UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Blaž Grebenšek

Splošnonamensko programiranje grafičnih naprav za potrebe strojnega učenja

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

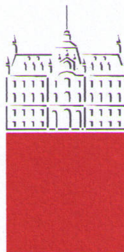
Mentor: doc. dr. Matjaž Kukar

Ljubljana, 2010

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si



Št. naloge: 01594/2009

Datum: 15.10.2009

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **BLAŽ GREBENŠEK**

Naslov: **SPLOŠNONAMENSKO PROGRAMIRANJE GRAFIČNIH NAPRAV ZA POTREBE STROJNEGA UČENJA**

GENERAL-PURPOSE PROGRAMMING OF GRAPHICAL DEVICES FOR MACHINE LEARNING

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Kandidat naj v diplomskem delu preuči področje splošnonamenskega programiranja grafičnih naprav (GPGPU). Opiše naj strojne in programske zahteve, ki jih zahteva uporaba GPGPU. Oriše naj programski model in nove tehnike programiranja, ki jih le-ta zahteva. V kontekstu strojnega učenja naj kandidat definira področja, ki so še posebej primerna za GPGPU implementacijo. S praktičnim primerom GPGPU implementacije nevronske mreže naj kandidat ilustrira opisane tehnike, pospešitve izvajanja pa naj prikaže tudi v primerjavi s konvencionalno CPU implementacijo.

Mentor:

doc. dr. Matjaž Kukar

Dekan:

prof. dr. Franc Solina

Rezultati diplomskega dela so intelektualna lastnina Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani **Blaž Grebenšek**, z vpisno številko **63030163**, sem avtor diplomskega dela z naslovom: **Splošnonamensko programiranje grafičnih naprav za potrebe strojnega učenja**.

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Matjaža Kukarja,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 15.1.2010 Podpis avtorja: ____________________

Zahvala

Zahvaljujem se doc. dr. Matjažu Kukarju za mentorstvo, pomoč in usmerjanje med izdelavo tega diplomskega dela. Prav tako bi se posebej zahvalil bratrancu Andreju za omogočanje testiranj na hitrejšem sistemu in družini, ki me je podpirala in omogočila študij.

Vsem, ki so kakorkoli pomagali.

Kazalo

Povzetek **1**

Abstract **2**

1 Uvod **3**

1.1 Pregled obstoječih programskih vmesnikov za programiranje grafičnih naprav 4

1.2 Pregled obstoječih implementacij nevronskih mrež za grafične naprave 5

1.3 Smernice za prihodnost 6

2 CUDA **15**

2.1 Strojna implementacija 16

2.2 Programski model 18

2.2.1 Jedra 18

2.2.2 Hiearhija niti 19

2.2.3 Hiearhija pomnilnika 21

2.2.4 Gostitelj in naprava 22

2.3 Programski vmesnik 22

2.3.1 Prevajanje s prevajalnikom nvcc 22

2.3.2 Glavni in deljeni pomnilnik 25

2.3.3 Teksturni pomnilnik 25

2.3.4 Delo z napakami 27

2.3.5 Razhroščevanje z uporabo emulacijskega načina 27

3 Umetne nevronske mreže **29**

3.1 Nevron in model nevrona 29

3.2 Vrste aktivacijskih funkcij 32

3.3 Lastnosti nevronskih mrež 33

3.4 Topologija nevronske mreže 35

3.5 Vrste učenja 35
3.6 Vrste nevronskih mrež 36
3.7 Enonivojski perceptron 37
3.8 Večnivojski perceptron 39

4 Implementacija nevronske mreže na GPE 46
4.1 Razlogi za implementacijo na GPE 46
4.2 Priprava zbirke podatkov 48
4.3 Način obdelave podatkov 50
4.4 Način dostopa do podatkov 50
4.4.1 Dostop do globalnega pomnilnika 50
4.4.2 Dostop do deljenega pomnilnika 55
4.5 Hitrost prenosa med pomnilniki 55
4.5.1 Seštevanje sprememb uteži 56
4.5.2 Uporaba bralnih pomnilnikov 57
4.6 Povečevanje zasedenosti jeder 58
4.7 Primerjava implementacije algoritma za CPE in GPE 61
4.8 Hitrejša implementacija algoritma 62

5 Rezultati poskusov 65
5.1 Lastnosti in uspešnost nevronske mreže 65
5.2 Merjenje časa 67
5.3 Časovna porazdelitev izvajanja jeder 68
5.4 Pohitritev GPE v primerjavi s CPE 71
5.5 Najpomembnejše optimizacije 73

6 Sklepne ugotovitve 75

A Namestitev orodja CUDA 76
A.1 Windows 7 Professional 76
A.2 Mac OS X 10.6 77

B Opis podatkovne zbirke 79

Seznam slik 81
Seznam tabel 83

Literatura 84

Seznam uporabljenih kratic in simbolov

ALE - Aritmetično Logična Enota

AoS - Array of Structures

API - Application Programming Interface (programski vmesnik)

B - Bajt

CPE - Centralno Procesna Enota

CS - Compute Shader

CTM - Close To Metal

CUDA - Compute Unified Device Architecture

ECC - Error Correcting Code

GLSL - OpenGL Shading Language

GPE - Grafična Procesna Enota

HLSL - High Level Shader Language

ILP - Instruction-Level Parallelism

ISA - Instruction Set Architecture

JIT - Just In Time

KB - Kilo Bajt

MB - Mega Bajt

OpenCL - Open Computing Language

PE - Procesni Element

PTX - Parallel Thread Execution (vzporedno izvajanje niti)

RBFN - Radial Basis Function Network

SDK - Source Development Kit (razvojno orodje)

SFU - Special Function Unit

SIMD - Single Instruction Multiple Data

SIMT - Single Instruction Multiple Thread

SoA - Structure of Arrays

SP - Scalar Processor core

VPE - Vektorska Procesna Enota

VS - Visual Studio

XOR - eXclusive OR (ekskluzivni OR)

Povzetek

Potrebe zabavne industrije na področju osebnih računalnikov zahtevajo vedno bolj realistične prikaze iger. V ta namen proizvajalci izdelujejo zmogljivejše grafične naprave, ki so po računskih zmogljivostih že zdavnaj prehitele centralne procesne enote (CPE). Takšno zmogljivost izračunov dosegajo z mnogimi preprostimi enotami, ki sicer ne vsebujejo dobre predikcije skokov, temveč posledice le-teh odpravljajo z ogromnim številom aktivnih niti.

V diplomski nalogi izkoristim to prednost grafičnih naprav in jih uporabim za splošnonamensko programiranje za potrebe strojnega učenja. Najpomembnejša koraka pri tem sta prava izbira računske arhitekture in algoritma. Odločil sem se za CUDA arhitekturo, na kateri sem kot vzorčno metodo implementiral algoritem vzvratnega razširjanja napake iz področja umetnih nevronskih mrež. Pri izvedbi na grafični napravi je podanih in uporabljenih več različnih optimizacijskih pristopov, s katerimi dosegam hitrejše izvajanje.

Cilj diplomske naloge je čim učinkovitejša oz. hitrejša izvedba konkretnega učnega algoritma na grafični napravi in posledično čim večja pohitritev v primerjavi s CPE izvedbo. Na danes najhitrejših grafičnih napravah sem tako dosegal tudi več kot 50-kratno pohitritev v primerjavi z zelo hitro CPE.

Ključne besede:

- CUDA
- splošnonamensko programiranje grafičnih naprav
- algoritem vzvratnega razširjanja napake
- pohitritev

Abstract

The needs of entertainment industry in the field of personal computers always require more realistic impressions of games. For this purpose manufacturers produce more powerful graphical devices, of which the compute capabilities have long overtaken central processing units (CPU). Such compute capabilities are achived through the capacity of many simple units, which do not contain good branching predictions, but better deal with the consequences through a huge number of active threads.

In this diploma thesis I use the advantage of graphical devices for general-purpose programming for the needs of machine learning. The most important steps are the right choice of compute architecture and algorithm. I have chosen CUDA architecture, where I implemented backpropagation algorithm as a sampling method in an artificial neural networks area. In graphical device implementation I used several different optimization approaches to achieve more rapid execution.

The purpose of thesis is to achieve as effective and fast concrete learning algorithm implementation on graphical device and thus to maximize speed up compared to the CPU implementation. At present-day fastest graphical devices I achieved more than 50-times speed up compared to the CPU implementation.

Key words:

- CUDA
- general-purpose programming of graphical devices
- backpropagation algorithm
- speed up

Poglavje 1

Uvod

Strojno učenje je eno izmed področij umetne inteligence, ki se uporablja za analizo podatkov, odkrivanje zakonitosti v podatkovnih bazah, ekspertne sisteme, razpoznavanje slik, klasifikacijo, regresijo ipd. Z metodami strojnega učenja namreč poskušamo avtomatsko opisati zakonitosti oz. lastnosti podatkov, iz katerih se nato generirajo neka pravila, sistemi enačb, relacije med njimi in podobno. Rezultate strojnega učenja se največkrat uporabi v obliki klasifikacijskih in regresijskih dreves, odločitvenih pravil in tudi v obliki nevronskih mrež, ki so uporabljene pri implementaciji v diplomski nalogi.

Pri umetnih nevronskih mrežah se pogosto srečujemo z masivno vzporednim in zato na običajnih arhitekturah časovno potratnim preračunavanjem, kjer sta učenje umetne nevronske mreže in klasifikacija novih primerov zelo zamudna postopka. Največ k temu prispeva ogromno število povezav v nevronski mreži in veliko število učnih primerov, ki so med seboj sicer neodvisni. Pri paketnem načinu preračunavanja nevronskih mrež je ta neodvisnost zelo uporabna, saj nam omogoča vzporedno obdelovanje več učnih primerov sočasno in je kot taka primerna za sočasno vzporedno obdelavo na več procesorjih.

Običajni štiri in dvoprocesorski sistemi so danes cenovno dokaj ugodni, vendar se za potrebe masivno vzporednega procesiranja najbolje obnesejo specializirane grafične naprave[1] (v nadaljevanju tudi le naprava). Vsebujejo namreč več deset oz. sto procesorjev in imajo veliko večjo prepustnost vodila kot CPE ter so zato zelo primerne naprave za vzporedno obdelovanje podatkov za potrebe strojnega učenja. Zaradi primernosti obdelovanja umetnih nevronskih mrež na grafičnih napravah smo se za cilj diplomske naloge odločili za vzporedno implementacijo algoritma vzratnega razširjanja napake na grafični

[1]Grafična naprava je v kontekstu diplomske naloge razumljena kot vse grafične kartice, mobilne naprave in ostale naprave, ki so zmožne izvajanja CUDA programov.

napravi in doseganje čim večje pohitritve (speed up) v primerjavi s CPE.

V diplomski nalogi je najprej podan kratek pregled obstoječih programskih vmesnikov za programiranje grafičnih naprav ter zatem pregled obstoječih implementacij nevronskih mrež na njih. Pred zaključkom poglavja 1 bom na hitro omenil še prihajajoče tehnologije, ki so v času pisanja še v razvojni fazi, vendar bodo zelo pomembne za splošnonamensko programiranje grafičnih naprav.

V poglavju 2 je podrobneje predstavljena strojna implementacija CUDA arhitekture, zatem njen programski model in programski vmesnik.

Umetne nevronske mreže so opisane v poglavju 3. Tam bom omenil, kako deluje nevron in njegov matematični model, katere aktivacijske funkcije se pri tem uporablja, lastnosti nevronskih mrež in njihove topologije, ter proti koncu poglavja eno in večnivojski perceptron, ki sta uporabljena pri implementaciji.

Poglavje 4 opisuje moji implementaciji nevronskih mrež za preračunavanje na grafični napravi. Podrobno so razložene vse uporabljene optimizacije ter vprašanje, zakaj seštevanje polj ni primerno opravilo za izvedbo na grafični napravi.

Zelo pomembno je poglavje 5, kjer so podani rezultati poskusov. Na začetku tega poglavja najprej podam uspešnost učenja realnega problema razpoznavanja črk in opišem merjenje časa posameznih izvedb. Zatem podrobneje primerjam čase izvajanja posameznih jeder na različnih napravah in nenazadnje podam pohitritev grafične naprave v primerjavi s CPE izvedbo.

Diplomsko nalogo bom sklenil s sklepnimi ugotovitvami v poglavju 6, v katerem ocenim rezultate in podam smernice za nadaljevanje.

1.1 Pregled obstoječih programskih vmesnikov za programiranje grafičnih naprav

Programiranje grafičnih naprav se je začelo okoli leta 2001 s prihodom prvih programabilnih grafičnih naprav Nvidia GeForce 3 [30], manj kot dve leti kasneje pa so nastali programski jeziki Cg, GLSL in HLSL v katerih so začeli nastajati prvi programi, ki niso namenjeni grafiki, izvajali pa so se na grafični napravi. Danes se ti programski jeziki za splošnonamensko programiranje grafičnih naprav množično ne uporabljajo več, saj so jih nadomestila boljša orodja, kot sta CUDA [26] in Stream SDK [1].

CUDA SDK, razvojno orodje podjetja Nvidia, je bilo prvič izdano leta 2006 in je do danes najpogosteje uporabljeno orodje. Približno istega leta je konkurenčno podjetje AMD izdalo CTM (Close To Metal), vendar ga kasneje niso več razvijali in so pričeli z orodjem Stream SDK, ki delno temelji na CTM.

Zaradi strojnih omejitev in (po poročanju mnogih uporabnikov) težav z grafičnimi gonilniki podjetja AMD znotraj linux sistemov, sem se pri izdelavi diplomske naloge odločil za uporabo razvojnega orodja CUDA SDK.

1.2 Pregled obstoječih implementacij nevronskih mrež za grafične naprave

Splošnonamensko programiranje grafičnih naprav je dokaj nov način programiranja in posledično je zanj zelo malo aplikacij za učenje nevronskih mrež na grafični napravi. Najstarejša najdena implementacija je napisana v programskem jeziku GLSL [24], namenjena bolj kot eksperiment, kaj se da narediti. Omenjena mreža ne omogoča učenja nevronske mreže, temveč le klasifikacijo primerov. Boljše različice so vse napisane v programskem jeziku C za CUDA in so opisane v nadaljevanju. Različici 1.2.2 in 1.2.3 sta se pojavili med pisanjem diplomskega dela in omogočata tudi učenje nevronske mreže na grafični napravi.

1.2.1 Razpoznavanje ročno napisanih števil

Ena izmed prvih javno dostopnih nevronskih mrež, v programskem jeziku C za CUDA, je bila mreža [5], ki razpoznava ročno napisana števila. Uporabljena je bila konvolucijska nevronska mreža, sestavljena iz petih nivojev nevronov, ki sprejme sliko v velikosti $29 * 29$ pikslov in po principu zmagovalne strategije izmed desetih izhodnih nevronov določi, katero število je napisano. Učni del v implementacijo ni vključen, tako da je to v bistvu preračunavanje, kjer se praktično nič ne zapisuje v globalni pomnilnik. Po avtorjevih trditvah je pri enem vhodnem vzorcu GPE različica okoli 270-krat hitrejša od CPE različice in 516-krat hitrejša od emulatorja. Če se meri še prenos podatkov, je faktor pohitritve okoli 10.

1.2.2 Multiple Back-Propagation Algorithm

Lopes N. [20] je v svojem programu "Multiple Back-Propagation Software" uporabil splošnejši algoritem vzvratnega razširanjanja napake, imenovan "Multiple Back-Propagation Algorithm". Programska oprema je prosto dostopna za učenje nevronskih mrež poljubnih razsežnosti z omenjenim algoritmom na CPE ali GPE. Po meritvah, podanih na portalu "CUDA ZONE" [6], je GPE

različica 40-krat hitrejša od CPE izvedbe, ki za povezave uporablja povezane sezname.

1.2.3 Učenje dvonivojskega perceptrona

Najboljša in najhitrejša implementacija nevronskih mrež, ki sem jih našel, je bila predstavljena na Elektrotehniški in računalniški konferenci ERK 2009 [32]. Program za učenje dvonivojskega perceptrona je napisan v programskem jeziku C. Kot navajajo v članku, je GPE izvedba tudi do 38-krat hitrejša od CPE izvedbe. Njihov matematični model namreč vsebuje več matričnih množenj, kjer je vzporednost na GPE mogoče učinkovito implementirati. Pri izračunih primerov se uporablja deljeni pomnilnik, ki najverjetneje zelo pripomore k pohitritvi, število prenosov med gostiteljem in napravo so minimizirali, koliko podatkov bo obdelala ena nit, pa je odvisno od velikosti nevronske mreže.

1.3 Smernice za prihodnost

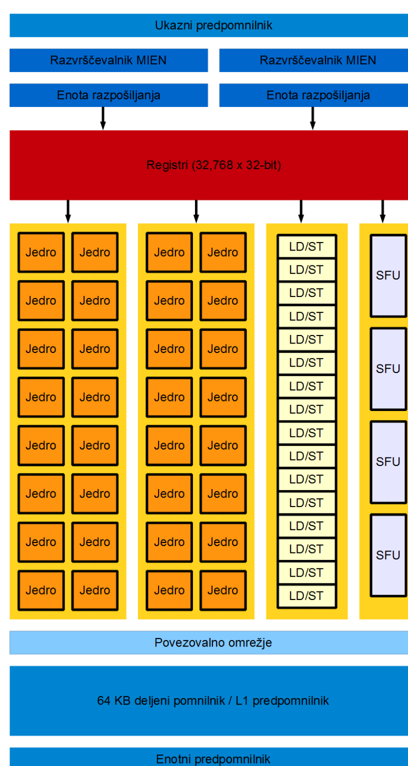
V času pisanja te diplomske naloge sta za splošnonamensko programiranje grafičnih naprav najbolj razširjena CUDA arhitektura in programski model [26]. Zaradi omejenosti na Nvidiine naprave, se bo to s prihodom OpenCL programskega jezika [17] verjetno spremenilo, saj se bodo programi napisani v OpenCL programskem jeziku, izvajali na grafičnih napravah različnih proizvajalcev in hkrati tudi na CPE-jih.

