SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1665

OPTIMIZACIJA PRIORITETNOG RASPOREĐIVANJA METODAMA STROJNOG UČENJA OSTVARENA NA GRAFIČKOM PROCESORU

Željko Findak

Zagreb, lipanj 2018.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 15. ožujka 2018.

DIPLOMSKI ZADATAK br. 1665

Pristupnik: **Željko Findak (0036471708)**
Studij: Računarstvo
Profil: Računarska znanost

Zadatak: **Optimizacija prioritetnog raspoređivanja metodama strojnog učenja ostvarena na grafičkom procesoru**

Opis zadatka:

Opisati problem raspoređivanja uporabom heurističkih postupaka, s posebnim naglaskom na raspoređivanje uz pomoć prioritetnih pravila. Istražiti postojeće načine stvaranja prioritetnih pravila uz pomoć postupaka strojnog učenja, kao što su neuronske mreže i genetsko programiranje. Ostvariti programsku podršku za paralelizaciju simulacije raspoređivanja te korištenih metoda strojnog učenja prilagođenu za izvođenje na grafičkim procesorima. Ispitati učinkovitost ostvarene paralelizacije u ovisnosti o korištenim algoritmima, strukturama podataka te svojstvima problema raspoređivanja. Radu priložiti izvorne tekstove programa, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Zadatak uručen pristupniku: 16. ožujka 2018.
Rok za predaju rada: 29. lipnja 2018.

Mentor:

Prof. dr. sc. Domagoj Jakobović

Predsjednik odbora za diplomski rad profila:

Prof. dr. sc. Siniša Srbljić

Djelovođa:

Doc. dr. sc. Tomislav Hrkać

Zahvaljujem roditeljima za prilike koje su mi omogućili.

Posebno hvala mentoru prof. dr. sc. Domagoju Jakoboviću na pomoći i savjetima tijekom izrade ovog rada, kao i tijekom studija.

SADRŽAJ

1. Uvod

Tema ovog diplomskog rada je prioritetno raspoređivanje poslova na strojeve s ograničenim resursima. Ciljevi su:

- istražiti postojeće načine stvaranja prioritetnih pravila metodama strojnog učenja,
- proučiti postojeću programsku podršku koja stvara ta pravila,
- istražiti mogućnosti paralelizacije navedenih metoda u svrhu ubrzanja izvođenja,
- istražiti algoritme i strukture podataka koji su prikladni za masivno paralelne arhitekture i programske okvire dostupne na GPU,
- implementirati podršku za izvođenje navedenih metoda na GPU,
- ispitati učinkovitost i isplativost ovakve implementacije.

Ovaj rad podijeljen je na sedam poglavlja. U drugom poglavlju opisuju se koncepti paralelnog programiranja. Zatim se ukratko opisuje arhitektura grafičkog procesora i trenutno dostupne tehnologije, potom specifičnosti programskog okvira CUDA. Na kraju je dan osnovni primjer CUDA programa. U trećem poglavlju opisuje se unaprijedna neuronska mreža. Prikazan je primjer propagacije signala i kako efikasno izračunati izlaz. Opisana je metoda paralelizacije i ostvarena implementacija na GPU korištenjem okvira CUDA. U četvrtom poglavlju opisano je genetsko programiranje. Metoda je korištena za rješavanje problema simboličke regresije funkcija realnih i logičkih varijabli. Opisan je efikasniji algoritam evaluacije rješenja zasnovan na postfiksnoj notaciji. Opisana je metoda paralelizacije i implementacijski detalji važni za efikasno ostvarenje algoritma na GPU. U petom poglavlju opisan je problem prioritetnog raspoređivanja, dane su metrike i opisani kriteriji za ocjenjivanje rasporeda. Opisana je primjena navedenih metoda u problemu raspoređivanja. U šestom poglavlju navedeni su rezultati i komentari. U sedmom poglavlju dan je zaključak.

2. Paralelno programiranje i CUDA

Sposobnosti današnjih računala često su poboljšane dodatkom grafičkog procesora, odnosno popularno nazvano grafičke kartice. To je uređaj koji se koristi za ubrzanje obrade i iscrtavanja grafičkih elemenata kao što su točke, teksture i animirani objekti.

2.1. Grafički procesor

Grafički procesor sastoji se od velikog broja mikroprocesora koji podržavaju mali skup instrukcija. Prednost ovakve računalne arhitekture dolazi do izražaja u programima koji imaju veliki potencijal za paralelizaciju. Na primjer, kod iscrtavanja nekog objekta (engl. *rendering*) postoji veliki broj točaka koje treba obraditi. Obrada pojedine točke često je neovisna o obradi drugih, pa je njihova obrada brža ako se izvodi paralelno.

Suvremeni grafički procesor sastoji se od nekoliko tisuća mikroprocesora. Zbog velike sposobnosti paralelizacije postao je zanimljiv i u brojnim primjenama izvan domene računalne grafike te se koristi u simulacijama gdje je za uspjeh presudna velika računalna moć (engl. *high performance computing*).

Pojavom sve snažnijih uređaja počele su se razvijati brojne programske biblioteke za primjenu u općenitim izračunima (engl. *General Purpose Graphics Processing Unit*). Najpoznatije takve biblioteke su OpenCL, CUDA, OpenACC.

OpenCL (engl. *Open Computing Language*) nastao je kao unificirani standard za GPGPU, podržan od Apple-a i kasnije AMD-a. Njegova prednost je biblioteka koja se izvodi na raznim platformama (RISC, GPU, CPU). Unatoč relativno strmoj krivulji učenja postao je poznati standard i koristi se u brojnim programima, često na jeftinijem potrošačkom sklopovlju.

CUDA (engl. *Compute Unified Device Architecture*) je programski okvir ostvaren kao proširenje C/C++ jezika. Obilježje je pristupačan API, kvalitetna dokumentacija, zajednica korisnika koja razvija biblioteke za razne primjene (cuBLAS, cuFFT, CULA). Podržan je na grafičkim procesorima proizvođača Nvidia.

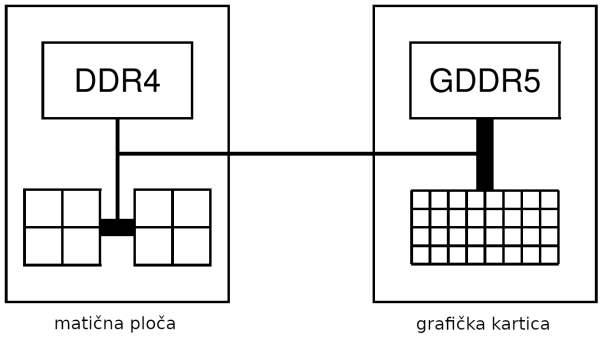
OpenACC je relativno novi standard čiji je cilj dostupnost na raznim platformama. Postoje implementacije prevoditelja (npr. Nvidia PGI) koje omogućuju da uz pomoć pretpro-

cesorskih direktiva ostvarimo veliko ubrzanje. Obilježje je *high-level* API koji omogućuje brz razvoj, ali i kombiniranje s nižim apstrakcijama kao što su CUDA i OpenCL. Zbog pristupačne krivulje učenja i dostupnosti često ima primjenu u znanstvenim proračunima (engl. *scientific computation*).

Za implementaciju ovog rada odabran je okvir CUDA zbog dostupnog sklopovlja, kvalitetne i široko dostupne dokumentacije i velike mogućnosti paralelizacije koju nudi svojom izražajnošću.

2.2. Sklopovlje

Tipično računalo na kojem se razvija CUDA program sastoji se od suvremenog višejezgrenog procesora arhitekture x64 i grafičkog procesora s velikim brojem *shadera* (grafičkih mikroprocesora). Oni su povezani PCIe sabirnicom i kroz nju međusobno komuniciraju.



Slika 2.1: Veza između računala i GPU

Sposobnost CUDA grafičkog procesora određuje se značajkom „*compute capability*". Ona govori kojoj verziji specifikacije uređaj pripada i koje mogućnosti mora podržavati. Trenutno se koriste tri izdanja grafičkih procesora:

- Kepler (*compute capability* 3.0):
  - objavljeno 2012. godine
  - donosi bolju podršku za *double* aritmetičku preciznost
- Maxwell (*compute capability* 5.0)
  - objavljeno 2014. godine
  - kartice predviđene za industriju računalnih igara, lošija *double* preciznost
- Pascal (*compute capability* 6.0 i 6.1)
  - objavljeno 2016. godine
  - značajno ojačane potrošačke GeForce kartice i profesionalne Tesla kartice

Neke specifikacije Pascal arhitekture:

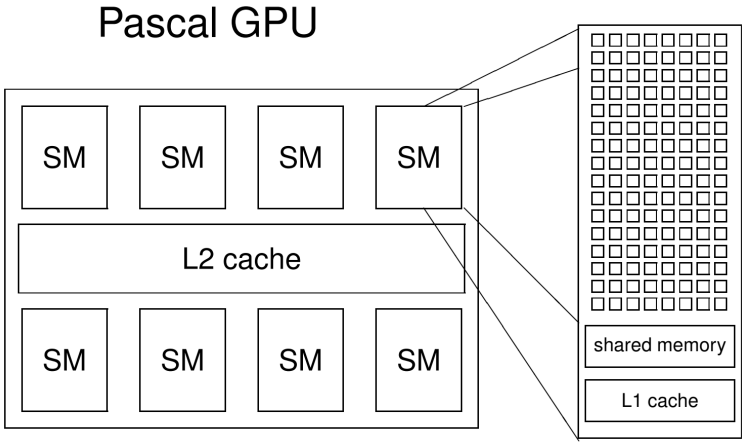
- Potrošačke kartice (GeForce):
  - GTX 1060: 1280 *shadera*, 6 GB videomemorije
  - GTX 1070: 1920 *shadera*, 8 GB videomemorije
  - GTX 1080 Ti: 3584 *shadera*, 11 GB videomemorije
  - Titan XP: 3840 *shadera*, 12 GB videomemorije
- HPC (Tesla):
  - P100 (PCIe): 3584 *shadera*, 12 GB videomemorije
  - P100 (PCIe): 3584 *shadera*, 16 GB videomemorije
  - P100 (Nvlink): 3584 *shadera*, 16 GB videomemorije

Kako su za potrebe pisanja ovog rada bile dostupne kartice Titan XP i GTX 1060, u daljnjem tekstu opisane su neke specifikacije Pascal arhitekture.

Osnovni element u arhitekturi je „*streaming multiprocessor*" (u daljnjem tekstu SM). U Pascal arhitekturi on se sastoji od:

- 128 grupiranih *shadera* i 64 tisuće registara
- 96 KB dijeljene memorije
- 48 KB L1 *cache* memorije
- 8 KB memorije za konstante

Jedan SM može držati do 2000 dretvi aktivno ili pauzirano.



Slika 2.2: Dijagram arhitekture SM-a, preuzeto iz (Giles, 2017)

Programski model ovih dretvi je SIMT (*Single Instruction Multiple Threads*). To znači da sve dretve u grupi istovremeno izvode istu instrukciju na različitim podatcima. Dretve se grupiraju u „val" (engl. *warp*) – skup od 32 dretve koje se izvode na 32 mikroprocesora u istom SM-u. Dretve koje se nalaze na istom SM-u imaju zasebne registre. Zbog velikog

broja dostupnih registara, moguće je istovremeno držati veliki broj dretvi na SM-u, a sve se izvršava vrlo brzo jer ne postoji zamjena konteksta. Dok jedne dretve čekaju podatke, druge dretve se izvode, a dretve koje čekaju postaju neaktivne.

2.3. Programska podrška

CUDA je programski okvir za GPGPU ostvaren kao proširenje programskih jezika C i C++. Nativno podržava i programski jezik FORTRAN kroz posebne prevoditelje. Prednost ovog okvira je velika baza primjera programskih kodova i kvalitetna dokumentacija. Također, okvir je dizajniran prema poznatim načelima paralelnog programiranja pa je olakšan prijelaz programerima s iskustvom u MPI-u i paralelnim arhitekturama. Osim toga, postoji aktivna zajednica korisnika koja pruža podršku na forumima proizvođača.

Tijek izvođenja CUDA programa je sljedeći:

1. inicijalizacija GPU-a,
2. zauzimanje memorije na GPU,
3. kopiranje podataka iz RAM memorije u globalnu memoriju GPU-a,
4. pokretanje skupine dretvi grupiranih u rešetku (engl. *grid*),
5. kopiranje rezultata iz globalne memorije GPU-a u RAM memoriju računala,
6. ponavljanje koraka 3. – 5. koliko je potrebno,
7. oslobađanje GPU memorije i završetak programa.

Bitno je napomenuti da standardom nije definirano kojim redoslijedom će se dretve izvoditi, pa je potrebno pridržavati se toga tijekom dizajna programa.

