dalje prema imitaciji učenja bliskog ljudskom. Inspiracija potječe od ljudskog mozga, gdje neurone zamjenjuju čvorovi (umjetni neuroni) te sinapsu predstavljaju veze s određenom težinom (eng. *weight*) i pragom aktivacije (eng. *threshold*). Osim praga aktivacije, često se koristi pojam varijable pristranosti (engl. *bias*). Čvorovi, kao reprezentacija neurona, u slučaju da vrijednost prelazi prag aktivacije, prosljeđuju podatke nadolazećim neuronima, odnosno šalju impulse. Čvorovi su raspodijeljeni u slojeve, s početnim ulaznim i završnim izlaznim slojem.

Pokažimo djelovanje umjetnog neurona na najjednostavnijem primjeru perceptrona.



Slika 2: Model perceptrona [26]

Perceptron (Slika 2) prima jedan ili više ulaznih podataka (x1, x2, x3…) koji mogu poprimiti jednu od dvije vrijednosti {0,1} te za svaki ulaz ima određenu težinu u obliku realnog broja koja određuje „važnost" ulaza. Izlaz perceptrona iz našeg primjera je uvijek 0 ili 1. Izlaz izračunavamo pomoću funkcije aktivacije. U našem primjeru, funkcija aktivacije je oblika:

$$output = \begin{cases} 0 \, if \sum w_i x_i \leq treshold \\ 1 \, if \sum w_i x_i > treshold \end{cases}$$

Zapišemo li sve ulaze (x1, x2, x3…) te sve pripadajuće težine (w1, w2, w3…) kao vektore X i W, te gore navedenim nejednadžbama pribrojimo b (varijabla pristranosti) kojeg definiramo kao $-treshold = bias = b$ dobijemo zapis oblika:

$$output = \begin{cases} 0 \, if \, W \cdot X + b \leq 0 \\ 1 \, if \, W \cdot X + b > 0 \end{cases}$$

u kojem $X \cdot W$ označava vektorski produkt. Ovakav jednostavni neuron podatke može klasificirati linearno, odnosno u dvije skupine, a redanjem i slaganjem perceptrona u slojeve, odnosno u neuralne mreže, možemo izvoditi i kompleksnije raspodjele (klasifikacije). Kod neuralnih mreža sa složenijim zadacima se koriste derivabilne funkcije aktivacije koje omogućuju proces učenja. Za razliku od perceptrona koji kao funkciju aktivacije koriste Heaviside step funkciju (jediničnu funkciju koraka), neuroni koji grade DNN najčešće koriste ReLU funkciju ili Sigmoidnu (logističku) funkciju, te neke ostale funkcije [7].

Pojam učenja (eng. *model training*) definiramo kao fazu u kojoj se za model neuralne mreže (algoritam strojnog učenja) traže vrijednosti varijabli pristranosti (b) i težina (w) takve da je objektivna (eng. *loss* ili *cost*) funkcija minimalna. Pritom su ključni pojmovi spomenuta loss funkcija koja nam daje informaciju o veličini greške te algoritam koji optimizira težine i varijable pristranosti.

Promotrimo primjer klasifikacije ručno pisanih znamenaka. Da bi neuralna mreža imala točnost veću od nasumičnog pogađanja znamenaka od 0 do 9 uzimamo trening podatke (eng. *training data*) koji se sastoje od očekivanog ulaza i točnih odgovora za izlaz (poznato i kao *labeled data*). Zatim objektivna funkcija uspoređuje rezultate dobivene prolaskom podataka kroz neuralnu mrežu s točnim podacima pridruženim ulazu (eng. *labels*). Nakon opisanog postupka ,zvanog propagacija unaprijed (eng. *front propagation*), slijedi propagacija unazad (eng. *back propagation*) u kojoj se pomoću optimizacijske funkcije, npr. funkcije stohastičkog gradijentnog pada, nadalje SGD (eng. *stochastic gradient descent*), određuju korekcije u vrijednostima težina i varijabli pristranosti tako da se smanjuje vrijednost loss (objektivne) funkcije. Potpuni proces ponavlja se određen broj puta, najčešće do dobivanja željene točnosti.

Trening podaci sastoje se od uzoraka (eng. *samples*), poznatih i pod nazivima vektor svojstva (eng. *feature vector*) i ulazni vektor. Podaci se najčešće raspoređuju u mini-serije (eng. *mini-batches*). Pritom se nakon svake mini-serije izvršava optimizacijski algoritam (npr. GD). Serijom nazivamo cjelokupni set trening podataka, dakle, set podataka sastoji se ili od n jednakih mini-serija ili jedne serije. Prolaz u kojem su svi uzorci iskorišteni u optimizacijskom algoritmu, odnosno prolaz kroz

sve uzorke jednom, naziva se epoha (eng. *epoch*) [8].