Iz stališča strojne opreme se največje novosti obetajo s prihodom Nvidiine arhitekture Fermi [27] in Intelove arhitekture Larrabee [33]. Sicer pa se za prihodnost pojavlja vedno več namigovanj, da naj bi se GPE in CPE združila v en fizični čip, kot se bo to zgodilo v primeru Intelove arhitekturne platforme Pine Trail [14], kjer bodo v CPE tudi GPE in pomnilniški krmilnik.

1.3.1 Fermi

Nvidia je javnosti na letošnji tehnološki konferenci o GPE prvič predstavila novo arhitekturo, poimenovano Fermi [27], ki prinaša zelo veliko novosti. Med najpomembnejše lahko štejemo tretjo generacijo tokovnih multiprocesorjev (streaming multiprocessor, v nadaljevanju multiprocesor) z več jedri, hitrejšim preračunavanjem v dvojni natančnosti, drugo generacijo arhitekture nabora ukazov za vzporedno izvanjanje niti (PTX ISA), napredejši pomnilniški podsistem in nov pogon NVIDIA GigaThread^{TM}.

Tretja generacija multiprocesorjev (slika 1.1) namesto dosedanjih 8 vsebuje



Slika 1.1: Shema multiprocesorja arhitekture Fermi.

32 procesorjev oz. CUDA jeder (CUDA cores), skupno torej kar 512 CUDA jeder, ki omogočajo preračunavanje števil v plavajoči vejici ali celoštevilsko preračunavanje in so porazdeljena v 16 multiprocesorjev. Med jedra za preračunavanje spada še 16 enot za izračun naslovov ter 4 enote za posebne funkcije (SFU), kot so sinus in kosinus. SFU je ločen od enote za razpošiljanje ukazov, kar omogoča zasedenost ostalih enot, medtem ko je SFU zaseden.

S prihodom Fermi arhitekture so uvedli tudi popolno heirarhično predpomnilniško zasnovo. Zunaj multiprocesorja se nahaja 768 KB velik L2 predpomnilnik, ki je dostopen vsem multiprocesorjem in je uporabljen za predpomnenje tekstur, branj in pisanj. Na multiprocesorju je 64 KB velik pomnilnik, v katerem sta sočasno L1 predpomnilnik in deljeni pomnilnik. Velikost obeh se določi programsko. Na voljo imamo konfiguraciji 16/48 KB oz. 48/16 KB, kjer je leva stran velikost deljenega pomnilnika in desna velikost L1 predpomnilnika.

V posodobljenem PTX ISA so poenotili operaciji `load` in `store` za različne naslovne prostore: lokalni naslovni prostor niti, deljeni pomnilnik znotraj blokov niti in naslovni prostor globalnega pomnilnika. Implementacija enotnega

naslovnega prostora je omogočila popolno podporo za programski jezik C++, kjer spremenljivke in funkcije obstajajo znotraj objektov, ki so nato podani preko referenc. Objekte se lahko podaja v katerikoli naslovni prostor, saj za nadaljno pretvorbo poskrbi strojna oprema.

Med zelo pomembne posodobitve zagotovo sodi popolna podpora v IEEE 754-2008 32-bitni in 64-bitni natančnosti, ki je v Fermiju do okoli 8-krat hitrejša kot na napravah tipa GT200. Pri napravah tipa GT200 je namreč vsak multiprocesor vseboval le eno enoto, zmožno računanja v dvojni natančnosti, Fermi arhitektura pa omogoča preračunavanje in izračun naslovov za 16 števil dvojne natančnosti v eni urini periodi na enem multiprocesorju. Iz opisanega sledi, da bodo naprave s Fermi arhitekturo v eni urini periodi teoretično do 8-krat hitreje preračunale števila v dvojni natančnosti od starejših naprav s 30 multiprocesorji.

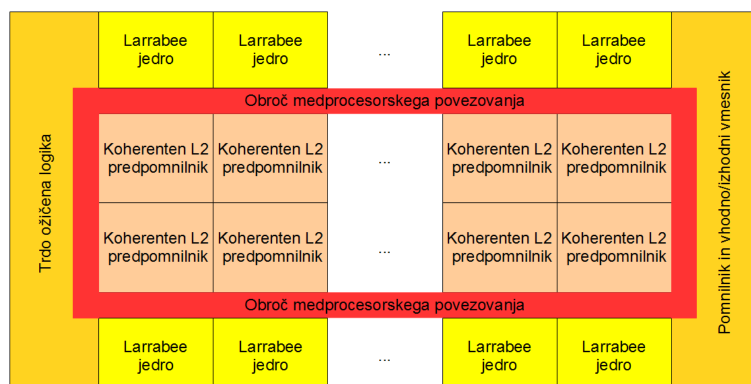
Ena izmed novosti je tudi NVIDIA GigaThread pogon, ki omogoča do desetkrat hitrejše preklope konteksta aplikacij in sočasno izvajanje različnih jeder. To pomeni, da se lahko sočasno izvaja 16 različnih jeder. Kot zadnjo pomembno izboljšavo naj omenim še odpravljanje enojih in zaznavanje dvojnih napak (ECC) v registrih, deljenemu pomnilniku, L1 in L2 predpomnilnikih in v globalnem pomnilniku.

Na konferenci [11] so predstavili še razvojno okolje Nexus, ki je narejeno za razvoj in razhroščevanje CUDA C, OpenCL in DirectCompute aplikacij. Nexus okolje je namenjeno za integracijo v razvojno okolje Microsoft Visual Studio (VS) in omogoča pisanje ter razhroščevanje programske kode za GPE z istimi orodji in vmesniki, kot za CPE, vključno s pregledom pomnilnika in dodajanjem prekinitvenih točk (breakpoint). Omogočeno je tudi, da se programer osredotoči na posamezno nit med tisočimi vzporedno se izvajajočimi nitmi in enostaven pregled izračunanih rezultatov.

1.3.2 Larrabee

Podjetje Intel pripravlja novo arhitekturo Larrabee [33] namenjeno za implementacijo v grafične naprave. Larrabee bo sestavljala množica procesorjev z razširjenim naborom ukazov x86. Vsak procesor bo podpiral večnitnost, kar bo povečalo zasedenost procesorja ob morebitnih zgrešitvah niti v predpomnilniku. Obogaten x86 nabor ukazov bo omogočal izvajanje programov in algoritmov, ki so se razvili že za obstoječe arhitekture, kar teoretično pomeni, da bodo jedra zmožna poganjanja tudi operacijskih sistemov.

Samo jedro (slika 1.3) sestavja množica registrov, L1 predpomnilnik, enoti za preračunavanje skalarjev ter vektorjev in dekodirnik ukazov (instruction decoder), ki razume razširjeni nabor x86 ukazov. Vsak procesor ima še dostop



Slika 1.2: Shema arhitekture Larrabee.

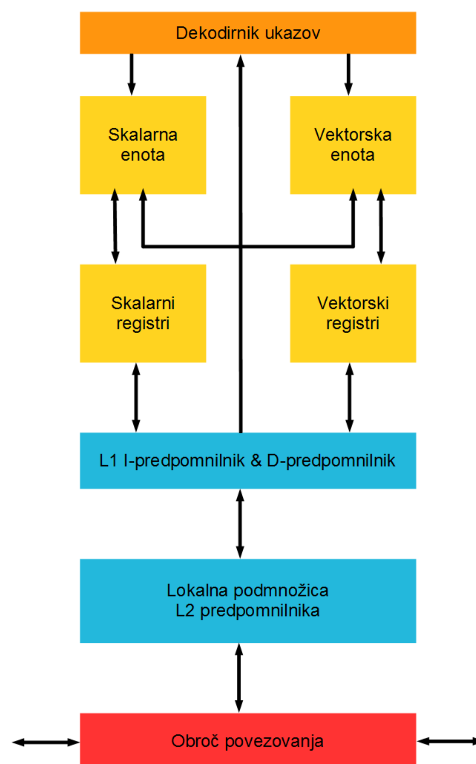
do L2 predpomnilnika, ki je ločen od L2 predpomnilnikov ostalih procesorjev. Skoraj tretjino sestave jedra zaseda vektorska procesna enota (VPE). V njej se izvaja celoštevilsko preračunavanje in preračunavanje v enojni in dvojni natančnosti.

Sporazumevanje med jedri poteka po t. i. dvosmernem obroču (slika 1.2). Širina podatkovnega vodila obroča je 512 bitna, smer je določena s sodostjo oz. lihostjo urine periode. Ker vsako jedro uporablja svoj L2 predpomnilnik, je potrebno preveriti konsistentnost podatkov pri dodajanju novih zapisov v ta predpomnilnik, kar se tudi izvede preko obroča.

Kljub temu da jedra razumejo x86 ukaze, obstoječe aplikacije verjetno ne bi v polnosti izrabljale procesorske moči grafične naprave. V času pisanja diplomske naloge so bile aplikacije, ki so bolje izkoriščale napravo, napisane v zbirniku. Na Intelovi spletni strani sicer že obstaja prototip C++ knjižnice, za kar pa še ni jasno, če bo v takšni obliki tudi v končni verziji.

1.3.3 OpenCL

Na predlog podjetja Apple se je leta 2008 osnoval nov odprt standard imenovan OpenCL [17]. Namenjen je vzporednemu programiranju aplikacij, ki se lahko izvajajo na raznih procesorjih, strežnikih, vgrajenih in drugih napravah, ki so skladne s standardom OpenCL. Vsako izmed računskih naprav obravnava kot soležnik (peer), ki je zmožen izvajanja aplikacije oz. preračunavanja dela programske kode. Standard ni omejen na specifičen operacijski sistem in kot tak le predpisuje minimalni nabor funkcij, ki jih implementacija standarda mora podpirati. Skrbstvo nad standardom je prevzela skupina Khronos Group, ki je znana po podobnih že uveljavljenih standardih, kot so OpenGL, OpenAL, OpenMAX ipd.



Slika 1.3: Zgradba jedra Larrabee.

Standard OpenCL:

- podpira vzporedni podatkovni in opravilni programski model,
- uporablja podmnožico ISO C99 programskega jezika z dodatki za vzporednost,
- definira numerično združljivost s standardom IEEE 754,
- definira konfiguracijski profil za ročne in vgrajene naprave,
- se učinkovito povezuje z OpenGL, OpenGL ES in drugimi grafičnimi programskimi vmesniki.

V času pisanja diplomske naloge so izšli prvič javnosti dostopni OpenCL prevajalniki in gonilniki za razne naprave. Nvidia kljub izidu standarda zagotavlja, da razvoja jezika C za CUDA ne bo opustila. S prihodom tehnologije Fermi bo le-ta omogočala dvojno natančnost in popolno podporo programskemu jeziku C++. OpenCL standard po drugi strani na tem področju še

CUDA	**OpenCL**	**CUDA**	**OpenCL**
nit	delovni element	deljeni pomnilnik	lokalni pomnilnik
blok niti	delovna skupina	skalarni procesor	procesni element
`gridDim`, `blockIdx`	globalni ID	sinhronizacija	prepreka
`blockDim`, `threadIdx`	lokalni ID	`__device__` `__global__`	`__kernel`

Tabela 1.1: Tabela sopomenk elementov v jeziku OpenCL in CUDA.

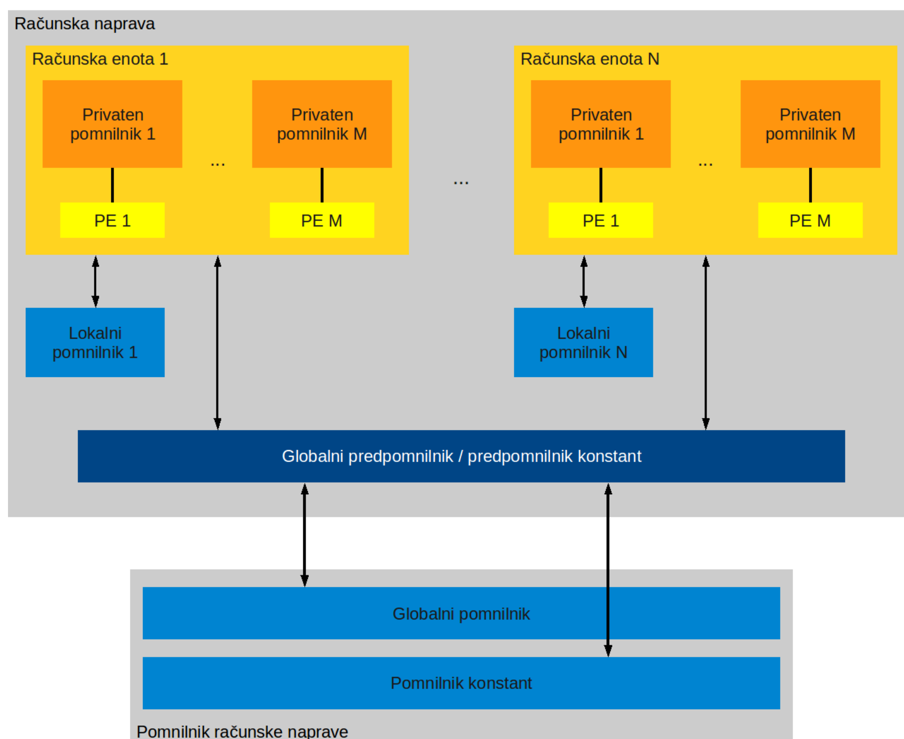
nima podpore rekurziji in dvojni natančnosti. Zadnja je omenjena le kot možen dodatek, če jo naprave podpirajo, in bo v prihodnjih različicah standarda najverjetneje obvezna.

Danes najpogostejši napravi, skladni s tem standardom, sta GPE in CPE. V primerjavi z razvojnim orodjem CUDA je programski jezik OpenCL po sintaksi najbolj podoben C za CUDA driver API vmesniku, ki je podrobneje pojasnen v poglavju 2.3. Zaradi novega poimenovanja elementov so primerjave s C za CUDA jezikom zajete v tabeli 1.1. Sam model OpenCL standarda je sicer zelo abstrakten in zato ni vezan na strojno zasnove naprave, podrobneje pa ga ločimo na model platforme, izvajanja, pomnilnika in programiranja.

Model platforme je sestavljen iz gostitelja (host), ki ima povezano vsaj eno OpenCL napravo, razdeljeno v eno ali več računskih enot, ta pa dalje v eno ali več procesnih elementov (PE). Samo preračunavanje se izvaja znotraj PE. V primerjavi s CUDA arhitekturo bi lahko rekli, da OpenCL napravo predstavlja grafična naprava, računsko enoto multiprocesor, procesni element pa predstavlja skalarni procesor.

Izvajanje OpenCL programov je opisano z modelom izvajanja in poteka v dveh delih. Gostiteljski program se izvaja na gostitelju in definira kontekst za izvajanje jeder ter upravlja z njihovim izvajanjem. Jedra (kernels) se nato izvajajo na OpenCL napravah. Bistvo OpenCL modela izvajanja je v definiranju, kako se izvedejo posamezna jedra. Določiti je potrebno število delovnih enot (work-item) znotraj delovnih skupin (work-groups) in število delovnih skupin.

Model pomnilnika je v OpenCL zelo podoben modelu v CUDA arhitekturi. Na voljo imamo globalni, lokalni, privatni pomnilnik in pomnilnik konstant. Dostopi do vseh so enaki kot pri CUDA arhitekturi, kar prikazuje tabela 2.1, lokacijo, kje se nahajajo, pa slika 1.4. Razlikuje se še v predpomnenju lokalnega



Slika 1.4: Konceptualna arhitektura OpenCL naprav.

in globalnega pomnilnika, kar je odvisno od zmogljivosti naprave.

Pri modelu programiranja sta podprta dva: vzporedni podatkovni model in vzporedni opravilni model. Najpomembnejši je prvi, kjer vsak delovni element deluje na nekem svojem podatku, medtem ko pri vzporednem opravilnem modelu definiramo le en delovni element in se vzporednost izkorišča z drugimi tehnikami vzporednosti.

Abstraktnost OpenCL programskega jezika omogoča neodvisnost od nabora ukazov za posamezno napravo. Združljivost programske kode z različnimi napravami se rešuje s pomočjo "just in time"(JIT) prevajalnikov, ki izvorno programsko kodo prevedejo ob zagonu programa. Za optimalno izvedbo nekega programa bo zato zaenkrat še vedno potrebno poznavanje dejanske strojne zasnove naprave. Izvajanje na trenutnih Nvidinih grafičnih napravah je tako močno odvisno od vezanega načina dostopa, konfliktov bank, števila niti za prekrivanje branja po pisanju v pomnilnik ipd. Programska koda, optimizirana za grafično napravo podjetja Nvidia, se verjetno ne bi optimalno izvajala tudi na napravah konkurenčnih proizvajalcev.

1.3.4 DirectCompute

Podjetje Microsoft je leta 1995 predstavilo nov programski vmesnik za delo z grafiko imenovan DirectX [21]. Sprva je bil namenjen le za igre, a se je sčasoma dopolnjeval in tako sedaj vsebuje še Direct3D, Direct2D, DirectWrite, DirectMusic, DirectPlay, DirectSound ipd. Del te kolekcije je tudi DirectCompute.

DirectCompute je API programski vmesnik, ki omogoča vzporedno programiranje aplikacij in izvajanje le-teh na grafičnih napravah. S tem se izkorišča številčnost grafičnih procesorjev in pospeši izvajanje programov. Implementiran bo šele v DirectX 11, vendar bo pri posodobljenih gonilnikih grafičnih naprav tekel tudi na napravah, ki so združljive z različico 10. Deloval naj bi na vseh napravah večjih proizvajalcev, a le v operacijskih sistemih Microsoft Windows.

Programski model, ki ga DirectCompute uporablja, je t. i. compute shader (CS) [22]. CS je nov podatkovno-vzporedni programski model, ki je v bistvu implementiran kot manjša sprememba HLSL standarda. Omogoča nadgradljivo (scalable) vzporednost brez zapletenih sprememb v programski kodi in enostavno razširljivost na več jeder.

Grafični cevovod DirectX-a je posodobljen tako, da v primeru uporabe CS-ja v stopnji točkovnega senčilnika (pixel shader) odda podatke splošni podatkovni strukturi, s katero nato upravlja CS.