Kako vidimo, programski kod može se podijeliti na onaj koji se izvodi na računalu (u dokumentaciji naziv „*host*") i onaj koji se izvodi na GPU (u dokumentaciji naziv „*device*"). Kod za GPU organizira se u kernel metode.

2.3.1. Poziv kernel metode

Osnovna kernel metoda poziva se ovako:

```
1 kernel_method<<<gridDim, blockDim>>>(arg1, arg2, ...);
```

Kernel metode grupiraju se u rešetku (u dokumentaciji naziv „*grid*"). Dimenzije rešetke određene su strukturama `gridDim` i `blockDim` tipa `dim3`. To je struktura koja ima tri dimenzije koje su inicijalno postavljene na 1. Na primjer, sljedeći kod postavlja rešetku kao matricu dimenzija 2x3 koja sadrži blokove dimenzija 16x16.

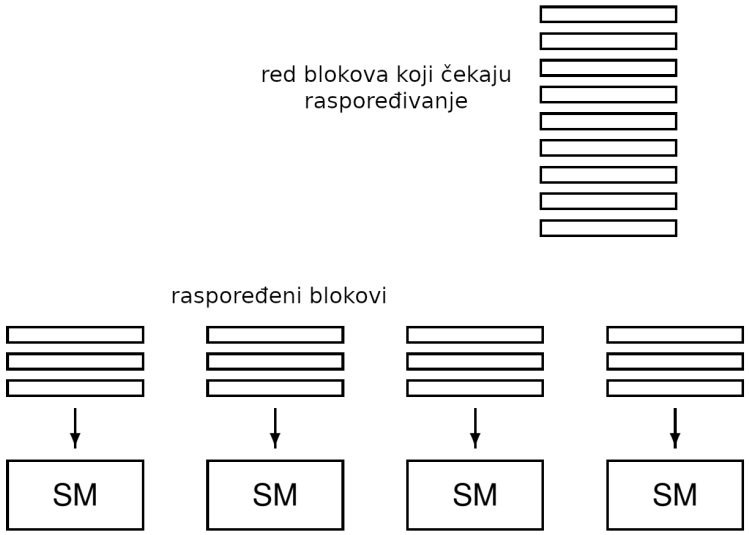
```
dim3 gridDim; gridDim.x = 2; gridDim.y = 3;
dim3 blockDim; blockDim.x = 16; blockDim.y = 16;
```

Ovakva konfiguracija pozvat će $(2*3*1)*(16*16*1) = 1536$ dretvi. U nastavku je primjer jednostavnog programa koji zbraja dva polja.

```
__global__ void sumArraysKernel(float* arr1, float* arr2,
                                float* arr3, int N) {
    int tIdx =  threadIdx.x +
                threadIdx.y * blockDim.x +
                threadIdx.z * blockDim.y * blockDim.x;
    if (tIdx >= N) return;
    arr3[tIdx] = arr1[tIdx] + arr2[tIdx];
}

int main() {
    const int N = 1200;

    float h_arr1[N], h_arr2[N], h_arr3[N];
    for (int i = 0; i < N; i++) {
        h_arr1[i] = rand() / RAND_MAX;
        h_arr2[i] = rand() / RAND_MAX;
    }

    float* d_arr1, d_arr2, d_arr3;
    sizet_t size = N * sizeof(float));
    cudaMalloc((void**) d_arr1, size);
    cudaMalloc((void**) d_arr2, size);
    cudaMalloc((void**) d_arr3, size);
    cudaMemcpy(d_arr1, h_arr1, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_arr2, h_arr2, size, cudaMemcpyHostToDevice);

    dim3 gridDim; gridDim.x = 2; gridDim.y = 3;
    dim3 blockDim; blockDim.x = 16, blockDim.y=16;
    sumArraysKernel<<<gridDim, blockDim>>>(
        d_arr1, d_arr2, d_arr3, N
    );
    cudaMemcpy(h_arr3, d_arr3, size, cudaMemcpyDeviceToHost);

    return 0;
}
```

Ovdje je važno primijetiti nekoliko koncepata. Prvo, svaka dretva ima svoj redni broj prema kojem se određuje koji posao mora obaviti. Drugo, kod se dizajnira iz perspektive pojedine dretve, kao u MPI okviru. Treće, iako postoje nastojanja da se ujedini adresiranje memorije između računala i GPU-a (CUDA Unified Memory, moguće u najnovijim izdanjima CUDA Toolkit-a), za većinu primjena potrebno je slijediti spomenuti obrazac: podatke eksplicitno kopirati u GPU memoriju, obraditi ih, rezultat eksplicitno kopirati u RAM memoriju računala.

Sljedeća slika prikazuje kako se blokovi raspoređuju za izvođenje.



Slika 2.3: Raspoređivanje blokova

Nakon što se pokrene izvođenje kernel metode, sustav odredi blokove dretvi i stavi ih u red čekanja. Nakon što se na pojedinom SM-u oslobodi mjesto (broj mjesta ograničen je slobodnim registrima i brojem *shadera*), blok se iz reda dodjeljuje na SM. Svaki SM ima mehanizam koji se zove „*warp scheduler*". Blokovi se dijele u valove, na svaki ciklus *warp scheduler* određuje koji val će se izvesti sljedeći, zapisuje instrukciju u registar svakog od 32 *shadera* i ta se instrukcija na njima izvršava istovremeno. Programer ne može utjecati na grupiranje u valove, već mora paziti da broj dretvi u bloku bude višekratnik broja 32 kako bi se izbjegla degradacija performansi.

2.3.2. Vrste memorije

U standardnom programiranju memorija se dijeli na:

- **programabilnu** – programer eksplicitno definira koji podatci se spremaju u nju;
- **neprogramabilnu** – programer nema kontrolu nad spremanjem u memoriju i oslanja se na sustav, npr. L1 i L2 *cache* u CPU memorijskoj hijerarhiji.

CUDA nudi nekoliko tipova memorije pa postoji više prostora za optimiranje, ali je zato hijerarhija složenija.

Kod ovakvih sustava do izražaja dolazi potreba za lokalnošću podataka. Postoji

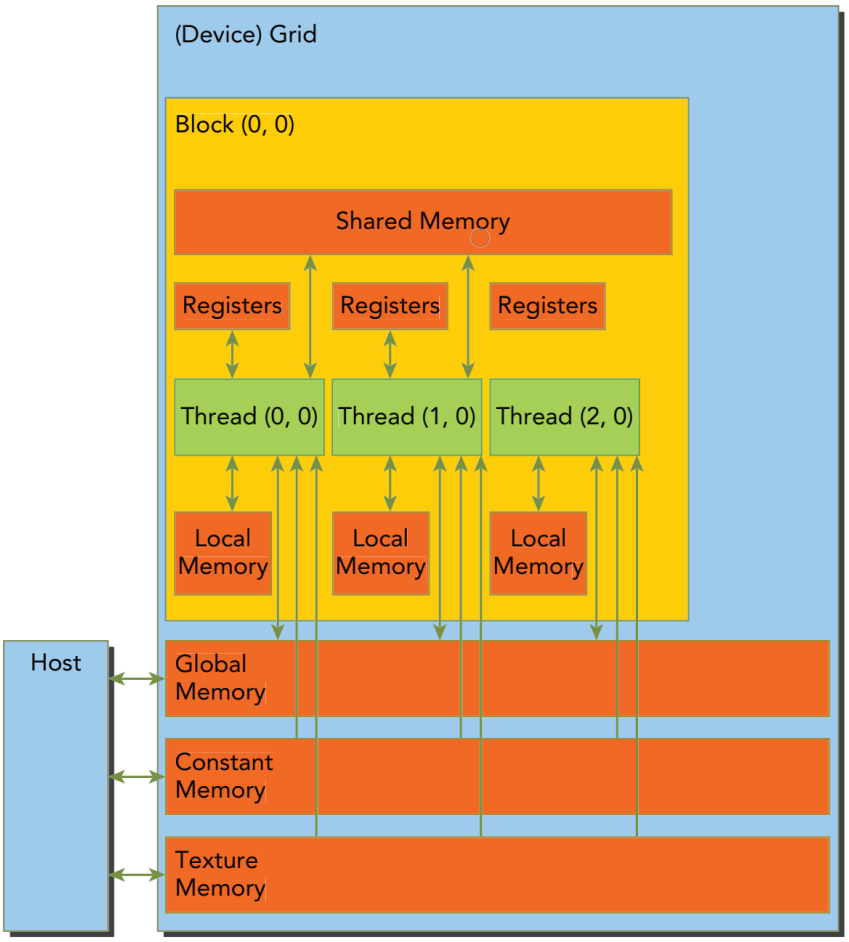
- **vremenska lokalnost** – podatak koji je korišten nedavno vjerojatno će se opet koristiti u bližoj budućnosti;
- **prostorna lokalnost** – podatci koji se u memoriji nalaze blizu nedavno korištenog vjerojatno će se opet koristiti u bližoj budućnosti.

Zbog toga je bitno da *cache* sustav dobro radi svoj posao kako bi se program izvodio efikasno.

Osim toga, kod dizajna kernel metoda programer treba voditi računa jesu li performanse ograničene količinom računanja ili obujmom podataka. U slučaju velike količine računanja više se isplati optimirati algoritam jer kopiranje memorije traje znatno kraće. U slučaju velikog obujma podataka sabirnica postaje usko grlo pa je potrebno istražiti kako najbolje organizirati program da sustav dobro iskoristi *cache* mehanizme, koje vrste memorije su prikladne za koje strukture podataka, istražiti isplati li se podijeliti posao u više tokova (CUDA streams i events), itd.

Vrste CUDA programabilne memorije su:

- **Globalna memorija**
  - najveća, ima najdulje vrijeme odziva (latenciju), najviše se koristi,
  - dostupna je tijekom cijelog izvođenja programa i svim elementima programa,
  - varijable se mogu deklarirati statički i dinamički, iz *host* i *device* funkcije,
  - dostupna je svim dretvama svih kernel metoda, zbog čega istovremeni pristup treba sinkronizirati ogradama (engl. *barrier*) ili atomičkim instrukcijama (CAS),
  - pristupa se u 32-, 64- ili 128-bajtnim transakcijama, pa zato pristup mora biti poravnat, što znači da prva adresa mora biti višekratnik brojeva 4, 8 ili 16,
- **Registri**
  - najbrža memorija na GPU,
  - tu se spremaju varijable deklarirane u kernel metodama,
  - spremaju se i statički alocirana polja, ako je veličina poznata prilikom prevođenja, inače se „prelijevaju" u L1 *cache* GPU-a, a potom u lokalnu memoriju,
  - vrlo ograničen resurs, dijeli se između aktivnih dretvi na SM-u,
- **Dijeljena memorija** (engl. *shared memory*)
  - nalazi se na svakom SM-u, ima veliku propusnost (engl. *bandwith*) i značajno manje vrijeme odziva u odnosu na lokalnu i globalnu memoriju,



Slika 2.4: Dijagram vrsta memorije, preuzeto iz (Cheng et al., 2014)

- može se koristiti slično kao L1 *cache* kod CPU-a, s razlikom da je programabilna,
- svaki SM ima ograničenu količinu dijeljene memorije koja se dijeli između blokova, pa previše zauzete dijeljene memorije ograničava broj blokova raspoređenih na SM,
- najčešće se koristi za komunikaciju između dretvi unutar istog bloka, one mogu dijeliti podatke tako da pišu i čitaju iz dijeljene memorije, zbog čega pristup ovoj memoriji mora biti sinkroniziran ogradom,

– **Lokalna memorija** (engl. *local memory*)

- sadrži varijable koje su deklarirane u kernel metodi, ali ne stanu u registre, npr. dinamički alocirana polja, velika polja i strukture,
- veliko vrijeme odziva jer je sklopovski izvedena isto kao globalna memorija,

