SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 220

KLASIFIKACIJA PODATAKA UPORABOM PARALELNIH NEURONSKIH MREŽA

Nenad Krpan

Zagreb, lipanj 2011.

Sadržaj

1. Uvod

Neuronske mreže služe za klasifikaciju i predviđanje i same mogu pronaći uzorke u podacima. Iako mreže rade brzo, njihovo učenje može dugo trajati. U većini današnjih računalnih sustava postoje programabilni grafički procesori i višejezgreni procesori s kojima se učenje može ubrzati. Stoga su u ovom radu istražene mogućnosti paraleliziranja učenja neuronskih mreža sa OpenCL-om. Napravljene su paralelne implementacije koje za učenje koriste algoritam roja čestica (engl. *particle swarm optimization*) i algoritam *backpropagation*. Ispitana su ubrzanja s obzirom na slijedne inačice algoritama.

Neuronska mreža primijenjena je na problem klasifikacije poslova na grozdovima računala. Za raspoređivanje poslova na grozdu može biti korisno znati unaprijed hoće li neki posao završiti uspješno ili s greškom. Mreža je učena podacima s grozdova i ispitano je koliko poslova završenih s pogreškom može pronaći.

U drugom poglavlju opisan je OpenCL i kako se koristi. U trećem poglavlju objašnjen je algoritam roja čestica, a u četvrtom neuronska mreža i načini učenja. U petom poglavlju opisano je na koji način su algoritmi paralelizirani radi ubrzanja. U šestom poglavlju ispitano je ubrzanje s obzirom na slijedne inačice algoritama i kako se ubrzanje mijenja s brojem uzoraka i veličinom mreže. U sedmom poglavlju se implementirana mreža koristila za predviđanje poslova s pogreškom.

2. OpenCL

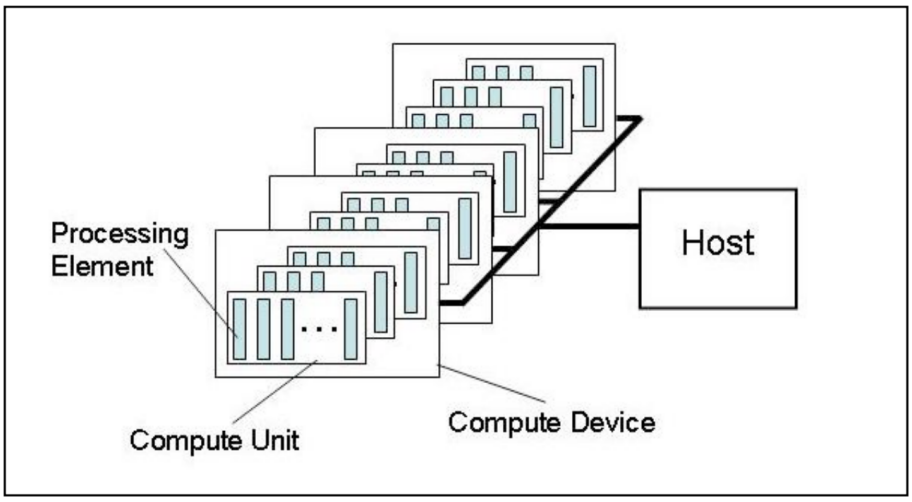
Glavni način ubrzanja kod modernih CPU (engl. *Central Processing Unit* – centralni procesor) arhitektura više nije povećanje frekvencije, nego dodavanje više jezgri. Grafički procesori (engl. *Graphics Processing Unit - GPU*) su se razvili iz uređaja za prikazivanje grafike u programabilne paralelne procesore s velikim brojem jezgri. Danas veliki broj računala ima takve vrste centralnih i grafičkih procesora i važno je da programeri iskoriste sve te heterogene platforme. Problem je što su uobičajene metode za programiranje višejezgrenih CPU-ova različite od metoda programiranja GPU-ova i zbog toga je stvoren OpenCL.

OpenCL (*Open Computing Language*) je otvoreni standard za paralelno programiranja opće namjene i može se koristiti za programiranje CPU-ova, GPU-ova i drugih paralelnih procesora (npr. Cell BE, DSP (*Digital Signal Processor*)). Izrađuje ga Khronos Group, a inačicu 1.0 specifikacije objavili su 06.10.2009.

Slika 2.1: OpenCL logo

OpenCL sastoji se od API (engl. *Application Programming Interface*) funkcija za koordiniranje paralelnog računanja preko heterogenih procesora i OpenCL C programskog jezika koji je podskup ISO C99 jezika sa dodacima za paralelizam. Iako su API funkcije napravljene za običan C, postoji i C++ omotač (engl. *wrapper*) za njihovo jednostavnije korištenje. [1]

2.1 Model platforme



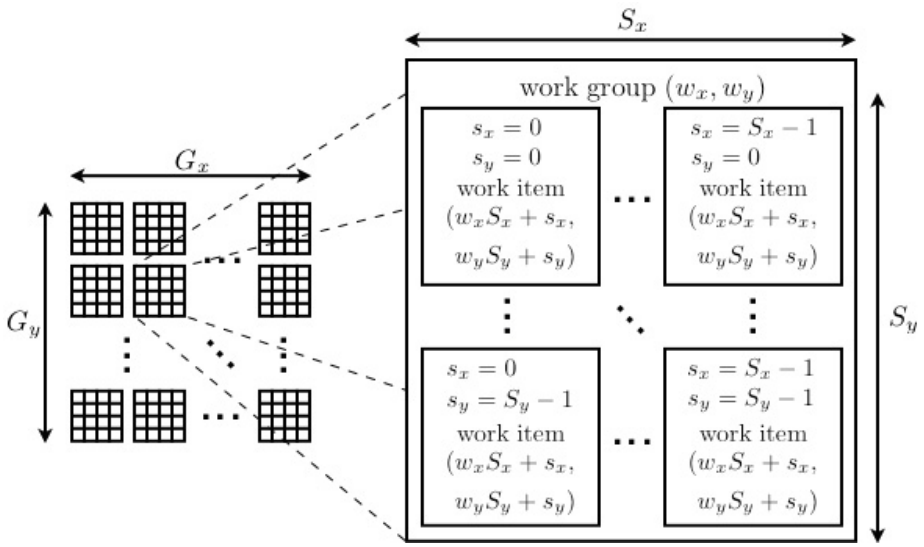
Slika 2.2: Model OpenCL platforme

Na slici 2.2 vidi se model OpenCL platforme koji se sastoji od domaćina – *host* (npr. CPU) na kojemu se izvršava glavni program (sa slijednim dijelovima i inicijalizacijom OpenCL-a). Domaćin je spojen na jedan ili više računskih uređaja (engl. *compute device*) – *device* (npr. GPU) na kojima se izvršava OpenCL C program, tj. *kernel* funkcije. Svaki računski uređaj ima jednu ili više računskih jedinica (engl. *compute unit*, za GPU je to protočni multiprocesor (engl. *streaming multiprocessor)*, a za CPU jezgra) koji se sastoji od jednog ili više procesirajućih elemenata (engl. *processing element – PE*).

Na jednom procesirajućem elementu se izvršava jedna radna jedinica – *work-item* (za GPU je to dretva), a na računskoj jedinici se izvodi radna grupa (engl. *work-group*). Radna grupa je uvijek na istoj računskoj jedinici i ne može se prebaciti na drugu, ali na jednoj radnoj jedinici može istovremeno biti više radnih grupa.

U OpenCL-u moguća je paralelizacija na razini podataka i na razini zadataka, ali OpenCL je uglavnom oblikovan za podatkovni paralelizam.

2.2 Programiranje *kernel* funkcija



Slika 2.3: Prostor indeksa (NDRange)

Kernel je funkcija napisana u OpenCL C-u koja se pokreće na uređaju. Kada domaćin preda *kernel* uređaju na izvođenje, definira se prostor indeksa (*NDRange – N-dimensional range,* $N \in \{1,2,3\}$) i jedna instanca *kernela* (radna jedinica - *work-item*) pokreće se za svaku točku u prostoru, tako da svaka radna jedinica ima različiti globalni indeks. Radne jedinice su organizirane u radne grupe (engl. *work group*). Svaka radna grupa ima svoj indeks i svaka radna jedinica ima svoj indeks unutar grupe i globalni indeks. Sve radne jedinice iz iste grupe se istodobno izvode na istoj računskoj jedinici.[2][3]

Na slici 2.3 je 2-D prostor indeksa sa radnim jedinicama koje su podijeljene u radne grupe. Veličina NDRangea - globalna veličina rada (engl. *global work-size*) je (G_x,G_y), a veličina radne grupe - lokalna veličina rada (engl. *local work-size*) je (S_x,S_y) za pojedine dimenzije. Indeksi počinu od 0. Sve radne grupe su jednako velike i globalna veličina rada mora biti višekratnik lokalne veličine rada.

```
__kernel void vector_add(__global float *A,
                         __global float *B,
                         __global float *C,
                         int N)
{
    int idx = get_global_id(0);
    if(idx < N)
         C [idx] = A[idx] * B[idx];
}
```

Slika 2.4: Primjer OpenCL kernela

Na slici 2.4 je primjer jednostavne kernel funkcije koja zbraja dva vektora A i B i sprema rezultat u C. Funkcija na početku mora imati ključnu riječ `__kernel` i vratiti `void`. Vektore dobije preko pokazivača u globalnoj memoriji (poglavlje 2.3). Za sve varijable unutar funkcije i za sve pokazivače potrebno je navesti vrstu memorije u kojoj se nalaze (ključne riječi: `__global`, `__constant`, `__local`, `__private`), a ako nije navedeno onda se varijabla nalazi u privatnoj memoriji. Sa ugrađenom funkcijom `get_global_id(uint dim)` radna jedinica saznaje svoj indeks za dimenziju `dim` u prostoru indeksa i računa svoj element u vektoru C. Budući da globalna veličina rada mora biti višekratnik lokalne veličine rada, može se dogoditi da globalna veličina bude veća od broja elemenata vektora (*N*) i zato je potrebno provjeriti je li indeks radne jedinice manji od *N*.

U tablici 2.1 se vide ugrađene funkcije za rad s indeksima koje se mogu koristiti unutar *kernela*.

Funkcije	Opis

`uint get_work_dim()`
`size_t get_global_size(uint dim)`
`size_t get_local_size(uint dim)`
`size_t get_num_groups(uint dim)`
	Funkcije vraćaju broj dimenzija, globalne/lokalne radne veličine i broj grupa u dimenziji `dim`.

`size_t get_global_id(uint dim)`
`size_t get_local_id(uint dim)`
`size_t get_group_id(uint dim)`
	Funkcije vraćaju globalne i lokalne indekse i indeks grupe u dimenziji `dim` za radnu jedinicu.

Tablica 2.1: Funkcije za rad s indeksima

Unutar *kernela* moguće je zvati druge funkcije (koje ne moraju imati ključnu riječ `__kernel`), ali rekurzija nije moguća.

Sinkronizacija na razini radne grupe radi se s funkcijom `barrier(flags)` a `flags` mogu biti `CLK_LOCAL_MEM_FENCE` ili `CLK_GLOBAL_MEM_FENCE` s kojima se osigurava da će sva pisanja u lokalnu ili globalnu memoriju biti gotova prije čitanja iz nje.

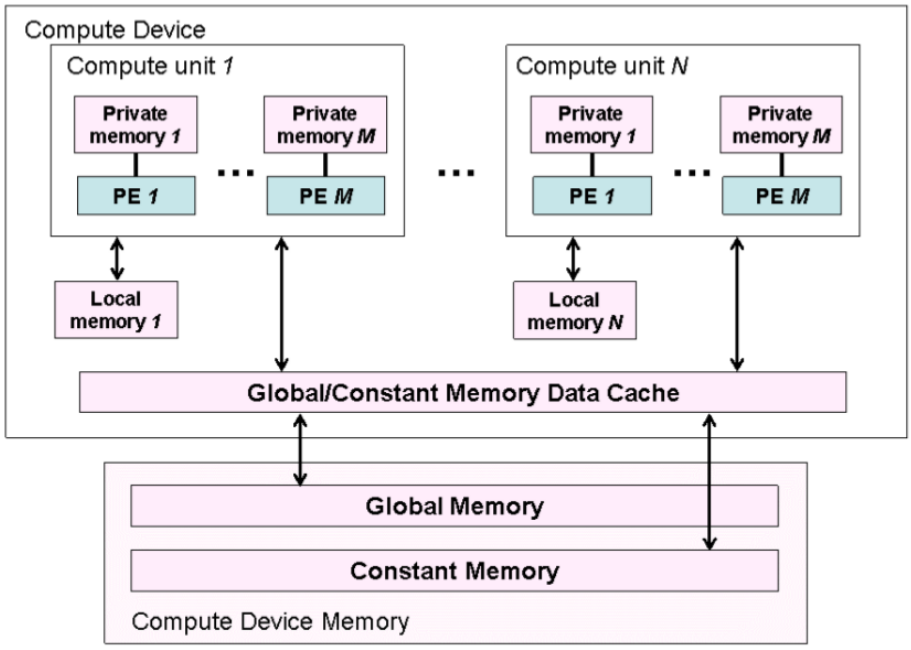
2.3 Model memorije

Na slici 2.5 vidi se razlika u podjela prostora na čipu za memoriju, upravljanje i računanje na CPU i GPU arhitekturama. Pri obradi podataka na procesorima, brzina glavne memorije u dosta slučajeva je usko grlo i zbog toga CPU veliki udio tranzistora potroši na priručnu (engl. *cache*) memoriju s više razina kako bi se smanjila latencija. S druge strane GPU više tranzistora potroši na elemente za

Slika 2.5: Razlika u CPU i GPU arhitekturama

računanje zbog čega ima više FLOPS-a (*floating point operations per second*), ali latenciju prema glavnoj memoriji mora skrivati s velikim brojem dretvi. Dok jedne dretve čekaju podatke iz glavne memorije, druge računaju. [4]

Zbog toga je u OpenCL-u programeru omogućeno direktno upravljanje s pojedinim vrstama memorija za njihovo učinkovitije korištenje.



Slika 2.6: OpenCL model memorije

Na slici 2.6 je shema jednog OpenCL računskog uređaja s njegovom memorijom. Postoje četiri vrste memorija:

- Globalna memorija (engl. *global memory*) je glavna memorija iz koje se čitaju podaci i spremaju rezultati. Najveća je, najsporija i može joj se pristupati s uređaja (iz *kernel* funkcije) i domaćina (s API funkcijama).
- Konstantna memorija (engl. *constant memory*) je dio globalne memorije koji se ne mijenja tijekom izvođenja *kernela*. Zato što se ne mijenja može

jednostavno biti keširana na čipu bez brige o nekonzistentnosti podataka te je zbog toga brza. Domaćin ju inicijalizira, a mogu joj pristupiti sve dretve s uređaja. Veličina je barem 64KB.

- Lokalna memorija (engl. *local memory*) postoji posebna za svaku računsku jedinicu. Mogu joj pristupiti samo radne jedinice unutar radne grupe i na taj način mogu dijeliti podatke. To je brza memorija i ako se neki podaci koriste više puta unutar jedne radne grupe, onda bi se radi ubrzanja trebali kopirati iz globalne memorija u lokalnu memoriju i tamo ih koristiti. Veličina je barem 32KB.
- Privatna memorija (engl. *private memory*) je najbrža memorija. To su ustvari registri i svaka radna jedinica ima svoju memoriju te joj samo ona može pristupiti. U slučaju da radne jedinice koriste više privatne memorije nego što je fizički postoji, onda se počinje koristiti globalna memorija što može dovesti do usporenja programa.

	Globalna	Konstantna	Lokalna	Privatna
Domaćin	Dinamička alokacija Čitanje/pisanje	Dinamička alokacija Čitanje/pisanje	Dinamička alokacija Nema pristup	Ne može alocirati Nema pristup
Kernel	Ne može alocirati Čitanje/pisanje	Statička alokacija Samo čitanje	Statička alokacija Čitanje/pisanje	Statička alokacija Čitanje/pisanje

Tablica 2.2: Vrsta alokacije i način pristupa memorijama

U tablici 2.2 vidi se na koje načine se mogu alocirati pojedine vrste memorije na uređaju i može li se s njih čitati ili po njima pisati s domaćina ili iz *kernela*.