Temeljne značilnosti CS:

- za sprožitev niti nam ni več potrebno "risati kvadrata", da bi se neka operacija v točkovnem senčilniku izvedla, temveč imamo sedaj na voljo eno, dvo in trorazsežna polja niti;
- registri so deljeni med nitmi in zato z njimi lahko eliminiramo nepotrebne izračune ter vhodno-izhodne dostope;
- razpršeno pisanje omogoča branje oz. pisanje v poljubne podatkovne strukture (drevesa, polja ...) in posledično nove vrste algoritmov.

Kot že omenjeno za sprožitev niti ni več potrebno klicati funkcij "draw", temveč "dispatch", kjer pri klicu kot argument podamo število niti. Ko se niti zaženejo, vse dobijo ID, ki bazira na njihovem indeksu v polju in je v programski kodi dostopen kot sistemska spremenljivka. Ker so niti združene v podskupine, imajo vse na voljo še dve dodatni sistemski spremenljivki: indeks skupine, v kateri se nit nahaja, in katera nit je to znotraj te skupine. Deljenje registrov poteka le znotraj podskupine. V različici modela CS 5 je trenutno maksimalno število niti na skupino 1024.

V CS-ju se z viri (resources) posplošeno poimenuje razne medpomnilnike (buffers), teksture in konstante. Medpomnilniki in teksture so lahko bralni in pisalni. Posebni vrsti medpomnilnikov sta t. i. strukturirani medpomnilnik (structured buffer) in "append/consume" medpomnilnik. Zadnji je zelo podoben tokovom (streams), saj lahko le dodajamo vrednosti oz. jih iz njih le pobiramo, medtem ko strukturirani medpomnilnik omogoča hranjenje štirih vrednosti v eni strukturi in je le bralni, tako kot konstante.

Pri uporabi CS modela imamo na voljo tudi deljeni pomnilnik, ki je deljen le med nitmi znotraj skupine. Ta je zaenkrat lahko le predznačenega in nepredznačenega celoštevilskega tipa. Med zanimivejšimi številskimi implementacijami velja omeniti še podporo dvojni natančnosti. Dvojna natančnost bo opcijsko podprta v različici 11 in ne bo na voljo kot vhod oz. izhod v funkcije, možno bo le interno preračunavanje in nato pretvorba ter shranjevanje v UINT2 tip.

Poglavje 2

CUDA

Arhitektura CUDA [26] je namenjena splošnonamenskemu izvajanju programov na grafičnih napravah. Podjetje NVIDIA jo je prvič predstavilo leta 2006 kot nov programski model za vzporedno računanje z novo arhitekturo nabora ukazov (Instruction Set Architecture).

CUDA je primerna predvsem za programe, ki potrebujejo intenzivno računsko moč in je zanje primerno izvajanje na arhitekturi tipa SIMD. V trenutnih verzijah (do vključno 2.3) je uradno podprt visokonivojski programski jezik C, v prihodnosti pa bo podprto tudi pisanje v OpenCL, DirectX Compute ter v FORTRAN-u. S pojavom večjedernih CPE se je razvoj iz povečevanja hitrosti v veliki meri usmeril predvsem v povečanje števila jeder, s čimer tako še vedno velja t. i. Moorov zakon[1].

Izziv za programerja pri tem je napisati aplikacijo, ki bo transparentno uporabljala kar največ jeder hkrati, podobno kot 3D grafične aplikacije transparentno uporabljajo mnogo jedrnost grafičnih naprav.

Zavedati se je potrebno treh ključnih vrst abstrakcije:

- hiearhije gruč niti,
- deljenega pomnilnika,
- sinhronizacijskih problemov.

Upoštevanje teh abstrakcij omogoča pisanje programov, ki se izvajajo z visoko stopnjo vzporednosti. Programerja usmerjajo v razdeljevanje problemov v neodvisne podprobleme, ki se izvajajo vzporedno, ter nato v podrobnejše

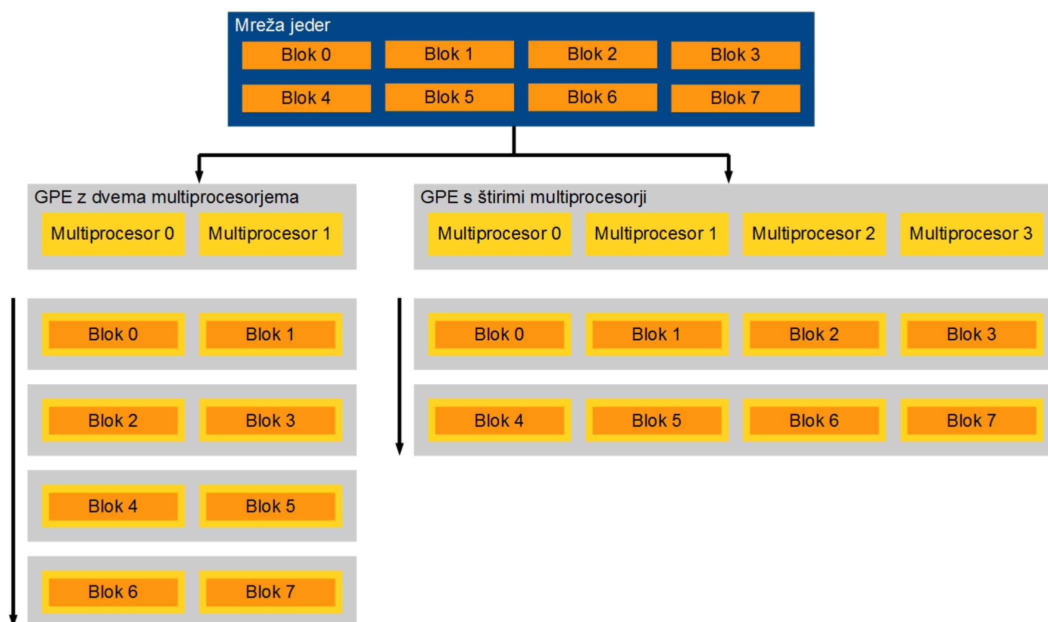
[1] Gordon Moore je leta 1965 napovedal, da se bo število tranzistorjev na čipu podvojilo na skoraj dve leti. Zaenkrat ta zakon še vedno velja in se pričakuje, da bo tako vsaj še kakšno desetletje.

probleme, ki se lahko rešijo vzporedno s sodelovanjem. Takšna dekompozicija problemov omogoča transparentno skaliranje reševanja problemov na poljubno veliko število jeder.

Za razumevanje delovanja programov na CUDA arhitekturi je zato pomembno razumevanje strojne implementacije ter predvsem konceptov programskega vmesnika in modela.

2.1 Strojna implementacija

CUDA arhitektura je zgrajena okoli večnitnih tokovnih multiprocesorjev (multithreaded Streaming Multiprocessors). Ko program, ki teče na CPE, zažene funkcijo, ki se bo izvajala na GPE (te imenujemo jedra), se vsi bloki niti (thread blocks, v nadaljevanju bloki) na GPE oštevilčijo in se razpošljejo na ustrezne multiprocesorje, kot to prikazuje slika 2.1. Niti, ki so znotraj blokov, se izvajajo sočasno na enem multiprocesorju. Ko se blok konča, se prične izvajanje naslednjega.



Slika 2.1: Prikaz izvajanja blokov na napravah z dvema in štirimi multiprocesorji. Naprava z več multiprocesorji obdeluje več blokov hkrati.

Vsak multiprocesor je sestavljen iz osmih jeder skalarnih procesorjev (scalar processor cores), večnitno ukazovno enoto (instruction unit), dveh enot

za preračunavanje posebnih funkcij ter iz deljenega pomnilnika (shared memory). Vsak multiprocesor ustvari, upravlja, izvršuje ter razporeja izvajanje dodeljenih niti brez kakršnih koli režij (overhead). Na tem nivoju je tudi implementirana sinhronizacija niti z ukazom `__syncthreads()`, medtem ko strojne sinhronizacije med bloki ni, jo lahko programer realizira na programskem nivoju med izvrševanji jeder.

Najmanjša enota, ki se izvede, je izvajanje skupine 32-ih niti, kar sem poimenoval snop niti[2] (v nadaljevanju snop). Ta se izvede v štirih urinih periodah in se nadalje deli še na polovici. Za ustvarjanje, upravljanje ter razporejanje izvajanja snopa skrbi enota, imenovana single-instruction multiple-thread (SIMT). Izvajanje vseh niti v snopu se začne na istem naslovu programa, vendar je vsaka nit svobodna v smislu, da se lahko veji in izvaja neodvisno.

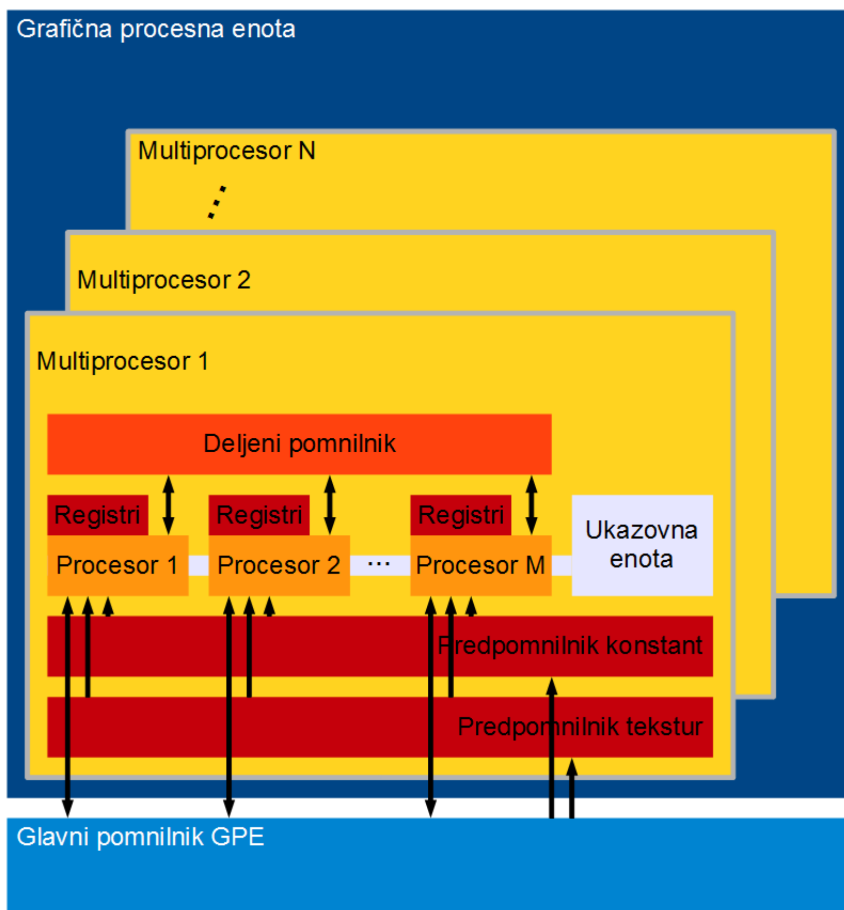
Ko multiprocesor dobi za izvedbo nekaj blokov, jih razdeli v snope, za katere nadalje skrbi SIMT. Ta enota vedno vzame snop, ki je pripravljen na izvajanje naslednjega ukaza. Zaradi SIMD zasnove sistema, se največi učinek doseže tedaj, ko vse niti izvajajo isti ukaz. Če bi pri izvajanju prišlo do vejitve, bi se izvedle različne poti izvajanja programa, kjer bi niti, pri katerih ni prišlo do vejitve, čakale, dokler se pot izvajanja ne bi nadaljevala združeno.

Vsak multiprocesor ima vgrajen pomnilnik štirih tipov (Slika 2.2):

- množico 32-bitnih registrov za vsak procesor;
- deljeni pomnilnik znotraj multiprocesorja;
- predpomnilnik konstant, s katerim se ob zadetkih zagotovi hitrejši dostop do pomnilnika konstant;
- predpomnilnik tekstur, s katerim se ob zadetkih zagotovi hitrejši dostop do pomnilnika tekstur.

Multiprocesor lahko obdeluje več blokov hkrati, vendar le, če ima na voljo dovolj prostih sistemskih sredstev. Vsak blok, ki se izvaja, potrebuje določeno število registrov in določeno velikost prostega deljenega pomnilnika. Če je teh sredstev dovolj za nekaj blokov, bo procesor obdeloval vse bloke hkrati. Te bloke imenujemo aktivni bloki. Če sredstev za procesiranje enega bloka ni dovolj, se jedro ne bo izvršilo.

[2] V angleščini je izraz za to warp. Termin izhaja iz prve vzporedne nitne tehnologije, imenovane weaving.



Slika 2.2: Shematični prikaz množice SIMT multiprocesorjev na čipu, s prikazanimi elementi, do katerih ima dostop posamezen procesor znotraj multiprocesorja.

2.2 Programski model

Programski model podaja osnovne vidike programiranja jeder ter razumevanje hiearhije niti in pomnilnika.

2.2.1 Jedra

Pisanje jeder poteka v programskem jeziku C z razširitvami, ki omogočajo programerju določitev, v katerem pomnilniku bodo shranjeni podatki in katera koda, imenovana jedro, se bo izvedla na GPE.

Jedro je funkcija, ki ima na začetku definirano `__global__` in jo izvede vsaka nit znotraj vsakega bloka. Vsaki niti je dodeljen unikaten indeks, ki je programerjem dosegljiv preko vgrajene spremenljivke `threadIdx`. V programski kodi, ki se izvaja na CPE, se klicu jedra doda posebne parametre `<<<...>>>` znotraj katerih se določi število niti, blokov, indeks podatkovnega toka in velikost dinamično dodeljenega pomnilnika. Primer poenostavljenega CUDA programa je prikazan v programu 1.

Program 1 CUDA program sešteje dve polji in vsoto shrani v prvega.

```
__global__ void function foobar(float* foo, float* bar) {
    int id = threadIdx.x;
    foo[id] = foo[id] + bar[id];
}

int main() {
    int threads = 32;
    int blocks = 64;
    foobar<<< blocks, threads >>>( fooArray, barArray );
    return 0;
}
```

2.2.2 Hiearhija niti

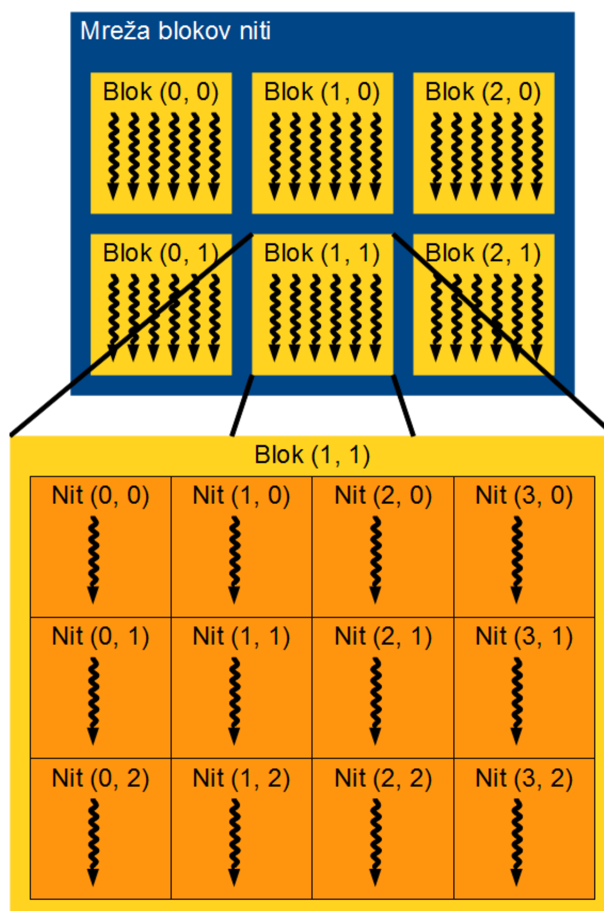
Zaradi priročnosti je vgrajena spremenljivka `threadIdx` vektor dolžine 3, s čimer označimo indeks niti (thread index) z enim, dvema ali tremi razsežnostmi. Na tak način dobimo eno, dvo ali trirazsežni blok. Več različnih razsežnosti označevanja bloka nam omogoča enostavnejše delo s samimi nitmi ter njenimi indeksi, kar se pogosto uporablja pri delu z matrikami in vektorji. Velikost blokov lahko znotraj programa izvemo iz vgrajene vektorske spremenljivke `blockDim`.

Indeks posamezne niti znotraj bloka je odvisen od njegove razsežnosti. V primeru enorazsežnega bloka je indeks določen kar z zaporedno številko niti, v primeru dvorazsežnega bloka velikosti (D_x, D_y) se indeks za nit (x, y) izračuna iz $(x + yD_x)$ in v primeru trorazsežnega bloka velikosti (D_x, D_y, D_z) se indeks za nit (x, y, z) izračuna iz $(x + yD_x + zD_xD_y)$.

Sodelovanje med nitmi je omogočeno preko deljenega pomnilnika. Na ta način si podatke lahko izmenjujejo samo niti znotraj istega bloka. Podobno

deluje tudi sinhronizacija niti. Ko je na vrsti ukaz za sinhronizacijo, niti znotraj bloka čakajo, dokler ne prispe do tega ukaza najpočasnejša nit.

Podobno, kot so niti razdeljene v več razsežnosti, so razdeljeni tudi bloki. Bloki so organizirani v eno, dvo ali trorazsežnost, imenovano mrežo blokov. Indeksi posameznih blokov se nahajajo v vektorski spremenljivki `blockIdx`, velikost mreže pa v vektorju `gridDim`. Dvorazsežnostno mrežo blokov prikazuje slika 2.3.



Slika 2.3: Mreža blokov niti.

V praksi je število blokov v mreži odvisno predvsem od velikosti podatkov, ki jih procesiramo. V primeru, da je blokov veliko in imamo na voljo poljubno število multiprocesorjev, se zahteva, da se lahko vsi bloki procesirajo neodvisno od vrstnega reda, in sicer vzporedno ali zaporedno. Ta neodvisnost omogoča blokom, da se razvrstijo na katerikoli skalarni procesor in na poljubno število

Pomnilnik	Lokacija na čipu	Pred-pomnjen	Način dostopa	Obseg	Življenjsko obdobje
Registri	Da	–	B/P[3]	1 nit	nit
Lokalni	Ne	Ne	B/P	1 nit	nit
Deljeni	Da	–	B/P	niti v bloku	blok
Globalen	Ne	Ne	B/P	vse niti+CPE	določi CPE
Konstantni	Ne	Da	B[4]	vse niti+CPE	določi CPE
Teksturni	Ne	Da	B	vse niti+CPE	določi CPE

Tabela 2.1: Tabela lastnosti pomnilnikov na grafični napravi.

le-teh. Programer tako ni omejen na določeno število multiprocesorjev in zato jih je lahko poljubno veliko.