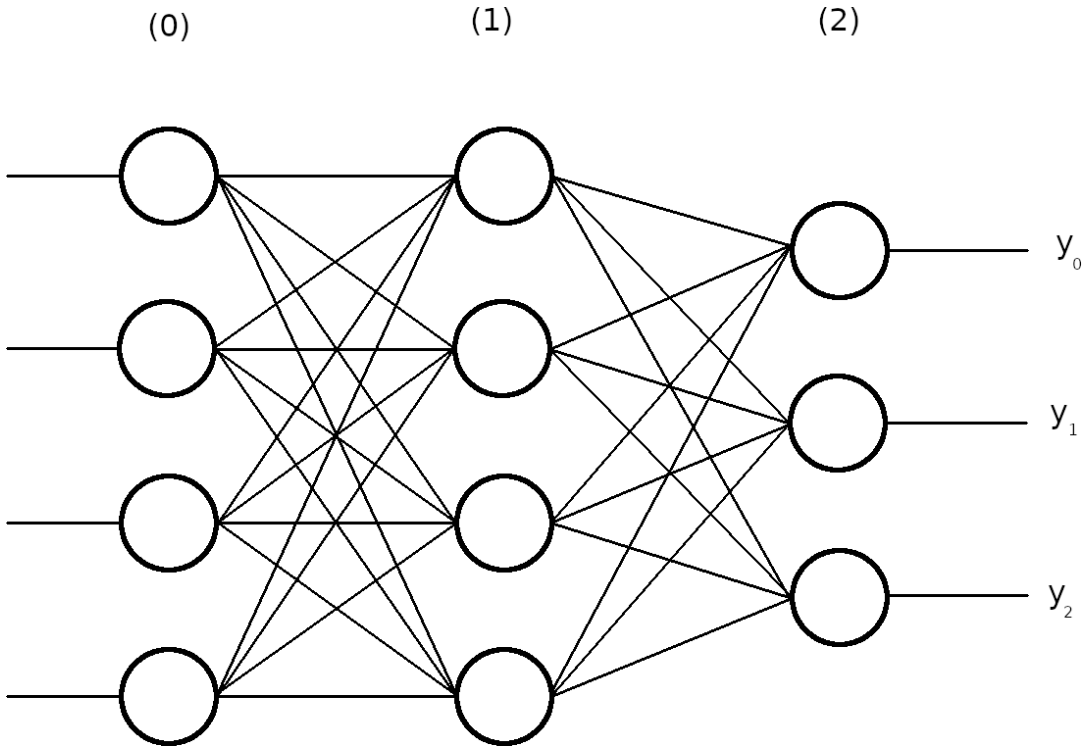
– **Konstantna memorija** (engl. *constant memory*)
    - sklopovski se nalazi isto gdje i globalna memorija, automatski se kopira u zasebni *cache* za svaki SM,
    - deklarira se u globalnom dosegu u metodama koje se izvode na CPU,
    - vidljiva je svim kernel metodama u programu i one ju mogu samo čitati,
    - koristi se npr. za neku konstantu koju sve dretve čitaju s iste memorijske adrese i koriste za izračun nad različitim podatcima,

– **Teksturna memorija** (engl. *texture memory*)
    - sklopovski se nalazi isto gdje i globalna memorija, automatski se kopira u zasebni *cache* za svaki SM,
    - samo za čitanje, koristi se za jako specifične obrasce pristupa zbog čega u nekim primjenama može biti brža od globalne memorije.

Važan faktor koji utječe na performanse programa je broj aktivnih blokova na SM-u. Svaki blok zahtijeva određene resurse: prostor za podatke o dretvama (*program counter*, *stack counter*, itd.), registre, dijeljenu memoriju. Zbog toga je broj blokova koji mogu biti istovremeno aktivni na pojedinom SM-u ograničen, od 1 do najviše 32. Zato je važno isprobati razne konfiguracije izvođenja kako bi se odabrao dovoljno dobar raspored koji ne ograničava brzinu izvođenja.

3. Neuronska mreža

Umjetna neuronska mreža je metoda strojnog učenja inspirirana radom mozga. Omogućuje izgradnju klasifikacijskih i aproksimacijskih sustava koji danas imaju niz različitih primjena u raznim domenama ljudske djelatnosti. Predstavnik je konektivističkog pristupa unutar područja umjetne inteligencije koji se temelji na kombiniranju niza vrlo jednostavnih procesnih elemenata koji zasebno ne pokazuju inteligentna obilježja, no kada ih se velik broj grupira dobivamo sustav sposoban riješiti neke za računalo teške probleme.



Slika 3.1: Jednostavna unaprijedna neuronska mreža

3.1. Primjer

Na slici 3.1 vidimo primjer mreže koja na ulaz dobiva vektor veličine 4, u srednjem sloju ima 4 neurona, a kao izlaz daje vektor veličine 3. Ovakva konfiguracija može se iskoristiti za rješavanje problema klasifikacije podatkovnog skupa Iris. Skup se sastoji od 150 vektora veličine 4, u kojima svaki element predstavlja vrijednost jedne od 4 značajke. Uzorci su podijeljeni u tri grupe po 50 uzoraka.

Unaprijedni prolaz kroz mrežu daje tri vrijednosti. Svaka pokazuje koliko taj uzorak pripada toj grupi. Vrijednosti su u intervalu $[0, 1]$, te se kod klasifikacije zaokružuju na cijeli broj iz skupa $\{0, 1\}$. Prolaz se provodi tako da se težine grupiraju u matrice i množe kako je prikazano u formuli 3.1.

$$\begin{bmatrix} x_0^{(1)} & x_1^{(1)} & x_2^{(1)} & x_3^{(1)} & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_0^{(i)} & x_1^{(i)} & x_2^{(i)} & x_3^{(i)} & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_0^{(n)} & x_1^{(n)} & x_2^{(n)} & x_3^{(n)} & 1 \end{bmatrix} \cdot \begin{bmatrix} w_{00}^{(1)} & w_{01}^{(1)} & w_{02}^{(1)} & w_{03}^{(1)} \\ w_{10}^{(1)} & w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{20}^{(1)} & w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \\ w_{30}^{(1)} & w_{31}^{(1)} & w_{32}^{(1)} & w_{33}^{(1)} \\ b_0^{(1)} & b_1^{(1)} & b_2^{(1)} & b_3^{(1)} \end{bmatrix} = \begin{bmatrix} o_0^{(1)} & o_1^{(1)} & o_2^{(1)} & o_3^{(1)} \\ \vdots & \vdots & \vdots & \vdots \\ o_0^{(i)} & o_1^{(i)} & o_2^{(i)} & o_3^{(i)} \\ \vdots & \vdots & \vdots & \vdots \\ o_0^{(n)} & o_1^{(n)} & o_2^{(n)} & o_3^{(n)} \end{bmatrix} \tag{3.1}$$

U prvoj matrici nalaze se ulazni vektori. Druga matrica sadrži težine prvog sloja $w_{ij}^{(1)}$, gdje indeks i označava i-ti neuron u prethodnom sloju, a indeks j označava j-ti neuron u trenutnom sloju. Rezultat množenja su izlazi neurona za svaki uzorak. Na te izlaze primjenjuje se funkcija nelinearnosti (u ovom slučaju sigmoidalna), te se taj rezultat prosljeđuje u sljedeći sloj i postupak se ponavlja.

3.2. Implementacija

U sklopu ovog rada implementiran je problem učenja jednostavne unaprijedne neuronske mreže algoritmom klonske selekcije. Neuronske mreže najčešće se uče gradijentnim metodama, no kako je cilj ovog rada napraviti brzu i efikasnu implementaciju za buduće korištenje u programskom okviru ECF, odlučeno je paralelizirati izvođenje evaluacije, dok je algoritam samo primjer primjene pa se izvodi slijedno. Ovakva strategija ima prednost jer u evolucijskim algoritmima kojima se uči neuronska mreža najviše vremena iziskuje upravo evaluacija, dok je izvođenje ostalih operatora nekoliko redova veličine brže.

3.2.1. Algoritam klonske selekcije

Pseudokod algoritma klonske selekcije prikazan je u nastavku.

Algorithm 1 ClonAlg

Ulaz: $prag$ – zadovoljavajuća dobrota
Izlaz: vektor težina koje daju najveću dobrotu
for $(i := 0; i < maxBrojIteracija; inc(i))$ **do**
  $klonovi \leftarrow klonirajPopulaciju()$
  $kandidati \leftarrow mutiraj(klonovi)$
  $novi \leftarrow generirajNove()$
  $kandidati.dodaj(novi)$
  $evaluirajPopulaciju(kandidati)$
  $najbolji \leftarrow odaberiNajboljegIzPopulacije(kandidati)$
  if $najbolji.dobrota > prag$ **then**
    $break$
  end if
end for

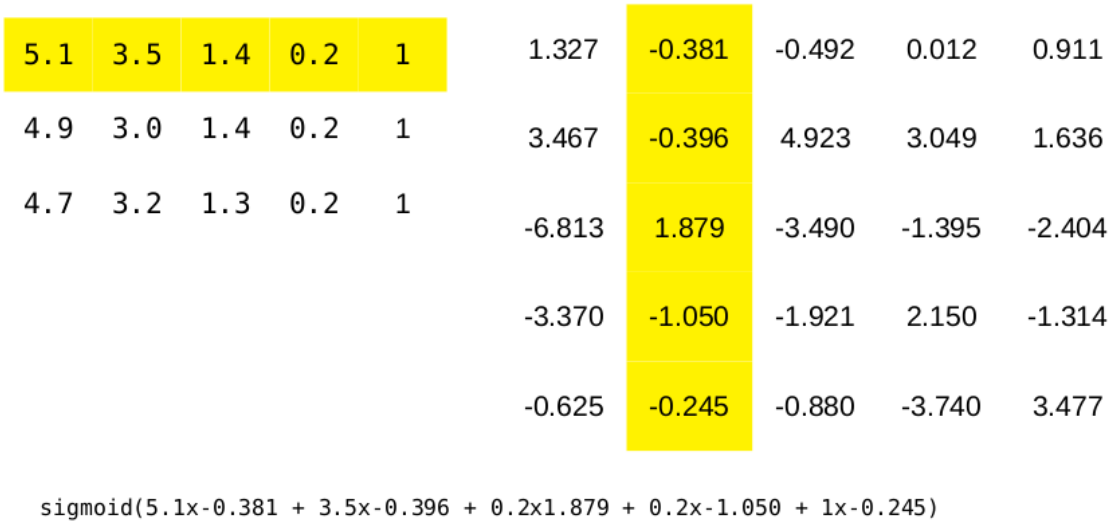
U svakoj generaciji nova populacija gradi se tako da se jedinke kloniraju proporcionalno njihovoj dobroti – bolja jedinka ima više klonova. Nakon toga svaka se jedinka mutira tako da se odabere nekoliko težina i njima se doda slučajni broj iz normalne razdiobe, s očekivanjem 0 i devijacijom 1. Dodaje se i određen broj slučajno generiranih novih jedinki i takav skup šalje se na evaluaciju. Zatim se iz njega odabire sljedeća populacija.

3.2.2. Prikaz rješenja

Svako rješenje je uređena lista težina neuronske mreže. Ono se evaluira tako da se jedinke prethodnog i sljedećeg sloja promatraju kao matrica, kao što je prije opisano. Izlaz se dobiva tako da se matrice množe uz dodavanje pomaka (engl. *bias*). Na to se primjenjuje funkcija nelinearnosti (ovdje sigmoidalna). Na slici 3.2 prikazano je računanje izlaza drugog neurona u prvom sloju za prvi uzorak (označeno žuto).

3.2.3. Kernel metoda

Paralelizacija se provodi na razini množenja matrica. Izlaz svakog neurona računa se u zasebnoj dretvi. Na primjer, izlaz drugog neurona u prvom sloju iz gornjeg primjera računa se tako da se vrijednosti prvog uzorka (prvi redak u prvoj matrici) množe težinama neurona (drugi stupac u matrici) u jednoj CUDA dretvi, kao na slici 3.2. Kernel metoda prikazana je u nastavku.



Slika 3.2: Primjer računanja izlaza

```
__global__ void forwardKernel(Matrix A, Matrix B, Matrix C) {
    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    if (x >= C.XX || y >= C.YY) return;

    double tmp = 0.;
    for (int k = 0; k < B.YY - 1; k++) {
        double valA = GetElement(A, k, y);
        double valB = GetElement(B, x, k);
        tmp += valA * valB;
    }

    tmp += GetElement(mB, x, rB - 1);    // bias
    tmp = activationFunction(tmp);       // sigmoid

    SetElement(C, x, y, tmp);
}
```

Izvođenje se konfigurira sljedećim kodom.

```
const uint BLOCK_SIZE = 16;
dim3 dimGrid;
dimGrid.x = (cols + BLOCK_SIZE - 1) / BLOCK_SIZE;
dimGrid.y = (dataset.SIZE + BLOCK_SIZE - 1) / BLOCK_SIZE;
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
```

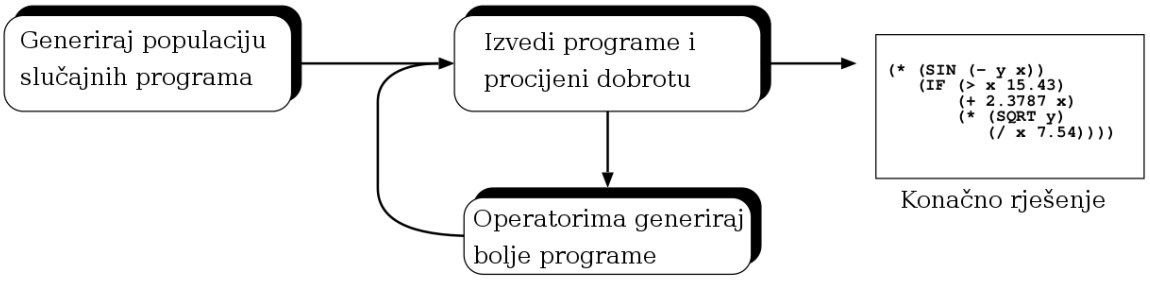
Na početku kernel metode određuju se koordinate nad kojima se provodi množenje. Te koordinate određene su konfiguracijom rešetke. U ovom slučaju rešetka se sastoji od blokova 16x16, što znači da se na jednom SM-u u jednom bloku u 16x16 dretvi izračuna izlaz iz 16 neurona dobivenih množenjem 16 redaka ulaza iz prve matrice i 16 stupaca težina za pripadne neurone iz druge matrice.