Aplikacija na domaćinu koristeći OpenCL API stvara memorijske objekte u globalnoj ili konstantnoj memoriji i s API funkcijama kopira podatke iz svoje memorije u te objekte i natrag.

Podaci u globalnoj i lokalnoj memorije su konzistentni unutar radne grupe nakon sinkronizacije, ali nije moguća globalna sinkronizacija (na razini svih radnih jedinica), tako da podaci u globalnoj memoriji ne moraju biti konzistentni za različite radne grupe.

2.4 Programiranje aplikacije za domaćina

Kernel funkcije se pokreću iz aplikacije na domaćinu gdje je potrebno definirati globalnu i lokalnu veličinu rada, uređaj na kojemu će se izvoditi i dr.

Prvo je potrebno odabrati platformu (npr. NVIDIA ili AMD; slika 2.7).

```
// provjeri se koliko ima platformi
cl_uint num_platforms;
clGetPlatformIDs(0,NULL,&num_platforms);

// alocira se memorija za cijeli popis i popuni se
cl_platform_id *platforms = new cl_platform_id[num_platforms];
clGetPlatformIDs(num_platforms,platforms,NULL);
```

Slika 2.7: Dohvaćanje popisa platformi

Zatim se stvara kontekst (slika 2.8). Kontekst (engl. *context*) je okruženje u kojem se *kernel* izvodi i uključuje računske uređaje, njihove memorije i redove za naredbe u koje se stavljaju *kerneli* i operacije sa memorijama.

```
cl_context_properties props[3];
props[0] = (cl_context_properties)CL_CONTEXT_PLATFORM;
props[1] = (cl_context_properties)platforms[p];
props[2] = (cl_context_properties)0;

// stvori se kontekst koji u sebi sadrzi GPU
cl_context context = clCreateContextFromType(props,
CL_DEVICE_TYPE_GPU,NULL, NULL, NULL);

// dohvati se prvi uredaj iz konteksta
size_t param_size;
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL,&param_size);
cl_device_id* devices = (cl_device_id*)malloc(param_size);
clGetContextInfo(context, CL_CONTEXT_DEVICES, param_size, devices,
                 NULL);
cl_device_id device = devices[0];
```

Slika 2.8: Stvaranje konteksta i dohvaćanje uređaja

Potrebno je odabrati uređaj na kojem će se izvoditi *kernel* (slika 2.8) te nakon toga stvoriti red za naredbe (engl. *command queue*) za taj uređaj (slika 2.9).

```
cl_command_queue queue;
queue = clCreateCommandQueue(context, device, 0, NULL);
```

Slika 2.9: Stvaranje reda za naredbe

Kerneli se ne prevode (engl. *compile*) zajedno sa aplikacijom za domaćina, nego se prevode tijekom izvođenja. Stvara se programski objekt (`cl_program`) iz *stringa* koji sadrži izvorni kod (slika 2.10). Program se prevodi i iz njega se napravi *kernel* objekt.

```
char *source;
// izvorni kod kernel funkcije ucita se u string source
...

cl_program program;
program = clCreateProgramWithSource (context,1,&source,NULL,NULL);

// prevodenje programa
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// stvaranje kernel objekta za funkciju vector_add
cl_kernel kernel;
kernel = clCreateKernel(program, "vector_add", 0);
```

Slika 2.10: Stvaranje kernel objekta

Memorija na uređaju se koristi tako da se naprave memorijski objekti (`cl_mem`) i u njih se kopiraju podaci (slika 2.11). Kod stvaranja objekata sa zastavicama se odabere za što će se memorija koristiti u *kernelu* (pisanje ili čitanje - `CL_MEM_READ/WRITE_ONLY`) i moguće je odmah dati pokazivač na memoriju na domaćinu i sa zastavicom `CL_MEM_COPY_HOST_PTR` odabrati da se podaci odmah kopiraju. Za kopiranje podataka u već postojeće memorijske objekte koristi se funkcija `clEnqueueWriteBuffer`.

```
// memorija za vektore na domacinu
float *h_A, *h_B, *h_C;
int N; //broj elemenata
// rezervira se memorija za  h_A, h_B i h_C i popune se h_A i h_B
...

// stvaranje memorijskih objekata na uredaju i kopiranje podataka
cl_mem d_A = clCreateBuffer(context, CL_MEM_READ_ONLY |
              CL_MEM_COPY_HOST_PTR, sizeof(float) * N, h_A, NULL);
cl_mem d_B = clCreateBuffer(context, CL_MEM_READ_ONLY |
              CL_MEM_COPY_HOST_PTR, sizeof(float) * N, h_B, NULL);
cl_mem d_C = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
              sizeof(float) * N, NULL, NULL);
```

Slika 2.11: Stvaranje i korištenje memorijskih objekata

Prije pokretanja *kernela* još je potrebno zadati argumente funkcije (slika 2.12) i odrediti neku veličinu radne grupe te izračunati koliki nam treba biti prostor indeksa. Tada pokrenemo *kernel* (stavimo ga u red za naredbe) i on se pošalje na uređaj.

```
// definiranje argumenata
clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_A);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_B);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_C);

// računanje veličine grupa i NDRange-a
size_t localWorkSize = 128;
int numGroups = (N+localWorkSize-1)/localWorkSize; // == ceil
size_t globalWorkSize = numGroups * localWorkSize;

// pokretanje kernela
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalWorkSize,
                       &localWorkSize, 0, NULL, NULL);
```

Slika 2.12: Pokretanje kernela

Nakon što *kernel* sve izračuna, rezultat se nalazi u vektoru C, ali na uređaju (`d_C`) i da bi se mogao koristiti potrebno ga je kopirati na domaćina (slika 2.13).

```
clEnqueueReadBuffer(queue_, d_C, CL_TRUE, 0, N*sizeof(float), h_C,
                    0, NULL, NULL);
```

Slika 2.13: Čitanje podataka s uređaja

Da se ne bi dogodilo da krenemo koristiti podatke u `h_C` prije nego se kopiraju, sa zastavicom CL_TRUE se označava da je to blokirajuće čitanje.

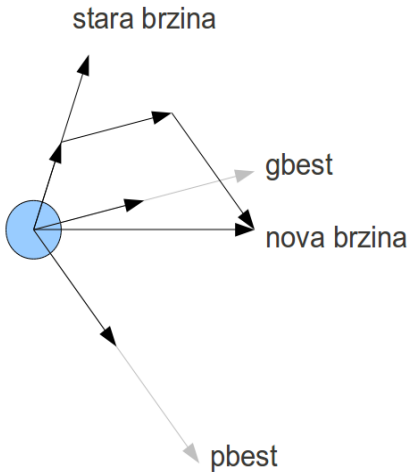
```
clReleaseMemObj(d_A);
clReleaseMemObj(d_B);
clReleaseMemObj(d_C);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(queue);
clReleaseContext(context);
```

Slika 2.14: Oslobađanje resursa

Kada više ne namjeravamo koristiti OpenCL resursi se mogu osloboditi (slika 2.14).

3. Algoritam roja čestica

Algoritam roja čestica (engl. *particle swarm optimization*) je optimizacijski algoritam inspiriran kretanjem jata ptica kojeg su 1995. godine opisali James Kennedy i Russell Eberhart u svom članku [5]. Čestice se nekom brzinom kreću u višedimenzijskom prostoru pretraživanja i traže optimalno ili skoro optimalno rješenje. Da bi algoritam bio bolji od slučajnog pretraživanja kod izbora slijedeće pozicije mora uzeti u obzir prethodno pretražena stanja i njihovu dobrotu. Pri određivanju smjera kretanja svaka jedinka uzima u obzir do tada najbolje pronađeno rješenje cijele populacije (socijalna komponenta) – *gbest* (engl. *global best solution*) i najbolje rješenje te jedinke (individualna komponenta) – *pbest* (engl. *particle's best solution*). Za sve čestice se smatra da imaju masu te zbog inercije ne mogu naglo mijenjati smjer kretanja. Zato kod ažuriranja brzine postoji inercijska komponenta koja ima smjer jednak prethodnoj brzini čestice ali manju (ili jednaku) amplitudu. Na slici 3.1 vidi se utjecaj pozicija *pbest* i *gbest* i stare brzine na novu brzinu čestice.



Slika 3.1: Računanje nove brzine čestice

Kada bi se koristila samo globalna i inercijska komponenta sve čestice bi se zaletile prema *gbest* poziciji koja bi se mijenjala jedino ako bi neka čestica slučajno prešla preko boljeg rješenja. Takav algoritam bi imitirao stvarno ponašanje ptica ali bi bilo jako malo istraživanja i čestice bi češće nalazile samo lokalne optimume. Zato je za bolji rezultat potrebno koristiti sve tri komponente.

Kako bi se spriječila divergencija čestica koristi se ograničenje brzine na neku unaprijed definiranu konstantu. [6][7]

$$\vec{v}_i = \omega \vec{v}_i + \varphi_p \vec{r}_p \left(\vec{pbest}_i - \vec{x}_i \right) + \varphi_g \vec{r}_g \left(\vec{gbest} - \vec{x}_i \right) \tag{3.1}$$

Ažuriranja brzine obavlja se izrazom (3.1) gdje su ω, φ_p i φ_g proizvoljno odabrani parametri za inerciju, individualnu i socijalnu komponentu, $\vec{r}_g$ i $\vec{r}_p$ slučajni vektori kojima su sve komponente iz intervala [0,1], a $\vec{v}_i$ i $\vec{x}_i$ brzina i položaj čestice *i.*

$$\vec{x}_i = \vec{x}_i + \vec{v}_i \tag{3.2}$$

Novi položaj *i*-te čestice računa se izrazom (3.2).

Dobar odabir za parametre je ω=0.7298 i φ_p=φ_g=1.4962. [8]

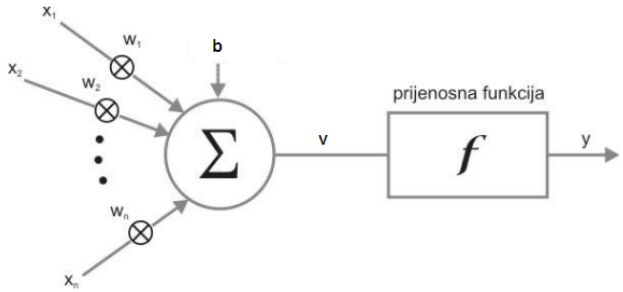
Na slici 3.2 vidi se pseudokod algoritma. Neki od uvjeta zaustavljanja mogu biti broj iteracija, vrijeme, dobrota najbolje jedinke, itd.

```
inicijaliziraj položaje i brzine čestica na slučajne
 vrijednosti
evaluiraj čestice
postavi pbest svake čestice na trenutni položaj
postavi gbest na položaj najbolje čestice

ponavljaj
    za svaku česticu
        izračunaj brzinu prema izrazu (3.1)
        ako je brzina veća od dozvoljene
            postavi brzinu na najveću dozvoljenu
        izračunaj novi položaj čestice prema izrazu (3.2)
        evaluiraj česticu na novom položaju
    kraj
dok nije zadovoljen uvjet zaustavljanja
```

Slika 3.2: Pseudokod algoritma roja čestica

4. Umjetna neuronska mreža

Umjetna neuronska mreža je sustav koji simulira rad bioloških neurona u mozgu. Jako dobro rješavaju probleme klasifikacije i predviđanja. U umjetnoj neuronskoj mreži koristi se pojednostavljeni model neurona (slika 4.1) gdje se izlaz računa tako da se sumiraju otežani ulazi (4.1), a zatim se na sumu primjeni prijenosna funkcija (4.2). [9][7]



Slika 4.1: Model neurona

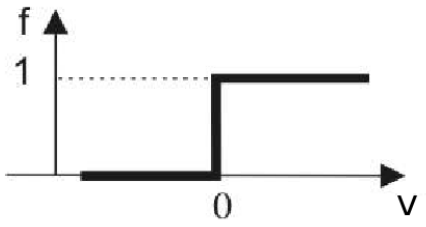
$$v = \sum_{i=1}^{n} w_i x_i - b \tag{4.1}$$

U jednadžbi (4.1) w_i označava težine, x_i ulaze u neuron, a b je pomak (engl. *bias*).

$$y = f(v) \tag{4.2}$$

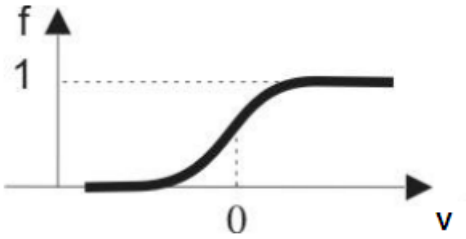
Izlaz neurona (y) zatim ide na ulaz drugog neurona ili na izlaz iz mreže. Neke od aktivacijskih funkcija su linearna (4.3), step (engl. *Threshold Logic Unit*, TLU, slika 4.2) (4.4) i sigmoidalna (slika 4.3) (4.5) funkcija.

$$f(v) = v \tag{4.3}$$



Slika 4.2: Step funkcija

$$f(v) = \begin{cases} 0, & v < 0 \\ 1, & v \geq 0 \end{cases} \tag{4.4}$$

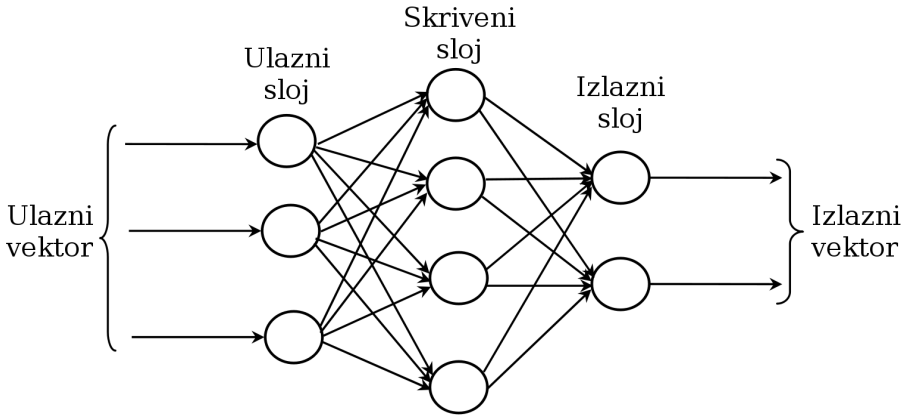


Slika 4.3: Sigmoidalna funkcija

$$f(v)=\frac{1}{1+e^{-a\cdot v}} \tag{4.5}$$

Prednost sigmoidalne funkcije u odnosu na TLU je u tome da je derivabilna, što može biti korisno kod učenja.

Neuronske mreže, za razliku od konvencionalnih metoda obrade podataka, su dobre u procjeni nelinearnih odnosa između uzoraka, mogu raditi s nejasnim ili manjkavim podacima i robusne su na pogreške u podacima.



Slika 4.4: Višeslojni perceptron

Najčešće korištena vrsta neuronske mreže je **višeslojni perceptron** (engl. *multilayer perceptron*, MLP) koji ima ulazni sloj, barem jedan skriveni sloj i izlazni sloj. Neuroni su povezani samo s neuronima iz susjednih slojeva.. Signal se šalje u jednom smjeru, od ulaza prema izlazu (engl. *feed-forward*). Za aktivacijsku funkciju najčešće se koristi sigmoidalna funkcija. Na slici 4.4 je primjer višeslojnog perceptrona sa 3 ulaza, jednim skrivenim slojem od 4 neurona i 2 izlaza.

Obradbena moć mreže pohranjena je u jačini veza između neurona odnosno težinama do kojih se dolazi postepenom prilagodbom tj. učenjem (treniranjem) iz skupa podataka za učenje. Uobičajen način učenja mreže je s algoritmom *backpropagation*, ali se može učiti i na druge načine, npr. s algoritmom roja čestica.