2.1. Duboko u dubokim neuralnim mrežama

Gruba podjela neuralnih mreža vrši se s obzirom na broj skrivenih slojeva (eng. *hidden layers*), slojeva koji se nalaze između ulaznog (eng. *input*) i izlaznog (eng. *output*) sloja. Prema tome, neuralne mreže s malim brojem skrivenih slojeva (1 do 2) nazivamo plitke neuralne mreže. U takvim mrežama obično su među slojevima svi neuroni povezani (eng. *fully connected layers*). Povećanjem broja skrivenih slojeva, rastu i mogućnosti neuralnih mreža. Takve mreže nazivamo dubokim neuralnim mrežama. Klasična DNN mjeri broj slojeva u desetcima, ako ne i stotinama. Izračunavanje rezultata funkcija aktivacija, propagacija unaprijed te propagacija unazad svodi se na operacije nad matricama, koje prikladno imaju veliki potencijal paralelizacije.

3. Paralelizam i heterogeni sustavi

Porast kompleksnosti i zahtjevnosti računalnih programa (softvera) iziskuje i dostatan hardver koji bi navedene programe mogao izvršavati. S vremenom postaje jasno kako fokusiranje snage izračuna u jednoj točci (jednojezgrenom procesoru) nije optimalno rješenje.

Revolucija počinje 2001. godine s prvim komercijalnim višejezgrenim procesorom IBM Power 4 [9]. Ideja višejezgrenih procesora donosi značajan napredak po pitanju energetske učinkovitosti CPU-a. Višejezgrenost podrazumijeva više manjih jezgri koje zajedno mogu postići jednake ili bolje performanse naspram monolitnih CPU-a i pritom koristiti manje energije, što je vidljivo iz izraza za procjenu potrošnje procesora: $P = C \times V^2 \times F$ [10] gdje je P snaga za rad procesora, C kapacitet, V napon i F frekvencija procesora.

Glavne ideje iza boljih performansi višejezgrenih procesora su višenitnost i multiprocesing (višeprocesnost) koje između ostalog podrazumijeva i paralelno izvođenje procesa (ili niti) u pravom smislu te riječi. Napomenimo kako je proces izvođenje instrukcija programa te se proces može sastojati od više niti. Niti unutar istog procesa dijele memorijski prostor, dok to među procesima nije slučaj.

3.1. Grafička procesorska jedinica

Grafički procesor elektroničko je sklopovlje posebno dizajnirano za brzo manipuliranje podacima, odnosno za izvođenje jednostavnijih operacija na velikom setu podataka (SIMD), kao na primjer za renderiranje trodimenzionalne grafike. Porastom programabilnosti GPU-a rađaju se nove primjene izvan domene računalne grafike za koju je izvorno dizajniran. Veliki broj jezgri GPU-a omogućuje visoku razinu paralelizacije izračuna što se pokazuje vrlo pogodnim za ubrzavanje treninga dubokih neuralnih mreža.

3.2. Flynnova taksonomija

Upravo prije spomenuta paralelnost u kombinaciji s grafičkim procesorima dovodi do nove revolucije u obliku heterogenih sustava. Heterogeni sustavi sastoje se od više različitih procesorskih jedinica, na primjer kombinacija CPU i GPU, koja će nama biti značajna. Takvi sustavi mogu izvoditi više različitih instrukcija na više različitih tokova podataka te time spadaju u MIMD (Multiple Instructions Multiple Data) strukturu.

Ostatak podjele računalnih arhitektura prema Michaelu J. Flynnu [11] s obzirom na instrukcije i tokove podataka glasi: SISD (Single Instruction Single Data) na primjer jednostavni jednojezgreni procesori, SIMD (Single Instruction Multiple Data) na primjer grafički procesori i višejezgreni procesori, MISD (Multiple Instructions Single Data) te prije spomenuti MIMD.

Paralelno programiranje granuliranje je nekog problema u niz jednostavnijih koraka koji se mogu izvoditi istovremeno na više procesorskih jedinica i time skratiti ukupno vrijeme izvođenja. Neke od primjena paralelnog programiranja su rijetka i gusta algebra, problemi kvadratične rešetke, pretraživanje u grafovima te simulacija konačnih automata. [12]

3.3. CUDA

Krajem 2006. godine NVIDIA upoznaje korisnike s platformom za paralelno programiranje koja koristi potencijal NVIDIA grafičkih kartica nazvanom CUDA (Compute Unified Device Architecture). CUDA kôd moguće je pisati u većini modernih jezika, uključujući u Python-u, C-u i C++-u. CUDA posjeduje širok raspon biblioteka i posrednih programa (eng. *middleware*) uključujući cuRAND (CUDA knjižnica generatora nasumičnih brojeva), CULA (CUDA knjižnica za linearnu algebru i sofisticiranu matematiku) i ostale te nama značajne cuBLAS [13], cuDNN [14], TensorRT [15], cuSPARSE.

Srž CUDA-e čine tri pojma: hijerarhijska podjela niti (nitni blokovi), dijeljena memorija i barijerna sinkronizacija niti. Navedeni pojmovi omogućuju sitnozrni paralelizam podataka i paralelizam niti. Paralelizam podataka (eng. *data parallelism*) predstavlja distribuiranje podataka na više niti, pri čemu niti izvode istu operaciju. Nasuprot tome je ideja paralelizma zadataka (eng. *task parallelism*) u kojoj se razne operacije pridružuju nitima koji ih izvršavaju na istom setu podataka. Spomenimo postojanje paralelizma na razini bita (eng. *bit-level parallelism*) te paralelizam na razini instrukcija (eng. *instruction level parallelism*).