2.2.3 Hiearhija pomnilnika

S stališča programskega modela obstaja na grafični napravi več različnih pomnilnikov:

- registri
- lokalni pomnilnik,
- deljeni pomnilnik,
- globalni pomnilnik,
- pomnilnik konstant,
- pomnilnik tekstur.

Med seboj se razlikujejo po načinu dostopa, fizični lokaciji na grafični napravi, predpomnjenju in življenskem obdobju (Tabela 2.1).

Vsaka nit ima zaseben lokalni pomnilnik. Vsak blok ima deljeni pomnilnik, ki je viden vsem nitim znotraj bloka in je dostopen, dokler se blok izvaja. Vsi bloki pa imajo dostop do istega globalnega pomnilnika. Obstajata tudi dva posebna pomnilniška naslovna prostora, iz katerih lahko samo beremo: konstantni in teksturni naslovni prostor. Fizično gledano lokalni, globalni in

[3]B/P - Bralno/Pisalni dostop

[4]B - Bralni dostop

teksturni naslovni prostori prebivajo znotraj globalnega pomnilnika, vendar se razlikujejo v načinih uporabe. Teksturni naslovni prostor tako poleg branja omogoča še različne načine naslavljanja, kot tudi načine filtriranja.

2.2.4 Gostitelj in naprava

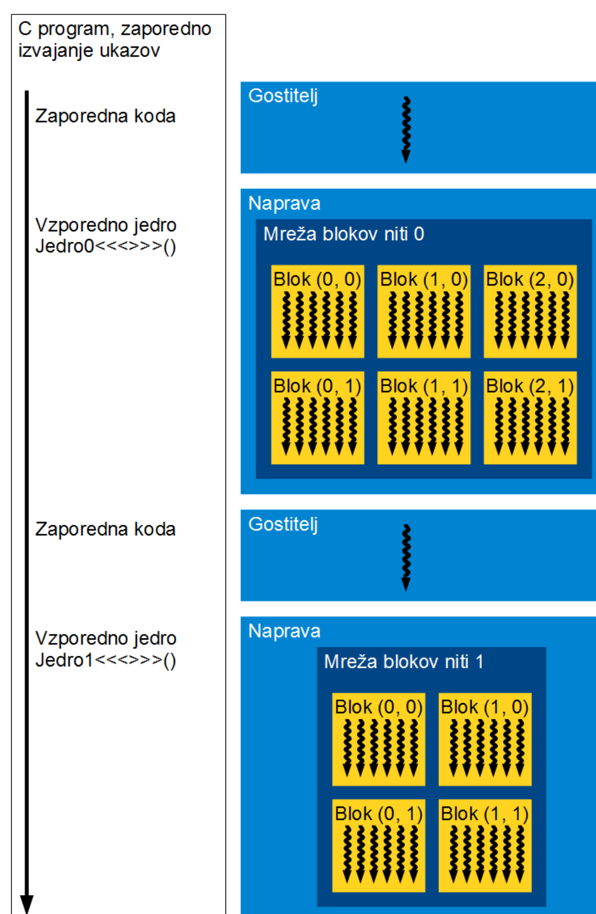
Pri izvajanju programa CUDA se grafična naprava (v nadaljevanju tudi naprava) obnaša kot dodaten koprocesor k CPE. Kot prikazuje slika 2.4 se program izvaja na gostitelju (host) do klica funkcije z argumenti `<<<...>>>`. Tedaj se generira mreža blokov in niti, kjer vsaka nit izvaja svojo funkcijo s svojim naslovnim prostorom. Ker so klici jedra asinhroni, CPE-ju ni potrebno čakati na končanje izvajanja GPE. CUDA programski model predpostavlja, da imata GPE in CPE lasten pomnilnik, in tako skrbi za dodeljevanje, sprostitev pomnilnika ter za prenose podatkov med gostiteljem in napravo.

2.3 Programski vmesnik

C za CUDA podpira dva vmesnika za pisanje programske kode: CUDA runtime API in CUDA driver API, ki se medsebojno izključujeta (mutual exclusive), zato lahko program uporablja le enega. CUDA driver API je nizkonivojski C API, ki vsebuje funkcije za nalaganje jeder kot module binarne ali asemblerske kode. To kodo se običajno dobi s prevajanjem jeder napisanih v C-ju. Oba API-ja vsebujeta funkcije za dodeljevanje in sproščanje pomnilnika na napravi, funkcije za prenose med pomnilnikoma naprave in gostiteljem, upravljata sisteme z več napravami . . . , medtem ko je runtime API zgrajen na driver API-ju in vsebuje še upravljanje s konteksti, moduli in inicializacijo. Program v driver API-ju je težje napisati in odpraviti napake, vendar omogoča boljši nadzor in je jezikovno neodvisen. Po drugi strani pa je program napisan v runtime API-ju bolj pregleden in uporabniku razumljivejši, na kar sem se osredotočil tudi v diplomski nalogi.

2.3.1 Prevajanje s prevajalnikom nvcc

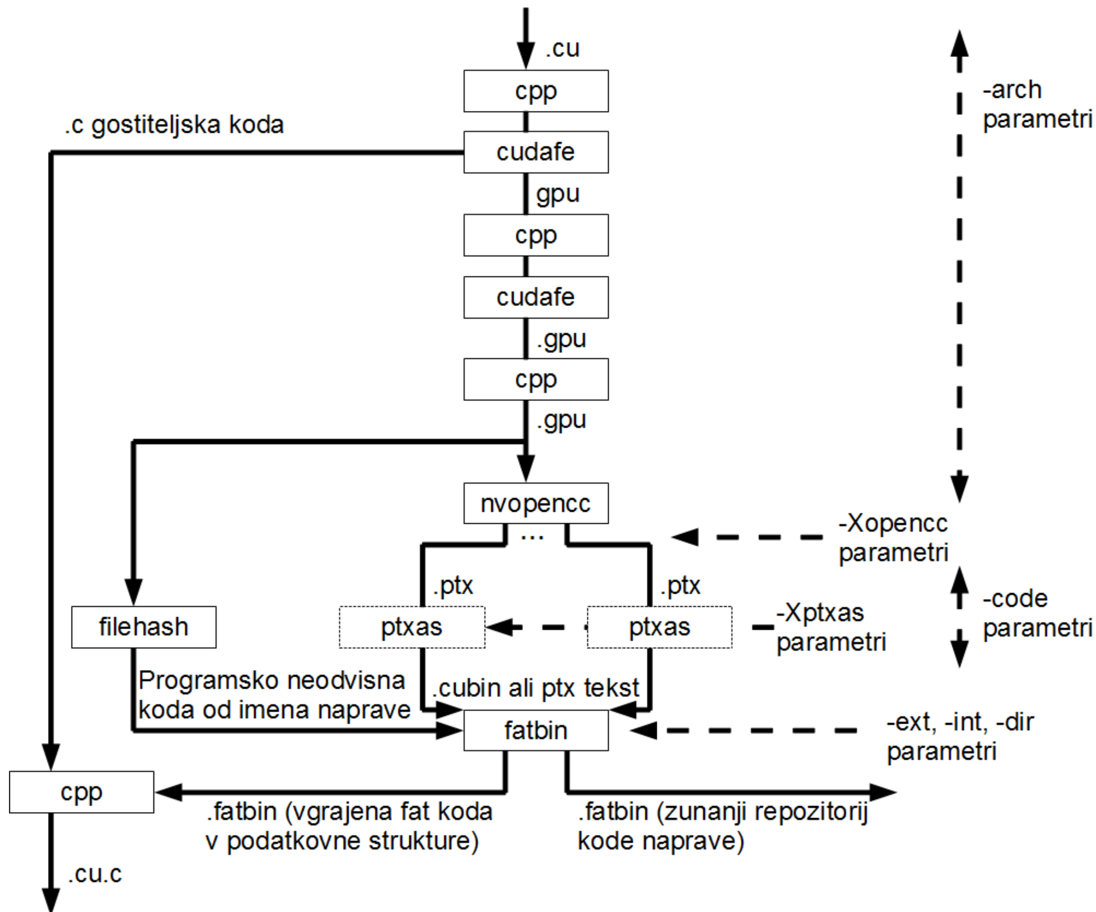
Kot že omenjeno, se programe piše v programskem jeziku C z razširitvami. Razširitve programskega jezika omogočajo programerju pisanje funkcij, ki se izvajajo na GPE, določanje, v katerem pomnilniškem prostoru se nahajajo podatki, in enostavno definiranje števila blokov in niti. Jedra se lahko napiše v arhitekturnem naboru ukazov imenovanem PTX (Parallel Thread Execution), vendar je običajno bolj učinkovito pisanje v visokonivojskem jeziku C.



Slika 2.4: Grafični prikaz izvajanja programske kode na gostitelju in napravi.

Vsa koda, kjer so uporabljene razširitve C-ja, se mora prevesti z NVIDIA prevajalnikom `nvcc` [29]. Prevajalnik vsebuje stopnje ločevanja, predprocesiranja, prevajanja in združevanja programske kode. Slednja se najprej loči na kodo, ki se izvaja na gostitelju in na napravi. Jedra za napravo se bodo prevedla v obliko PTX ali v binarno (cubin) obliko, preostanek kode, ki je namenjena izvajanju na gostitelju, pa se prevede z zunanjimi prevajalniki, kot sta `gcc`, `vc` . . . V zadnji stopnji prevajanja se prevedeno kodo lahko ponovno združi. Shemo prevajanja prikazuje slika 2.5.

Nekateri PTX ukazi so podprti samo pri napravah višje računske zmo-



Slika 2.5: Shema prevajanja programske kode s končnicami .cu

gljivosti (compute capability[5]). Pri napravah z računsko zmogljivostjo >1.1 so dodane atomske operacije, medtem ko naprave >1.3 že vsebujejo podporo dvojni natančnosti. V primeru, da programska koda vsebuje dvojno natančnost, jo je potrebno prevesti s posebnim stikalom `-arch sm_13`, sicer se bo vsa dvojna natančnost prevedla na enojno.

[5] NVIDIA vse svoje naprave označuje z glavno (major revision number) in s pomožno številko izdaje (minor revision number), ki sta ločeni s piko. Naprave, ki imajo isto veliko številko izdaje, so istega tipa arhitekture, medtem ko mala številka pomeni napredke oz. dodatke k osnovni arhitekturi ter s tem posledično večjo računsko zmogljivost.

2.3.2 Glavni in deljeni pomnilnik

Kot je bilo omenjeno v poglavju 2.2.4, imata gostitelj in naprava lasten pomnilnik. Jedra lahko neposredno berejo le iz pomnilnika naprave, kar pomeni, da mora runtime API poskrbeti za dostopnost do podatkov. Glavni pomnilnik je lahko dodeljen kot:

- **CUDA polje (CUDA array)** – je struktura, ki je optimizirana za branje iz teksturnega pomnilnika. Ta je podrobneje opisan v podpoglavju 2.3.3.

- **Linearno polje** – obstaja na napravi v 32-bitnem naslovnem prostoru. Entitete, ustvarjene v tem prostoru, lahko preko kazalcev kažejo na podobne entitete znotraj istega prostora. Tak primer bi bilo recimo binarno drevo.

Linearno polje se tipično dodeli z uporabo ukaza `cudaMalloc()` in sprosti s `cudaFree()`. Za prenos podatkov med različnimi pomnilniki se uporablja ukaz `cudaMemcpy()`, kjer smer kopiranja določimo z argumentom. S podobnimi ukazi se linearni pomnilnik lahko dodeli tudi za večrazsežna polja, kjer je že poskrbljeno za poravnanost elementov znotraj polja. To je zelo pomembno za doseganje večje učinkovitosti med branjem vrstic ali kopiranjem večrazsežnih polj na druge regije pomnilnika naprave.

Kadar je mogoče, se želimo izogniti uporabi glavnega pomnilnika in namesto večkratnega branja vrednosti le-tega shranimo v deljenega. Deljeni pomnilnik je namreč mnogo hitrejši od glavnega in je zato vsako nepotrebno dostopanje do glavnega priporočeno zamenjati z začasnim deljenim pomnilnikom.

Če želimo uporabljati deljeni pomnilnik, moramo pred deklaracijo polja definirati, da se to nahaja v deljenem pomnilniku. To storimo s kvalifikatorjem `__shared__` pred tipom polja. Program 2 prikazuje primer uporabe deljenega in glavnega pomnilnika.

2.3.3 Teksturni pomnilnik

CUDA podpira uporabo podmnožice strojne opreme, ki jo GPE uporablja za grafični dostop do teksturnega pomnilnika. Teksturni pomnilnik je del globalnega pomnilnika, v katerem se nahajajo teksture, ki so običajno dvorazsežno polje števil namenjenih za uporabo v računalniški grafiki in v katere ne moremo pisati s strani GPE. Uporaba tekstur omogoča več različnih načinov na-

Program 2 CUDA program, ki inicializira polje v glavnem pomnilniku na vrednost 0f. Jedro pri tem uporablja deljeni pomnilnik kot števec za vsako nit.

```
__device__ float array[BLOCK*THREADS];
__shared__ int k[THREADS];
__global__ void clearArrays() {
    int startDIdx = threadIdx.x+(blockDim.x*LENGTH)*blockIdx.x;

    for(k = 0; k<LENGTH; k++) {
        array[startDIdx+blockDim.x*k] = 0.0f;
    }
}
```

slavljanj, filtriranj ipd. Vse so predpomnjene po principu prostorske lokalnosti in dostop ob morebitnih zgrešitvah je vedno v vezanem načinu dostopa.

Uporaba teksturnega pomnilnika je iz CUDA programov mogoča le preko funkcij za zajem tekstur (texture fetch). Pri teh funkcijah je najpomembnejši parameter referenca teksture. Ta definira kateri del teksturnega pomnilnika (v nadaljevanju teksture) je zajet in mora biti definiran že pred izvajanjem jeder. Na isto teksturo se lahko sklicuje več različnih tekstur, ki se lahko prekrivajo. Za definiranje reference teksture moramo določiti razsežnost teksture, izhodni tip in način branja. Pri branju lahko izbiramo med možnostjo branja vrednosti ali normaliziranih vrednosti. Pri slednjih bo strojna oprema poskrbela za normalizacijo vrednosti na interval med 0 in 1 oz. -1 in 1.

Ostale pomembne nastavitve dostopanja do tekstur vključujejo, ali so koordinate normalizirane ali ne, način naslavljanja in filtriranja tekstur.

- **Normalizirane koordinate** – pri privzeti vrednosti se do tekslov (texel), to je elementov teksture, dostopa v obsegu od [0, N), kjer je N velikost teksture v neki razsežnosti. V primeru normaliziranih koordinat pa se do tekslov dostopa v obsegu [0, 1) za vse razsežnosti. Tak način dostopanja se uporablja pri aplikacijah, kjer so koordinate neodvisne od velikosti teksture.

- **Način naslavjanja** – ta način določa, kaj se zgodi s koordinatami, ki so izven obsega teksture. Pri ne-normaliziranih koordinatah je na voljo le način rezanja (clamp), ki deluje tako, da se vrednosti manjše od 0 preslikajo na 0 in večje od N preslikajo na N-1. Za normalizirane koordinate pa imamo na voljo način rezanja ali zvijanja (wrap). Zadnji način

je uporaben za periodične signale, koordinata 1.25 se namreč preslika v 0.25, -1.25 pa v 0.75.

- **Filtriranje tekstur** – je omogočeno samo za teksture, ki vračajo vrednosti v obliki plavajoče vejice. Ko je ta opcija vklopljena, so vrnjene vrednosti linearne interpolacije med dvema, štirimi ali osmimi sosednjimi teksli.

Pred dejansko uporabo teksture v programski kodi jo je potrebno še povezati z nekim poljem. To se naredi z ukazom `cudaBindTexture()` ali z `cudaBindTextureToArray()`.

2.3.4 Delo z napakami

Vse funkcije izvajanja vračajo kodo napake (error code). Vendar funkcije, ki se izvajajo asinhrono, kodo napake, ki se zgodi na napravi, ne morejo sporočiti. Jedra se namreč izvajajo asinhrono in se klic funkcije vrne takoj in ne tedaj, ko je zaključeno izvajanje na napravi. Koda napake sporoča le napake na napravi, ki so se že zgodile.

Edini način za preverjanje takih napak je sinhronizacija takoj po klicu asinhrono izvajajočih se funkcij. To storimo z ukazom `cudaThreadSynchronize()`, ki čaka, da se funkcija dokonča, in nato vrne kodo napake.

CUDA za vsako nit gostitelja vzdržuje spremenljivko, ki je inicializirana na vrednost `cudaSuccess`. V primeru napake se bo vrednost te spremenljivke prepisala s kodo napake, ki jo resetiramo z branjem zadnje napake.

Klici jeder ne vračajo nobenih napak. Če želimo preveriti napake, do katerih je prišlo med izvajanjem jeder, moramo takoj po klicu postaviti spremenljivko na `cudaSuccess` in zatem izvesti gostiteljsko sinhronizacjo niti.

2.3.5 Razhroščevanje z uporabo emulacijskega načina

Za odpravljanje napak v jedrih se uporablja emulacijski način na napravah z računsko zmogljivostjo večjo od 1.0, lahko pa tudi razhroščevalnik `CUDA-GDB`. Prevajalnik `nvcc` bo ob uporabi posebnega stikala `-deviceemu` jedro prevedel za izvedbo v emulacijskem načinu. S tem načinom koda, namenjena za izvajanje na napravi, teče na gostitelju in jo lahko preučimo z gostiteljevim razhroščevalnikom. Pri uporabi omenjenega stikala se definira predprocesorjeva spremenljivka `__DEVICE_EMULATION__`, kar nam omogoča pisanje kode, ki sicer ni podprta na sami napravi, npr. `printf()` ukaz, ki ga lahko uporabimo za izpis podatkov.