4.1 Algoritam *backpropagation*

Učenjem mreže težine veza se ugađaju kako bi se smanjila pogreška između stvarnog i željenog izlaza mreže, ali teško je odrediti koliko je koji skriveni neuron

odgovoran za pogrešku. Taj problem riješen je algoritmom propagacije pogreške unatrag (engl. *error back-propagation algorithm*). Pomoću pogreške na izlazu podešavaju se težine od skrivenog do izlaznog sloja i tako redom do ulaznog sloja s ciljem minimiziranja izlazne pogreške. Pogreška se tako propagira unatrag i istovremeno se podešavaju težine. Problem kod ovakvog učenja je što nakon što prođemo sve primjere za učenje, težine koje su se namjestile za prvi primjer, na kraju su se promijenile u sasvim drugom smjeru zbog čega je potrebno obaviti više prolazaka kroz sve primjere. Takvim algoritmom ustvari radimo gradijentni spust u lokalni minimum, zbog čega je potrebno da aktivacijska funkcija bude derivabilna. Sigmoidalna funkcija je prikladna za takav postupak zato što je derivabilna.[10]

Pogreška na izlaznom neuronu *j* gdje je d_j željeni izlaz, a y_j stvarni izlaz je:

$$e_j = d_j - y_j \tag{4.6}$$

Derivacija sigmoidalne funkcije je:

$$\phi_j'(v_j) = y_j \cdot (1 - y_j) \tag{4.7}$$

Gradijent izlaznog neurona *j* je:

$$\delta_j = e_j \cdot \phi_j'(v_j) \tag{4.8}$$

Gradijent neurona iz skrivenog sloja je (težina w_{kj} ulazi u neuron *k* iz neurona *j*):

$$\delta_j = \phi_j'(v_j) \sum_k \delta_k \cdot w_{kj} \tag{4.9}$$

Konačno, težine neurona se korigiraju na sljedeći način (η je stopa učenja):

$$\begin{aligned} \Delta w_{ji} &= \eta \delta_j \cdot y_i \\ w_{ji} &= w_{ji} + \Delta w_{ji} \end{aligned} \tag{4.10}$$

4.2 Učenje algoritmom roja čestica

Prednosti učenja algoritmom roja čestica su to da aktivacijska funkcija ne mora biti derivabilna i veća je vjerojatnost da će algoritam naći globalni minimum, a ne samo lokalni.

Dimenzionalnost prostora u kojem se čestice kreću jednaka je broju težina neuronske mreže (uključujući i pomake tj. *bias*). Broj dimenzija (težina) može se izračunati izrazom (4.11), uz to da je *L* broj slojeva, a N_i je broj neurona u sloju *i* $(i \in \{0, \ldots, L-1\})$. To znači da svaka čestica predstavlja jednu inačicu mreže i njena pozicija predstavlja težine te mreže.

$$D = \sum_{i=1}^{L-1} N_i * N_{i-1} \tag{4.11}$$

Dobrota čestice računa se tako da se za svaki primjer za učenje izračuna izlaz iz te mreže koju čestica predstavlja i izračuna srednja kvadratna pogreška (engl. *mean*

squared error - MSE) s obzirom na željeni izlaz (4.12). U jednadžbi (4.12) *N* je broj uzoraka za učenje, *O* veličina izlaznog vektora (broj izlaznih neurona), a y_{ij} i d_{ij} stvarni i željeni izlaz *i*-tog primjera za učenje *j*-tog izlaznog neurona.

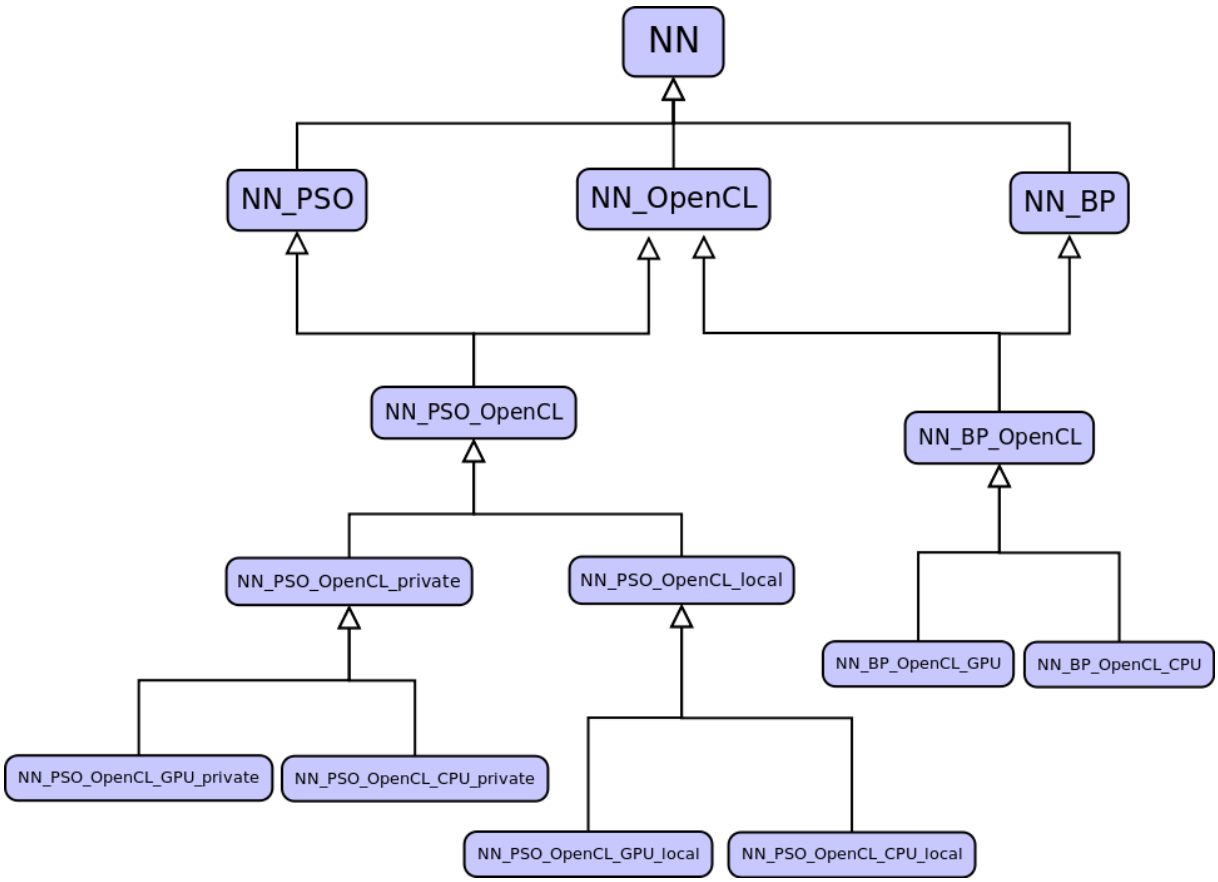
$$f = \frac{1}{N * O} \sum_{i=1}^{N} \sum_{j=1}^{O} (y_{ij} - d_{ij})^2 \tag{4.12}$$

Nakon učenja odabire se čestica s najboljom dobrotom čije se težine onda mogu koristiti za obradu novih podataka.

5. Implementacija neuronske mreže s OpenCL-om

Neuronske mreže su brze dok su već naučene, ali njihovo učenje može dugo trajati. Zato je napravljena implementacija koja koristi OpenCL za ubrzanje učenja. Mreža je višeslojni perceptron, a aktivacijska funkcija na svim neuronima je sigmoidalna (4.5). Ulazi u mrežu se skaliraju na raspon [-1,1], a izlazi iz mreže se skaliraju sa [0,1], koliki je raspon sigmoidalne funkcije, na željeni raspon. Korišten je algoritam roja čestica (poglavlje 4.2) i algoritam *backpropagation* (poglavlje 4.1). U radovima [11] i [12] predložene su metode paralelizacije algoritma *backpropagation* sa različitim arhitekturama mreže i podjelama podataka, a u radovima [12] i [13] je napravljena implementacija algoritma *backpropagationa* s CUDA-om za grafičke procesore.

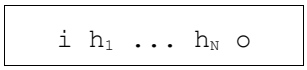
Na slici 5.1 je hijerarhija implementiranih klasa.



Slika 5.1: Dijagram klasa

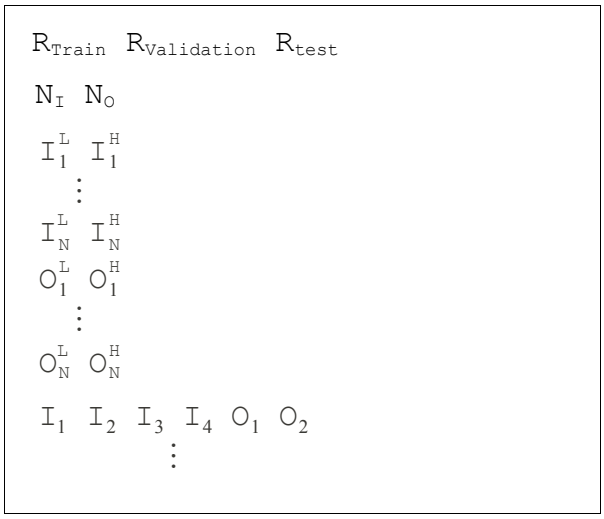
Klasa **NN** je apstraktna klasa koja radi zadatke zajedničke svim drugim ostvarenim neuronskim mrežama. Učitava datoteke s arhitekturom neuronske mreže i primjerima za učenje.

Format datoteke s arhitekturom mreže je na slici 5.2, gdje su `i` i `o` broj neurona u ulaznom i izlaznom sloju, a h_j broj neurona u `j`-tom skrivenom sloju. Npr. mreža s 4 ulaza, 2 izlaza i jednim skrivenim slojem s 3 neurona će imati "4 3 2" u datoteci.



Slika 5.2: Format datoteke s arhitekturom mreže

Format datoteka s primjerima za učenje je na slici 5.3. U prvom redu se određuje u kojem omjeru će se podijeliti primjeri na skup za učenje (R_{train}), skup za validaciju ($R_{validation}$) i skup za ispitivanje (R_{test}). Mogu se pisati npr. postoci "40 30 30" ili točan broj primjera "360 270 270". U drugom redu je broj ulaza (N_I) i izlaza (N_O). Nakon toga su rasponi ulaza i izlaza gdje je raspon j-tog ulaza $[I_j^L, I_j^H]$, a j-tog izlaza $[O_j^L, O_j^H]$. U ostalim redovima su parovi ulaza i izlaza: za ulaz (I_1,I_2,I_3,I_4) dobije se izlaz (O_1,O_2).



Slika 5.3: Format datoteke s primjerima za učenje

Učenje se obavlja sa skupom za učenje, a svakih nekoliko iteracija računa se srednja kvadratna pogreška (4.12) skupa za validaciju i pamti se do tada najbolja nađena mreža. Ta mreža se koristi nakon učenja. Za provjeru rezultata učenja koristi se skup za ispitivanje.

Klasa NN ima i člansku funkciju `train()` (koja je implementirana u izvedenim klasama) s kojom se mreža uči. Postoji i nadgrađeni (engl. *overloaded*) operator () koji za neki ulaz vraća izlaz naučene mreže.

Klasa **NN_PSO** uči mrežu koristeći slijedni algoritam roja čestica za CPU. Uz to još i radi stvari zajedničke svim mrežama koje koriste algoritam roja čestica kao što je učitavanje datoteke s parametrima za PSO. Format datoteke s parametrima je takav da je u svakom redu naziv atributa i vrijednost, odvojeni s ':'. Npr. "Population size:32" znači da će broj čestica biti 32. U tablici 5.1 vide se svi atributi koji se mogu postaviti iz datoteke te vrijednost koja će se koristiti ako atribut nije postavljen. Ako je zadovoljen barem jedan uvjet zaustavljanja, učenje se zaustavlja, a ako je neki uvjet 0 on se ne uzima u obzir.

Naziv	Opis	Unaprijed postavljeno
Population size	Broj čestica	64
omega	ω izraza 3.1	0.72984
phiP	φ_p izraza 3.1	1.49618
phiG	φ_g izraza 3.1	1.49618
Lower limit	Donja i gornja granica prostora pretraživanja unutar kojih se na početku slučajno razmjeste čestice	-1
Upper limit		1
Strip length	Svakih koliko iteracija se računa *MSE* skupa za validaciju	10
Iterations	Nakon koliko iteracija se zaustavlja učenje	0
Time	Nakon koliko sekundi se zaustavlja učenje	60
MSE	Ako *MSE* skupa za validaciju padne ispod zadanog broja, učenje se zaustavlja	0

Tablica 5.1: Atributi za algoritam roja čestica

Klasa **NN_OpenCL** radi stvari zajedničke svim mrežama koje koriste OpenCL. Inicijalizira OpenCL: stvara kontekst i red za naredbe, rezervira memoriji na uređaju (skupove za učenje, validaciju i ispitivanje u globalnoj memoriji i arhitekturu mreže i raspone izlaza u konstantnoj memoriji) i kopira podatke s domaćina te stvara i prevodi OpenCL program za ciljani uređaj.

NN_PSO_OpenCL je apstraktna klasa za algoritam roja čestica s OpenCL-om i ima dvije izvedene klase koje se razlikuju po vrsti memorije koju koriste.

NN_PSO_OpenCL_local je brža inačica koja koristi OpenCL i koristi lokalnu memoriju.

NN_PSO_OpenCL_private je sporija inačica koja koristi OpenCL i koristi privatnu memoriju.

NN_BP koristi slijedni algoritam *backpropagation* za CPU. Uz to još i učitava datoteku s parametrima za *backpropagation*. Format datoteke jednak je kao kod parametara za PSO, ali postoje drugi atributi (tablica 5.2). Uvjeti zaustavljanja su također isti kao u PSO.

Naziv	Opis	Unaprijed postavljeno
Learning rate	Stopa učenja (η iz (4.10))	0.02
Strip length	Svakih koliko iteracija se računa *MSE* skupa za validaciju	10
Lower limit	Donja i gornja granica unutar kojih se na početku slučajno namjeste težine	-1
Upper limit		1
Iterations	Nakon koliko iteracija se zaustavlja učenje	0
Time	Nakon koliko sekundi se zaustavlja učenje	60
MSE	Ako *MSE* skupa za validaciju padne ispod zadanog broja, učenje se zaustavlja	0

Tablica 5.2: Atributi za algoritam backpropagation

Klasa **NN_BP_OpenCL** koristi algoritam *backpropagation* implementiran s OpenCL-om.

Dodatno svaku klasu koja koristi OpenCL nasljeđuju dvije klase od kojih jedna računa na grafičkoj kartici, a druga koristi sve jezgre procesora, ali obje koriste isti algoritam.

5.1 Paralelni algoritam roja čestica

Algoritam učenja opisan u poglavlju 4.2 paraleliziran je s OpenCL-om tako da *kernel* funkcija računa izlaz iz mreže za jedan uzorak za učenje. Ako postoji S primjera za učenje i N čestica, u svakoj iteraciji pokreće se N*S radnih jedinica.

Na slici 5.4 vidi se pseudokod implementacije. Uzorci se ne mijenjaju i zato se samo jednom kopiraju na uređaj. Težine se mijenjaju u svakoj iteraciji i zato se moraju kopirati svaki put. Zatim se pokreće *kernel* koji računa izlaz za svaki uzorka za svaku česticu.

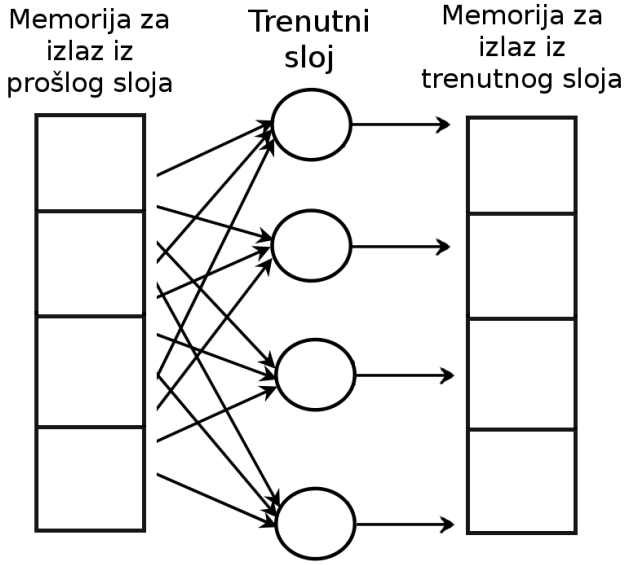
```
kopiraj uzorke na uredaj

ponavljaj
    kopiraj težine mreža (pozicije čestica) na uredaj
    paralelno simuliraj neuronske mreže
    izračunaj mse svake čestice
    kopiraj MSE-ove natrag na domacina
    izračunaj novi položaj čestice prema izrazu (3.2)
dok nije zadovoljen uvjet zaustavljanja
```

Slika 5.4: Pseudokod paralelnog algoritma roja čestica

Rezultati simulacija nalaze se na uređaju, i da se ne bi svi morali kopirati natrag, pokreće se još jedan *kernel* koji računa MSE svake čestice. MSE-i se kopiraju natrag

i na temelju njih se računaju nove pozicije čestica. To se računa na domaćinu zato što ne traje dugo s obzirom na ostatak algoritma i ne isplati se implementirati na OpenCL-u. Nove težine (pozicije) se opet kopiraju na uredaj i tako se ponavlja dok uvjet zaustavljanja ne bude zadovoljen.