Rad s CUDA platformom vrlo je jednostavan, sastoji se od domaćina (eng. *host*) u obliku CPU-a i uređaja (eng. *device*) u obliku GPU-a. Pritom programer piše ljuske (eng. *kernels*), svojevrsne funkcije, koje se nakon prijenosa na globalnu memoriju uređaja pozivaju na nitima. Niti su raspoređene u blokove, te ovisno o ograničenjima hardvera postoje ograničenja na niti po bloku. Također postoje gotove biblioteke specijalizirane za razne zadatke, na primjer cuFFT (brze Fourierove transformacije), cuRAND (generatori nasumičnih brojeva) ili Thrust (algoritmi i strukture podataka) te prije spomenute značajne biblioteke.

3.3.1. CuBLAS

BLAS (Basic Linear Algebra Subprograms) označava skup rutina (funkcija) koje vrše operacije nad vektorima i matricama. CuBLAS biblioteka je dakle implementacija BLAS-a za paralelno izvođenje na NVIDIA grafičkim procesorima. BLAS se generalno dijeli u tri razine [16]:

1. Razina 1: Operacije s vektorima
2. Razina 2: Operacije s matricama i vektorima
3. Razina 3: Operacije s matricama

3.3.2. SAXPY primjer

Kako bi razjasnili osnovnu ideju iza rada cuBLAS biblioteke i CUDA-e, pogledajmo primjer rutine razine 1 zvane SAXPY (Single-Precision A * X Plus Y). SAXPY je funkcija koja izvodi zbrajanje vektora X i Y, gdje je vektor X pomnožen skalarom a. Sukladno osnovi rada s CUDA-om, izvođenje programa sastoji se od koraka:

1. Deklariranje i alociranje memorije na domaćinu (host) i uređaju (device)
2. Inicijalizacija podataka na domaćinu
3. Prijenos podataka s domaćina na uređaj
4. Izvođenje zrna (kernel)
5. Prijenos rezultata s uređaja na domaćin

Započnimo definiranjem zrna koje ćemo pozivati na grafičkom procesoru. Ovako definirana funkcija pojednostavljenje je rutine „cublasSaxpy()“ iz cuBLAS biblioteke [13].

```
__global__
void saxpy(int n, float a, float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}
```

Pogledajmo „main“ funkciju, koja započinje deklariranjem dva para pokazivača (pritom je N = 1048576) kojima ćemo pristupati memoriji domaćina (x, y) te memoriji uređaja (d_x, d_y). Nakon deklaracije alocira se memorija na domaćinu i uređaju, čime smo završili s prvim korakom.

```
int main(void)
{
  int N = 1<<20;
  float *x, *y, *d_x, *d_y;
  x = (float*)malloc(N*sizeof(float));
  y = (float*)malloc(N*sizeof(float));

  cudaMalloc(&d_x, N*sizeof(float));
  cudaMalloc(&d_y, N*sizeof(float));
```

Zatim polja na koja pokazuju x i y punimo vrijednostima te ih prenosimo s domaćina na uređaj (2. i 3. korak), odnosno kopiramo x na d_x te y na d_y.

```
for (int i = 0; i < N; i++) {
    x[i] = 1.0f;
    y[i] = 2.0f;
  }

  cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);
```

Na kraju izvodimo zrno na nitima uređaja i kopiramo podatke nazad na domaćina (4. i 5. koraci). Pritom je izraz unutar trostrukih zagrada („<“) konfiguracija izvođenja koja nam govori koliko niti na uređaju paralelno izvodi pozvano zrno. Preciznije, prvi argument („(N+255) / 256“) odnosi se na broj nitnih blokova u rešetki, a drugi („256“) na broj niti unutar bloka. Dakle imamo 4096 blokova od kojih svaki sadrži 256 niti.

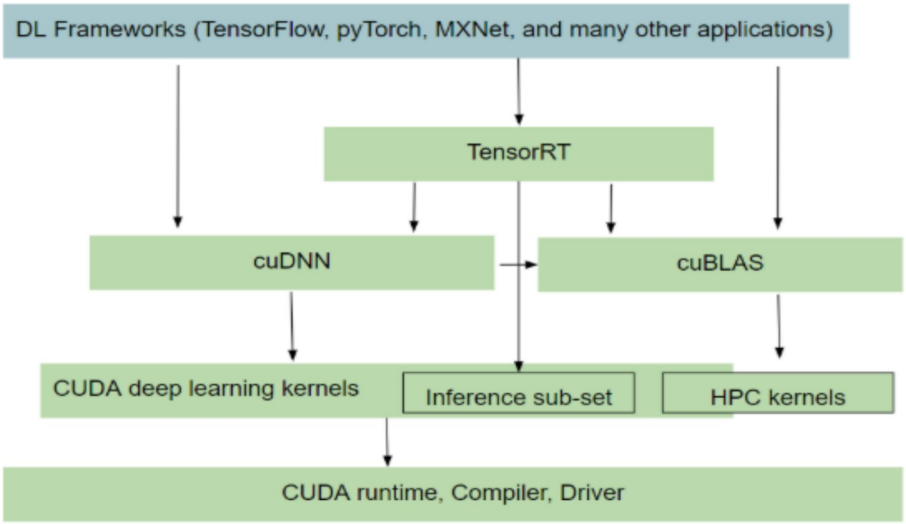
```
// Perform SAXPY on 1M elements
  saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);

  cudaMemcpy(y, d_y, N*sizeof(float), cudaMemcpyDeviceToHost);
```

Primjerni kod je preuzet sa [17]; tamo dostupna detaljnija pojašnjenja istoga.