Pred prevajanjem za emulacijski način izvajanja moramo paziti, da v ta način prevedemo tudi knjižnice in ostalo kodo, sicer bo prišlo do napake `cudaErrorMixedDeviceExecution`.

Z izvedbo aplikacije v emulacijskem načinu bo CUDA emulirala programski model. Za vsako nit v bloku, se bo ustvarila nit na gostitelju. Programer mora pri tem zagotoviti, da:

- je gostitelj sposoben poganjanja vseh niti namenjenih za napravo in eno nit, namenjeno za gostitelja;
- je dovolj pomnilnika za poganjanje vseh niti, ker vsaka nit dobi 256 KB sklada.

Programer se mora zavedati, da način emuliranja emulira napravo, ne simulira. Čeprav je ta način zelo uporaben za ugotavljanje napak v algoritmu, je vseeno določene napake zelo težko odkriti:

- tekmovanje za vire[6] (race condition) je težko odkriti, ker je število niti v emulacijskem načinu običajno precej manjše, kot pri izvajanju na napravi;
- ko dereferenciramo kazalce na globalni pomnilnik na gostitelju ali na napravi, se bo izvajanje na napravi zagotovo končalo v nedefiniranem stanju, kjer lahko emulacija proizvede pravilne rezultate;
- večino časa rezultati ne bodo čisto enaki kot na drugih napravah. To je sicer pričakovano, saj je točnost odvisna od možnosti nastavitev prevajalnika, različnih prevajalnikov, različnih množic ukazov in arhitektur. V splošnem nekatere gostiteljske platforme shranijo vmesne rezultate enojno-natančnih operacij v plavajoči vejici v registrih z večjo natančnostjo, kar lahko pripelje do velikih razlik v natančnosti pri emuliranem izvajanju in izvajanju na napravi.

[6]Tekmovanje za vire [23] je stanje, ko vsaj dva dogodka (procesa, niti ...) poskušata doseči isti vir v istem trenutku.

Poglavje 3

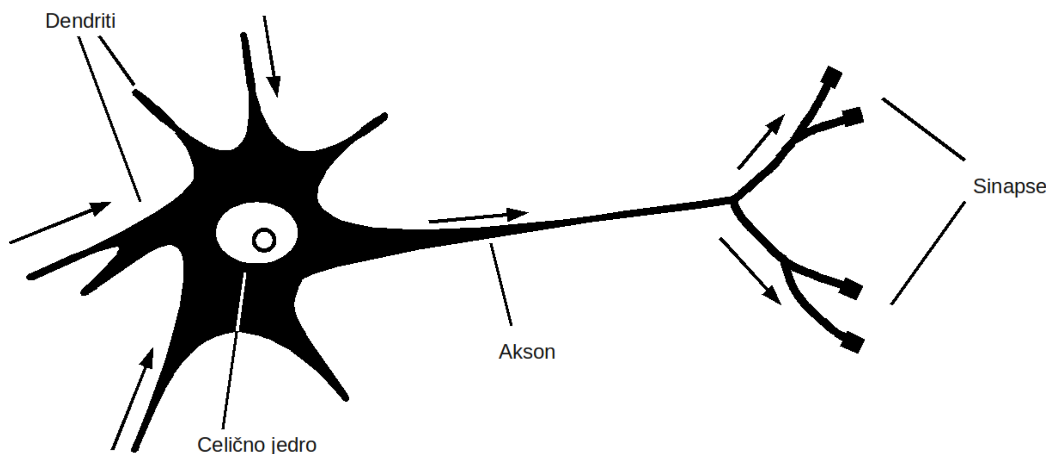
Umetne nevronske mreže

S prihodom računalnikov in razvojem modernih teorij učenja procesiranja nevronov v možganih v štiridesetih letih 20. stoletja je računalnik postal orodje za preučevanje posameznih nevronov in gruč nevronov, ki jih imenujemo umetne nevronske mreže [12, 18, 2] (v nadaljevanju nevronske mreže). Že v začetku preučevanja nevronskih mrež so odkrili, da so možgani zelo kompleksni in da podatke vzporedno obdelujejo. Možgani so sposobni organizirati nevrone tako, da lahko opravljajo zahtevne računske operacije (npr. računalniški vid) veliko hitreje kot kakšen namenski računalnik. V računalništvu so kot analogijo z možgani uvedli različne modele nevronskih mrež, ki temeljijo na poenostavljenih modelih nevronov.

3.1 Nevron in model nevrona

Biološki nevron je element nevronskega sistema, ki procesira, pomni in prenaša informacije. Sestavljen je iz dendritov, celičnega jedra, aksona in sinaps, kot shematično prikazuje slika 3.1. Sinapse so povezave, preko njih pa med seboj komunicirajo nevroni. Najbolj običajna je kemična sinapsa (chemical synapse), ki električni signal pretvarja v kemijskega in obratno. Akson je namenjen prenosu na daljše lokacije, npr. celično jedro nevrona, ki nadzoruje palec na nogi, leži v hrbtenjači (spinal cord) in njegov akson poteka po celotni dolžini noge [2]. Dendriti pa so področja prejemanja signalov. Nevrone v glavnem delimo v tri skupine: za prenos signalov, nadzor motoričnih sposobnosti in na nevrone za zaznavanje. Po obliki se med seboj lahko močno razlikujejo, vendar so po strukturi v splošnem enaki.

Močno poenostavljeno delovanje lahko najlažje prikažemo s primerom. Predpostavimo, da imamo nevron s štirimi vhodi označenimi s črkami A, B, C, D



Slika 3.1: Shematični prikaz možganskega nevrona.

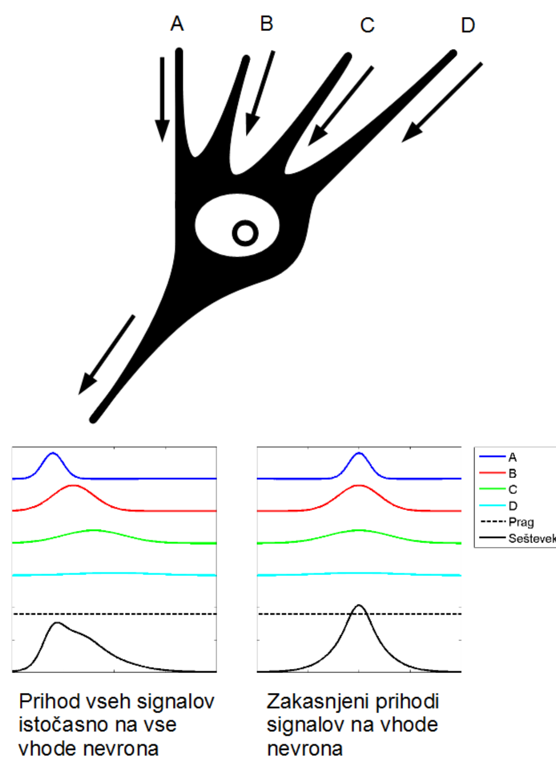
in enim izhodom (slika 3.2). Pri vhodih je pomembna dolžina posameznih dendritov do jedra. V primeru, da na vse vhode istočasno prispe signal, bo do jedra najprej prispel preko dendrita od najbližjega vhoda A, nato od B, C in na koncu od D. V tem primeru skupna vrednost vhodov ne bo dosegla praga (threshold), ki je potreben, da se celično jedro aktivira in pošlje izhodni signal. Zato je potrebno upoštevati tudi čas in zaporedje, v katerem se morajo signali sprožiti. Najprej mora signal priti na vhod D, nato C, B in na koncu na vhod A, saj je ta najbližje jedru. To si lahko predstavljamo kot senzorje za zaznavanje svetlobe. Nevron se odziva le, če svetloba prihaja od desne proti levi. Ne odizva pa se na stacionarne objekte oz. premikajoče se iz leve proti desni zaradi nesimetrične porazdelitve dendritov glede na jedro.

Kot analogijo biološkim nevronom so v računalništvu uvedli model nevrona (slika 3.3). Sestavljajo ga trije osnovni elementi:

- **Sinapse**, na katerih se nahajajo numerične vrednosti – uteži. Z njimi določamo moč posamezne sinapse. Na vhodu sinaps so vhodni podatki označeni z x_j pri j-ti sinapsi, ki je povezana na k-ti nevron. Ta vhod se pomnoži z utežjo w_{kj}. Prvi indeks pri označevanju pomeni, od katerega nevrona je utež, drugi pa, na katero sinapso se nanaša.

- **Seštevalnik,** ki izračuna uteženo vsoto vhodnih signalov in uteži.

- **Aktivacijska funkcija:** vrednost seštevalnika se v zadnjem elementu omeji glede amplitude, zato vrednost običajno preslikamo na območje od -1 do 1 oz. od 0 do 1, odvisno od tipa funkcije.



Slika 3.2: Prikaz aktivacije nevrona.

Matematično delovanje nevrona k opisuje par enačb:

$$u_k = \sum_{j=1}^{m} w_{kj} x_j \tag{3.1}$$

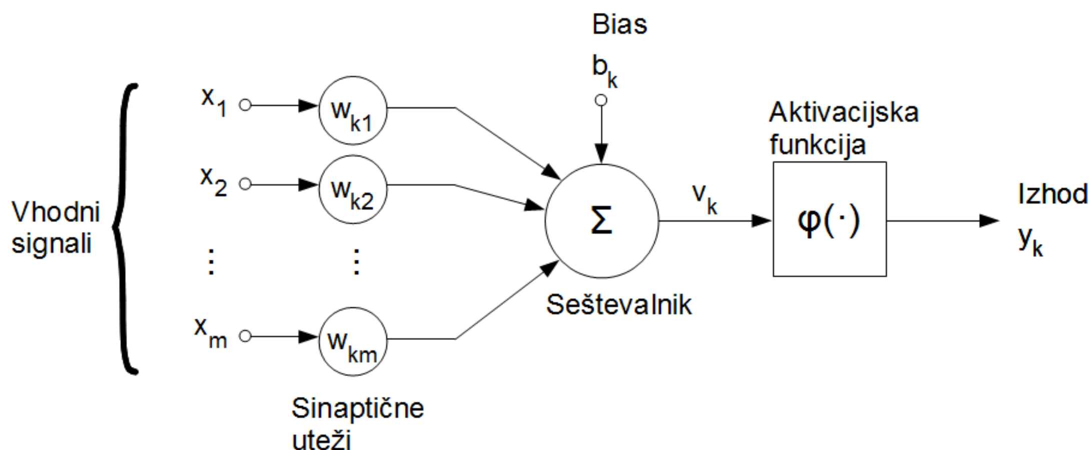
in

$$y_k = \varphi(u_k + b_k) \tag{3.2}$$

kjer so x_1, x_2, ...,x_m vhodni signali; w_{k1}, w_{k2}, ..., w_{km} uteži sinaps za nevron k; u_k je izhod seštevalnika; b_k je bias; $\varphi(\cdot)$ je aktivacijska funkcija in y_k je izhodna vrednost nevrona.

Vloga biasa je, da prišteje oz. odšteje neko vrednost k izhodu seštevalnika in pred vstopom v aktivacijsko funkcijo. S tem povzroči premik krivulje oz. premice iz izhodišča koordinatnega sistema.

Ker je bias zunanji parameter k umetnemu nevronu k, ga lahko vključimo v enačbi 3.1 in 3.2 tako, da začnemo seštevati pri $j = 0$, na vhod x_0 pripeljemo konstanto 1 in jo namesto množenja z utežjo pomnožimo z vrednostjo biasa.



Slika 3.3: Model nevrona.

Tako dobimo poenostavljeni enačbi:

$$v_k = \sum_{j=0}^{m} w_{kj} x_j \tag{3.3}$$

in

$$y_k = \varphi(v_k) \tag{3.4}$$

kjer je w_0 bias.

3.2 Vrste aktivacijskih funkcij

Preslikavo vsote seštevalnika na interval, ki običajno obsega števila med -1 in 1 oz. 0 in 1, se izvede z uporabo aktivacijske funkcije. Ločujemo tri osnovne skupine funkcij, katerih grafični prikaz je podan na sliki 3.4.

1. **Pragovne funkcije.** Primer takih funkcij je funkcija enotine stopnice. Ko je izhod seštevalnika manjši od 0, tedaj je vrednost funkcije 0, sicer pa 1.

$$\varphi(v) = \begin{cases} 1, & \textit{če } v \geq 0 \\ 0, & \textit{če } v < 0 \end{cases} \tag{3.5}$$

2. **Odsekovno linearne funkcije.** V to skupino spadajo različne funkcije, ki jih omejimo na delovanje v določenem intervalu in so naslednje oblike:

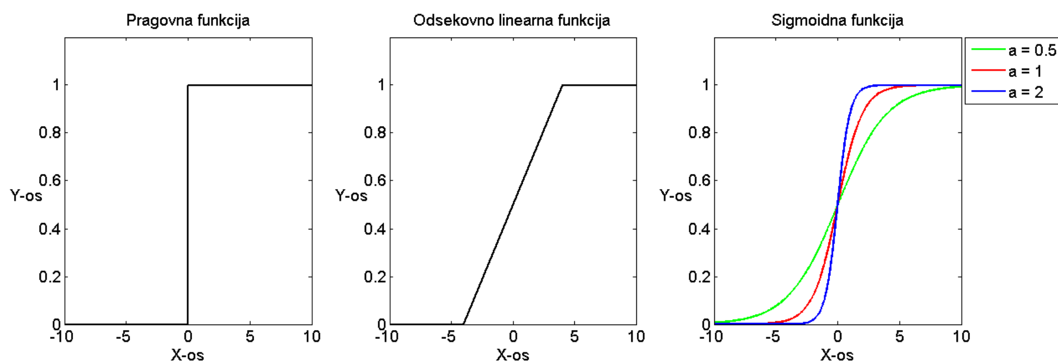
$$\varphi(v) = \begin{cases} 1, & \textit{če } v \geq +\frac{1}{2} \\ v, & \textit{če } +\frac{1}{2} > v > -\frac{1}{2} \\ 0, & \textit{če } v \leq -\frac{1}{2} \end{cases} \tag{3.6}$$

3. **Sigmoidne funkcije.** Njihov graf je S-oblike in so daleč najpogosteje uporabljene. So strogo naraščujoče in odvedljive funkcije. Prvi odvod zato nikoli ni negativen. Primer funkcije je podan v enačbi 3.7, kjer s parametrom a spreminjamo naklon funkcije.

$$\varphi(v) = \frac{1}{1+e^{-av}} \tag{3.7}$$

$$\begin{aligned}\varphi'(v) &= \frac{a\ e^{-av}}{(1+e^{-av})^2} \\ &= a\ \varphi(v)(1-\varphi(v))\end{aligned} \tag{3.8}$$

V limiti naklona bi funkcija postala podobna pragovni funkciji. Razlikovala bi se v tem, da bi bila še vedno zvezna in bi vsebovala vsa števila od 0 do 1 medtem, ko pragovna ne bi. Ker je funkcija povsod zvezna in je zvezno odvedljiva, se funkcijo in njen odvod 3.8 uporablja tudi v večnivojskih nevronskih mrežah.



Slika 3.4: Prikaz različnih aktivacijskih funkcij.

3.3 Lastnosti nevronskih mrež

Umetne nevronske mreže imajo veliko dobrih in nekaj slabih lastnosti. Med pomembnejše dobre lastnosti spadajo: biološka podobnost, večsmerno izvajanje, prilagodljivost in robustnost, posploševanje iz primerov, učenje in matematična podlaga. Kombinacijo dobrih in slabih lastnosti vsebuje paralelizem, največja slabost nevronskih mrež pa je razlaga odločitve.

Biološka podobnost je v tem, da se z umetnimi nevronskimi mrežami skuša oponašati delovanje možganskih celic v poenostavljeni obliki. Namesto celic se uvede poenostavljen model nevrona, s katerimi nato tvorimo mrežo nevronov.

Večsmerno izvajanje: Ko je v nevronski mreži vsak nevron povezan z vsakim, je vsak nevron hkrati vhoden in izhoden. V takih primerih nevronska mreža ne dela razlik med vhodom ter izhodom in je lahko poljubna podmnožica nevronov vhodnih ali izhodnih. Mreža tedaj deluje tako, da pri danem vhodu aproksimira neznane vrednosti, ki so izhod.

Prilagodljivost in robustnost: Nevronska mreža ima zaradi načina delovanja že vgrajeno zmogljivost prilagajanja uteži v sinapsah glede na okolje, v katerem deluje. V primeru, da deluje v okolju, ki ni stacionaren (ko se statistika s časom spreminja), se lahko mrežo načrtuje tako, da spreminja uteži v realnem času. Kot posplošeno pravilo lahko rečemo, da bolj kot naredimo prilagodljivo mrežo v stacionarnem okolju, bolj robustne bodo njene zmogljivosti v nestacionarnem okolju. Ker je mreža sestavljena iz mnogo nevronov, kjer vsak predstavlja delček celote, odstranitev oz. nedelovanje posameznih nevronov ali sinaps ne povzroči izgube podatkov v celoti. Taka okvara povzroči zgolj manjšo napako pri končnem rezultatu.

Posploševanje iz primerov: Pri realnih problemih se pogosto zgodi, da pri nekem atributu podatek manjka. V primeru nevronskih mrež imajo lete pomembno lastnost: učijo se iz izkušenj in zato interpolirajo manjkajoči podatek iz že videnih primerov. Tako lahko podajo tudi pravilne odzive na podatke, ki jih predhodno še niso videle.

Učenje poteka s predstavljanjem učnih podatkov nevronski mreži. Ta nato preko preračunavanja uteži v pomnilniških celicah oz. sinapsah izračuna izhodne vrednosti, ki se zatem ocenijo. Če vrednosti niso enake željenim izhodom, se uteži popravi. Postopek učenja se ponavlja, dokler ne dosežemo željene točnosti. Ločujemo več različnih načinov učenja, kar je podrobneje opisano v poglavju 3.5.

Matematična podlaga: Med ključne dobre lastnosti, zakaj so nevronske mreže pogosto uporabljene, sodi tudi dobro utemeljena matematična podlaga. Teoretično podlago so osnovali v času 50-ih let 20. stoletja, ko je Donald Hebb definiral Hebbovo pravilo in Frank Rosenblatt ustvaril perceptron.