Kada se simulira mreža, svakoj radnoj jedinici potrebna je određena memorija za međurezultate. Mreža se simulira sloj po sloj i da bi se izračunao izlaz iz sloja potrebno je u memoriji imati izlaz prošlog sloja i prostora za izlaz iz trenutnog sloja (slika 5.5). Jednom će u tom prostoru biti i izlaz iz najvećeg sloja i zato svaka radna jedinica treba imati unaprijed alocirano dva puta više memorije od one količine koja je dovoljna za najveći sloj.



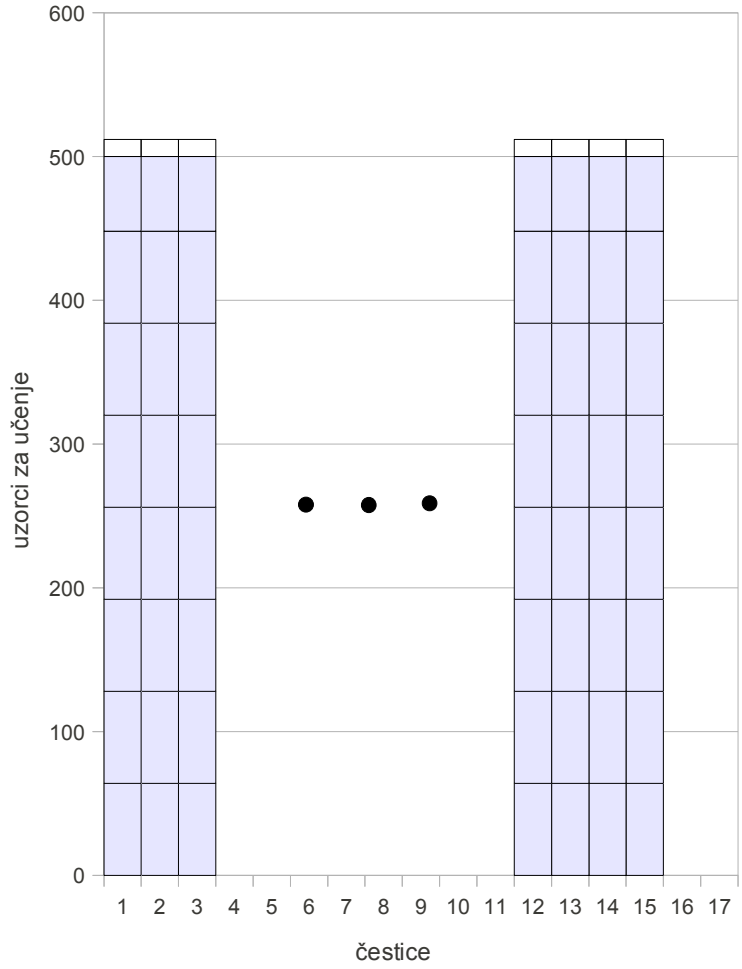
Slika 5.5: Memorijski zahtjevi neuronske mreže

Postoje dvije inačice *kernela* za simulaciju: sporiji i brži.

Sporiji za međurezultate koristi privatnu memoriju, koju je jedino moguće alocirati statički unutar *kernela*. Ali kako se OpenCL program prevodi tijekom izvođenja, izvorni kod se mijenja prije prevođenja pa veličina memorije ne mora biti unaprijed definirana.

Brži koristi lokalnu memoriju za međurezultate, ali i za težine. Na slici 5.6 se vidi primjer organizacije radnih jedinica u radne grupe za populaciju od 15 jedinki, 500 primjera za učenje i veličinu grupe od 64 radnih jedinica.

U jednoj grupi su samo jedinke koje koriste težine od iste čestice, ali računaju različite uzorke. Budući da sve radne jedinice u grupi koriste iste težine, moguće ih je samo jednom učitati u lokalnu memoriju i onda ih iz nje koristiti. Sve radne jedinice zajedno učitavaju težine tako da svaka učita jedan dio.



Slika 5.6: Radne grupe

Prema tome ukupna potrošnja lokalne memorije po grupi je opisana izrazom (5.1), uz to da N_W ukupan broj težina u mreži, G je broj radnih jedinica u grupi i W_{Lmax} je broj neurona u najvećem sloju i još se treba uračunati da se parametri od *kernela* prenose preko lokalne memorije. Zato grupe ne smiju biti prevelike i brži *kernel* ne može raditi za prevelike mreže.

$$L = 4 * (N_W + 2 * G * W_{Lmax})\ B \tag{5.1}$$

U primjeru na slici 5.6 se može primijetiti da budući da broj uzoraka (500) nije djeljiv s brojem grupa (64) grupe pri vrhu imaju radne jedinice koje ništa ne rade.

Računanje srednje kvadratne pogreške je također ostvareno OpenCL-om, ali kako je udio potrošenog vremena mali s obzirom na simulaciju, *kernel* za *MSE* nije jako optimiziran. Prednost računanja na uređaju je u tome da se svi izlazi neće morati kopirati natrag na domaćina, nego samo jedna vrijednost po čestici (što je S*O puta manje podataka; S-broj uzoraka, O-broj izlaznih neurona). Svaka grupa računa *MSE* za jednu česticu. Grupa se sastoji od 64 radne jedinice i svaka jedinica zbraja razlike kvadrata svog dijela izlaza iz mreže i sprema rezultat u lokalnu memoriju. Zatim ih jedna radna jedinica iz grupe sve zbroja, dijeli s brojem uzoraka i sprema u globalnu memoriju.

5.2 Paralelni algoritam *backpropagation*

Algoritam *backpropagation* (poglavlje 4.1) paraleliziran je tako da izlaz svakog neurona računa posebna radna jedinica. Dok se računa jedan sloj, svi izlazi iz prošlog sloja već trebaju biti izračunati. To bi se moglo osigurati jedno kada bi sve radne jedinice bile u istoj grupi, jer u OpenCL-u nema globalne sinkronizacije, ali to bi značilo da bi se koristila samo jedna računska jedinica i to bi bilo sporo, a i možda bi bilo previše neurona i ne bi svi stali u istu grupu. Zato je to ostvareno na način da *kernel* računa izlaz za samo jedan sloj i takav *kernel* se onda zove više puta sve do zadnjeg sloja.

```
// ulazni sloj je 0; L je broj slojeva
za i=1 do L-1
    izračunaj izlaz iz i-tog sloja (4.5)
kraj
// sada je izračunat izlaz iz mreže
za i=L-1 do 1
    izračunaj gradijent iz i-tog sloja (4.8), (4.9)
        i izračunaj Δw za i-ti sloj prema (4.10)
kraj

dodaj Δw težinama
```

Slika 5.7: Pseudokod paralelnog algoritma backpropagation

Na slici 5.7 je pseudokod paralelnog algoritma *backpropagation*. Za svaki sloj tijekom propagacije unaprijed zove se po jedan *kernel* i za svaki sloj tijekom propagacije unatrag također. Na kraju se još pozove jedan *kernel* koji dodaje izračunati Δw težinama.

Tako da se svake iteracije pokreće N_K (5.2) *kernela*, gdje je L broj slojeva, a S broj uzoraka.

$$N_K = S*(2*(L-1)+1) \tag{5.2}$$

Ako npr. mreža ima 4 sloja i uči se s 10.000 primjera, to je 70.000 pokretanja *kernela* svake iteracije. Kako svako pokretanje ima neki mali *overhead* ne može se očekivati da će implementacija raditi brže od procesor za male mreže.

6. Ispitivanje brzine paralelne implementacije neuronske mreže

Kako bi se utvrdilo ubrzanje postignuto s OpenCL-om mjerena su učenja mreže bez OpenCL-a i s OpenCL-om (na grafičkom i centralnom procesoru). Na CPU-u cijela radna grupa se izvršava na jednoj jezgri, a radne jedinice unutar grupe se izvršavaju slijedno.

Korišten hardver:

- GPU – NVIDIA GeForce GTS 250
- CPU – AMD Phenom II x6 1055T (6 jezgri)

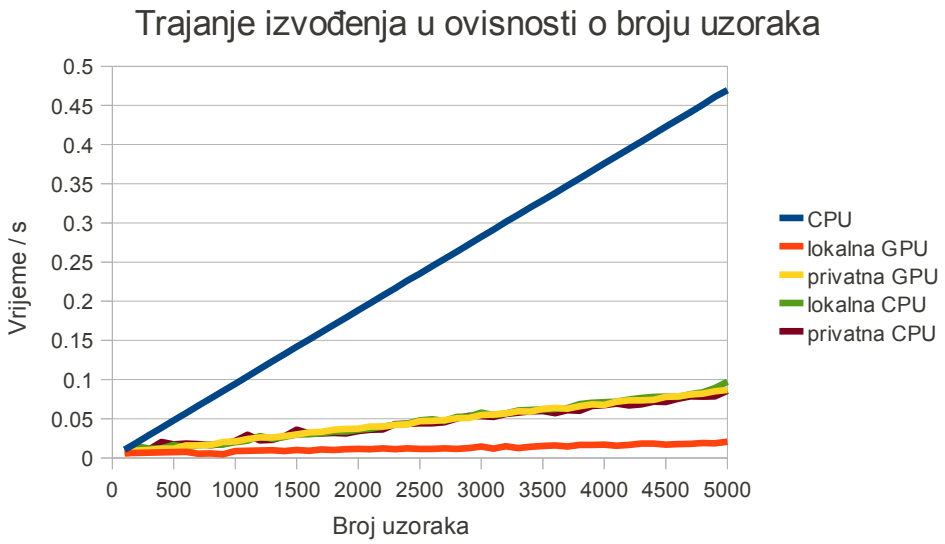
Primjeri za učenje su zapisi iz grozda "CTC-SP2-1996-2.1-cln" (više u poglavlju 7.) Neuronska mreža ima 9 ulaza i jedan izlaz.

6.1 Ispitivanje paralelnog algoritma roja čestica

Provedena su mjerenja slijednog algoritma roja čestica za CPU i paralelnog algoritma sa lokalnom i privatnom memorijom za CPU i GPU. CPU nema dodatnu brzu memoriju (nego koristi više razina priručnih memorija) i lokalna memorija je ustvari na hardverskom nivou na istom mjestu gdje i globalna (*RAM*). Broj neurona po slojevima mreže koja je korištena je: `9 15 6 1` (slika 5.2), a veličina populacije je 64 čestice.

6.1.1 Ispitivanje algoritma u ovisnosti o broju uzoraka

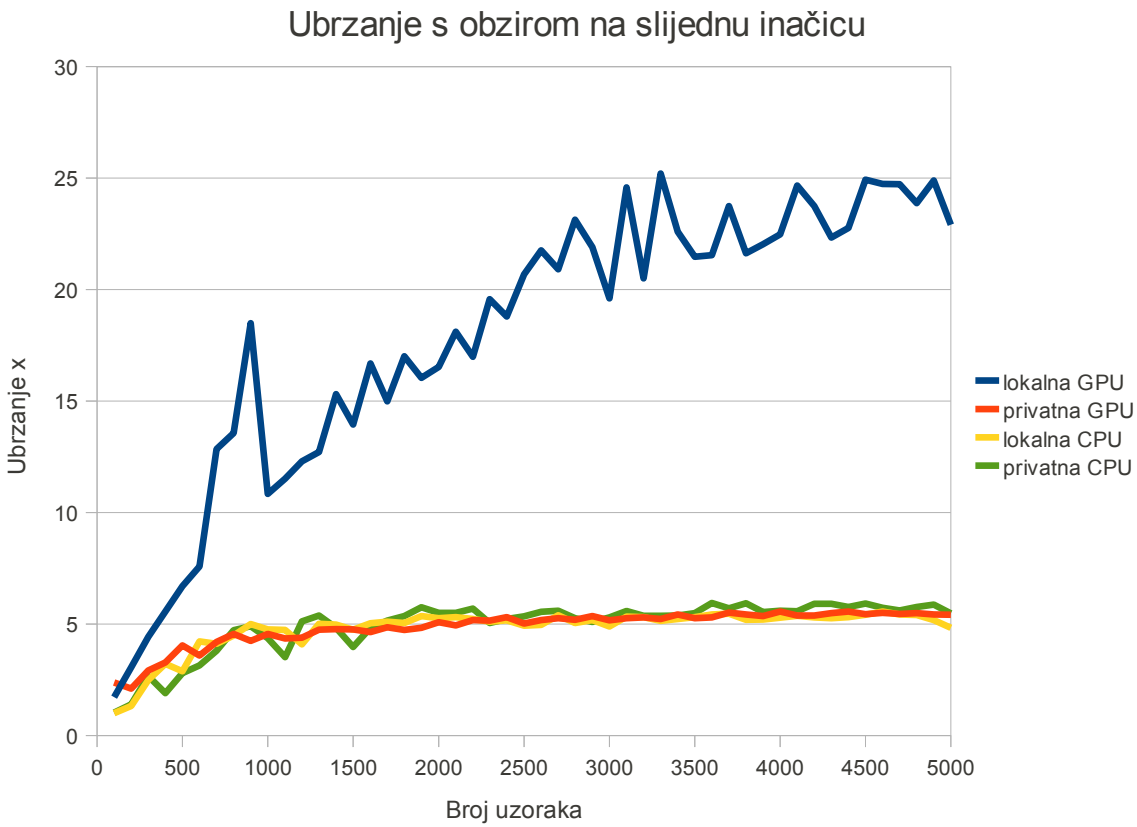
Mjerena su trajanja jedne iteracije algoritma roja čestica (svaka čestica računa pogrešku za sve uzorke) za različite brojeve uzoraka.



Slika 6.1: Trajanje izvođenja u ovisnosti o broju uzoraka

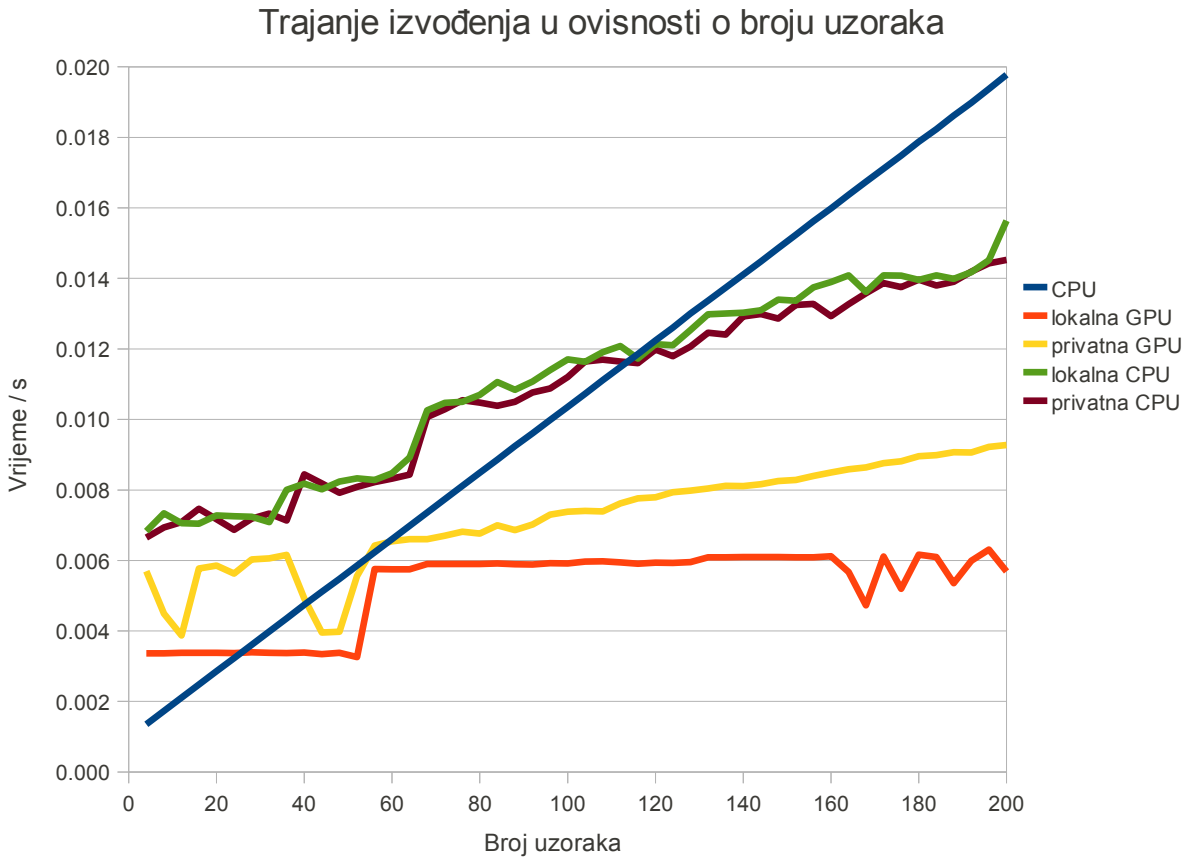
Na slici 6.1 se vidi da slijedni algoritam najduže traje, paralelni s lokalnom memorijom za GPU je najbrži, dok ostali imaju slična trajanja. Trajanje svake točke je prosjek 10

iteracija.