3.3.3. CuDNN

CuDNN biblioteka je GPU-om ubrzanih operacija karakterističnih za rad s neuralnim mrežama. Neke od implementiranih rutina su: aktivacijske funkcije (ReLU, Sigmoidna, TANH), operacije nad tenzorima (cuBLAS), konvolucija, unakrsna korelacija (eng. *cross-correlation*) te uzorkovanje (eng. *pooling*). Većina rutina je jednostavna paralelna izvedba postojećih algoritama vezanih uz DNN, izuzev konvolucije koju je potrebno svesti na umnožak matrica. Više o vrstama konvolucija unutar cuDNN, te o primjeru implementacije dostupno je u [14] [18].



Slika 3: Odnos okvira i biblioteka u radu sa DNN [14]

Slika (Slika 3) prikazuje odnos biblioteka i razvojnih okvira u radu s dubokim neuralnim mrežama. Na vrhu se nalaze platforme za rad s DNN koje povezuju rad svih biblioteka u jednostavnije sučelje za korisnika. Nešto više o samim platformama napisano je kasnije. Kombinacija cuDNN i cuBLAS biblioteka zadužena je za trening modela, pritom funkcije iz cuDNN biblioteke koriste cuBLAS rutine. Nakon treninga vrši se optimizacija modela za izvođenje koju provodimo pomoću TensorRT biblioteke.

3.3.4. Alternative

Rad s DNN-om uvelike ovisi o paralelizaciji izračuna korištenjem GPU-a što NVIDIA dobro prepoznaje i započinje razvoj CUDA-e. Spomenimo postojanje AMD-ove platforme ROCm (Radeon Open eCosystem) i MIOpen kao alternative otvorenog koda za AMD-ove grafičke

procesore. Uz ROCm, AMD pruža potporu okviru za razvoj programa na heterogenim sustavima OpenCL (Open Computing Language) i OpenMP-u (Open Multi-Processing). Osim OpenCL okvira, Khronos grupa također razvija SYCL koji dopušta generaliziraniji pristup paralelnom programiranju u jeziku C++ unutar jedne datoteke.

4. Paralelizacija dubokih neuralnih mreža

Zaključili smo da GPU postiže najbolje rezultate kada je moguće rad rasporediti na velik broj radnika (procesorskih niti), točnije, kada je posao moguće raspodijeliti na puno manjih zadataka, koje je, prikladno, moguće izvoditi paralelno. Prema radu „Parallel and Distributed Deep Learning“ [19], propagacija unaprijed kroz konvolucijski sloj te kroz potpuno povezani sloj neurona postiže ubrzanje od 40 puta prilikom korištenja jednostavnog heterogenog sustava (CPU-a i GPU-a) naspram korištenja samog jednojezgrenog CPU-a. Prikažimo u nastavku neke od metoda paralelizacije DNN-a.

4.1. Lokalni trening modela

Navedeni oblik treninga podrazumijeva jedno ključno ograničenje - spremanje čitavog modela i seta trening podataka na jednom uređaju, što bi odgovaralo primjerima izvođenja treninga na CPU ili kombinaciji CPU i GPU. Kompleksni modeli s revolucionarnim mogućnostima, dakako, trebaju veliki set podataka i veliki prostor za spremanje cijelog modela te kao takvi nisu izvedivi u obliku lokalnog treninga. Podijelimo lokalni trening na tri načina izvođenja [19]:

1. Višejezgreno procesiranje
2. GPU za zahtjevne izračune
3. Višejezgreno procesiranje i GPU u kombinaciji

Višejezgreno procesiranje možemo izvesti na dva načina Prvi, trivijalni način je procesiranje više ulaznih podataka od sloja do sloja neuralne mreže. Drugi način podrazumijeva paralelno izvođenje optimizacijskog algoritma (npr. SGD) na više mini-serija.

4.2. Distribuirani trening

Prema [20] postoji 6 stupnjeva (spomenut ćemo 3 relevantna) paralelizacije izračuna DNN od kojih, shodno porastu stupnja, raste i dodatak vremena utrošen na komunikaciju između čvorova distribuiranog sustava, pri čemu se distribuirani sustav sastoji od više softverskih komponenti na više računala koji zajedno rade kao jedan sustav. Odvojene jedinice sustava nazivamo čvorovi.