Paralelizem: Visoka stopnja paralelizma je posledica dejstva, da vsak nevron deluje relativno neodvisno od ostalih. To omogoča izredno hitro izvajanje v celoti, zato so se mreže sposobne prilagajati okolju v realnem času. Kljub paralelnemu izvajanju nevronov v mreži pa je slabost, da je zaporedna implementacija še vedno počasna.

Razlaga odločitve: V primerjavi z običajnimi računalniki, kjer obstaja nek algoritem, po katerem preračunava podatke, moramo nevronske mreže učiti iz primerov. Primeri morajo biti pazljivo izbrani, sicer se mreža lahko napačno nauči. Pri tem obstaja velika slabost. Ko že naučeno mrežo uporabimo za reševanje realnih problemov, pri odločitvah nevronske mreže uporabnik ne bo imel nobenih podatkov, zakaj se je mreža tako odločila.

3.4 Topologija nevronske mreže

V splošnem ločujemo tri topologije nevronskih mrež: enonivojske usmerjene, večnivojske usmerjene in ciklične usmerjene nevronske mreže.

Enonivojske usmerjene nevronske mreže: V najbolj preprostih mrežah so vsi vhodi povezani z vsemi nevroni, iz katerih sledijo izhodi. Nevroni med seboj niso povezani in s takšno topologijo povezljivosti tvorijo "en nivo" nevronov, med katere ne štejemo vhode. Mreža je strogo usmerjena in aciklična.

Večnivojske usmerjene nevronske mreže so sestavljene iz več enonivojskih usmerjenih nevronskih mrež. Mrežo sestavlja en vhodni in izhodni nivo ter poljubno število skritih nivojev – to je nivojev med vhodnim in izhodnim nivojem. Z dodajanjem enega ali več skritih nivojev je mreža sposobna reševanja kompleksnejših problemov, npr. XOR problem. Vmesni nivoji so med seboj lahko polno ali delno povezani. V primeru, da so polno povezani, je vsak nevron na nekem nivoju povezan z vsemi na naslednjem, sicer pa le z določenimi. Običajno se večnivojske mreže označuje v stilu $v - s_1 - \ldots - s_n - i$, kar pomeni v-nevronov na vhodnem nivoju, i-nevronov na izhodnem nivoju in n-nivojev skritih nevronov, kjer je s_i nevronov na i-tem skritem nivoju.

Ciklične usmerjene nevronske mreže: Mreže te vrste imajo vsaj eno ciklično povezavo. Primer take mreže bi bila mreža, ki bi imela nivo nevronov, kjer je vsak izhod nevrona povezan z vsemi vhodi nevronov na tem nivoju, razen s samim seboj. Te mreže običajno vsebujejo tudi poseben element zakasnitve, tako da so izhodni podatki šele v naslednjem koraku vhodni.

3.5 Vrste učenja

Ločujemo dve vrsti učenja: nadzorovano (supervised learning) in nenadzorovano (unsupervised learning). V nenadzorovano učenje spada še t. i. spodbujevano učenje (reinforcement learning).

Nadzorovano učenje pojmujemo kot učenje, kjer imamo podane vhodne in željene izhodne podatke. Nevronska mreža najprej preračuna vhodne po-

datke, kar pripelje do nekih izhodnih podatkov. Ti se nato ocenijo s pomočjo t. i. učitelja, zato to vrsto učenja imenujemo tudi učenje z učiteljem. Pri ocenjevanju izhodnih podatkov se običajno izračuna napako izhoda, ki je definirana kot razlika med željeno in dejansko vrednostjo. Željena vrednost predstavlja optimalen odziv nevronske mreže. Napako izhoda se uporabi za popravljanje nevronske mreže k pravim željenim izhodom. Nadzorovano učenje to lahko stori z upoštevanjem gradienta površine napake v primerjavi z obnašanjem sistema. Gradient površine napake je vektor, ki kaže v smeri najstrmejšega spusta in kaže v smeri lokalnega oz. globalnega minimuma. Kot mero učinkovitosti se uporablja srednjo kvadratično napako ali vsoto kvadratov napake.

Nenadzorovano učenje poteka brez nadzora učitelja in brez željenih vrednosti izhodnih nevronov. Uči se tako, da začne ločevati statistične zakonitosti med posameznimi primeri in razvije svojo predstavitev kodiranja značilk vhoda ter tako po potrebi tudi avtomatsko generiranje novih razredov. Za implementacijo tega učenja lahko uporabimo zmagovalno strategijo, kjer izhodni nevron z največjo vrednostjo zmaga in se sproži.

Spodbujevano učenje spada v zvrst nenadzorovanega učenja, katerega cilj je minimiziranje funkcije "cost-to-go". Ta je definirana kot zbrana vrednost akcij preko več korakov, namesto neposredne vrednosti. Izkaže se lahko, da so določene predhodne akcije v zaporedju najboljši faktor ocenitve celotnega obnašanja sistema.

3.6 Vrste nevronskih mrež

V desetletjih razvoja je nastalo mnogo različnih mrež, kjer med najbolj osnovne skupine štejemo: naprej povezane (feedforward), "radial basis function network" (RBFN), samo-organizirajoče (self-organizing maps) in rekurentne mreže (recurrent network).

Naprej povezane nevronske mreže: Nevroni so povezani z usmerjenimi povezavami tako, da ne tvorijo nobenega cikla. Signal vedno potuje od vhodnih do izhodnih nevronov. Najbolj osnovna tipa te vrste sta perceptron in večnivojski perceptron, ki sta podrobneje opisana v poglavju 3.7 in 3.8.

RBFN: So mreže, ki so po topologiji zelo podobne naprej povezanim mrežam. Imajo torej usmerjene povezave brez ciklov in signal potuje od vhoda do izhoda. Glavna razlika v primerjavi z naprej povezanimi mrežami je v uporabi aktivacijske funkcije, kjer te mreže uporabljajo "radial basis" funkcije. Vhodne podatke obravnavajo kot vektorje v prostoru od katerih se izračuna razdalja do središčnega vektorja za posamezni nevron in iz česar se

nato izračuna uteženo vsoto. Za razdaljo se ponavadi vzame evklidsko razdaljo. Metoda je zelo primerna za interpoliranje v večrazsežnem prostoru.

Samo-organizirajoče mreže: To je primer mreže, ki se uči brez nadzora učitelja in deluje po principu zmagovalne strategije. Nevroni so običajno razporejeni v eno ali dvorazsežno mrežo. Možne so tudi večrazsežne mreže, vendar niso pogoste. Nevroni se v fazi učenja sčasoma selektivno uglasijo na različne vhodne vzorce in se s tem razporedijo na mreži tako, da se uredijo z upoštevanjem medsebojnih odnosov in značilk. Samo-organizirajoče mreže si zato lahko predstavljamo kot topografski zemljevid, kjer se vsak vhod preslika na neke koordinate zemljevida, ki predstavlja značilke tega vhodnega vzorca.

Hopfieldove nevronske mreže: So primer rekurentnih mrež, ki imajo po topologiji usmerjene povezave med posameznimi nevroni. Od običajnih naprej povezanih usmerjenih mrež se razlikujejo v tem, da so nevroni povezani tudi z nevroni, bližje vhodnim nevronom, s čimer tvorijo zanke. Vhodi v posamezne nevrone so zato odvisni od trenutnega in od predhodnjih vhodov. Poseben sestavni element mrež je operator enotine časovne zakasnitve, s katerim se izhod nekega nevrona upošteva pri preračunavanjih naslednjih vhodov v preostale nevrone. Hopfieldove nevronske mreže se lahko uporablja tudi za vsebinsko naslovljiv pomnilnik, saj je dokazano, da bo mreža pri nekem delnem vhodu konvergirala k najbližjemu znanemu vhodu, ki si ga je v preteklosti zapomnila.

3.7 Enonivojski perceptron

Kot prvi model učenja z učiteljem je Rosenblatt leta 1958 predstavil perceptron [12]. To je najenostavnejša oblika nevronskih mrež, primerna za razpoznavanje vzorcev, ki so linearno ločljivi (to so takšni primeri, ki so bodisi na levi bodisi na desni strani hiper ravnine).

Najbolj osnovna verzija je sestavljena iz enega nevrona s prilagodljivimi sinapsami in biasom. Algoritem za nastavitev prostih parametrov je prvi predstavil Rosenblatt okoli leta 1960. Dokazal je tudi, da če za učenje uporabimo primere iz dveh linearno-ločljivih razredov, bo algoritem za učenje perceptrona konvergiral in nastavil ločnico v obliki hiper ravnine med razredoma. Dokaz konvergence algoritma je poznan kot teorem konvergence perceptrona (perceptron convergence theorem). Perceptron zgrajen iz enega nevrona lahko klasificira dva razreda oz. hipotezi. V primeru, da bi želeli klasificirati več med seboj sicer linearno ločljivih primerov, bi lahko uporabili več nevronov, kjer bi vsak opravil klasifikacijo za nek razred.

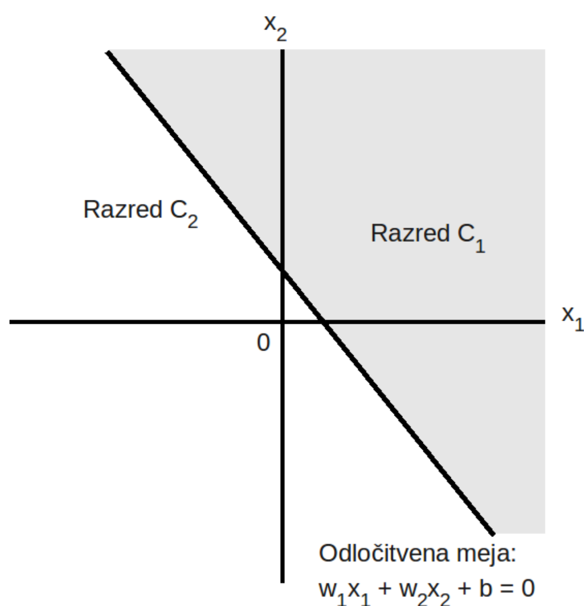
Kot je že opisano v poglavju 3.1, je matematično ozadje delovanja nevrona

definirano z enačbama 3.1 in 3.2. Cilj perceptrona je pravilno klasificirati vhodne primere. S preračunavanji po omenjenih enačbah izračunamo izhod y, na podlagi katerega se klasificira vhodni primer. Npr. če je izhod -1, se primer uvrsti v razred C_1, in če je izhod +1, v C_2.

Pri najenostavnejšem primeru si klasifikacijo lahko v grafični obliki predstavljamo kot premico v dvorazsežni ravnini (slika 3.5), kjer so na eni strani primeri razreda C_1 in na drugi C_2. Premica, ki ločuje razreda je definirana kot:

$$\sum_{i=1}^{m} w_i x_i + b = 0 \tag{3.9}$$

kjer so x_i vhodni podatki, w_i uteži za i-ti vhodni podatek in b-bias.



Slika 3.5: Grafični prikaz odločitve dvorazrednega klasifikacijskega problema. Odločitev je določena z $w_1x_1 + w_2x_2 + b = 0$.

Uteži sinaps se tekom učenja spreminjajo. Za spreminjanje lahko uporabimo pravilo popravljanja napake (error-correction), ki je poznano tudi pod imenom algoritem konvergence perceptrona.

Algoritem konvergence perceptrona

Pri delovanju algoritma bias obravnavamo kot utež, ki je povezana na konstanten vhod +1, kot v enačbah 3.3 in 3.4. Predpostavimo, da vsi učni primeri

izhajajo iz linearno ločljivih razredov. Množica H_1 vsebuje podmnožico učnih primerov $x_1(1), x_1(2), \ldots$, ki pripadajo razredu C_1, in množica H_2 vsebuje preostanek učnih primerov $x_2(1), x_2(2), \ldots$, ki pripadajo razredu C_2. Unija H_1 in H_2 tvori celotno množico učnih primerov. Pri podanih množicah H_1 in H_2 učenje klasifikatorja pomeni nastavljanje vektorja uteži w tako, da sta razreda C_1 in C_2 linearno ločljiva. To pomeni, da obstaja vektor uteži w, kjer velja:

- $w^T x > 0$ za vsak vhodni primer iz razreda C_1;
- $w^T x \leq 0$ za vsak vhodni primer iz razreda C_2.

Vektor uteži w se določi z učenjem mreže na učni množici tako, da mreža pravilno klasificira učne primere. Algoritem učenja je podan v programu 3. Ta na koraku 4 uporablja pravilo delta, ki upošteva razliko med dejanskim izhodom in željenim izhodom. Pravilo se imenuje tudi gradientno pravilo, saj se vektor uteži spremeni v smeri manjše napake oz. v smeri negativnega odvoda.

3.8 Večnivojski perceptron

Naprednejša oblika združevanja enonivojskih perceptronov so večnivojski perceptroni. Tipična mreža je sestavljena iz več nivojev: vhodnega, izhodnega in enega ali več vmesnih nivojev, ki jih imenujemo skriti nivoji (hidden layer).

Delovanje mreže lahko ločimo na dve fazi: 1) prehod naprej (feedforward pass) in 2) prehod nazaj (backward pass). V prvi fazi se na vhod mreže pošlje vhodni vektor (primer) in se izračuna odziv nevronov na te vhode na prvem nivoju. To se ponavlja preko ostalih nivojev, le da ostali nivoji dobijo kot vhod izhode predhodnega nivoja. Na ta način dobimo odziv večnivojskega perceptrona na nek vhod. Sledi druga faza, kjer se opravi vzvratno razširjanje napake in t. i. dejansko učenje mreže. Poteka tako, da izračunamo napako odziva kot razliko med dejanskim odzivom in željenim odzivom mreže na izhodu. Napako se nato razširi na predhodnje nivoje v obratni smeri, od izhoda proti vhodu. Od tod tudi ime vzvratno razširjanje napake (backpropagation of error). Na ta način se izračuna napako, ki jo generira vsaka sinapsa, in se ustrezno popravi uteži na sinapsah.

Algoritem vzvratnega razširjanja napake

Pred predstavitvijo je potrebno definirati notacijo, s katero bo opisan algoritem:

Program 3 Psevdo algoritem konvergence perceptrona.

```
Spremenljivke in parametri:
x(n) = vektor vhodnih podatkov dolžine m+1
     = [+1,x_1(n),x_2(n),...,x_m(n)]^T
w(n) = vektor uteži dolžine m+1
     = [b(n),w_1(n),w_2(n),...,w_m(n)]^T
b(n) = bias
y(n) = dejanski odziv
d(n) = željeni odziv
η   = parameter stopnje učenja, majhna pozitivna konstanta

1) Inicializacija:
   Nastavi w(0)=0, nato izvajaj sledeči postopek za korake n=1,2,...
2) Aktivacija:
   Na vsakem časovnem koraku n, aktiviraj perceptron z zaporednimi
   vhodi x(n) in željenimi izhodi d(n).
3) Izračun dejanskega odziva:
   Izračunaj odziv perceptrona: y(n) = sgn[w^T(n)x(n)],
   kjer je sgn(·) - funkcija predznaka (signum function).
4) Nastavitev vektorja uteži:
   Posodobi vektor uteži perceptrona po pravilu:
   w(n+1) = w(n) + η[d(n)-y(n)]x(n), kjer je
   d(n) = { +1   če x(n) ∈ C_1
          { -1   če x(n) ∈ C_2
5) Ponavljaj:
   Za eno povečaj časovni korak n in pojdi na korak 2.
```

- Indeksi i, j in k se nanašajo na različne nevrone v mreži. Če signal potuje od leve (vhoda) proti desni (izhodu), tedaj j-ti nevron leži desno od i-tega in k-ti desno od j-tega.
- V n-tem časovnem koraku (ponovitvi) je n-ti učni primer predstavljen mreži.
- Energija $E(n)$ je vsota kvadratov napake v ponovitvi n.
- Simbol $e_j(n)$ pomeni napako izhoda nevrona j v ponovitvi n.
- Željen izhod nevrona j v ponovitvi n označimo s simbolom $d_j(n)$.
- Simbol $y_j(n)$ pomeni dejanski izhod nevrona j v ponovitvi n.
- Uteži označujemo z $w_{ji}(n)$. To so uteži na sinapsi med nevronom i in j.
- Spremembo uteži predstavlja simbol $\Delta w_{ji}(n)$.
- Uteženo vsoto izhodov in bias-ov nevronov j v ponovitvi n označimo z $v_j(n)$.
- Aktivacijska funkcija j-tega nevrona je označena s simbolom $\varphi(\cdot)$.
- Zaradi enostavnejše predstavitve bias-a, le-tega obravnavamo v vektorju uteži, tako je bias j-tega nevrona shranjen v w_{j0} uteži, ki je povezana na konstanten vhod $+1$.
- i-ti element vektorja vhoda označujemo z $x_i(n)$.
- k-ti element celotnega izhoda je označen z $o_k(n)$.
- Oznaka m_l označuje število nevronov na nivoju l. $l = 0, 1, \ldots, L$, kjer je L globina mreže. m_0 predstavlja nivo vhoda, m_L nivo izhoda, vmesni nivoji pa skrite nivoje.

Najprej je potrebno izvesti prehod naprej, pri čemer se vrednosti uteži ne spreminjajo in se nevrone le "proži". Izhod j-tega nevrona se izračuna kot vrednost aktivacijske funkcije utežene vsote vseh m-vhodov v ta nivo nevronov:

$$y_j(n) = \varphi(\sum_{i=0}^{m} w_{ji}(n) y_i(n)) \tag{3.10}$$

kjer nevron prejme kot vhod izhod nevronov na predhodnem nivoju. V primeru, da računamo izhod vhodnega nivoja nevronov, tedaj velja $y_i(n) = x_i(n)$.

Po tem postopku izračunamo izhode nevronov na vseh nivojih. Dejanski odziv mreže na nek vhodni primer x je izhod o, kar je izhod nevronov na izhodnem nivoju. To zapišemo kot $y_j(n) = o_j(n)$, ko $j = L$. Iz dejanskega odziva $o_j(n)$ in željenega odziva $d_j(n)$ se izračuna napako izhoda $e_j(n)$, kar se uporabi v drugi fazi algoritma.