Iako je ovakva implementacija značajno brža od izvođenja na CPU kod mreža s velikim brojem težina, kod malih postoji problem pripremnih poslova (engl. *overhead*). Naime, prije izvođenja na GPU, težine se moraju kopirati u globalnu memoriju kartice, a to je postupak koji zahtijeva blokiranje dijela memorije u glavnoj memoriji računala, uspostavljanje veze s GPU-om i kopiranje podataka preko sabirnice. Samo kopiranje je izrazito brzo jednom kada počne, no *overhead* koji je potrebno obaviti prije toga može prevagnuti kod mreža s malim brojem težina. Zbog toga je potrebno minimizirati kopiranje između glavne memorije i GPU-a.

Nešto manji *overhead* javlja se kod pokretanja izvođenja kernel metode. Iako novije implementacije CUDA Toolkit-a značajno smanjuju vrijeme potrebno za pokretanje kernel metode, za mali broj operacija više se isplati izvesti ih slijedno na računalu.

4. Genetsko programiranje

Genetsko programiranje (u daljnjem tekstu GP) je metoda evolucijskog računarstva kojom je moguće pronaći rješenje problema bez zahtjeva da se unaprijed odredi kakvog ono mora biti oblika ili u kojem smjeru ga je potrebno tražiti. Osnovna ideja genetskog programiranja sadržana je u sljedećem dijagramu.



Slika 4.1: Tijek izvođenja algoritma genetskog programiranja

Genetsko programiranje je slično ostalim metodama evolucijskog računarstva; na početku imamo početnu populaciju kandidata, posebnim operatorima ona evoluira i kroz određeni broj generacija dobivamo najboljeg kandidata koji (najčešće) predstavlja dovoljno dobro rješenje. Razlika je u prikazu rješenja – ovdje se evoluira populacija (jednostavnih) računalnih programa, sastavljenih od skupa instrukcija koje su definirane u domeni primjene.

GP sustav procjenjuje dobrotu svakog programa tako da ga izvede (evaluira) i dobiveni rezultat uspoređuje sa željenim rezultatom. Dobrota nekog rješenja određena je izlazom funkcije dobrote (engl. *fitness*) i ona pokazuje koliko je rezultat blizak željenom. Iz generacije u generaciju opstaju najbolji kandidati, množe se i generiraju nove. Osnovni operatori kojima se to postiže su operatori križanja (engl. *crossover*) i operatori mutacije (engl. *mutation*). U osnovnoj implementaciji GP sustava operator križanja odabire dva kandidata, iz njih slučajno uzima dio stabla i stvara novog. Operator mutacije odabire nekog kandidata, slučajno odabire neki njegov dio i zamijeni ga slučajno generiranim dijelom.

Postoje različiti načini kako zapisati rješenja u memoriju računala. Ona mogu biti zapisana kao klasičan zapis programa, no najčešće se koristi zapis apstraktnog sintaksnog stabla. Takvo stablo često se zapisuje u prefiksnoj notaciji, kao u programskom jeziku Lisp.

GP ima mnoštvo primjena u raznim domenama, npr. predviđanje vremenskih nizova ili generiranje *fuzzy* pravila za neki industrijski proces.

4.1. Simbolička regresija

Simbolička regresija problem je pronalaženja matematičkog izraza koji najbolje opisuje ulazne podatke. U odnosu na običnu regresiju, simbolička regresija ne pretpostavlja model, već se početni izrazi slučajno kombiniraju iz matematičkih elemenata kao što su aritmetički operatori, funkcije, konstante i varijable stanja.

Zato što se na početku ne određuje model, ljudska pristranost ne utječe na rezultat algoritma. Umjesto toga, algoritam iz skupa podataka može pronaći veze koje nisu očite tijekom pažljive analize. Postoje razne tehnike u genetskom programiranju koje mogu osigurati da rješenja koja algoritam pronađe zadovoljavaju i neke strukturne zahtjeve, ne samo funkciju gubitka.

Razlika u odnosu na klasičnu regresiju je značajno veći prostor pretraživanja u odnosu na metode koje pretražuju parametre. To može biti prednost jer, kako vrijeme protječe, nova rješenja su kombinacija najboljih prethodnih i novih slučajno generiranih koji su dodani u populaciju.

4.1.1. Realne funkcije

Za potrebe ovog rada korištena je simbolička regresija realnih funkcija. To su funkcije koje imaju ulazne varijable čija je domena iz skupa realnih brojeva, a izlaz je također realni broj.

Skup operatora je

$$\left\{+, *, -, /, sin, cos, \sqrt{\ }\right\},$$

a skup literala je

$$\left\{x0, x1, ..., xn, slucajna\_konstanta\right\}.$$

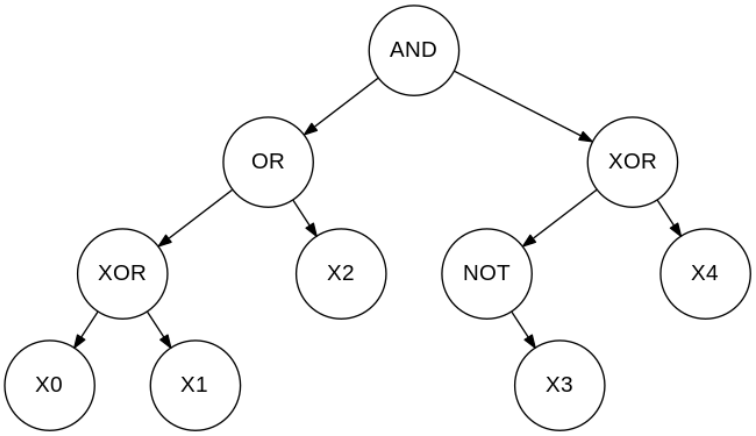
4.1.2. Logičke funkcije

U ECF-u postoji mogućnost dodavanja vlastitih operatora, pa je za buduće potrebe implementiran i skup operatora za logičku domenu:

$$\left\{AND, OR, XOR, XNOR, NAND, NOR, NOT\right\}.$$

4.2. Primjer

Na slici vidimo primjer jednog rješenja simboličke regresije logičke varijable. Ono se sastoji od operatora $\{AND, OR, XOR, NOT\}$ i literala $\{X0, X1, X2, X3, X4\}$.



Slika 4.2: Stablo izraza $AND(OR(XOR(X0, X1), X2), XOR(NOT(X3), X4))$

4.3. Implementacija

Algoritmi evolucijskog računarstva (genetski algoritam, genetsko programiranje, itd.) uspješno rješavaju veliki broj teških optimizacijskih problema. To su često kombinatorički problemi koji nisu rješivi iscrpnom pretragom u razumnom vremenu. Navedene metode pomažu nam da dođemo do dovoljno dobrog rješenja uz relativno malu cijenu, zbog čega su tema velikog broja istraživačkih radova.

Ovakvi algoritmi često su procesorski vrlo zahtjevni, kod velikog broja problema njihovo izvođenje traje dugo i zbog toga se ne mogu primijeniti u nekim slučajevima kada nam je odgovor potreban u relativno kratkom vremenu. Za druge slučajeve, kada imamo nešto više vremena za traženje dovoljno dobrog rješenja, performanse možemo značajno popraviti paralelizacijom. To nam omogućuje da algoritam u danom vremenu šire i kvalitetnije pretraži prostor rješenja, u odnosu na slijedni algoritam.

4.3.1. Metode paralelizacije u evolucijskom računarstvu

Paralelizacija se provodi iz nekoliko razloga:

– **velika količina podataka** – postoje problemi čija su rješenja prevelika da bi stala u memoriju jednog računala, odnosno problemi kod kojih evaluacija generira veliku količinu podataka koja ne stane u memoriju jednog računala;

– **velika količina posla** – postoje problemi čija rješenja se sporo evaluiraju (npr. razne simulacije), problemi kod kojih je potrebno za kvalitetno rješenje pretražiti veliki prostor rješenja, problemi kod kojih je potrebno održavati veliku populaciju kandidata;

– **skraćivanje vremena izvođenja** – često je potrebno izvršiti konstantnu količinu posla, a ako se dijelovi mogu paralelizirati, čitav postupak će biti gotov prije i može se više vremena posvetiti finijem pretraživanju manjeg prostora rješenja.

U svakom od slučaja, paralelizacijom možemo postići:

– brži dolazak do prihvatljivih rješenja,

– veću kvalitetu pronađenih rješenja (zbog većeg prostora koji se može pretražiti u danom vremenu)

– veću robusnost algoritma (manja vjerojatnost zaglavljivanja u lokalnim optimumima zbog mogućnosti održavanja veće populacije).

Paralelizacija evolucijskog algoritma može se izvesti na tri načina:

– **na razini algoritma** – istovremeno izvođenje više instanci algoritma, može biti suradna (razmjenjuju informacije) i nesuradna (ne razmjenjuju informacije);

– **na razini populacije** – paralelno se izvodi generiranje kandidata (npr. kod genetskog algoritma radnik obavlja selekciju, križanje, mutaciju i vrednovanje i kao rezultat vraća gotovo rješenje);

– **na razini jedne iteracije algoritma** – paralelizacija algoritma na najnižoj razini (npr. jedan radnik izvodi isključivo jedan operator, predaje rezultat i čeka sljedećeg kandidata za obradu).

U svakom od navedenih slučaja potrebno je odlučiti kakav način paralelizacije je prihvatljiv s obzirom na dostupno sklopovlje. Paralelizaciju možemo izvesti kroz nekoliko scenarija:

– ako imamo **višejezgreni procesor**, na svaku jezgru možemo pokrenuti proces s jednom instancom algoritma, oni međusobno komuniciraju; nedostatak ovog pristupa je potreba za sinkronizacijom;

– ako imamo **više računala**, možemo ih povezati u posebnu topologiju (npr. grozd, rešetka) ili im omogućiti da komuniciraju putem lokalne mreže; nedostatak ovog pristupa je značajna količina pripremnog posla kojeg stvara komunikacija mrežom (serijalizacija podataka, protokoli, itd.); primjer tehnologije koja ovo omogućuje je MPI (engl. *Message Passing Interface*);

– ako imamo **grafički procesor** (GPU), kod nekih problema možemo postići izrazito veliko ubrzanje; nedostatak ovog pristupa je obavezno korištenje *shadera* koji su zna-

čajno slabiji od CPU-a pa nisu prikladni za sve vrste problema; prisutan je i problem značajne količine pripremnih poslova koji se izvode tijekom slanja podataka u globalnu memoriju GPU-a pa zato možemo postići ubrzanje od nekoliko stotina puta, ali se može dogoditi i znatno sporije izvođenje u usporedbi s CPU verzijom, što ovisi o zahtjevima konkretnog problema.

Uzimajući u obzir dostupne resurse i prirodu problema, za implementaciju ovog rada odabrana je paralelizacija na razini iteracije korištenjem tehnologije CUDA. To znači da se u svakoj iteraciji algoritma pojedina rješenja šalju na evaluaciju u grafički procesor i po završetku se dobrota kandidata vraća algoritmu.

Razlozi zbog kojih je odabran GPU su:

- u trenutnoj implementaciji GP algoritama u programskom okviru ECF najsporiji dio je upravo evaluacija, često nekoliko redova veličine u usporedbi s operatorima;
- evaluacija je spora u trenutnim primjenama zbog velike količine računanja koju je potrebno izvesti kako bi se ocijenilo pojedino rješenje;
- u današnjem vremenu generiraju se velike količine podataka, zbog čega je bitno da se implementacija dobro skalira dodavanjem više resursa i jačeg sklopovlja;
- evaluacija kod problema simboličke regresije genetskim programiranjem je zapravo skup primitivnih računskih operacija koje su u današnjim računalnim arhitekturama većinom sklopovski implementirane i izvode se u jednom ciklusu, te najčešće nekoliko redova veličine brže na velikom broju GPU *shadera* u odnosu na klasične procesore s kompleksnim skupom instrukcija i malim brojem jezgri.

Posebno važan razlog je skaliranje – trošak dodavanja novog računala u klaster značajno je veći u odnosu na trošak dodavanja nove grafičke kartice u postojeće računalo, a dobiveno ubrzanje je kod nekih primjena zanemarivo u odnosu na ubrzanje dobiveno novom grafičkom karticom (kao što pokazuju rezultati).