4.2.1. Paralelizam treninga (eng. Training session parallelism)

Prvi stupanj paralelizacije je izvođenje istog modela DNN-a na svim čvorovima (računalima koja su dio distribuiranog sustava) s različitim inicijalnim vrijednostima težina, pri čemu se na kraju treninga izabire model s najboljim rezultatima. Ovakav način paralelizacije ne zahtijeva komunikaciju među čvorovima te time ne postoji gubitak vremena utrošen na internu komunikaciju. Takav način paralelizacije nije ovisan o drugim čvorovima sustava. Prema tome, porast broja čvorova (GPU-a) uvjetuje porast učinkovitosti.

4.2.2. Paralelizam podataka

Paralelizam podataka, slično prethodnome, zahtjeva kopiju modela na svakom od čvorova sustava, prilikom čega su kopije potpune jednake. Trening podaci, mini-serije, najčešće se ravnomjerno raspoređuju na čvorove, gdje se lokalno vrši propagacija u oba smjera te je zatim potrebno ažurirati sve modele u sustavu shodno dobivenim rezultatima. Ovisno o načinu ažuriranja težina modela, paralelizam podataka dijelimo na sinkroni i asinkroni. Kod sinkronog načina gradijenti (rezultati izračuna, na primjer SGD-a) iz čvorova se prosljeđuju na parametarski server. Parametarski server je čvor koji je zadužen za pohranu parametara modela i posluživanje istih ostalim čvorovima. Kod sinkronog načina, na primjer BSP-a (Bulk Synchronous Parallelism), čekaju se svi izračuni koji se zatim spajaju i takvi vraćaju na sve čvorove sustava. Količina podataka koja je potrebna za ovakav tip komunikacije proporcionalna je broju težina unutar modela i broju čvorova koji vrše izračune. Vidljivo je kako ovakav princip dovodi do velikih zastoja i čekanja, što umanjuje iskorištenost GPU-ova. U svrhu poboljšanja iskorištenosti čvorova koji vrše izračune (radnika) koristi se asinkroni tip paralelizma podataka. Asinkroni tip dopušta radnicima prelazak na fragment sljedeće mini-serije bez čekanja najažurnijih parametara modela DNN-a. Ovakav tip paralelizma podataka smanjuje vrijeme po iteraciji, ali povećava broj potrebnih iteracija do željene preciznosti samog modela.

4.2.3. Paralelizam modela

S obzirom na porast broja slojeva u modelu DNN-a na stotine i tisuće, jasno je kako u jednom trenu nema dovoljno memorije za pohranu istog na jednom uređaju (čvoru). U takvim situacijama koristi se ideja paralelizma modela, teoretski suprotna paralelizmu podataka. Mini-serije se ne raspoređuju po čvorovima (radnicima) već je svaki radnik zadužen za određeni sloj ili više njih. Takav model znatno smanjuje trošak komunikacije među čvorovima. Međutim, u naivnoj izvedbi to dovodi do velike neefikasnosti po pitanju GPU-a, odnosno radnika. Do ovakvog ishoda dolazi upravo zbog raspodjele radnika na slojeve; set podataka u trenutku prolazi kroz jedan sloj dok su drugi, nadolazeći slojevi neaktivni. Prema tome, i radnici zaduženi za navedene slojeve ostaju neaktivni.

4.2.4. Usporedba paralelizma modela i podataka

U nastavku je priložena tablica koja nam daje slikovitu usporedbu dva ključna modela paralelizma. Efikasnost treninga definiramo kao najmanji mogući broj prolazaka kroz sve trening podatke u svrhu postizanja određene točnosti modela, pri čemu se točnost modela ocjenjuje s obzirom na učestalost točnog rješavanja problema, npr. ispravnog prepoznavanja rukom nacrtane znamenke na slici.

	Paralelizam podataka	**Paralelizam modela**
Raspodjela izračuna na GPU-ove	Ravnomjerna raspodjela trening podataka na sve radnike.	Različita kompleksnost slojeva dovodi do nejednake raspodjele rada.
Troškovi komunikacije	Potrebna je komunikacija velike količine podataka(ažuriranje parametara).	Zahtjeva prijenos male količine podataka između slojeva(prijenos međuizračuna).
Efikasnost treninga	Veličina particioniranih trening podataka na radnike utječe na učinkovitost. Manji broj radnika s većim setom podataka povećava učinkovitost, ali smanjuje preciznost modela.	Naivna izvedba ima maksimalnu efikasnost treninga. Aktualni i korišteni oblici paralelizma modela imaju manju točnost zbog ustajalih vrijednosti.

Tablica 1: Usporedba paralelizma modela i podataka

Porast samih modela DNN-a, te samim time izračunskih zahtjeva, raste i potreba za većim i „jačim“ distribuiranim sustavima. Unatoč jednostavnosti i učinkovitosti paralelizma podataka za manje sustave, činjenica da se spomenuti način paralelizacije teško skalira dovodi do češćeg korištenja paralelizma modela ili kombinacije dva spomenuta paralelizma u recentnijim radovima i istraživanjima.

Konkretnije, porast broja čvorova u distribuiranom sustavu (prilikom korištenja paralelizma podataka) uvjetuje i porast vremena izdvojenog na međusobnu komunikaciju (eng. *communication overhead*). Posebno je vidljiv porast troška vremena prilikom rada DNN-a s potpuno povezanim slojevima. U svrhu optimizacije u primjenu ulazi hibridni paralelizam, spoj paralelizma podataka i modela ovisno od sloja do sloja [21].