Slika 6.2: Ubrzanje s obzirom na slijednu inačicu

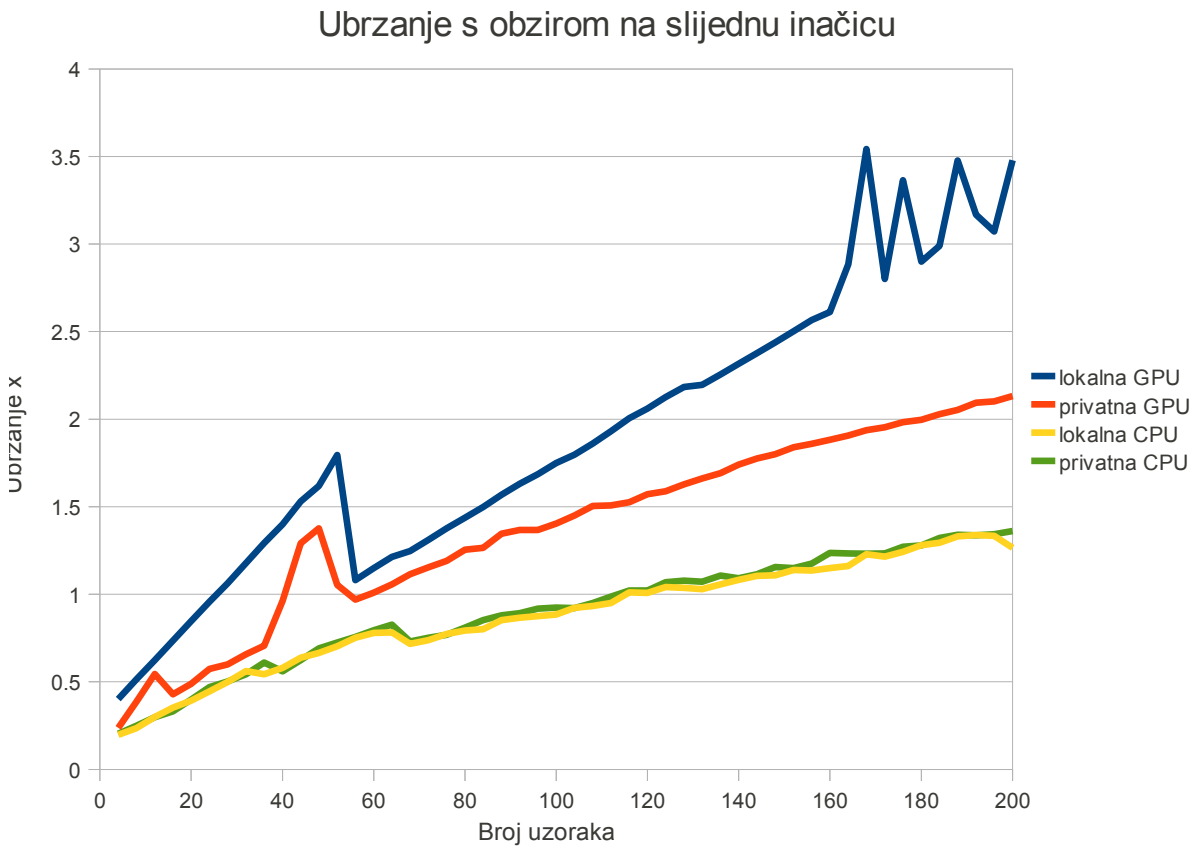
Na slici 6.2 se može bolje vidjeti ubrzanje ostvareno u odnosu na slijedni algoritam. Ubrzanje lokalne inačice za GPU brzo raste do otprilike 3000 uzoraka gdje je skoro 25 puta brža. Ostalim inačicama ubrzanje raste do 1000 uzoraka i brže su 5 do 6 puta od slijednog algoritma.



Slika 6.3: Trajanje izvođenja u ovisnosti o broju uzoraka

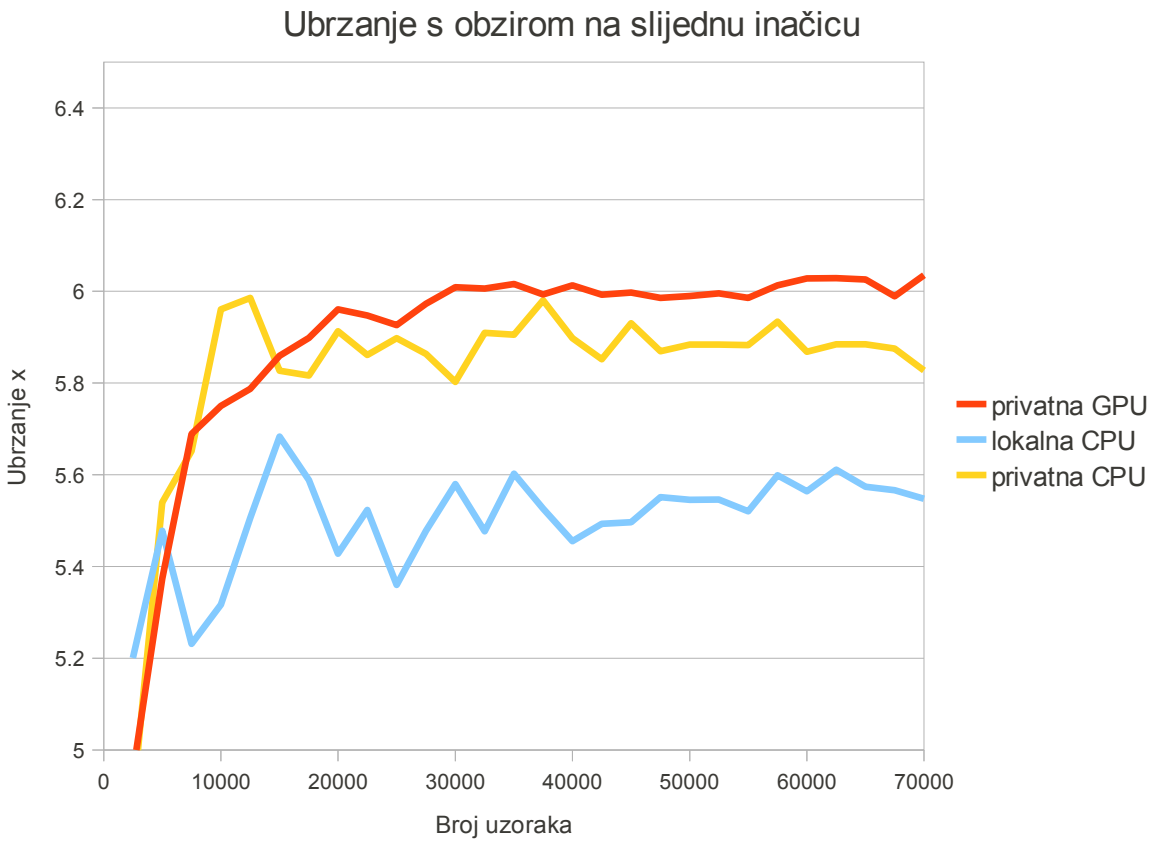
Provedeno je preciznije mjerenje uz manji broj uzoraka kako bi se vidjelo koliko je minimalno uzoraka potrebno da bi paralelne inačice bile brže od slijedne (slike 6.3 i 6.4, svaka točka je prosjek 500 iteracija), tj. da bi im ubrzanje bilo veće od 1.

GPU inačici s lokalnom memorijom je dovoljno samo 28 uzoraka da bi bila brža, a s privatnom memorijom 44 uzorka. CPU inačice brže su nakon 116 uzoraka.



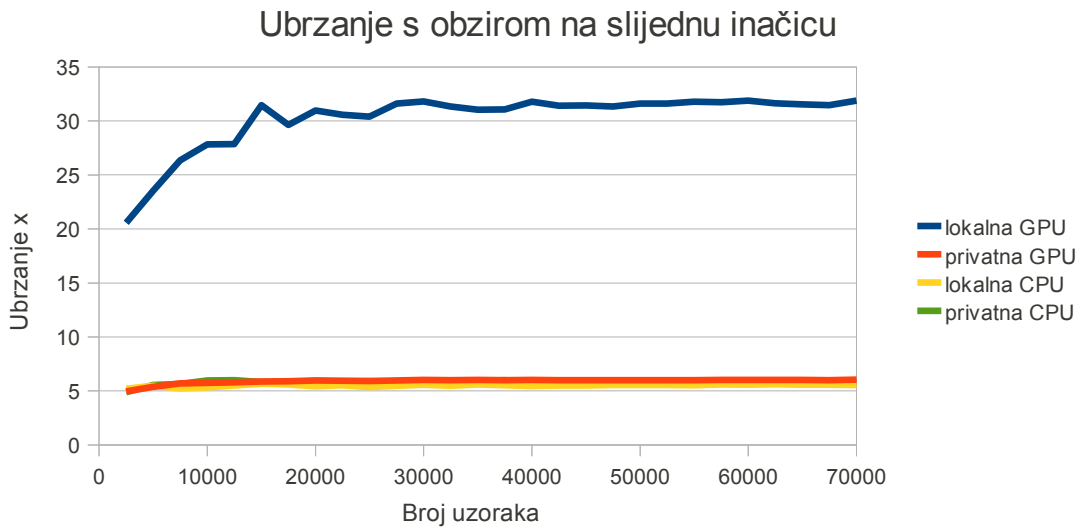
Slika 6.4: Ubrzanje s obzirom na slijednu inačicu

Zanimljivo je primijetiti kod GPU inačice kako se nakon otprilike 50 uzoraka smanji ubrzanje (tj. naglo poveća trajanje iteracije). To je zato što se za svaku česticu stvara nova radna grupa (slika 5.6) koja će računati dodatne uzorke, ali budući da ima malo dodatnih uzoraka većina radnih jedinica u tim grupama neće ništa raditi, ali zato što raspoređivač dretvi na GPU-u tako radi, te dretve će ipak zauzimati jezgre (procesirajuće elemente) dok drugi iz grupe računaju. Na CPU-u se to ne događa zato što se svaka radna grupa nalazi na jednoj jezgri i te radne jedinice koje ništa ne računaju su brzo gotove.



Slika 6.5: Ubrzanje s obzirom na slijednu inačicu

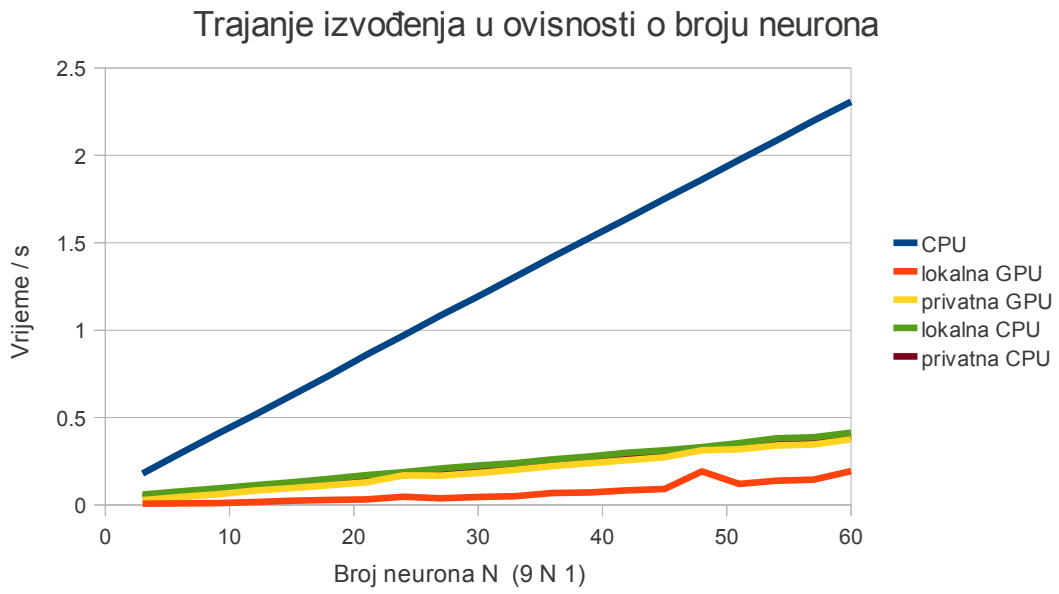
Uz veliki broj uzoraka gdje su ubrzanja konstantna mogu se uspoređivati i GPU privatna inačica i paralelne CPU inačice (slika 6.5, svaka točka je prosjek 5 iteracija). GPU inačica je dalje najbrža, oko 6 puta od slijedne, lokalna CPU oko 5.5 puta, a privatna CPU oko 5.9 puta. Zanimljivo je da je lokalna CPU inačica sporija od privatne, a to je zato što CPU nema dodatnu bržu lokalnu memoriju, nego su lokalna i globalna jednako brze. Tako da radne jedinice u lokalnoj CPU inačici samo gube vrijeme s kopiranjem podataka iz globalne u lokalnu memoriju i zato je sporija. Ubrzanje paralelne CPU inačice blizu je idealnom (CPU ima 6 jezgri).



Slika 6.6: Ubrzanje s obzirom na slijednu inačicu

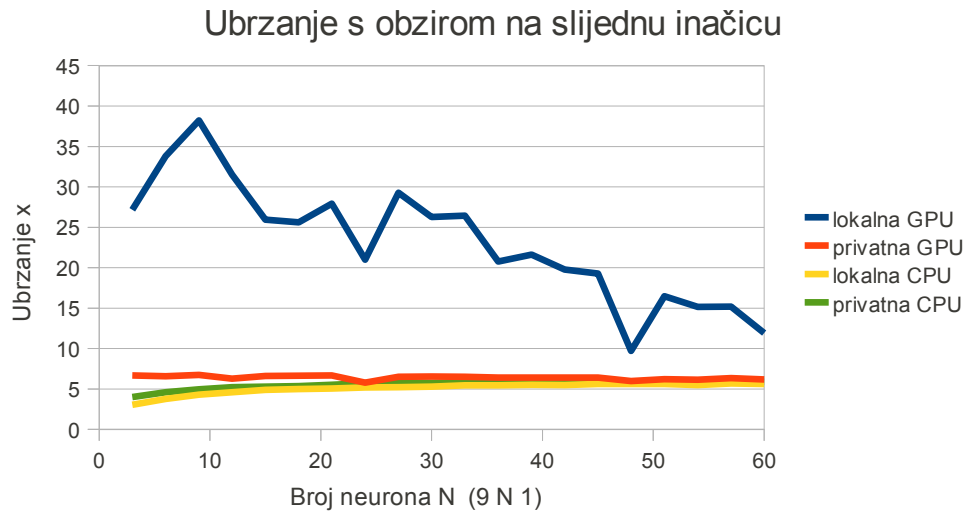
Na slici 6.6 se vidi i lokalna GPU inačica i nakon 30000 uzoraka dostigne ubrzanje od 32 puta i računa oko $5*10^9$ sinapsi po sekundi.