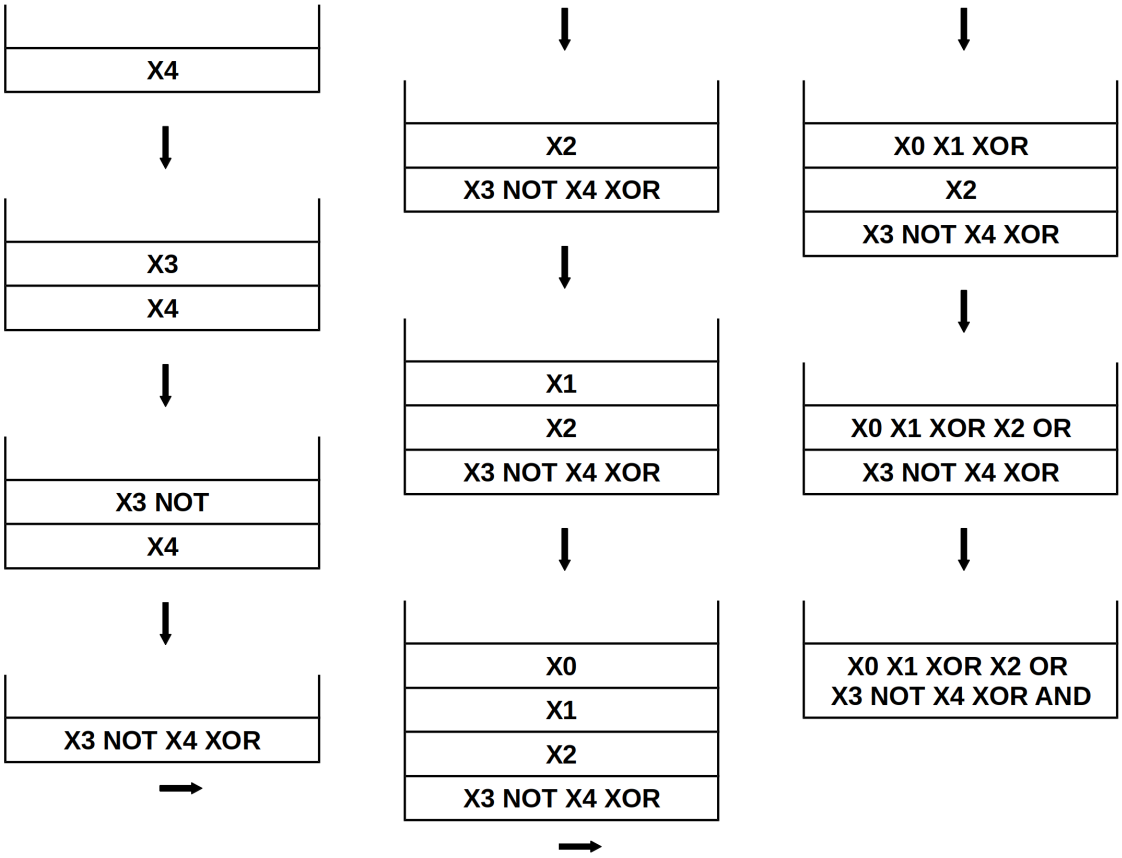
4.3.2. Izvođenje algoritma u okviru ECF

U programskom okviru ECF izvođenje započinje inicijalizacijom operatora evaluacije. U operator se pohrani ulazni skup podataka i proslijede parametri potrebni za računanje. Stablo je zapisano u jednodimenzionalno polje u prefiksnom zapisu, te se evaluira rekurzivnim pozivima za svaki element.

4.3.3. Prevođenje stabla u postfiksni zapis

Zbog rekurzivnih poziva procesiranje dubokih stabala obavlja i veliku količinu pripremnog posla. Osim toga, rekurzivno pozivanje funkcija u CUDA-i trenutno je sporo, zbog čega je takav zapis rješenja neprikladan za paralelizaciju.

Rješenje oba problema je prijevod stabla iz prefiksnog zapisa u postfiksni. Time se kod evaluacije uvodi dodatno polje za stog te se eliminira potreba za rekurzivnim pozivima kratkih funkcija. Takav zapis pogodan je i za GPU jer se zbog jednostavne evaluacije može izvesti brzo, a najveći *overhead* jest kopiranje polja u globalnu memoriju GPU-a. Algoritam prevođenja u postfiksni zapis prikazan je na stranici 22, a izvođenje na stablu iz primjera prikazano je na slici 4.3.



Slika 4.3: Prevođenje stabla sa slike 4.2 u postfiksni zapis

Algorithm 2 Pretvorba u postfiksni zapis

Ulaz: $stablo$ – lista, stablo u prefiksnom zapisu

Izlaz: $rezultat$ – lista, stablo u postfiksnom zapisu

$duljina = stablo.duljina()$

for $(i := duljina - 1; i >= 0; dec(i))$ **do**

$brojArgumenata = stablo[i].brojArgumenata()$

if $brojArgumenata == 3$ **then**

$op1 \leftarrow stog.uzmi()$

$op2 \leftarrow stog.uzmi()$

$op3 \leftarrow stog.uzmi()$

$op1.dodaj(op2)$

$op1.dodaj(op3)$

$op1.dodaj(stablo[i])$

$stog.stavi(op1)$

else if $brojArgumenata == 2$ **then**

$op1 \leftarrow stog.uzmi()$

$op2 \leftarrow stog.uzmi()$

$op1.dodaj(op2)$

$op1.dodaj(stablo[i])$

$stog.stavi(op1)$

else if $brojArgumenata == 1$ **then**

$op1 \leftarrow stog.uzmi()$

$op1.dodaj(stablo[i])$

$stog.stavi(op1)$

else

$tmp \leftarrow stablo[i]$

$stog.stavi(tmp)$

end if

end for

$rezultat \leftarrow stog.uzmi()$

4.3.4. Evaluiranje na GPU

Nakon pretvorbe iz prefiksnog u postfiksni zapis, stablo se evaluira kako je prikazano na slici 4.4 na stranici 26.

U primjenama se evaluira od tisuću do milijun ulaznih vektora. Za svaki primjer poziva se kernel metoda koja izvodi evaluaciju, opisana u nastavku.

```
1  __global__ void d_evaluateIndividual(uint *d_program,
2      BOOL_TYPE *d_datasetInput, BOOL_TYPE *d_datasetOutput,
3      BOOL_TYPE *d_resultOutput, uint *d_resultFitness,
4      int N, int DIM, int PROG_SIZE) {
```

Metodi kao argumente predajemo GP program zapisan u prefiksnoj notaciji u polje, u kojem je svaka operacija predstavljena predefiniranim cjelobrojnim kodom.

```
5      uint tid = blockIdx.x * blockDim.x + threadIdx.x;
6      if (tid == 0) *d_resultFitness = 0;
```

Na početku metode odredi se redni broj dretve u rešetci. Ako je dretva prva, inicijalizira varijablu u globalnoj memoriji u koju će se zapisati dobrota rješenja, odnosno rezultat evaluacije.

```
7      extern __shared__ uint shared_programCache[];
8      for (uint idx = threadIdx.x;
9          idx < PROG_SIZE;
10         idx += THREADS_IN_BLOCK) {
11         shared_programCache[idx] = d_program[idx];
12     }
13     __syncthreads();
14
15     if (tid >= N) return;
```

U navedenom odsječku bitno je obratiti pažnju na korištenje dijeljene memorije. Kako smo već spomenuli, globalna memorija je najveća, ali i najsporija memorija u CUDA arhitekturi. Kako je za neko prosječno stablo najveća dubina 11, stablo može imati najviše $2^{11} - 1$ elemenata koji zauzimaju 4 B $* 2047$ `unsigned int` zapisa, odnosno 8 KB. U vrijeme pisanja teksta čak i pet godina stare kartice imaju najmanje 48 KB dijeljene memorije, pa je logično iskoristiti ju kao programski upravljan *cache*. Svaka dretva pročita $duljina\_stabla/broj\_dretvi\_u\_bloku$ elemenata stabla i zapiše ih u polje koje se nalazi u dijeljenoj memoriji. Veličina tog polja određena je brojem elemenata u stablu i određuje se prije pozivanja kernel metode. Na ovaj način svaka dretva čita $duljina\_stabla$ puta iz brze dijeljene memorije, umjesto iz spore globalne memorije.

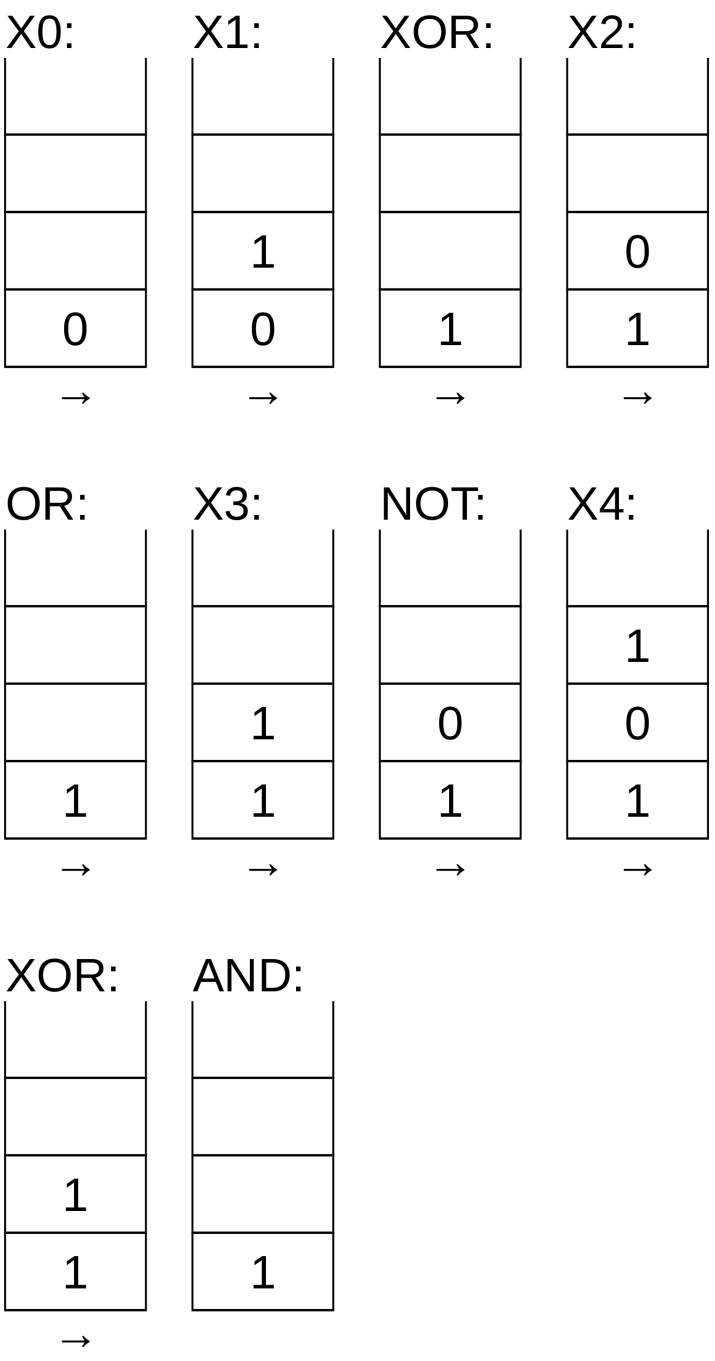
```
16      BOOL_TYPE stack[MAX_STACK_SIZE];
17      BOOL_TYPE *inputSample = d_datasetInput + tid * DIM;
18      int SP = 0;      // pokazivac na vrh stoga
19      BOOL_TYPE o1, o2, tmp;  // varijable za operande
20      uint code;       // predefinirani kod operatora
21      uint idx;        // indeks ulazne varijable
22
23      for (int i = 0; i < PROG_SIZE; i++) {
24          if (shared_programCache[i] >= ARITY_3) {
25              o3 = stack[--SP];
26              o2 = stack[--SP];
27              o1 = stack[--SP];
28              switch (shared_programCache[i]) {
29                  case IF:
30                      tmp = o1 ? o2 : o3; break;
31              }
32          } else if (shared_programCache[i] >= ARITY_2) {
33              o2 = stack[--SP];
34              o1 = stack[--SP];
35              switch (shared_programCache[i]) {
36                  case AND:
37                      tmp = o1 && o2; break;
38                  case OR:
39                      tmp = o1 || o2; break;
40              }
41          } else if (shared_programCache[i] >= ARITY_1) {
42              o1 = stack[--SP];
43              switch (shared_programCache[i]) {
44                  case NOT:
45                      tmp = !o1; break;
46              }
47          } else if (shared_programCache[i] >= VAR
48                  && shared_programCache[i] < CONST) {
49              code = shared_programCache[i];
50              idx = code - VAR;
51              tmp = inputSample[idx];
52          }
53          stack[SP++] = tmp;
54      }
```

U ovom odsječku prikazan je kod za evaluaciju stabla. Petlja prolazi poljem, prema kodu operatora određuje koliko je operanda potrebno za trenutni operator, uzima ih sa stoga, odredi o kojem se operatoru radi i izračuna rezultat. Taj rezultat zatim stavlja na stog i postupak se ponavlja.

```
55      BOOL_TYPE result = stack[--SP];
56      if (result != d_datasetOutput[tid]) {
57          atomicAdd(d_resultFitness, 1);
58      }
59  }
```

Nakon evaluacije jedini podatak na stogu je rezultat izvršavanja programa. Razlika između rezultata i točne vrijednosti za taj ulazni primjer (u slučaju logičkih varijabli iznosi 1) pribroji se atomičkom (nedjeljivom, višedretveno sigurnom) operacijom u varijablu u globalnoj memoriji. Nakon što su sve dretve gotove, glavna metoda kopira rezultat u RAM memoriju računala kako je prikazano u primjeru zbrajanja dva vektora.