4.3. Rješenja ponuđena u aktualnim istraživanjima

Do sada su u radu spomenute sve značajne vrste unutarserijskih paralelizacija (eng. *intra-batch parallelism*), odnosno paralelizacija u kojima je jedna iteracija treninga (epoha) raspoređena na čvorove, bilo po slojevima, bilo po samim podacima. U nastavku su nabrojena aktualna rješenja koja se oslanjaju na ideju cjevovoda (eng. *pipeline*). Cjevovodno procesiranje (eng. *pipeline processing*) je preklapanje operacija pomicanjem podataka ili instrukcija u konceptualnom cjevovodu u kojem se svaki dio cjevovoda, svaka cijev (stadij), izvodi paralelno.

4.3.1. Gpipe

U izvornom radu koji opisuje Gpipe [22] prvi puta se spominje ideja paralelizma cijevi (eng. *pipeline parallelism*). Ključna ideja je gledati model DNN-a kao niz slojeva (paralelizam modela) te se pritom uzastopne skupine slojeva mogu rasporediti u ćelije. Ćelije pritom konceptualno odgovaraju stadiju (cijevi) u cjevovodu te svakoj se ćeliji dodjeljuje radnik (GPU). Mini-serije raspodjeljuju se u manje mikro-serije te se prosljeđuju u cjevovod. Naposljetku, rezultati se svih mikro-serija spajaju te se vrši optimizacija modela. GPipe je dakle skalabilna biblioteka koja kombinira paralelizam modela i cjevovod te tako nadmašuje paralelizam podataka.

4.3.2. PipeDream

GPipe model odvaja izvođenje propagacije unaprijed i propagacije unazad, što u kombinaciji s odbacivanjem podataka o aktivaciji između propagacija zahtjeva česti *pipeline flush*. Nasuprot tome, PipeDream izvedba optimizira raspodjelu rada te pohranjuje rezultate aktivacija i gradijenata za mikro-serije, što uzrokuje lošiju memorijsku efikasnost i bržu izvedbu. Detaljnije o izvedbi i radu PipeDream-a dostupno je na [21]. Spomenimo još aktualni Xpipe [23] koji uvodi predviđanje težina kao kompromis između GPipe-a i PipeDream-a.

5. Usporedba izvedbe na DNN modelu

U nastavku je obrađen jednostavan primjer duboke neuralne mreže s tri sloja. Uspoređeno je vrijeme izvođenja na CPU-u i heterogenom sustavu (CPU-u i GPU-u). Za samu izvedbu korištena je platforma TensorFlow 2.0 te njen integrirani Keras API. Recimo nešto o specifikacijama računala:

1. CPU: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
2. GPU: NVIDIA GeForce GTX 1050 (3 GB)

5.1. Opis korištenog softvera

5.1.1. TensorFlow

TensorFlow je sveobuhvatna platforma otvorenog koda za strojno učenje. Posjeduje alate i biblioteke za razvoj modela DNN-a, treninga DNN-a te primjene i optimizacije modela spremnog za korištenje. Osnovna podrška u obliku API-ja dostupna je za (programske) jezike Python, JavaScript, C++ i Java. Zajednica koja okružuje platformu TensorFlow potaknuta je na samostalno održavanje nekih od jezika: Haskell, C#, Julia, R, Ruby i ostalih. Kao što je vidljivo na (Slika 3) TensorFlow spaja rad biblioteke cuDNN i platforme CUDA (time i biblioteke cuBLAS) s kojima je omogućeno paralelno programiranje, odnosno korištenje resursa GPU-a. TensorFlow također omogućuje rad na distribuiranim sustavima.

Osnovna ideja rada TensorFlow-a je graf izračuna (eng. *computational*/*dataflow graph*), usmjereni graf u kojem vrhovi opisuju operacije, a bridovi predstavljaju protok podataka (tenzora) među tim operacijama. Tenzore pritom definiramo kao apstraktni natpojam matrica i vektora. Osnovna specifikacija tenzora je dimenzija. Shodno tome, skalar, vektor i matrica predstavljaju tenzore nulte, prve i druge dimenzije respektivno.

Vrhovi grafa raspoređuju se na uređaje (CPU, GPU ili TPU), odnosno radnike, pomoću algoritma dodjeljivanja (eng. *placement algorithm*). Više o modelu izvođenja i implementaciji TensorFlow-a dostupno je u radu autora Petera Goldsborougha „A Tour of TensorFlow” [24].

5.1.2. Keras

Keras je API za strojno učenje napisan u jeziku Python te se koristi kao sučelje za TensorFlow. Osnovni pojmovi i strukture u Kerasu pojmovi su iz strojnog učenja (slojevi neurona i modeli), čime je rad s neuralnim mrežama sveden na slaganje slojeva i određivanje funkcija aktivacije te loss funkcije.