V drugi fazi se napako $e_j(n)$ propagira proti vhodnim nivojem. Pri prehodu nazaj se za vsak nevron izračuna lokalni gradient δ, ki ga uporabimo pri posplošenem pravilu delta za spremembo uteži:

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n) \tag{3.11}$$

Izračun $\delta_j(n)$ se razlikuje za skrite in izhodne nivoje:

Izhodni nivo: $\delta_j(n)$ je v tem primeru enak napaki $e_j(n)$, pomnoženi z odvodom aktivacijske funkcije izhoda nevrona:

$$\delta_j(n) = e_j(n) \varphi'_j(\sum_{i=0}^{m} w_{ji}(n) y_i(n)) \tag{3.12}$$

Skriti nivoji: Pri skritih nivojih je potrebno upoštevati še vrednosti uteži, s katerimi posamezne sinapse prispevajo k skupni napaki. Enačba 3.13 je zato rekurzivna, kjer moramo najprej izračunati lokalne gradiente izhodnega nivoja, nato zadnjega skritega nivoja, zatem predzadnjega in tako dalje proti vhodnemu nivoju.

$$\delta_j(n) = \varphi'_j(\sum_{i=0}^{m} w_{ji}(n) y_i(n)) \sum_k \delta_k(n) w_{kj}(n) \tag{3.13}$$

Uteži naslednjega koraka, za nivo l, se posodobi s prištevanjem delta uteži:

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \Delta w_{ji}^{(l)}(n) \tag{3.14}$$

Matematične enačbe lahko utemeljimo s parcialnimi odvodi. Napako celotne mreže lahko izračunamo kot polovico vsote kvadratov po vseh izhodnih nevronih:

$$E(n) = \frac{1}{2} \sum_{j \in C} e_j^2(n) \tag{3.15}$$

kjer množica C vsebuje vse nevrone izhodnega nivoja.

Če enačbo 3.10 razčlenimo na

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \tag{3.16}$$

$$y_j(n) = \varphi_j(v_j(n)) \tag{3.17}$$

potem je $\Delta w_{ji}(n)$ sprememba sinaptične uteži $w_{ji}(n)$ sorazmerna parcialnemu odvodu $\partial E(n)/\partial w_{ji}(n)$, ki ga lahko zapišemo po verižnem pravilu:

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = \frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} \tag{3.18}$$

Parcialni odvod $\partial E(n)/\partial w_{ji}(n)$ predstavlja faktor občutljivosti, iz katerega lahko izračunamo smer, v kateri moramo iti, da pridemo do lokalnega minimuma, od tod tudi negativen predznak v enačbi 3.24.

Pri odvajanju obeh strani enačbe 3.15 po $e_j(n)$, je prvi člen desne strani enačbe 3.18

$$\frac{\partial E(n)}{\partial e_j(n)} = e_j(n) \tag{3.19}$$

Preostali posamezni odvodi enačbe 3.18 pa so:

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{3.20}$$

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)) \tag{3.21}$$

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_j(n) \tag{3.22}$$

Z vstavljanjem posameznih odvodov v enačbo 3.18 dobimo:

$$\frac{\partial E(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n) \tag{3.23}$$

oz. če to vstavimo v pravilo delta, ki pravi, da se uteži $w_{ji}(n)$ prišteje $\Delta w_{ji}(n)$, ki je definirana kot:

$$\Delta w_{ji}(n) = -\eta \frac{\partial E(n)}{\partial w_{ji}(n)} \tag{3.24}$$

potem z združevanjem enačbi 3.23 in 3.24 dobimo enačbo 3.25, ki je identična 3.11:

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n) \tag{3.25}$$

S ponovno uporabo verižnega pravila definiramo še lokalni gradient:

$$\begin{aligned} \delta_j(n) &= -\frac{\partial E(n)}{\partial v_j(n)} \\ &= -\frac{\partial E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \\ &= e_j(n)\varphi_j'(v_j(n)) \end{aligned} \tag{3.26}$$

V primeru, da je nevron j na izhodnem nivoju, potem velja enačba 3.26, ki je ob uporabi enačbe 3.16 ekvivalentna enačbi 3.12. Razlika se pojavi pri nevronih na skritih nivojih. Takrat nimamo na voljo željenega odziva za skriti nevron in je potrebno napako rekurzivno izračunati iz izhodnega nivoja.

$$\begin{aligned}
\delta_j(n) &= -\frac{\partial E(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \\
&= -\frac{\partial E(n)}{\partial y_j(n)} \varphi_j'(v_j(n)) \\
&= \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n)
\end{aligned} \tag{3.27}$$

kjer je j-nevron skritega nivoja. Enačba lokalnega gradienta skritega nevrona je ob uporabi enačbe 3.16 ponovno ekvivalentna enačbi 3.13.

Zaradi izračuna lokalnega gradienta z odvodom aktivacijske funkcije morajo biti funkcije zvezne in zvezno odvedljive, zato se v večnivojskih nevronskih mrežah kot aktivacijsko funkcijo uporablja sigmoidna funkcija, ki je podrobneje opisana v poglavju 3.2.

Pred pričetkom opisovanja implementacije je potrebno omeniti, da se opisani algoritem nanaša na stohastično učenje (oz. online učenje), pri katerem se uteži posodobi po vsakem učnem primeru. Za vzporedno implementacijo je potrebno uporabiti paketno (batch oz. offline) učenje. To učenje deluje tako, da najprej predstavimo vse učne primere. Pri vsakemu izračunamo napako in popravek uteži shranimo. Shranjene uteži šele po predstavitvi vseh učnih primerov končno posodobimo. Pri tem učenju je zato potrebno definirati povprečno napako mreže, kot:

$$E_{av} = \frac{1}{2N} \sum_{n=1}^{N} \sum_{j \in C} e_j^2(n) \tag{3.28}$$

kjer je N število primerov, in ustrezno prilagoditi spremembo uteži Δw_{ji}:

$$\begin{aligned}
\Delta w_{ji} &= -\eta \frac{\partial E_{av}}{\partial w_{ji}} \\
&= -\frac{\eta}{N} \sum_{n=1}^{N} e_j(n) \frac{\partial e_j(n)}{\partial w_{ji}}
\end{aligned} \tag{3.29}$$

Nadaljni postopek je enak kot pri enačbi 3.18.

Oba algoritma imata določene slabe in dobre lastnosti. Pri online učenju lahko izpostavimo hitro konvergenco, manj porabljenega pomnilnika za izračun

novih uteži in zaradi možnosti naključnega vrstnega reda predstavitve podatkov, manjšo možnost konvergence v lokalni minimum. Zadnje je hkrati tudi slabost. Zaradi naključnosti je zato težje vzpostaviti teoretične pogoje za konvergenco algoritma. Pri paketnem učenju naključnost ni pomembna, saj se najprej preračunajo vse razlike in šele na koncu prilagodi vrednost uteži. Dobra lastnost paketnega učenja je ravno prilagajanje vrednosti uteži po predstavitvi vseh učnih primerov, kar omogoča natančno oceno vektorja gradienta. Konvergenca v vsaj lokalni minimum je zato zagotovljena pod enostavnimi pogoji. Kot drugo dobro lastnost naj omenim še, da je zaradi narave algoritma le-tega zelo enostavno prilagoditi za vzporedno računanje, precej lažje kot online učenje.

Največji problem opisanega algoritma je običajno veliko število ponovitev, potrebnih za učenje mreže, in morebitno preveliko prileganje mreže učnim primerom. Prvi problem se rešuje s prištevanjem momentnega člena, ki predstavlja vztrajnost predhodnih sprememb:

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n) \tag{3.30}$$

Momentni člen α je definiran na območju $0 \leq \alpha < 1$ in teži k stabilni spuščajoči se smeri. Ker upošteva delne spremembe uteži predhodnjih korakov, omogoča hitrejšo konvergenco v minimum in izhod iz nekaterih lokalnih minimumov. Poseben primer nastane, ko $\alpha = 0$, tedaj je enačba 3.30 enaka enačbi 3.25 in deluje brez momenta.

Do drugega problema prevelikega prileganja učni množici lahko pride, če nevronska mreža uporablja preveč skritih nevronov in se začenja zapomnjevati učne primere. Posledično to pomeni, da primerov, ki jih še ni videla, ne bo znala pravilno klasificirati. Ta problem rešujemo s pomočjo odstranjevanja uteži [19]. Napakam na posameznih nivojih je potrebno prišteti dodaten člen, kar nam da novo napako E_{nova}:

$$E_{nova} = E + \lambda \sum_{i,j} \frac{\frac{w_{ij}^2}{w_0^2}}{1 + \frac{w_{ij}^2}{w_0^2}} \tag{3.31}$$

Novi člen vsebuje dva parametra: λ in w_0. Parameter λ mora biti dovolj majhen, da preveč ne pokvari originalne napake in dovolj velik, da se odstrani odvečne uteži. Medtem ko želimo imeti pri majhnem parametru w_0 rešitev z manj velikimi utežmi, želimo imeti pri velikem w_0 rešitev z veliko majhnimi utežmi [4].

Poglavje 4

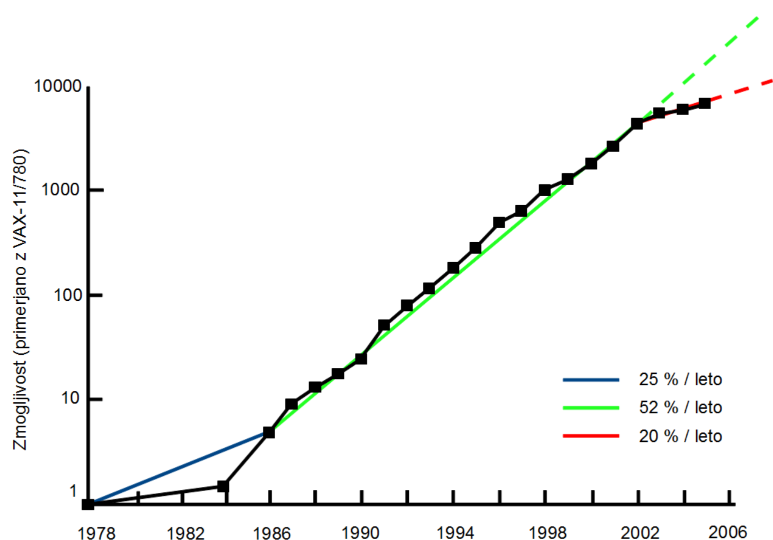
Implementacija nevronske mreže na GPE

Za začetek je bilo potrebno preveriti, kateri program bi bil primeren za izvajanje na grafični napravi. Najprimernejši programi so namreč tisti, ki zahtevajo veliko računsko zmogljivost in malo število prenosov podatkov. Med klasične probleme preračunavanja velike količine podatkov spadajo tudi nevronske mreže, zato smo se odločili za implementacijo algoritma vzvratnega razširjanja napake na grafični napravi. Ena izmed najpomembnejših optimizacij je način implementacije vzporednosti algoritma. Tako sta nastali dve različici, kjer je zadnja precej hitrejša od prvotne, vendar sta zaradi pomembnih vidikov optimizacije opisani obe. Hitrejša različica, ki sicer uporablja skoraj vse optimizacije omenjene v nadaljnih poglavjih, je podrobneje opisana v razdelku 4.8.

4.1 Razlogi za implementacijo na GPE

V današnjem času so glavni problem upad povečanja zmogljivosti poimenovali "the brick wall" [3], ki je seštevek preprek iz preteklosti. Najprej se je pojavil "power wall", ko so bili tranzistorji poceni, a je bila energija draga. Na čip so tako lahko natisnili več tranzistorjev, kot so imeli energije za vklop vseh. Za tem se je pojavila prepreka "memory wall". Ukaza load in store sta bila zelo počasna, a množenje je bilo hitro. Dostop do pomnilnika je bil zato nekaj večdesetkrat počasnejši od aritmetičnih operacij. Kot zadnji se je pojavil "ILP wall", kar lahko interpretiramo kot problem predolgega cevovoda.

Zaradi omenjenih preprek se je zmogljivost CPE iz 52 %/leto med leti 1985 in 2002 zmanjšala na manj kot 20 %/leto do leta 2006 (slika 4.1). Pri



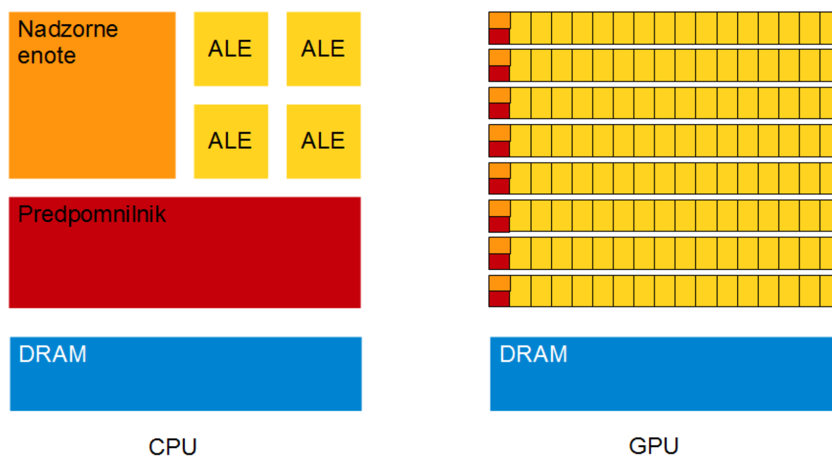
Slika 4.1: Prikaz rasti zmogljivosti za CPE in GPE. Povzeto po [13].

GPE ostaja zmogljivost še vedno okoli 50 %/leto. Kljub prepreki Moorov zakon še vedno velja. Ves razvoj se namreč usmerja iz kompleksnih enojedernih procesorjev na enostavne večjederne procesorje in tako lahko v prihodnosti pričakujemo ekonomsko upravičljiv čip, na katerem bo mnogo enostavnih procesorjev. Prednost takšnih procesorjev na istem čipu bo zelo majhen dostopni čas in zelo velika pasovna širina.

Hitrost CPE že dolgo poskušajo izboljšati s predikcijskimi tabelami, daljšim cevovodom, predpomnilnikom, hitrejšim izvajanjem posamezne stopnje cevovoda ipd. Z drugimi besedami rečeno so CPE usmerjene v kontrolo pretoka (flow control) in predpomnjenje podatkov (data-caching), medtem ko GPE vsebujejo večje število procesorjev in so usmerjene bolj v intenzivno preračunavanje podatkov. Shematično to ponazarja slika 4.2.

Več enostavnih procesorjev znotraj čipa sicer ne omogoča dobrih predvidevanj skokov, kazni so ob zgrešitvi velike. Vendar se to odpravlja z veliko številčnostjo teh procesorjev. Če predpostavimo, da pri nekaterih procesorjih pride do zgrešitve in vsi hkrati dostopajo do glavnega pomnilnika, potem nam ni potrebno izvesti dostop do pomnilnika za vsakega posebej, ampak lahko to izvedemo z enim branjem za več procesorjev hkrati. To posledično omogoča tudi širina podatkovnega vodila, ki pri grafičnih napravah omogoča veliko večjo prepustnost, kot pri CPE. Leta 2007 so grafične naprave dosegale prepustnost vodila malenkost več kot 100 GB/s, medtem ko CPE-ji manj kot 20 GB/s.

Podobno velja za aritmetične operacije. Več procesorjev hkrati dosega več



Slika 4.2: Shematični prikaz zasedenost čipa CPE in GPE z ALE, predpomnilnikom, pomnilnikom in ukazovnimi enotami.

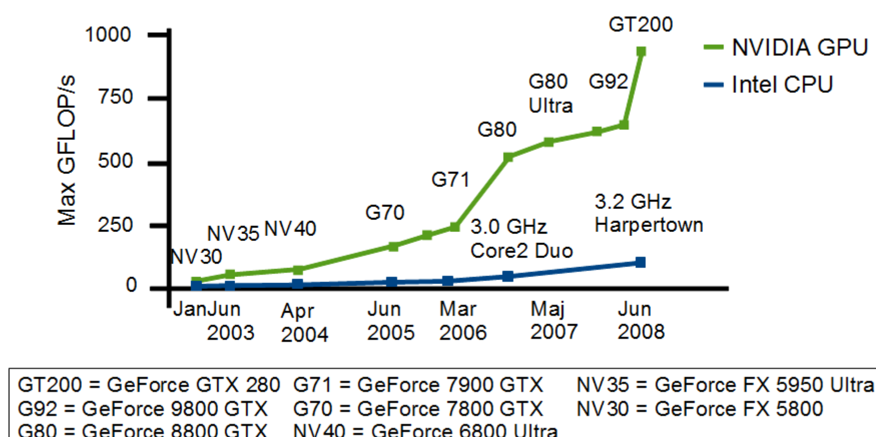
operacij v plavajoči vejici na sekundo (FLOPS/s) kot običajni CPE. Razmerje je približno 1200 GFLOPS/s proti manj kot 100 GFLOPS/s pri najboljših CPE. Slika 4.3 prikazuje maksimalno GFLOPS/s za GPE in CPE.

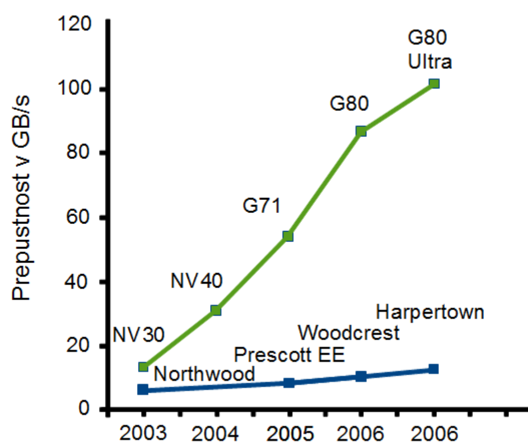
Razlogov za implementacijo na GPE je sicer še precej več, med pomembnejša bi lahko uvrstil še enostavnost razširitve kode na več GPE in programiranje v visokonivojskem programskem jeziku. Kot zanimivost lahko omenim še največjo prepreko, s katero se srečujejo v superračunalništvu – to je namreč poraba električne energije. Z ekstrapolacijo obstoječe arhitekture in tehnologije bi po ocenah podjetja Cray za en exaflop potrebovali 100 MW samo za vklop. Če bi primerjali porabo CPE z GPE, bi ugotovili, da grafične naprave omogočajo 20 do 40-krat več FLOPS/vat kot CPE.