6.1.2 Ispitivanje algoritma u ovisnosti o broju neurona unutar sloja



Slika 6.7: Trajanje izvođenja u ovisnosti o broju neurona

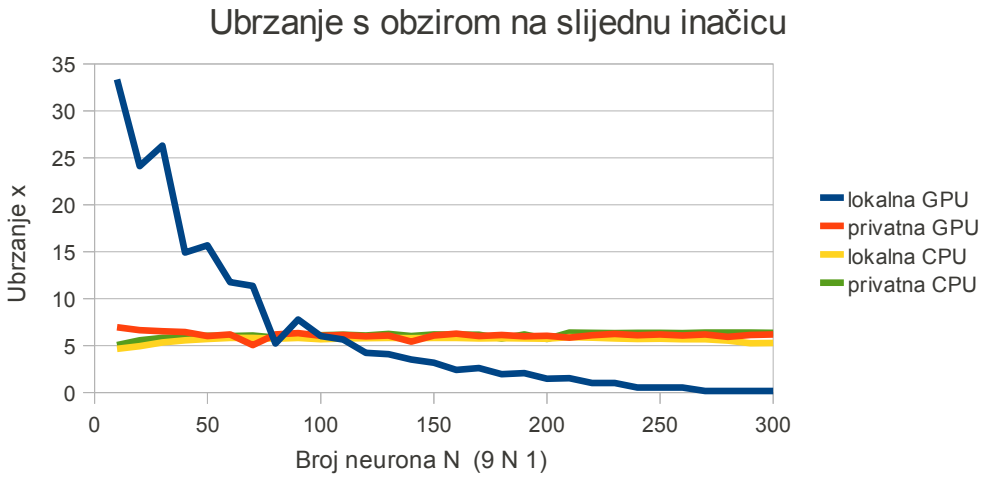
Ispitivanje je izvedeno s 10000 primjera za učenje sa mrežom s jednim skrivenim slojem unutar kojeg se mijenja broj neurona. Na slici 6.7 su trajanja jedne iteracije za različit broj neurona u tom sloju, a na slici 6.8 ubrzanje s obzirom na slijedni algoritam. Svaka točka je prosjek trajanja 20 iteracija.



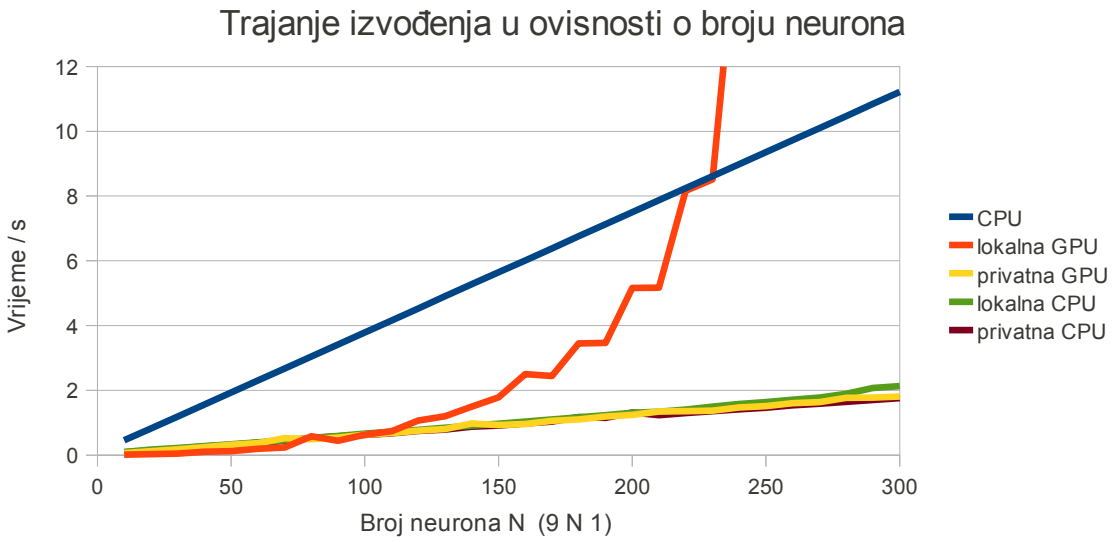
Slika 6.8: Ubrzanje s obzirom na slijednu inačicu

Ubrzanje lokalne GPU inačice na početku raste zato što se udio *overheada* smanjuje u odnosu na ukupno računanje, ali kako raste broj neurona u sloju, ubrzanje počinje padati. Do toga dolazi zbog toga što lokalna memorija potrebna radnoj jedinici ovisi o veličini najvećeg sloja (slika 5.5), i tako što je veći sloj, jedinica zauzima više memorije. Lokalna memorija na računskoj jedinici je ograničena i ako jedna radna jedinica zauzima puno memorije, na jednu računsku jedinicu stane manje radnih jedinica (radne grupe su manje). Kako pada broj radnih jedinica istovremeno aktivnih tako se teže skriva velika latencija prema globalnoj memoriji, a u jednom trenutku može doći i do toga da na računsku jedinicu stane manje radnih jedinica nego što ona ima jezgri i neke jezgre budu neiskorištene.

Nakon 90 neurona lokalna GPU inačica postaje sporija od ostalih paralelnih inačica (slika 6.9) koje imaju stalno ubrzanje za veće brojeve neurona. Lokalna CPU inačica se ne usporava zato što je dovoljno da u grupi bude jedna radna jedinica budući da se ionako cijela grupa izvodi na jednoj jezgri.



Slika 6.9: Ubrzanje s obzirom na slijednu inačicu

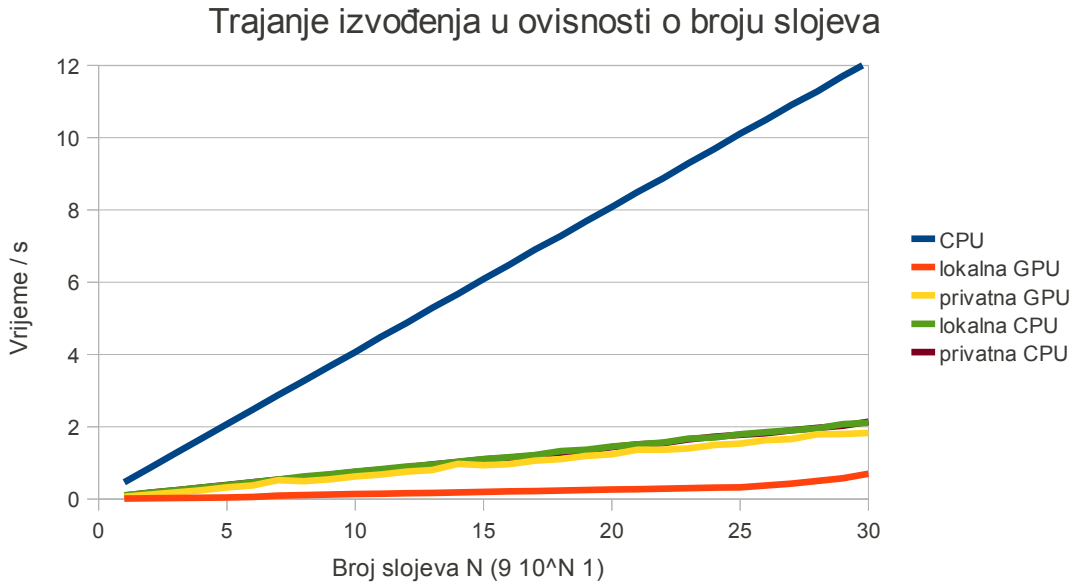


Slika 6.10: Trajanje izvođenja u ovisnosti o broju neurona

Kada broj neurona naraste iznad 240 (slika 6.10) lokalna GPU inačica čak postaje sporija od slijedne i broj jedinki u grupi padne na 2.

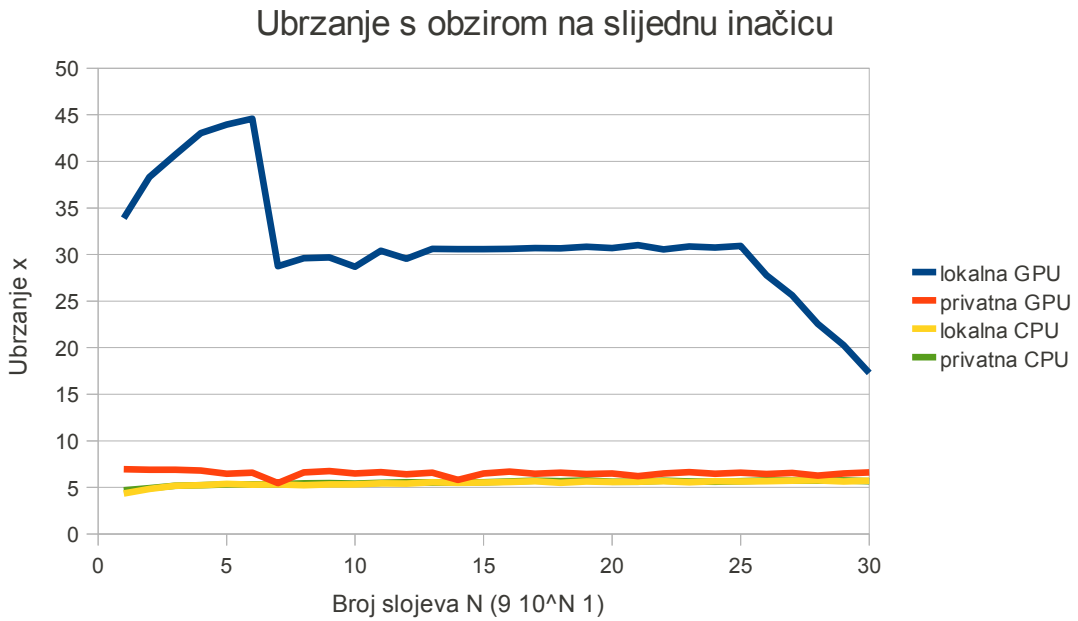
6.1.3 Ispitivanje algoritma u ovisnosti o broju slojeva

Provedeno je ispitivanje s 10000 primjera za učenje i uz mijenjanje broja slojeva (svaki sloj ima 10 neurona i svaka točka je prosjek 10 iteracija).



Slika 6.11: Trajanje izvođenja u ovisnosti o broju slojeva

Na slici 6.11 su rezultati mjerenja, a na slici 6.12 se vidi usporedba sa slijednom inačicom.

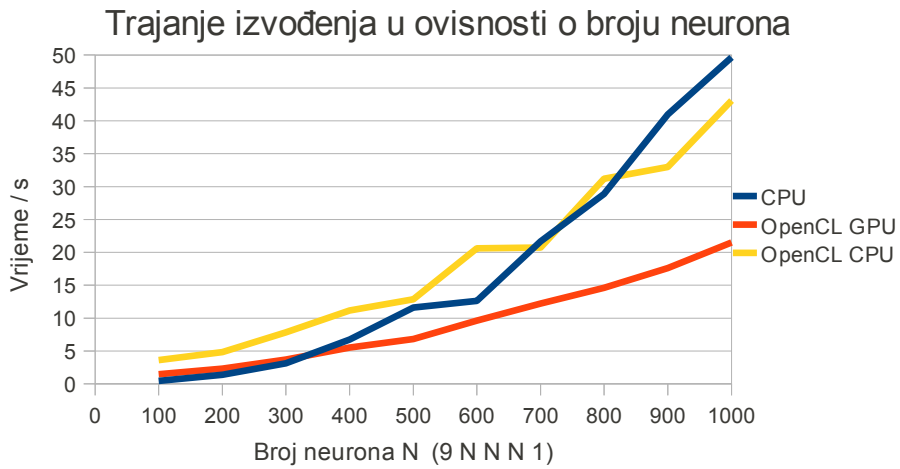


Slika 6.12: Ubrzanje s obzirom na slijednu inačicu

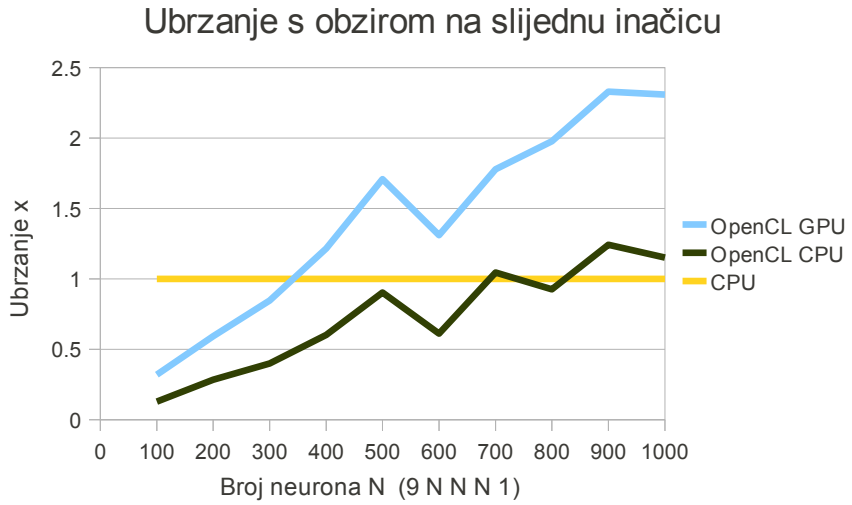
Na početku do 6 skrivenih slojeva ubrzanje raste, nakon čega padne i ostaje stabilno neko vrijeme, a na kraju počinje padati zato što u lokalnoj memoriji svake grupe su učitane sve težine njihove mreže i kako mreža raste, postoji manje prostora za radne jedinice, pa su računske jedinice sve manje popunjene. Sa 6 slojeva brzina mreže je oko $7*10^9$ sinapsi po sekundi.

6.2 Ispitivanje paralelnog algoritma *backpropagation*

Za ispitivanje ovog algoritma su korištene veće mreže i 1000 uzoraka po mjerenju. Mreža ima 9 neurona u ulaznom sloju, 3 skrivena sloja i 1 izlaz. U svakom skrivenom sloju jednak je broj neurona i provedena su mjerenja za različite brojeve neurona u tim slojevima. Na slici 6.13 je rezultat mjerenja jedne iteracije. Za manje mreže paralelne inačice su sporije, ali postaju brže iznad 400 neurona po sloju za GPU inačicu i 800 neurona za CPU inačicu (slika 6.14).

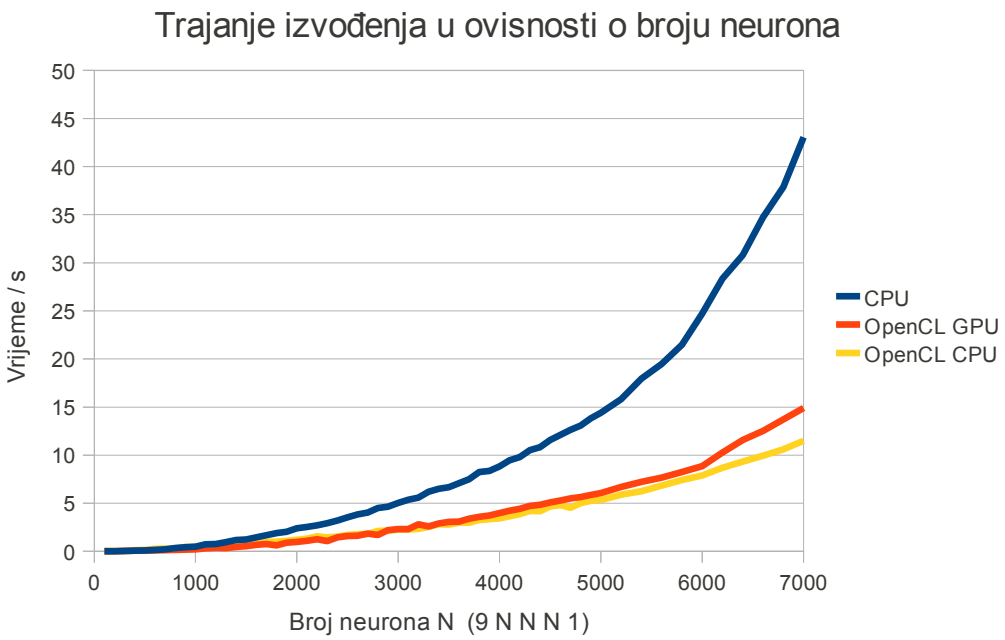


Slika 6.13: Trajanje izvođenja u ovisnosti o broju neurona u slojevima

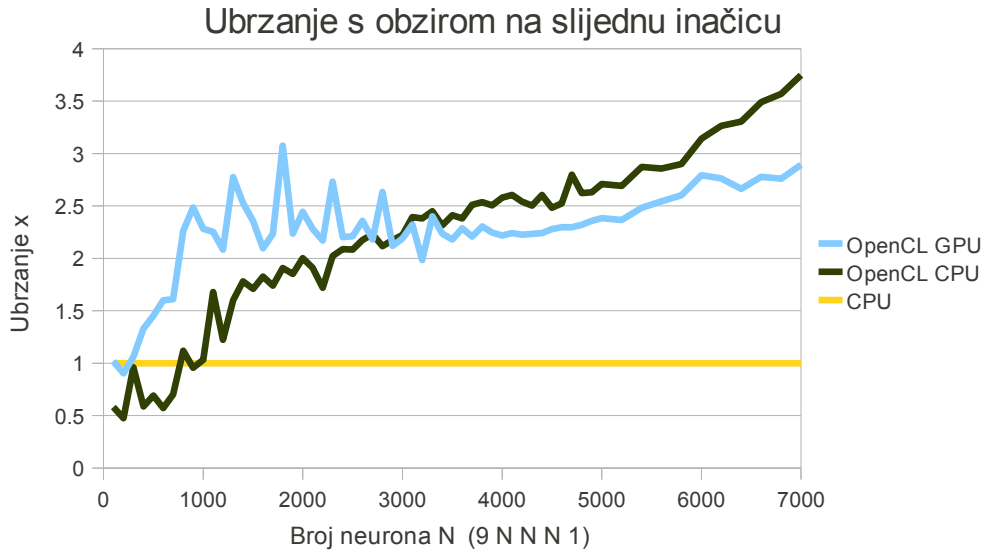


Slika 6.14: Ubrzanje s obzirom na slijednu inačicu

Rezultati za još veće mreže (do 7000 neurona po sloju, 10 primjera za učenje) su na slici 6.15. GPU inačica je oko 2.5 puta brža od slijedne (slika 6.16), a CPU inačici ubrzanje postepeno raste i nakon 3000 neurona postaje brža i od GPU inačice.