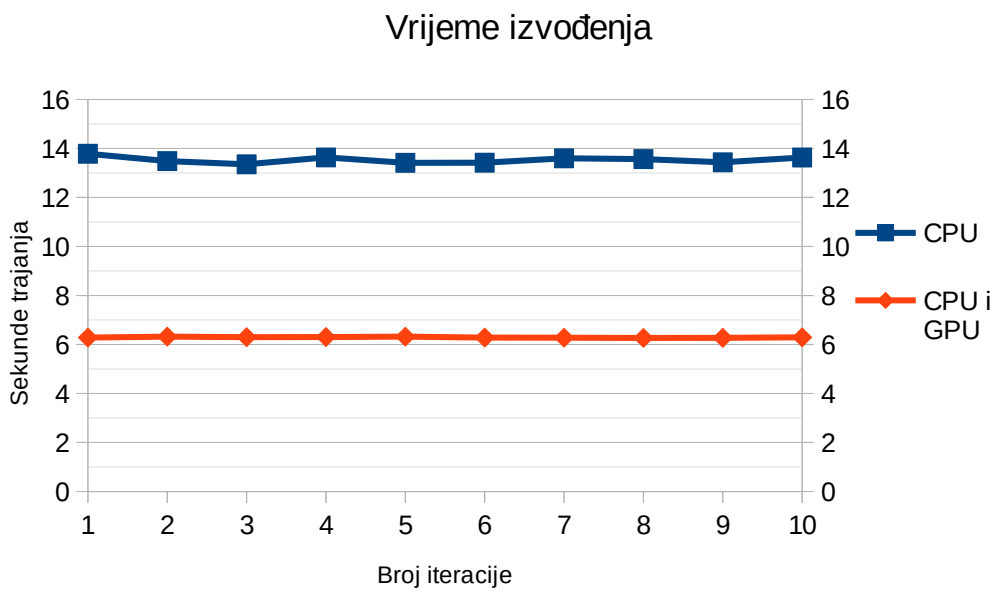
5.2. Model

Kao primjer za usporedbu izvedbe treninga DNN-a koristit ćemo prethodno spomenuti primjer prepoznavanja rukom pisanih znamenaka. Točnije, model će prepoznavati brojeve od 0 do 9 na slikama dimenzija 28 px * 28 px iz standardne baze podataka za praktične primjere, zvane „MNIST” (Modified National Institute of Standards and Technology database).

Model se sastoji od 3 gusto povezana sloja od kojih prvi ima 512, drugi 256 i treći 10 neurona. Aktivacijske funkcije koje se primjenjuju su prije spomenuta ReLu i Softmax funkcija. Prilikom propagacije unazad koristi se optimizirajuća funkcija RMSprop te kategorička unakrsna entropija (eng. *categorical cross-entropy*) kao loss funkcija.

Model je temeljen na praktičnom primjeru iz knjige „Deep learning with Python” [25].

5.3. Rezultati



Graf 5: Grafički prikaz vremena trajanja po izvođenju treninga modela

Nakon 10 iteracija, prosječno vrijeme izvođenja na kombinaciji CPU-a i GPU-a iznosi 6.291 sekundi, dok na samom CPU-u iznosi 13.531 sekundi. Prilikom izvođenja na kombinaciji CPU-a i GPU-a na vrhuncu se postiže 61% iskorištenja jezgara GPU-a (uz korištenje 3303 MiB GPU memorije) te približno 15% iskorištenosti svih 8 jezgara CPU-a. Na drugom primjeru (izvođenje na CPU-u) očekivano je vidljiv porast korištenosti (svih 8) CPU jezgara na 70%. Ubrzanje postignuto korištenjem GPU-a, odnosno paradigmi paralelnog programiranja je dakle 2.151 puta što jasno opisuje važnost paralelizma i korištenja GPU-a prilikom treninga DNN-a. Razlika u vremenu izvođenja, dakako, raste proporcionalno s porastom kompleksnosti modela i ulaznih podataka.

6. Zaključak

Ideje dubokog učenja postoje od kraja 20. stoljeća, međutim, usko grlo stvaraju nedostaci u softveru, hardveru i količini podataka dostupnih za treniranje modela. Do novih otkrića na području ne dolazi prije početka korištenja GPU-a, u tada, neuobičajene svrhe. Činjenica da trening jednostavnog modela DNN-a više ne mora trajati satima (ponekad i danima) te dostupnost velike količine podataka, bilo strukturiranih, bilo nestrukturiranih dovodi do nove revolucije u polju umjetne inteligencije. Kao što je vidljivo u poglavlju (5), razvoj i rad s DNN-om danas je dostupan svakome tko posjeduje osobno računalo s grafičkom karticom (grafičkim procesorom). Jačanje hardvera i velika količina podataka dovode do razvoja novih algoritama u radu s neuralnim mrežama. Kako danas postoje grafički procesori dovoljno snažni za izvođenje svih operacija u razumnom vremenu, usko grlo razvoja kompleksnih dubokih neuralnih mreža polako prelazi s hardvera na komunikaciju. Tako glavni problem u radu masivnih računalnih klastera stvara prijenos podataka u modelu, a ne samo izvođenje operacija nad spomenutim podacima. Paralelizacija neuralnih mreža i razvoj GPU-a su dakle kamen temeljac razvoja umjetne inteligencije koja pronalazi primjenu u svim područjima ljudskog života.