X	Y
X0	0
X1	1
X2	0
X3	1
X4	1



Slika 4.4: Evaluacija stabla u postfiksnom zapisu

5. Prioritetno raspoređivanje

Raspoređivanje je problem u kojem se određen skup resursa dodjeljuje određenom skupu poslova. Rješenje problema raspoređivanja je raspored koji određuje koji poslovi će biti izvedeni na kojim resursima i kojim redoslijedom. Problem raspoređivanja javlja se u mnogim domenama ljudske djelatnosti, npr. raspored poslova u tvorničkom procesu, izrada rasporeda sati za studente itd.

Rješenje problema raspoređivanja je raspored koji optimalno zadovoljava određene uvjete (npr. minimalno kašnjenje poslova, minimalni broj rupa u rasporedu). Izrada ovakvih rasporeda najčešće je izrazito težak kombinatorički problem. Zbog toga ovakve probleme nije moguće riješiti metodama iscrpne pretrage jer vrijeme koje je potrebno za to nije realno prihvatljivo, a teoretski može trajati mjesecima ili godinama. Iz navedenog razloga za rješavanje ovog problema najčešće se koriste određene heurističke metode koje ne jamče da će pronađeno rješenje biti optimalno, ali će zato konvergirati prema njemu i kao rezultat dati rješenje koje je dovoljno dobro, u prihvatljivom vremenskom razdoblju.

U literaturi se resursi često nazivaju strojevima. Iako postoje različite varijante raspoređivanja s obzirom na vrstu strojeva, kao što su paralelni identični strojevi, jednoliki strojevi, nesrodni strojevi i slično, u ovom radu pažnja je posvećena samo raspoređivanju u okruženju nesrodnih strojeva.

Okruženje nesrodnih strojeva je okruženje u kojem svaki stroj obrađuje svaki pojedini posao proizvoljno definiranom brzinom, što znači da je za svaki uređeni par $(stroj, posao)$ definirana brzina kojom se zadani posao izvršava na zadanom stroju. Zato takve strojeve zovemo nesrodnima, jer iako je poznata brzina izvršavanja određenih poslova na strojevima, iz nje ne možemo zaključiti ništa o odnosu brzina između dva stroja. Naime, ako je poznato da se svi promatrani poslovi izvršavaju dvostruko brže na jednom stroju u odnosu na drugi, to ne znači da je prvi stroj dvostruko brži, jer može postojati posao koji će se izvršavati jednakom brzinom ili brže na drugom.

Broj ovakvih rasporeda stoga može biti izrazito velik, zbog čega se takvi problemi najčešće rješavaju heurističkim optimizacijskim postupcima.

5.1. Obilježja poslova

Posao je određena aktivnost koju postupkom raspoređivanja želimo dodijeliti nekom stroju kako bi se na njemu izvršila. Ovisno o okruženju u kojem se obavlja raspoređivanje, poslovi se mogu sastojati od više aktivnosti ili od samo jedne nedjeljive aktivnosti. U daljnjem tekstu navedena su najvažnija obilježja kojima se opisuju poslovi i koja predstavljaju temelj za razumijevanje postupaka raspoređivanja. Tekst je preuzet iz (Jakobović, 2005).

Skup svih poslova označuje se s J, dok se pojedini poslovi označuju s J_j, pri čemu indeks j predstavlja redni broj posla. Najvažnija svojstva koja su, ovisno o okolini, pridružena svakom poslu su:

- **Trajanje izvođenja posla** p_{ij} (engl. *processing time*) – označava trajanje koje je potrebno da se posao J_j obavi na stroju i.
- **Vrijeme pripravnosti posla** r_j (engl. *ready time*, *release time*) – označava vrijeme, odnosno trenutak u kojem posao postaje raspoloživ za raspoređivanje. Prije svog vremena pripravnosti niti jedan posao ne može se rasporediti niti izvoditi na nekom stroju. S druge strane, posao može čekati na početak izvođenja neodređeno dugo nakon svog vremena pripravnosti, što može biti posljedica zauzetosti svih strojeva ili neispunjavanja nekog drugog uvjeta.
- **Vrijeme željenog završetka posla** d_j (engl. *due date*) – označava vrijeme do kojeg je poželjno, odnosno do kojeg se očekuje da će pojedini posao završiti. Vrijeme završetka ne osigurava da će se pojedini posao stvarno završiti prije tog vremena, ali time je ipak osigurano da se, ako posao završi nakon tog vremena, stvori određeni trošak.
- **Težina posla** w_j (engl. *weight*) – označava prioritet nekog zadatka u sustavu. Najčešće se koristi pri određivanju troškova kod ocjenjivanja rasporeda, gdje predstavlja neku stvarnu mjeru kvalitete nekog rasporeda. U slučaju da se ocjena rasporeda temelji na više kriterija, poslu se može pridružiti više težina koje definiraju prioritete za svaki pojedini kriterij.

5.2. Ocjena kvalitete rasporeda

Svaki raspored generiran nekom metodom potrebno je na neki način ocijeniti. Potrebno je uzeti u obzir da nisu svi kriteriji u svim primjenama jednako važni. Stoga je potrebno razviti nekoliko različitih načina ocjenjivanja rasporeda. Prije opisa načina ocjenjivanja, potrebno je definirati neke izlazne veličine sustava, koje će kasnije poslužiti za ocjenjivanje pojedinog rasporeda.

– **Vrijeme završetka** C_j (engl. *completion time*) – trenutak u kojem je posao s indeksom j završio s izvođenjem.

– **Protjecanje** F_j (engl. *flowtime*) – količina vremena koju je neki posao proveo u sustavu, odnosno količina vremena od kada je posao postao pripravan za izvođenje do trenutka kada je završio s izvođenjem.

$$F_j = C_j - r_j$$

– **Kašnjenje** L_i (engl. *lateness*) – razlika između vremena završetka i željenog vremena završetka posla (važno je uočiti da može biti pozitivna i negativna).

$$L_i = C_j - d_j$$

– **Zaostajanje** T_j (engl. *tardiness*) – pozitivan iznos kašnjenja nekog posla (ako je kašnjenje negativno, postavlja se na nulu).

$$T_j = \max\{0, L_j\}$$

– **Preuranjenost** E_j (engl. *earliness*) – negativan iznos kašnjenja nekog posla, odnosno koliko je taj posao završio ranije od željenog vremena završetka.

$$E_j = \max\{0, -L_j\}$$

– **Zakašnjelost** U_j – označuje je li neka aktivnost prekoračila željeno vrijeme završetka.

$$U_j = \begin{cases} 1 : T_j > 0 \\ 0 : T_j = 0 \end{cases}$$

Nakon što su opisane sve potrebne izlazne veličine sustava, možemo opisati pojedine kriterije vrednovanja dobivenih rasporeda.

– **Ukupna duljina rasporeda** C_{max} (engl. *makespan*) – posljednje vrijeme završetka svih poslova u sustavu.

$$C_{max} = \max\{C_j\}$$

– **Najveće kašnjenje** L_{max} (engl. *maximum lateness*) – najveće vrijeme kašnjenja nekog posla u sustavu.

$$L_{max} = \max\{L_j\}$$

– **Težinsko protjecanje** F_w (engl. *weighted flowtime*) – suma težinskog protjecanja svih poslova u sustavu.

$$F_w = \sum_j w_j F_j$$

– **Težinsko zaostajanje** T_w (engl. *weighted tardiness*) – težinska suma zaostajanja svih poslova.

$$T_w = \sum_j w_j T_j$$

– **Težinski broj zaostalih poslova ili težinska zakašnjelost** U_w (engl. *weighted number of tardy jobs*) – težinska suma svih zaostalih poslova.

$$U_w = \sum_j w_j U_j$$

– **Težinska preuranjenost i težinsko zaostajanje** ET_w (engl. *weighted earliness and weighted tardiness*) – zbroj težinske preuranjenosti i težinskog zaostajanja, uz uvjet da su težinski faktori posebno definirani za obje navedene vrijednosti.

$$ET_w = \sum_j \left(w_{E_j} E_j + w_{T_j} T_j \right)$$

Prethodno navedeni načini ocjene nisu jedini koji se koriste. Međutim, iako postoji mnogo načina ocjenjivanja rasporeda, ovi predstavljaju skup najčešće korištenih.

Koji od navedenih kriterija za ocjenjivanje rasporeda koristimo ovisi o tome koji su prioriteti postavljeni na raspored. Tako je u pojedinim slučajevima važno da svi poslovi što prije završe, dok je za druge važnije da je vrijeme kašnjenja pojedinačnih poslova što manje, pri čemu nije toliko bitno hoće li ukupno trajanje poslova biti najmanje moguće. Osim toga, pojedinim poslovima moguće je dati i veći prioritet, što rezultira većom ovisnošću o tome kako su ti poslovi raspoređeni. Također, raspored je moguće ocijeniti i pomoću nekoliko kriterija odjednom, pri čemu se odabire onaj raspored koji najbolje zadovoljava odabrane kriterije. Kako vidimo, postoji više mogućnosti ocjenjivanja rasporeda, pa odabir načina ocjenjivanja ovisi o uvjetima i prioritetima koji su pretpostavljeni.

5.3. Implementacija

Središnja tema ovog diplomskog rada je optimizacija prioritetnog raspoređivanja ostvarena na GPU s ciljem brzog izvođenja. Stoga je najsporiji dio, evaluacija rješenja, određen za implementaciju u CUDA tehnologiji i izvođenje na Nvidia GPU-u.

U postojećoj implementaciji program je generirao rasporede simboličkom regresijom realnih varijabli. Rješenja su bila zapisana kao GP stabla, a nakon simulacije svako rješenje evaluiralo se rekurzivnim pozivom za svaki element stabla. Kako je navedeno u prethodnom poglavlju, ovakva evaluacija je u nekim slučajevima izrazito spora, pa je logični korak implementirati postupak tako da se može izvesti na GPU. Zbog navedenog razloga trenutna implementacija evaluacije rješenja zamijenjena je onom opisanom u prethodnom poglavlju.

Postoji mogućnost korištenja neuronske mreže umjesto GP simboličke regresije. Međutim, kod ovog problema pokazalo se da je kvaliteta rješenja slična kao kod GP-a, no za veći broj neurona nije moguće objasniti rješenje, što je nedostatak. U slučaju GP-a, moguće je izvesti pojednostavljenje simboličkog izraza i time dobiti kompaktniju formulu koja se zatim može analitički proučavati.

6. Rezultati

Cilj ovog rada bio je implementirati opisane metode tako da se mogu izvoditi na GPU. U ovom poglavlju opisana je metodologija mjerenja, prikazani su rezultati i dani njihovi kratki komentari.

Sva mjerenja izvedena su na računalu sljedećih specifikacija:

- CPU: Intel i7-7700, 8 jezgri, 4.2 GHz;
- RAM: 16 GB DDR4 2400 MHz;
- SSD: Samsung 960 EVO NVMe M.2, 3200 MB/s read, 1500 MB/s write, 500 GB;
- GPU: Nvidia GeForce GTX 1080 TI, 3584 jezgri, 1594 MHz, 11 GB GDDR5X;
- GPU: Nvidia TITAN Xp, 3840 CUDA jezgri, 1582 MHz, 12 GB GDDR5X.
- OS: Ubuntu MATE 17.10 x86_64 GNU/Linux distribucija, 4.13.0-generic kernel

Iako su dostupne dvije grafičke kartice, za mjerenja je korištena Nvidia TITAN Xp. Na računalu je dostupan CUDA Toolkit 9.1 i GCC 6.4.0.