Slika 6.15: Trajanje izvođenja u ovisnosti o broju neurona u slojevima



Slika 6.16: Ubrzanje s obzirom na slijednu inačicu

Mreža s 7000 neurona po sloju ima ukupno 100 milijuna sinapsi i jedan prolaz kroz mrežu na svih 6 jezgri centralnog procesora traje 1.1 s, što je oko $9*10^7$ sinapsi po sekundi.

Ubrzanje GPU inačice na ovom algoritmu je lošije u odnosu na PSO zato što izlaz iz jednog neurona ovisi o svim izlazima iz prethodnog sloja i težinama samo od tog neurona, a za ovako velike mreže nije moguće sve učitati u lokalnu memoriju i raditi u njoj. Težine mreže se također ne mogu dijeliti među radnim jedinicama zato što svaka jedinica ima različite težine.

7. Predviđanje statusa poslova na grozdu računala

Arhiva paralelnih opterećenja (engl. *Parallel Workloads Archive*) [14] sadrži datoteke sa zapisima o konačnim statusima poslova sa raznih grozdova računala. Poslovi su mogli završiti uspješno ili sa pogreškom (zbog greške u programu, zbog prekoračenja dodijeljenog vremena ili jer ga je korisnik sam zaustavio). Ako su poznate neke značajke o poslu, može se pokušati predvidjeti hoće li posao završiti s pogreškom i za to se može primijeniti neuronska mreža (poglavlje 4.). Slično ispitivanje je obavljeno u radu [15] gdje se za klasifikaciju koristio Bayesov klasifikator.

Popis korištenih datoteka sa zapisima o poslovima nalazi se u tablici 7.1.

Naziv	Ukupno poslova	Poslovi koji su završili s pogreškom	Udio poslova koji su završili s pogreškom
CTC-SP2-1996-2.1-cln	77222	16669	21.59%
LANL-CM5-1994-3.1-cln	122060	20368	16.69%
LLNL-Atlas-2006-1.1-cln	42725	21546	50.43%
LLNL-Thunder-2007-1.1-cln	121039	16877	13.94%
LPC-EGEE-2004-1.2-cln	234889	27064	11.52%
SDSC-Par-1995-2.1-cln	53970	906	1.68%
SDSC-Par-1996-2.1-cln	32135	814	2.53%

Tablica 7.1: Popis datoteka sa zapisima o poslovima

Datoteke sadrže razne atribute poslova, ali neke atribute nije moguće znati unaprijed, pa ih se ne može koristiti. Atributi korišteni u ovom predviđanju nalaze se u tablici 7.2 [16]. Neke od tih značajki nisu poznate u vrijeme dolaska posla (sive boje u tablici), ali poznate su za prethodne poslove. Svi grozdovi nisu zapisivali sve značajke, pa u svim datotekama nedostaju neke značajke.

Identifikator posla	Vrijeme dolaska posla
Trajanje posla	Identifikator korisnika
Identifikator aplikacije	Zatraženi broj procesora
Procijenjeno trajanje izvođenja	Zatražena količina radne memorije
Status završetka posla	Prosječno vrijeme izvođenja po procesoru

Tablica 7.2: Popis korištenih značajki poslova

7.1 Predviđanje statusa na temelju prošlog sličnog posla

U radu [16] autor je primijetio jednostavno pravilo: ako je prošli slični posao (posao s istim indentifikatorom korisnika i aplikacije) završio s pogreškom, velika je vjerojatnost da će i taj posao završiti s pogreškom.

Provedeno je ispitivanje može li neuronska mreža pronaći to pravilo. Korištena je

mreža učena s PSO na grafičkom procesoru (poglavlje 5.1) s jednim ulazom – status prošlog sličnog posla i jednim izlazom – status tog posla, bez skrivenih slojeva. Ulaz može biti 0 (prethodni posao je uspješno završio) ili 1 (prethodni posao je završio s greškom), a ako je izlaz veći od 0.5 mreža predviđa da će posao završiti s pogreškom, a ako je manje od 0.5 predviđa da će uspješno završiti. Posebna mreža je rađena za svaku datoteku sa zapisima. Primjeri za učenje podijeljeni su na skup za učenje (40%), skup za validaciju (30%) i skup za ispitivanje (30%). Za PSO su korištene 64 čestice i uvjet zaustavljanja je 1000 iteracija, a za ostale parametre su korišteni unaprijed postavljeni (tablica 5.1).

Nakon učenja, mreže su uspjele naučiti to pravilo osim za grozd SDSC-Par (obje datoteke). Ako se pogleda tablica 7.2 vidi se da u tim datotekama postoji sam oko 2% poslova završenih s greškom (dok kod ostalih ima 11% do 50%) i zato pri učenju imaju premali utjecaj. To je rješeno tako da je promijenjen udio poslova završenih s greškom u skupu za učenje i validaciju, pri tome skup za ispitivanje nije mijenjan. Iz skupova za učenje i validaciju izbačeno je toliko uspješnih poslova da udio završenih s greškom bude barem kao željeni udio, ali ako je udio već dovoljno veliki, ništa nije izbačeno (slika 7.1).

```
Promjeni_udio(skup_za_učenje, skup_za_validaciju, željeni_udio)
  spoji skupove za učenje i validaciju
  Dok je udio završenih s greškom manji od željenog_udjela
    odaberi slučajno jedan posao
    ako je odabrani posao završio uspješno
      izbaci ga iz skupa
  kraj

  podjeli skup na skup za učenje i validaciju u istom omjeru
   kao na početku
```

Slika 7.1: Pseudokod funkcije za mijenjanje udjela poslova završenih s greškom

Nakon promjene udjela poslova završenih s greškom u skupovima za učenje i validaciju na 50%, mreže su uspjele naučiti pravilo za sve datoteke. Svi prikazani rezultati odnose se na skup za ispitivanje.

		Stvarno stanje	
		Pozitivan *Positive*	Negativan *Negative*
Predviđeno stanje	Pozitivan *Positive*	Točno pozitivan True Positive = **TP**	Lažno pozitivan False Positive = **FP**
	Negativan *Negative*	Lažno negativan False Negative = **FN**	Točno negativan True Negative = **TN**

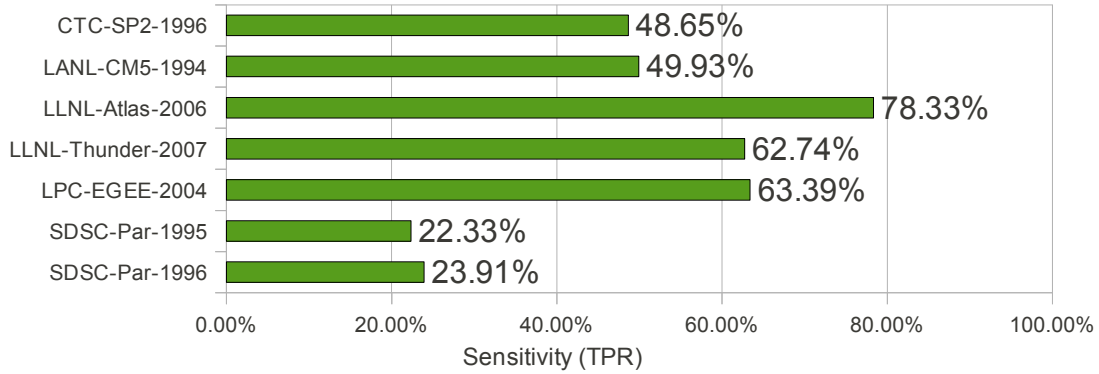
Tablica 7.3: Tablica zabune

U tablicama 7.3 i 7.4 su objašnjene mjere s kojima se prikazuju rezultati u sljedećim grafovima. [17]

Osjetljivost (engl. *Sensitivity* ili *True Positive Rate - TPR*)	$=\frac{TP}{TP+FN}$
Točnost (engl. *accuracy*)	$=\frac{TP+TN}{TP+TN+FP+FN}$
Preciznost (engl. *precision*)	$=\frac{TP}{TP+FP}$

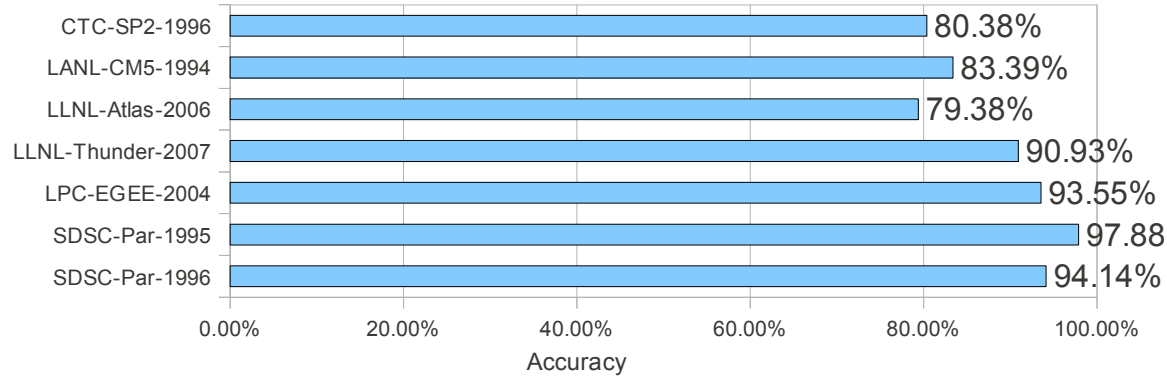
Tablica 7.4: Evaluacijske mjere

Na slici 7.2 prikazana je osjetljivost klasifikacije, tj. koliko poslova završenih s pogreškom je pronađeno.



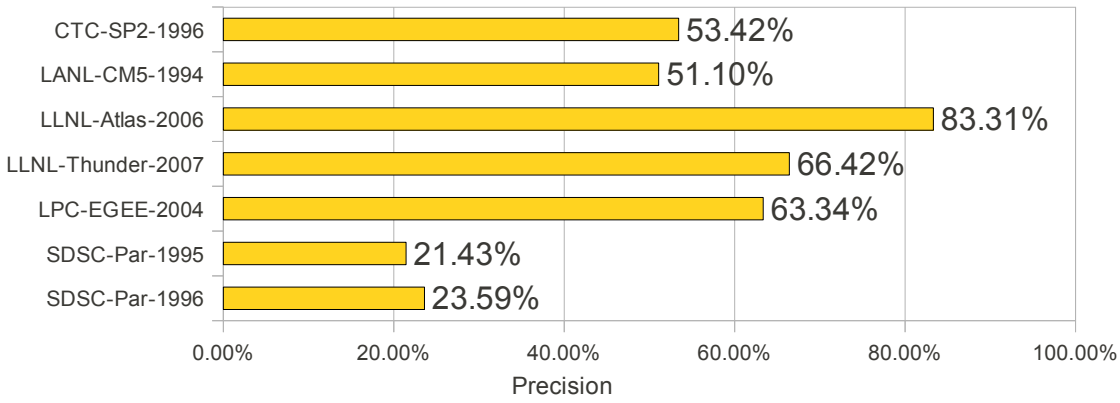
Slika 7.2: Osjetljivost

Na slici 7.3 prikazana je točnost, tj. koliko je poslova točno klasificirano (i završenih s pogreškom i uspješno završenih).



Slika 7.3: Točnost

Na slici 7.4 prikazana je preciznost, tj. od svih poslova koje je predvidio da će završiti s greškom, koliko ih je stvarno tako završilo.



Slika 7.4: Preciznost

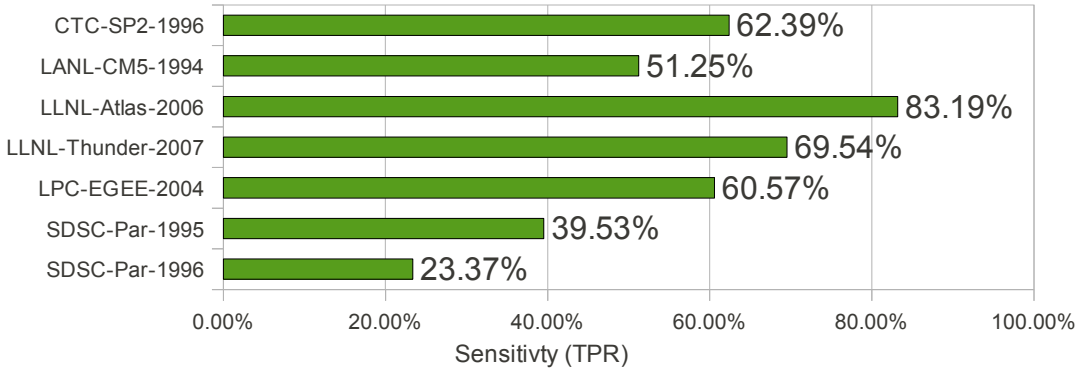
7.2 Predviđanje statusa posla na temelju više značajki

Za ovo ispitivanje korištene su sljedeće značajke:

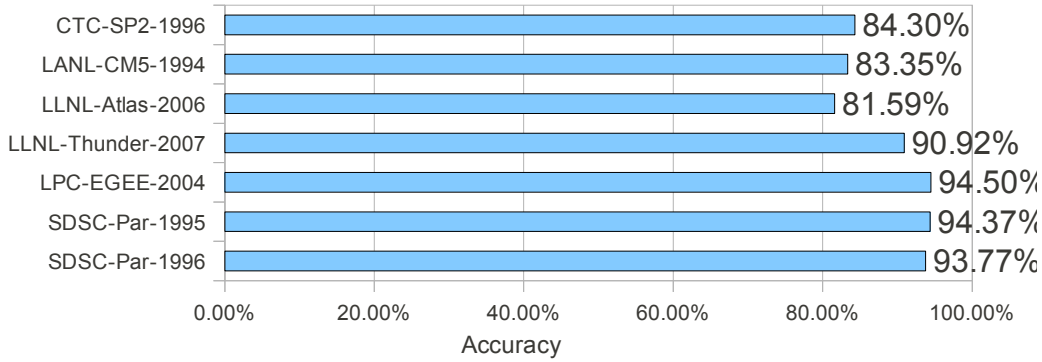
- Zatraženi broj procesora
- Zatražena količina radne memorije
- Procijenjeno trajanje izvođenja
- Status prošlog posla od istog korisnika
- Status prošlog posla od iste aplikacije
- Status prošlog posla od istog korisnika s istom aplikacijom
- Trajanje prošlog posla od istog korisnika s istom aplikacijom
- Prosječno trajanje izvođenja po procesoru prošlog posla od istog korisnika s istom aplikacijom
- Proteklo vrijeme otkad je korisnik zadnji put koristio tu aplikaciju
- Proteklo vrijeme otkad je korisnik zadnji put koristio grozd.