Literatura

[1] J. McCarthy, M. L. Minsky, N. Rochester, i C. E. Shannon, „A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence, August 31, 1955“, *AI Mag.*, sv. 27, izd. 4, str. 12, pros. 2006, doi: 10.1609/aimag.v27i4.1904.

[2] A. M. Turing, „Computing machinery and intelligence“, *Mind*, sv. 59, izd. October, str. 433–60, 1950.

[3] C. Kumar, „Artificial Intelligence: Definition, Types, Examples, Technologies“. Artificial Intelligence: Definition, Types, Examples, Technologies

[4] Rockship, „[No. 4 — AI] Related Fields“. https://medium.com/@social.rockship/no-4-ai-related-fields-6cf2facda32f

[5] IBM Cloud Education, „Machine Learning“. https://www.ibm.com/cloud/learn/machine-learning

[6] T. Mitchell, *Machine Learning*. McGraw-Hill, 1997. [Na internetu]. Dostupno na: https://www.cin.ufpe.br/~cavmj/Machine%20-%20Learning%20-%20Tom%20Mitchell.pdf

[7] S. Sharma, „Activation Functions in Neural Networks“, *Activation Functions in Neural Networks*. https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

[8] J. Brownlee, „Difference Between a Batch and an Epoch in a Neural Network“, *Difference Between a Batch and an Epoch in a Neural Network*. https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/

[9] IBM, „Power 4, The First Multi-Core, 1GHz Processor“, [Na internetu]. Dostupno na: https://www.ibm.com/ibm/history/ibm100/us/en/icons/power4/

[10] P. Gepner i M. F. Kowalik, „Multi-Core Processors: New Way to Achieve High System Performance“, u *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)*, ruj. 2006, str. 9–13. doi: 10.1109/PARELEC.2006.54.

[11] M. J. Flynn, „Some Computer Organizations and Their Effectiveness“, *IEEE Trans. Comput.*, sv. C–21, izd. 9, str. 948–960, ruj. 1972, doi: 10.1109/TC.1972.5009071.

[12] „Introduction to Parallel Computing“, *Introduction to Parallel Computing*. https://www.geeksforgeeks.org/introduction-to-parallel-computing/

[13] NVIDIA, „CUDA Toolkit documentation“. NVIDIA Developer Zone. [Na internetu]. Dostupno na: https://docs.nvidia.com/cuda/cublas/index.html

[14] NVIDIA, „NVIDIA cuDNN Documentation“. NVIDIA Accelerated Computing. [Na internetu]. Dostupno na: https://docs.nvidia.com/deeplearning/cudnn/index.html

[15] NVIDIA, „NVIDIA TensorRT documentation“. NVIDIA Accelerated Computing. [Na internetu]. Dostupno na: https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html

[16] „BLAS (Basic Linear Algebra Subprograms)“. Netlib. [Na internetu]. Dostupno na: http://www.netlib.org/blas/#_blas_routines

[17] M. Harris, „An Easy Introduction to CUDA C and C++“. https://developer.nvidia.com/blog/easy-introduction-cuda-c-and-c/

[18] S. Chetlur *i ostali*, „cuDNN: Efficient Primitives for Deep Learning“. 2014.

[19] V. Hegde i S. Usmani, „Parallel and Distributed Deep Learning“, Tech report. [Na internetu]. Dostupno na: https://web.stanford.edu/~rezab/classes/cme323/S16/projects_reports/hedge_usmani.pdf

[20] R. K. L. Kennedy, T. M. Khoshgoftaar, F. Villanustre, i T. Humphrey, „A parallel and distributed stochastic gradient descent implementation using commodity clusters“, *J. Big Data*, sv. 6, izd. 1, str. 16, velj. 2019, doi: 10.1186/s40537-019-0179-2.

[21] D. Narayanan *i ostali*, „PipeDream: Generalized Pipeline Parallelism for DNN Training“, u *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2019, str. 1–15. doi: 10.1145/3341301.3359646.

[22] Y. Huang *i ostali*, „Gpipe: Efficient training of giant neural networks using pipeline parallelism“, *Adv. Neural Inf. Process. Syst.*, sv. 32, str. 103–112, 2019.
[23] L. Guan, W. Yin, D. Li, i X. Lu, „XPipe: Efficient pipeline model parallelism for multi-GPU DNN training“, *ArXiv Prepr. ArXiv191104610*, 2019.
[24] P. Goldsborough, „A Tour of TensorFlow“. 2016. [Na internetu]. Dostupno na: https://arxiv.org/pdf/1610.01178.pdf
[25] F. Chollet, *Deep learning with Python*. Manning. [Na internetu]. Dostupno na: https://www.manning.com/books/deep-learning-with-python
[26] A. Dhalla, „The Rise and Fall of the Perceptron“. https://ai.plainenglish.io/the-rise-and-fall-of-the-perceptron-c04ae53ea465

Popis slika

Popis tablica

Popis priloga

Uz ovaj rad priložen je kod korišten u svrhu mjerenja opisanog u 5. poglavlju:

- model_DNN.py – program koji izvodi trening modela DNN