6.1. Prioritetno raspoređivanje

Algoritam koji rješava problem prioritetnog raspoređivanja koristi genetsko programiranje za generiranje rasporeda. Kako je dosadašnja implementacija bila dosta spora i izvođenje trajalo danima, cilj je bio ubrzati ju implementacijom evaluacije na GPU. Međutim, zbog prirode problema nije moguće postići ubrzanje. Naime, skup podataka na kojem algoritam uči ima mali broj strojeva (do 10) i mali broj poslova (do 100), zbog čega *overhead* pretprocesiranja i kopiranja svih parametara na GPU traje nekoliko redova veličine dulje nego samo izvođenje evaluacije na CPU. Zato, iako je evaluacija na GPU-u uspješno implementirana, rezultati ubrzanja nisu zadovoljavajući.

6.2. Učenje neuronske mreže algoritmom klonske selekcije

U programskom okviru ECF trenutno ne postoji implementirana neuronska mreža kao genotip, no kako je to u planu u skorim verzijama, moja implementacija je dizajnirana s ciljem lake integracije u okvir. Danas su snažni grafički procesori široko dostupni, zbog čega je važno ostvariti podršku za njih kako bi ECF bio ažuran u odnosu na ostale programske okvire slične namjene.

Implementacija je testirana na problemu klasifikacije skupa podataka. Cilj testiranja je usporediti brzinu GPU implementacije u odnosu na slijednu CPU implementaciju. Algoritam klonske selekcije ima sljedeće parametre:

- populacija: 100;
- prag dobrote: 0.001;
- najveći broj iteracija: 1000.

Ulazni skupovi podataka sastoje se od 150, 600 i 1500 vektora koji sadrže 4 značajke i grupirani su u 3 klase. Testirane su tri konfiguracije neuronske mreže:

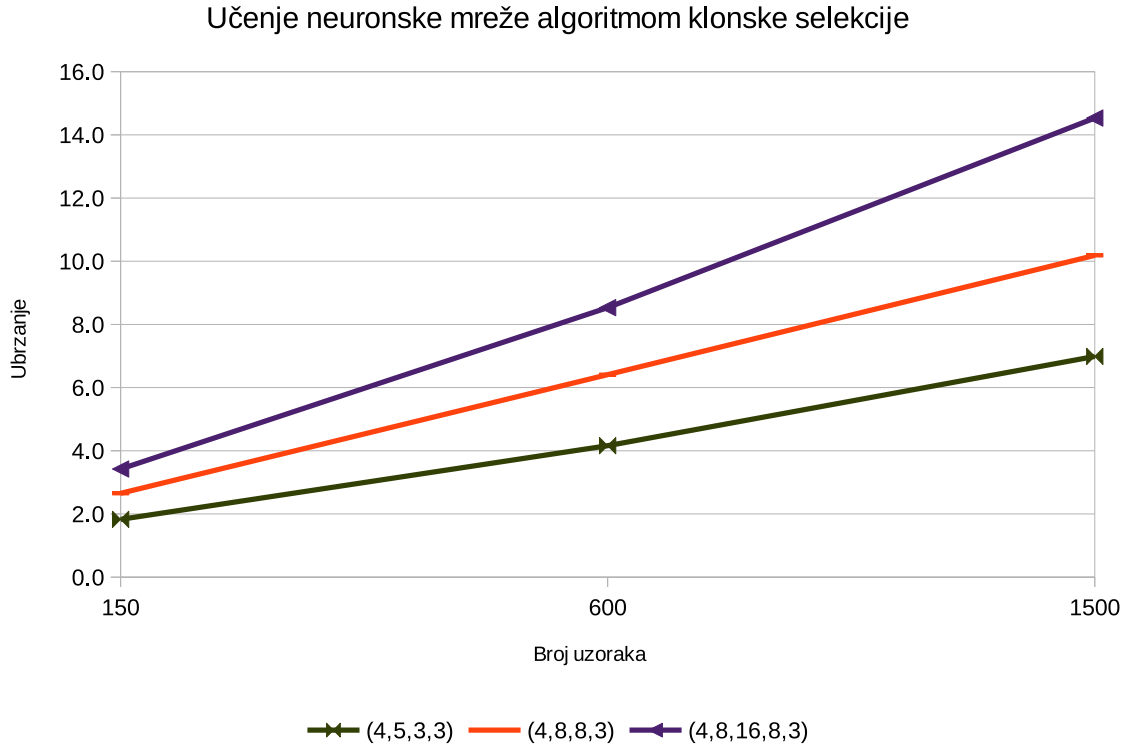
- (4, 5, 3, 3), broj parametara: 56;
- (4, 8, 8, 3), broj parametara: 140;
- (4, 8, 16, 8, 3), broj parametara: 348.

Rezultati su prikazani u tablici 6.1, a ubrzanje je prikazano na slici 6.1.

Tablica 6.1: Rezultati

Konfiguracija	Broj primjera	CPU $[s]$	GPU $[s]$	GPU vs CPU
(4, 5, 3, 3)	150	34.536	18.885	1.82875
(4, 5, 3, 3)	600	121.434	29.194	4.15955
(4, 5, 3, 3)	1500	309.508	44.302	6.98632
(4, 8, 8, 3)	150	69.443	26.171	2.65343
(4, 8, 8, 3)	600	241.733	37.722	6.40828
(4, 8, 8, 3)	1500	585.885	57.518	10.1861
(4, 8, 16, 8, 3)	150	151.145	44.179	3.42120
(4, 8, 16, 8, 3)	600	516.546	60.545	8.53160
(4, 8, 16, 8, 3)	1500	1231.770	84.768	14.5310

Iz rezultata možemo zaključiti da se najveće ubrzanje postiže kod velikih skupova ulaznih primjera.



Slika 6.1: Graf ubrzanja

6.3. Simbolička regresija funkcija realnih varijabli

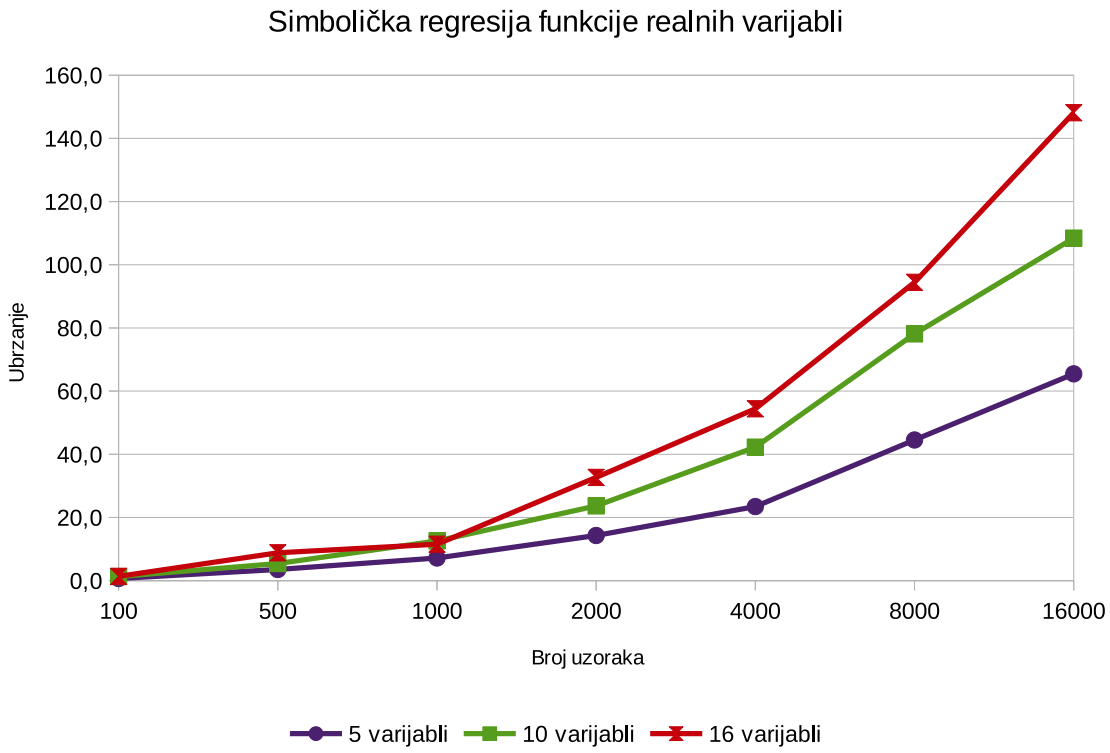
Kao što je prije spomenuto, performanse problema simboličke regresije neke funkcije ograničene su količinom potrebnog računanja, a ne protokom podataka. Stoga je važno raspodijeliti posao tako da je većina *shadera* zauzeta računanjem.

Implementacija je testirana regresijom triju različitih realnih funkcija. Svaka ima fiksni broj varijabli, dok broj ulaznih primjera varira od 100 do 16000. Ulazni primjeri slučajno su uzorkovani uniformnom razdiobom na intervalu $[-5, 5]$. Parametri algoritma su:

- najveći broj generacija: 100,
- najveći broj generacija bez napretka: 30,
- veličina populacije: 500,
- broj turnira u selekciji: 3,
- najveća dubina stabla: 10,
- skup funkcija: $\{+, -, *, /, sin, cos\}$,
- skup terminala: $\{1, X_0, ..., X_N, C \in [-1, 1]\}$.

U tablici 6.2 prikazani su rezultati. U trećem, četvrtom i petom stupcu navedeno je ukupno vrijeme izvođenja operatora evaluacije. Operatori evaluacije su implementacija u

programskom okviru ECF, implementacija postfiksnog interpretera izvedena na CPU, implementacija postfiksnog interpretera izvedena na GPU, tim redoslijedom. U šestom stupcu navedeno je ukupno vrijeme utrošeno na pretvorbu iz prefiksnog u postfiksni zapis GP stabla i to vrijeme već je uračunato u vrijeme izvođenja CPU i GPU verzija postfiksnih interpretera. U preostalim stupcima navedena su ubrzanja za pojedine slučajeve. Graf ubrzanja prikazan je na slici 6.2.



Slika 6.2: Graf ubrzanja

Iz rezultata možemo zaključiti sljedeće:

- izvođenje na GPU najviše se isplati kod velikih skupova primjera za učenje,
- isplati se i kod regresije funkcija s većim brojem varijabli (ovdje 10 i više),
- implementacija u ECF-u sporija je najviše zbog rekurzivnih poziva kratkih funkcija,
- vrijeme potrebno za pretvorbu u postfiksni zapis zanemarivo je već kod manjih problema,
- rješenje je skalabilno: dodavanjem GPU-a s većim brojem *shadera* postiže se veće ubrzanje kod računski zahtjevnijih problema,
- tijekom izvođenja programa na posljednjem skupu (16 varijabli, 16000 uzoraka) zauzeće memorije iznosilo je svega 450 MB, što je malo u odnosu na dostupnu količinu na suvremenim GPU-ima,