Podjela na skupove je također 40%, 30%, 30%, a broj iteracija je 2000. Udio poslova završenih s pogreškom u skupovima za učenje i validaciju je 50%. Mreža ima 10 ulaza, 2 skrivena sloja, prvi s 10 neurona i drugi s 5. Za svaku datoteku sa zapisima učene su 3 mreže i odabrana ona s najboljim *MSE*-om validacijskog skupa.

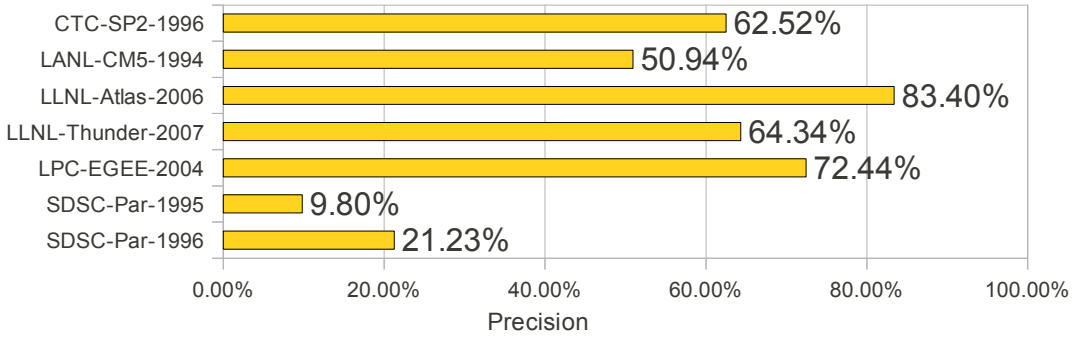
Rezultati se nalaze na slikama 7.5, 7.6 i 7.7.



Slika 7.5: Osjetljivost

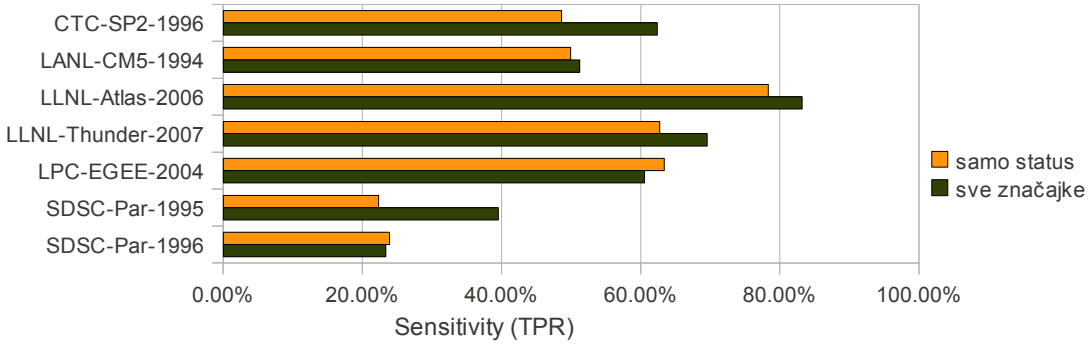


Slika 7.6: Točnost

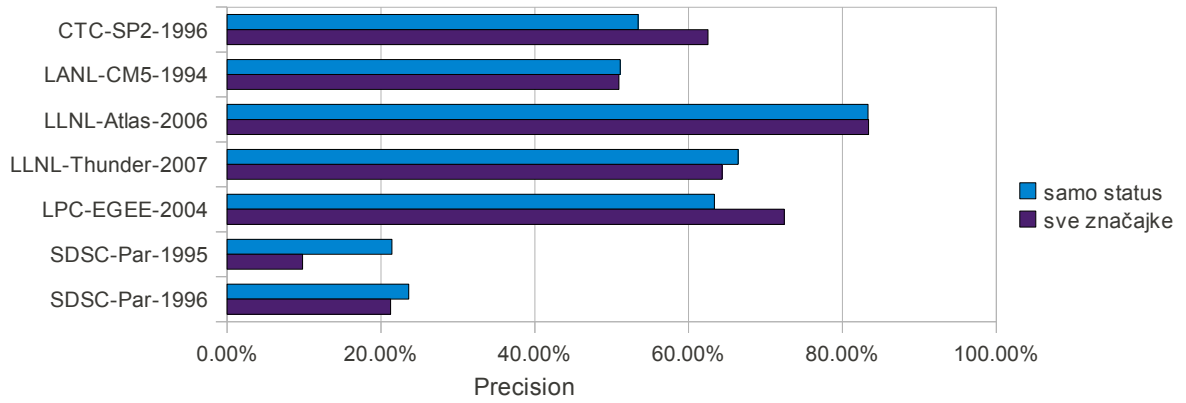


Slika 7.7: Preciznost

Ako se usporede osjetljivosti za mreže koje su koristile samo status prošlog sličnog posla i mreže koje su koristile više značajki (slika 7.8) vidi se da CTC-SP2, LLNL-Atlas i LLNL-Thunder imaju veću osjetljivost s tim novim značajkama (4-14%), a CTC-SP2 čak ima i veću preciznost (slika 7.9). Za LANL-CM5 nema bitne promjene. Kod LPC-EGEE osjetljivost je malo manja, ali preciznost je dosta veća, udio FP u poslovima koji više nisu klasificirani kao poslovi s pogreškom je 83%, što je puno bolje od slučajnog. Za SDSC-Par-1995 ima dosta veću osjetljivost, ali je preciznost puno manja, a kod SDSC-Par-1995 nema promjena. SDSC-Par datoteke imaju puno manju osjetljivost i preciznost od ostalih, a to može biti zbog toga što ima manje primjera za učenje, a i mali udio poslova s pogreškom (udio poslova u skupu za ispitivanje je nepromijenjen).

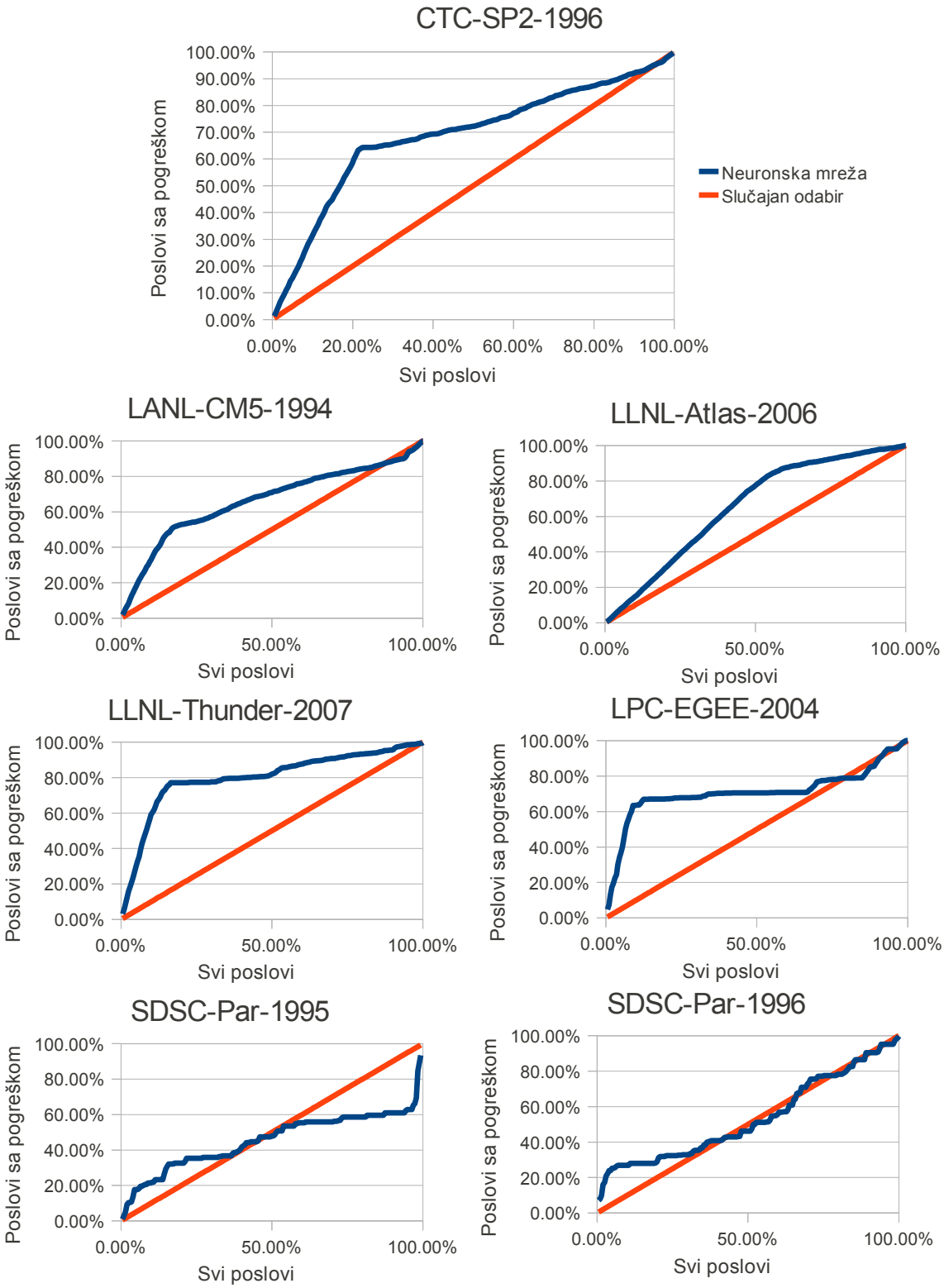


Slika 7.8: Osjetljivost



Slika 7.9: Preciznost

Još jedan način prikaza rezultat je sa *lift* krivuljom (slika 7.10) gdje se može vidjeti koliko je klasifikacije bolja od slučajnog odabira. Na osi x je udio odabranih poslova, a na osi y je udio poslova sa pogreškom u tom odabranom dijelu. Poslovi se biraju tako da se prvi odaberu oni s najvećim izlazom iz mreže. Npr. ako se iz CTC-SP2-1996 odaberu 22% poslova, u njima će biti 64% od svih poslova s pogreškom. Vidi se da mreže rade bolje od slučajnog odabira za sve osim za SDSC-Par grozd gdje je na početku bolje, ali s većim brojem poslova čak postaje i lošije od slučajnog odabira.



Slika 7.10: Lift krivulje

8. Zaključak

U ovom radu implementirane su paralelne metode učenja neuronske mreže na OpenCL-u i brzine izvođenja uspoređene su sa slijednim inačicama. Učenje sa paralelnim algoritmom roja čestica ima ubrzanje, s obzirom na slijednu inačicu, do 32 puta na grafičkom procesoru, dok na centralnom procesoru ima blizu 6 puta (sa 6 jezgri). Za veće mreže koje ne stanu u brzu memoriju grafičkog procesora ubrzanje je oko 6 puta.

Učenje s paralelnim algoritmom *backpropagation* je oko 2.5 puta brže na grafičkom procesoru, dok se ubrzanje na centralnom procesoru povećava s brojem neurona do 4 puta. Takav algoritam ne radi dobro za male mreže zbog velikih troškova komunikacije i sinkronizacije. Ubrzanje algoritma *backpropagation* je lošije nego kod algoritma roja čestica zato što kod računanja pogreške neurona ona ovisi o svim neuronima iz prethodnog sloja i stoga se ne može dobro paralelizirati. Algoritam bi bilo moguće ubrzati tako da se istovremeno odvojeno računa pogreška na različitim podskupovima primjera za učenje i nakon toga se sve promjene težina zajedno zbroje [11]. Drugi način bi bio da mreža ne bude potpuno povezana, nego da su neuroni povezani samo sa dijelom neurona iz prethodnog sloja, tj. lokalno povezana [12]. Sve paralelne implementacije za CPU moguće je ubrzati tako da se u OpenCL-u koriste vektori čime se omogućuje prevođenje koda u SSE (engl. *Streaming SIMD Extensions*) i AVX (engl. *Advanced Vector Extensions*) instrukcije. I za ATI GPU-ove je također dobro koristiti vektore.

Neuronska mreža korištena je za klasifikaciju poslova na grozdovima računala. Tražili su se poslovi koji završavaju s pogreškom i u većini slučajeva mreža je uspješno pronašla od 51% do 83% poslova s pogreškom uz preciznost od 50% do 83%, osim za dvije datoteke sa zapisima koje imaju malo uzoraka s pogreškom. Kvalitetu mreže bi možda bilo moguće poboljšati ponavljanjem učenja svakih nekoliko tjedana i dodavanjem indentifikatora korisnika na ulaz u mrežu čime bi se moglo uzeti u obzir ponašanje pojedinih korisnika [16].

9. Literatura

1. Khronos OpenCL Working Group, "The OpenCL Specification Version: 1.1", Khronos Group, 2010.
2. "OpenCL and the AMD APP", http://developer.amd.com/documentation/articles/pages/OpenCL-and-the-AMD-APP-SDK.aspx, 06.04.2011.
3. A. Trbojević, "Izvedba algoritama računalnog vida na grafičkim procesorima", završni rad, FER, 2010.
4. NVIDIA, "OpenCL Programming Guide for the CUDA Architecture", 2010.
5. J. Kennedy, R. Eberhart, "Particle Swarm Optimization", Proc. IEEE Int'l. Conf. on Neural Networks, 1995.
6. Marko Čupić, "Prirodom inspirirani optimizacijski algoritmi", Zagreb, FER, 2009.-2010.
7. J. Fulcher, L.C. Jain, "Computational Intelligence: A Compendium", Berlin, Springer, 2008.
8. Ioan Cristian Trelea, "The particle swarm optimization algorithm: convergence analysis and parameter selection", Information Processing Letters, 2003.
9. B. Dalbelo Bašić, M. Čupić, J. Šnajder, "Umjetne neuronske mreže", Zagreb, FER, 2008.
10. S. Lončarić, "Neuronske Mreže - predavanja", Zagreb, FER, 2010.
11. A. Pétrowski, G. Dreyfus, C. Girault, "Performance Analysis of a Pipelined Backpropagation Parallel Algorithm", IEEE Transactions on Neural Networks, 1993.
12. R. Uetz, S. Behnke, "Large-scale Object Recognition with CUDA-accelerated Hierarchical Neural Networks", Proceedings of the 1st IEEE International Conference on Intelligent Computing and Intelligent Systems, 2009.
13. X. Sierra-Canto, F. Madera-Ramírez, V. Uc-Cetina, "Parallel Training of a Back-Propagation Neural Network using CUDA", Ninth International Conference on Machine Learning and Applications, 2010.
14. Parallel Workloads Archive, http://www.cs.huji.ac.il/labs/parallel/workload/
15. I. Grudenić, N. Bogunović, "Job Status Prediction – Catch Them Before They Fail", Lecture Notes in Computer Science, 2010.
16. I. Grudenić, "Prilagodljivo dinamičko raspoređivanje skupnih poslova na grozdu računala", doktorska disertacija, FER, 2010.
17. B. Dalbelo Bašić, "Strojno učenje - predavanja", Zagreb, FER, 2009.

Klasifikacija podataka uporabom paralelnih neuronskih mreža

Sažetak:

Učenje umjetnih neuronskih mreža može trajati dugo, zato su implementirane paralelne inačice za učenje. Implementiran je paralelan algoritam roja čestica (engl. *particle swarm optimization*) za učenje mreže i paralelan algoritam *backpropagation*. Ispitano je ubrzanje s obzirom na slijedne inačice. Neuronska mreža je primijenjena na problem predviđanja statusa poslova na računalnim grozdovima i ispitane su mogućnosti mreže da pronađe poslove koji će završiti s greškom.

Ključne riječi: paralelna umjetna neuronska mreža, OpenCL, algoritam roja čestica, backpropagation, predviđanje statusa poslova

Classification of data using parallel neural networks

Abstract:

Training neural networks can take a long time, so parallel versions of training algorithms were implemented. Parallel particle swarm optimization and parallel backpropagation were implemented to train the network. Speed-up in regards to a sequential version was measured. The neural network was trained to predict job statuses on computer clusters and network's ability to predict a job's failure was tested.

Keywords: parallel neural network, OpenCL, particle swarm optimization, backpopagation, job status prediction