Tablica 6.2: Rezultati

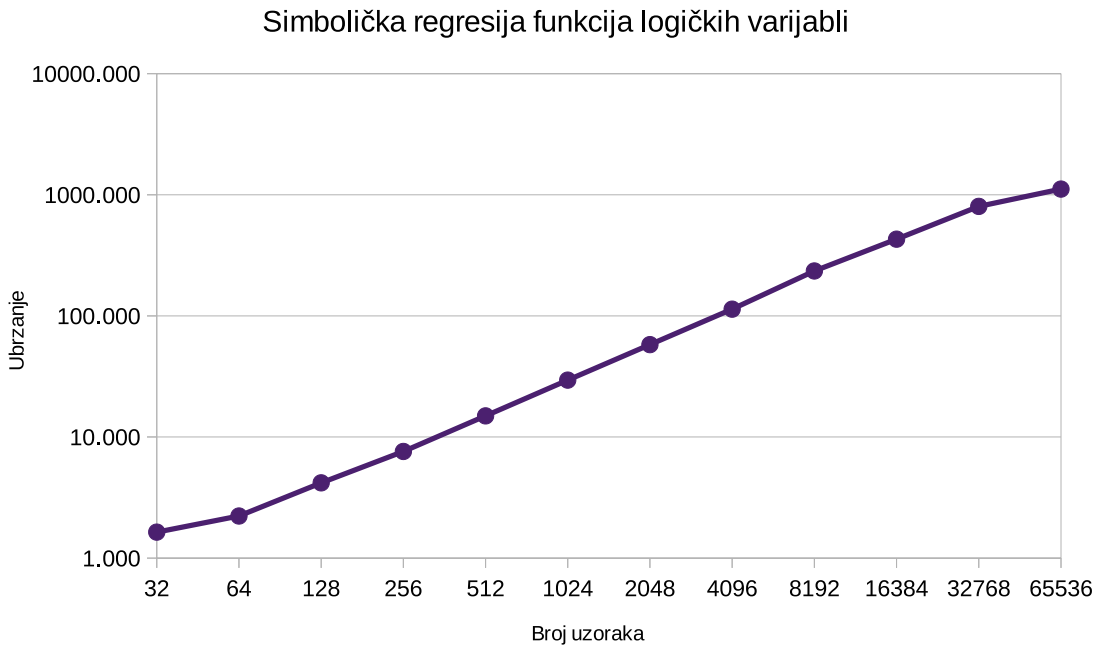
Broj varijabli	Broj uzoraka	ECF $[\mu s]$	CPU $[\mu s]$	GPU $[\mu s]$	Pretvorba $[\mu s]$	CPU vs ECF	GPU vs CPU	GPU vs ECF
5	100	972675	1169447	1447198	550862	0.83174	0.80808	0.67211
5	500	5896546	3568170	1669326	496951	1.65254	2.13749	3.53229
5	1000	11474265	6154359	1601463	443495	1.86441	3.84296	7.16486
5	2000	128560454	113565699	8997987	3572249	1.13204	12.6212	14.2876
5	4000	44667906	23922617	1905301	523119	1.86718	12.5558	23.4440
5	8000	142567179	100649648	3202116	864806	1.41647	31.4322	44.5228
5	16000	522612904	312366460	7984937	1454637	1.67308	39.1194	65.4498
10	100	2460650	1892442	1907520	707632	1.30025	0.99210	1.28997
10	500	27896370	14098967	5150367	1943287	1.97861	2.73747	5.41638
10	1000	32237596	14314990	2565690	833173	2.25202	5.57939	12.5648
10	2000	76825806	43808600	3248317	1172169	1.75367	13.4865	23.6509
10	4000	142058941	73870574	3361479	1069600	1.92308	21.9756	42.2608
10	8000	216105911	61067966	2766466	514628	3.53878	22.0743	78.1162
10	16000	499672411	200535999	4608645	972691	2.49168	43.5130	108.420
16	100	4529229	3953549	3348533	1587400	1.14561	1.18068	1.35260
16	500	23136550	7583762	2626801	1005512	3.05080	2.88707	8.80788
16	1000	79487066	57591909	6892904	3493593	1.38018	8.35525	11.5317
16	2000	96528375	32660104	2954788	1022516	2.95554	11.0532	32.6684
16	4000	126087334	60644593	2318185	917163	2.07912	26.1603	54.3905
16	8000	293973327	126850118	3113043	1081056	2.31749	40.7479	94.4327
16	16000	1020484065	524735082	6889716	2077850	1.94476	76.1620	148.117

6.4. Simbolička regresija funkcija logičkih varijabli

Implementacija je testirana regresijom jedanaest različitih logičkih funkcija. Svaka funkcija ima broj varijabli od 5 do 16. Funkcije nisu uzorkovane, već je ulazni skup podataka potpuna tablica istinitosti. Tako je u slučaju funkcije 5 varijabli broj primjera $2^5 = 32$, a u slučaju 16 varijabli $2^{16} = 65536$. Parametri algoritma su:

- najveći broj generacija: različito s obzirom na broj varijabli,
- najveći broj generacija bez napretka: 10,
- veličina populacije: 200,
- broj turnira u selekciji: 3,
- najveća dubina stabla: 10,
- skup funkcija: $\{AND, OR, XOR, NOT\}$,
- skup terminala: $\{X_0, ..., X_N\}$.

U tablici 6.3 prikazani su rezultati. Značenje stupaca isto je kao i kod tablice 6.2. Graf ubrzanja prikazan je na slici 6.3.



Slika 6.3: Graf ubrzanja

Iz rezultata možemo zaključiti sljedeće:

- u odnosu na regresiju funkcija realnih varijabli, ovdje se izvođenje na GPU isplati već kod manjeg broja primjera,
- najveće ubrzanje javlja se kod regresije funkcija s većim brojem varijabli,

Tablica 6.3: Rezultati

Broj varijabli	Broj uzoraka	ECF $[\mu s]$	CPU $[\mu s]$	GPU $[\mu s]$	Pretvorba $[\mu s]$	CPU vs ECF	GPU vs CPU	GPU vs ECF
5	32	1703435	1039691	1039668	695955	1.63841	1.00002	1.63844
6	64	7067200	4344904	3176098	2151650	1.62655	1.36800	2.22512
7	128	15778386	7311869	3773399	2535674	2.15791	1.93774	4.18148
8	256	14646819	5367124	1929114	1272205	2.72899	2.78217	7.59251
9	512	96968258	35210393	6494450	4379884	2.75397	5.42161	14.9309
10	1024	66557099	21426709	2258350	1482271	3.10627	9.48777	29.4715
11	2048	248011781	83522298	4286620	2883968	2.96941	19.4844	57.8571
12	4096	154196361	43392521	1356403	885157	3.55352	31.9908	113.680
13	8192	472219677	137779262	2009335	1326042	3.42736	68.5695	235.012
14	16384	307932311	77804608	715978	446570	3.95776	108.668	430.086
15	32768	550201350	134191568	685830	403850	4.10012	195.663	802.241
16	65536	741499028	181242577	665248	265863	4.09120	272.443	1114.62

– implementacija u ECF-u sporija je iz istog razloga kao i kod funkcija realnih varijabli, zbog velikog *overheada* tijekom rekurzivnih poziva kratkih funkcija,

– ovdje je vrijeme potrebno za pretvorbu u postfiksni zapis također zanemarivo u odnosu na vrijeme izvođenja u ECF-u,

– ovo rješenje je također skalabilno,

– tijekom izvođenja posljednjeg primjera (16 varijabli, 65536 primjera) zauzeće memorije na GPU iznosi svega 180 MB kod zapisa tipa `bool` tipom `unsigned char`, odnosno 220 MB kod zapisa tipom `unsigned int`, što je također zanemarivo u odnosu na dostupnu količinu memorije na GPU,

– prema rezultatima vidimo da kod problema regresije funkcije s velikim brojem varijabli (16+) GPU implementacija omogućava pronalazak kvalitetnog rješenja u razumnom vremenu, što nije moguće u ostalim implementacijama.

Kod ovog problema bitno je pažljivo odabrati primitivni tip za zapis logičkih varijabli. Naime, pokazalo se da najbolje performanse daje tip `unsigned char`, dok najlošije daje `bool` (zbog implicitne pretvorbe u `unsigned short`).

7. Zaključak

Cilj ovog diplomskog rada bio je ubrzati trenutnu implementaciju algoritma optimizacije prioritetnog raspoređivanja strojeva. Iako je algoritam uspješno implementiran, rezultati ubrzanja nisu zadovoljavajući. Postoji nekoliko razloga zbog kojih je to slučaj, od kojih navodimo dva najvažnija.

Prvi razlog je neprikladnost algoritma za izvođenje na GPU. Algoritam za generiranje rasporeda mora obaviti veliku količinu složenog računanja. Kako je GPU arhitektura dizajnirana za obradu velike količine podataka korištenjem jednostavnog skupa operacija na velikom broju mikroprocesora, ni teoretski nije moguće ostvariti dobar rezultat u ovakvoj kombinaciji. Kao što je ilustrirano u primjerima, program prvo mora kopirati podatke potrebne za računanje u GPU memoriju i nakon računanja mora kopirati podatke iz GPU memorije. To nije isplativo raditi često, a upravo to zahtijeva dostupni algoritam raspoređivanja.

Drugi razlog je nedovoljna količina podataka. Kao što je već rečeno u prethodnom poglavlju, za problem raspoređivanja dostupan je relativno mali skup podataka nad kojim se izvodi složena programska logika, tijekom evaluacije rješenja generiraju se novi podatci i takav postupak se ponavlja veliki broj puta. Ti novi podatci su također mali, ali se svejedno moraju slati između glavnog programa i kernela, što zahtijeva uspostavljanje komunikacije i kopiranje preko sabirnice. Zbog velike količine *overheada* cijeli proces nije isplativ i više se isplati implementirati klasičnu višedretvenost na CPU.

Koliki je onda doprinos ovog diplomskog rada? Živimo u vremenu kada se svakodnevno generiraju petabajti podataka. Brojni važni problemi u raznim granama znanosti zahtijevaju obradu velikih količina podataka kako bi se pronašlo rješenje koje može dovesti do novih otkrića i unaprijediti istraživanja.

U zadnjih nekoliko godina većina programskih okvira ovakve namjene uvodi podršku za izvođenje na GPU. Nova postignuća u razvoju masovno paralelnih arhitektura omogućila su čitav niz novih otkrića u znanosti (napredne vizualizacije, pojava dubokih modela, itd.). Iz preostalih rezultata u prethodnom poglavlju možemo vidjeti da GPU implementacije daju ubrzanja od nekoliko redova veličine. To najviše dolazi do izražaja u problemu simboličke regresije funkcija, gdje je kod velikih problema toliko ubrzanje presudna razlika između uspješnog pronalaska rješenja i nemogućnosti pristupanja problemu.

Osim vremena koje je potrebno za izračun, bitan faktor je i ekonomski trošak. U današnje vrijeme implementacije algoritama koje se izvode na GPU u nekim problemima daju performanse usporedive s nekoliko puta skupljim klasterima računala. Iako su takve konfiguracije godinama služile svrsi, danas je njihov najčešći problem ograničena skalabilnost. Uz to se javlja i problem velike potrošnje električne energije. Iz tih razloga brojna starija superračunala zamjenjuju se novim efikasnijim i kompaktnijim arhitekturama u kojima je glavni izvor računske snage upravo veliki broj grafičkih procesora.

Iz navedenih razloga možemo zaključiti da ova implementacija omogućuje programskom okviru ECF da se i u narednim godinama uspješno nosi s novim izazovima.

LITERATURA

ECF - Evolutionary Computation Framework. http://ecf.zemris.fer.hr/.

John Cheng, Max Grossman, i Ty McKercher. *Professional CUDA C Programming*. Wrox Press Ltd., Birmingham, UK, UK, 1. izdanju, 2014. ISBN 1118739329, 9781118739327.

Marko Čupić. Prirodom inspirirani optimizacijski algoritmi, metaheuristike, prosinac 2013. URL http://java.zemris.fer.hr/nastava/pioa/.

Vipul K. Dabhi i Sanjay Chaudhary. Solution modeling using postfix genetic programming. *Cybernetics and Systems*, 46(8):605–640, 2015. doi: 10.1080/01969722.2015.1058662. URL https://doi.org/10.1080/01969722.2015.1058662.

Mike Giles. Course on CUDA Programming on NVIDIA GPUs. https://people.maths.ox.ac.uk/gilesm/cuda/, 2017.

Domagoj Jakobović. *Raspoređivanje zasnovano na prilagodljivim pravilima*. Doktorska disertacija, Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva, 2005.

William B. Langdon. A many threaded cuda interpreter for genetic programming. U Anna Isabel Esparcia-Alcázar, Anikó Ekárt, Sara Silva, Stephen Dignum, i A. Şima Uyar, urednici, *Genetic Programming*, stranice 146–158, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-12148-7.

NVIDIA Corporation. Nvidia CUDA C Programming Guide, svibanj 2018. URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

Riccardo Poli, William B. Langdon, i Nicholas Freitag McPhee. A field guide to genetic programming, 2008. URL http://www.gp-field-guide.org.uk. (With contributions by J. R. Koza).

Marko Đurasević. Optimizacija raspoređivanja u okruženju nesrodnih strojeva. Magistarski rad, Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva, 2014.

Optimizacija prioritetnog raspoređivanja metodama strojnog učenja ostvarena na grafičkom procesoru

Sažetak

U ovom radu opisan je problem optimizacije prioritetnog raspoređivanja u okolini nesrodnih strojeva. Opisane su metode strojnog učenja koje se koriste u generiranju rasporeda, genetsko programiranje i neuronska mreža. Opisana je arhitektura grafičkog procesora i tehnologija CUDA. Opisane su tehnike paralelizacije evaluacije rješenja navedenih metoda. Implementirana je programska podrška za izvođenje navedenih metoda na grafičkom procesoru u sklopu programskog okvira ECF. Izmjereni su rezultati ubrzanja i komentirani problemi koji se javljaju.

Ključne riječi: prioritetno raspoređivanje, CUDA, GPU, optimizacija, neuronske mreže, genetsko programiranje, nesrodni strojevi

Optimization of priority scheduling using machine learning on GPU

Abstract

This thesis describes the problem of priority scheduling optimization in the environment of unrelated machines. It also describes the machine learning methods, genetic programing and neural networks used for generating schedules. Furthermore, this thesis describes the architecture of a graphics processor and the CUDA technology, as well as the paralelization techniques that are used for the solution evaluation of these methods. Software support for GPU execution is implemented within the ECF framework. Finally, the thesis reports on speedup measures and comments on the problems that arise.

Keywords: priority scheduling, CUDA, GPU, optimization, neural networks, genetic programming, unrelated machines