4.2 Priprava zbirke podatkov

Za učenje mreže in testiranje sem uporabil podatkovno zbirko razpoznavanja črk [34] z 20000 primeri, 16 atributi in 26 razredi klasifikacije. Podatkovna zbirka je bila generirana iz dvajsetih različnih pisav in vsaka črka znotraj teh pisav je naključno popačena. Opis posamezne popačene črke je shranjen v 16-ih numeričnih atributih, kjer lahko vsak obsega cela števila od 0 do 15. Kaj pomeni posamezen atributov podatkovne zbirke, je podano v dodatku B.

Omenjena podatkovna zbirka je najprej pretvorjena v binarno obliko. Vsak atribut zato nadomestimo z binarno vrednostjo le-tega, kar pomeni, da imamo skupno 64 atributov, ki določajo en primer. Razred klasifikacije je določen z





Slika 4.3: Graf hitrosti preračunavanja in prepustnosti vodila za GPE in CPE. Povzeto po [26].

zmagovalno strategijo, 26 razredov zato pomeni 26 izhodnih atributov, kjer zmaga nevron, ki doseže največjo vrednost. Zapis spremenjene podatkovne zbirke je tako sestavljen iz "Y, X"-prve vrstice zapisa, kjer število Y pove, koliko je izhodnih nevronov, in X, ki pove, koliko je vhodnih nevronov. Število skritih nevronov je določeno znotraj programa in ni vezano na samo podatkovno zbirko. Vsaka nadaljna vrstica predstavlja po en primer iz zbirke podatkov, kjer se prvih Y števil nanaša na razred in zadnjih X števil na vrednosti atributov.

4.3 Način obdelave podatkov

Nadalje je implementacija napisana ločeno za CPE in GPE. CPE verzija uporablja le eno nit in zaporedno obdeluje vsak naslednji primer. Za GPE verzijo se določi število niti in blokov, s katerimi se zažene jedra na napravi. Vsaka nit obdeluje svoj podatek vendar z enakimi funkcijami kot ostale niti znotraj bloka. S tem se upošteva t. i. SIMD arhitekturo naprave.

V vseh implementacijah se podatki oz. uteži nahajajo v poljih oz. matrikah. Preračunavanje in izvajanje algoritma je zato množenje matrik. Kljub temu, da CUDA omogoča pisanje programske kode gostitelja v C++, sem se odločil za C zaradi primerljivosti testiranja med CPE in GPE. Posledično ni uporabljenih nobenih struktur kompleksnejših od polj. Podatki relevantni za izvajanje nevronske mreže se tako nahajajo:

- bodisi v poljih vhodnih, skritih ali izhodnih nevronov
- bodisi v poljih uteži med posameznimi nivoji nevronov.

Vsi podatki so v enojni natančnosti (float) in indeksi v celoštevilski obliki (integer). Pri trenutnih grafičnih napravah na trgu tipa GeForce 8 in 9, le-te ne omogočajo računanje v dvojni natančnosti, temveč samo v enojni. Računanje omogočajo grafične naprave družine GT200 in novejše, vendar so zelo počasne. Pri Nvidii obljubljajo, da se bo situacija bistveno spremenila s prihodom Fermija in bo računanje v dvojni natančnosti do okoli 8-krat hitrejše kot na predhodnjih napravah, kar je podrobneje razloženo v poglavju 1.3.1.

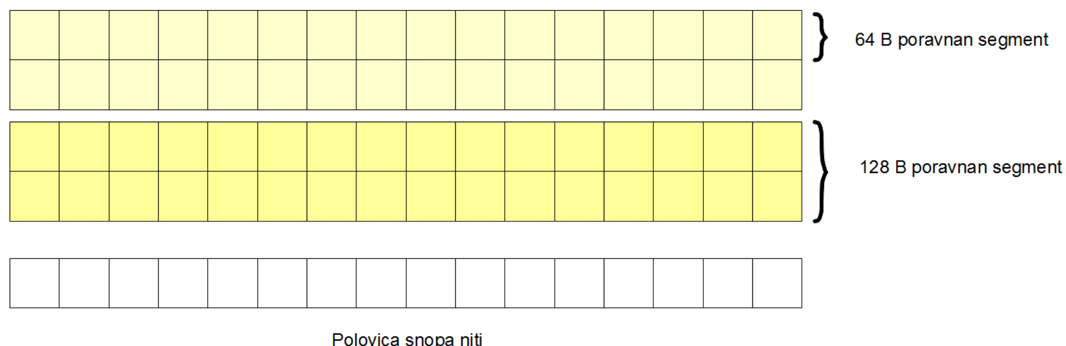
4.4 Način dostopa do podatkov

Med najpomembnejše optimizacije sodi dostop posamezne niti do globalnega oz. deljenega pomnilnika.

4.4.1 Dostop do globalnega pomnilnika

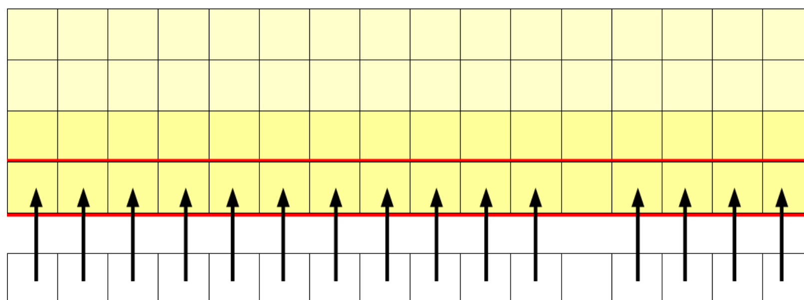
Branje in pisanje v globalni pomnilnik se za niti znotraj polovice snopa izvede z enim prenosom, ko so izpolnjeni določeni pogoji. Način takega dostopanja do pomnilnika se imenuje vezan način dostopanja (coalesced). Za razumevanje pogojev moramo na globalni pomnilnik gledati kot na segmente poravnanih 16 oz. 32 bitnih besed. Shematično je vezan način dostopa do pomnilnika prikazan na sliki 4.4, kjer se dostopa do 32 bitnih besed. Globalni pomnilnik je prikazan kot vrste 64 bajtnih poravnanih segmentov. Dve vrsti enakih barv

predstavljata 128 bajtno poravnavo segmentov. Dostop polovice snopa do podatkov je nakazan na dnu slike.



Slika 4.4: Segment linearnega pomnilnika in niti v polovici snopa.

Pogoji za vezan dostop se razlikujejo pri napravah različnih računskih zmogljivosti. Najstrožja omejitev velja za starejše naprave z računsko zmogljivostjo 1.0 in 1.1. Na teh napravah mora k-ta nit v polovici snopa dostopati do k-te besede v poravnanem segmentu. Pri tem ni omejitev, da vse niti dejansko sodelujejo. Grafično to prikazuje slika 4.5, kjer je z rdečim pravokotnikom prikazano branje iz pomnilnika. Vse niti, razen ene, ki ne bere, dostopajo do podatkov.



Slika 4.5: Vezan način dostopa, kjer vse niti razen ene dostopajo do besed znotraj segmenta.

Opisan dostop se izvede z enim 64 bajtnim prenosom. Kljub nesodelujočim nitim se prenesejo vsi podatki znotraj tega segmenta. V primeru, da bi bili dostopi niti znotraj rdečega pravokotnika permutirani, bi se na napravah računske zmogljivosti 1.2 ali novejše še vedno izvedel en prenos, medtem ko bi se pri starejših izvedlo 16 zaporednih prenosov.

Pogoj so pri napravah 1.2 in novejših še malce omilili. Te naprave namreč omogočajo tudi delne prenose npr. 32, 64 in 128 bajtne, kar pri zamaknjenem 64 bajtnem segmentu še ne pomeni 16 zaporednih prenosov, ampak 64 bajtni in 32 bajtni prenos oziroma dva 32 bajtna prenosa.

Iz omenjenega sledi, da morajo biti podatki, ki jih potrebujejo niti znotraj polovice snopa, skupaj v pomnilniku. Pri uporabi struktur je zato zelo pomembno zavedanje, kako se podatki v struktri dejansko nahajajo v pomnilniku. Najbolj naravno je teženje k celovitosti podatkov, s čimer do različnih podatkov nekega elementa dostopamo preko istega odmika, npr data[i]. Dostop do različnih spremenljivk tega elementa je tako: data[i]→x, data[i]→y. Opisano lahko realiziramo kot program 4 – kar imenujemo polje struktur (Ar-

Program 4 Polje struktur (Array of Structures - AoS).

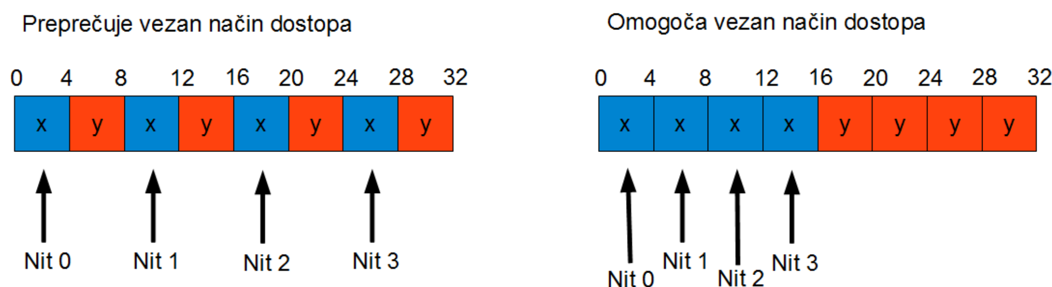
```
struct S {
 float x;
 float y;
};
struct S data[N];
```

ray of Structures - AoS). Omenjeni strukturi enakovredna je struktura polj (Structure of Arrays - SoA), prikazana v programu 5. Tu gre za strukturo,

Program 5 Struktura polj (Structure of Arrays - SoA).

```
struct S {
 float x[N];
 float y[N];
};
struct S data;
```

ki ima definirano polje za vsako spremenljivko v prejšni strukturi. Dostop do podatkov takšne strukture je tako: $data \rightarrow x[i]$ in $data \rightarrow y[i]$. V primerjavi med AoS in SoA je bistvena razlika v fizični lokaciji hranjenja podatkov za spremenljivki x in y. Pri AoS se x in y izmenjujeta po naraščajočih indeksih lokacij, pri SoA pa si najprej sledijo vsi x-i in nato vsi y-i po naraščajočih indeksih lokacij, kar prikazuje slika 4.6.



Slika 4.6: Dostop do podatkov pri AoS in SoA.

Pri SIMD arhitekturi bodo ob prihodu ukaza za dostop do spremenljivke x vse niti začele z branjem iz globalnega pomnilnika in v primeru, da so indeksi niti pravilno poravnani z indeksi pomnilnika in je uporabljena struktura polj, bo branje v vezanem načinu dostopanja do podatkov in s tem en dostop za polovico snopa.

V programu algoritma vzvratnega razširjanja napake vsaka nit uporablja več različnih polj. Polja za vhodne, skrite in izhodne nevrone so enorazsežna. Ob uporabi B-blokov, N-niti, V-skritih nevronov, S-skritih nevronov in I-izhodnih nevronov se vsaki niti dodeli svoje polje dolžine $V+1$ za vhodne nevrone, $S+1$ za skrite nevrone in I za izhodne nevrone. Skupna dolžina posameznih polj za vse niti je tako:

1) $B * N * (V + 1)$ – za polje vseh vhodnih nevronov,

2) $B * N * (S + 1)$ – za polje vseh skritih nevronov,

3) $B * N * I$ – za polje vseh izhodnih nevronov.

Velikost polja pri vhodnih in skritih nevronih se poveča za 1 zaradi bias-a, kamor shranimo konstanto +1.

Dostop do vseh polj je v vezanem načinu, če je število niti mnogokratnik števila 16. Najlažje način dostopa v vezanem načinu opišemo s primerom, za lažje razumevanje pa uporabimo le štiri niti in dva bloka, skupno torej osem niti. Vsaka nit dostopa do treh vhodnih podatkov, kjer je četrti podatek konstanten vhod +1, uporabljen za bias. Dostop do vhodnega nivoja nevronov bi izgledal takole:

$$
\begin{array}{cccc}
n_0[0], & n_1[1], & n_2[2], & n_3[3], \\
n_0[4], & n_1[5], & n_2[6], & n_3[7], \\
n_0[8], & n_1[9], & n_2[10], & n_3[11], \\
n_0[12], & n_1[13], & n_2[14], & n_3[15], \\
\\
n_4[16], & n_5[17], & n_6[18], & n_7[19], \\
n_4[20], & n_5[21], & n_6[22], & n_7[23], \\
n_4[24], & n_5[25], & n_6[26], & n_7[27], \\
n_4[28], & n_5[29], & n_6[30], & n_7[31].
\end{array}
\tag{4.1}
$$

Oznaka $n_i[j]$ pomeni nit z indeksom i, ki dostopa do lokacije z indeksom j. Vrstice v tabeli lahko obravnavamo kot indeks, do katerega podatka dostopa posamezna nit, npr. v tretji vrstici vse niti dostopajo do tretjega vhodnega podatka podanega na zaporednih naslovih 8, 9, 10 in 11. Na stolpce pa lahko gledamo kot na dostop posamezne niti do fizične lokacije pomnilnika. Prvi del pomnilnika do indeksa 15 je uporabljen za hranjenje podatkov za niti iz prvega bloka. Pri ostalih blokih je postopek enak, le indeksi niti so drugačni in lokacija dostopa se začne z odmikom [število niti znotraj bloka] $*$ [število podatkov na nit] $*$ [indeks bloka]. V tabeli je prikazan tudi drugi blok, kjer se podatki zanj začnejo na lokaciji 16 in končajo na 31.

Zgornja implementacija ločevanja podatkov po posameznih blokih je zelo intiuitivna, ker nam že ločuje podatke na bloke in tako omogoča lažje razhroščevanje za posamezne niti znotraj blokov. Vendar je boljša implementacija, če gledamo na bloke in niti, kot samo na "niti". Podatki se tako med seboj bolj prepletajo, vendar so vsi podatki istega tipa soležni:

$$
\begin{array}{cccc}
n_0[0], & n_1[1], & n_2[2], & n_3[3], \\
n_4[4], & n_5[5], & n_6[6], & n_7[7], \\
\\
n_0[8], & n_1[9], & n_2[10], & n_3[11], \\
n_4[12], & n_5[13], & n_6[14], & n_7[15], \\
\\
n_0[16], & n_1[17], & n_2[18], & n_3[19], \\
n_4[20], & n_5[21], & n_6[22], & n_7[23], \\
\\
n_0[24], & n_1[25], & n_2[26], & n_3[27], \\
n_4[28], & n_5[29], & n_6[30], & n_7[31].
\end{array}
\tag{4.2}
$$

Vezan način dostopa se precej bolj zakomplicira pri dvorazsežnih poljih. V tem primeru je ob morebitnih uporabah for-zank za premikanje po polju obvezno pravilno razvitje dvorazsežnega polja v enorazsežno tako, da z notranjo zanko niti vedno jemljejo zaporedne elemente, nato vse naslednje itn. V moji implementaciji je pri enorazsežnih poljih uporabljena prva implementacija (tabela 4.1), medtem ko pri dvorazsežnih poljih zadnja (tabela 4.2) zaradi nadaljne obdelave podatkov oz. seštevanja delnih sprememb uteži.

4.4.2 Dostop do deljenega pomnilnika

Malo manj pomemben kot način dostopa do glavnega pomnilnika je dostop do deljenega. Deljeni pomnilnik je razdeljen v šestnajst različnih modulov pomnilnika, imenovanih bank, ki so lahko dostopani istočasno. V primeru, da več niti hkrati dostopa do iste banke, pride do t. i. konflikta banke. Prenos se pri tem izvrši za maksimalno število bank, ki niso v konfliktu. Posebna izjema nastane, ko vse niti dostopajo do iste banke. Tedaj se naredi razpršeno oddajanje (broadcast), ki omogoča sočasno branje vseh niti te banke.

Deljeni pomnilnik je namenjen zgolj začasnemu hranjenju podatkov. S hitrostjo enako kot dostop do registrov, če ni nobenih konfliktov, je namenjen kot prostor za lokalno obdelavo podatkov in s tem zmanjševanju dostopa do glavnega pomnilnika. V kodi nevronskih mrež, se deljeni pomnilnik uporablja predvsem kot nadomestek za registre, kar je podrobneje opisano v razdelku 4.6, in v primeru večkratnega dostopanja do istega podatka. Tedaj se ta podatek začasno shrani v deljeni pomnilnik.

Ker je ogromno dela s polji nevronov, bi bila med boljšimi implementacijami, če bi imeli vsa ta polja v deljenem pomnilniku. To zaenkrat zaradi velikega števila niti in premajhnega pomnilnika žal še ni mogoče. Pri napravah z računsko zmogljivostjo 1.0 ima programer na voljo 16 KB deljenega pomnilnika na en multiprocesor. Če predpostavimo, da bi želeli imeti sočasno na voljo za obdelavo vsaj dva bloka, kjer ima vsak vsaj 64 niti, bi po preprostem izračunu dobili $16384/(2 * 64) = 128$ B oz. 32 podatkov z enojno natančnostjo na nit. Slabost te implementacije bi bila seveda ravno strojna omejenost s številom niti in količino podatkov, ki bi jo lahko hranili v deljenem pomnilniku.

4.5 Hitrost prenosa med pomnilniki

Pri uporabi grafične naprave je izrednega pomena zmanjševanje prenosa podatkov med pomnilniki. Najbolj kritičen je prenos med pomnilnikom naprave in pomnilnikom gostitelja. Hitrost prenosa podatkov na sistemu z grafično

napravo 9500GT je znotraj naprave za kar več kot 10x hitrejši od hitrosti med pomnilnikom naprave in gostiteljem. V določenih primerih se temu prenosu lahko izognemo tako, da napišemo jedro, ki na napravi shrani podatke v pomnilnik. Primer brisanja celotnega polja na napravi je prikazan v programu 2, kjer je omejitev hitrosti dostopa do pomnilnika naprave, namesto hitrosti prenosnega vodila med pomnilnikoma.

4.5.1 Seštevanje sprememb uteži

Vsaka nit, ki se izvaja na grafični napravi, potrebuje za svoje delovanje polja nevronov in uteži. Pri vzporedni paketni implementaciji se polje uteži dejansko posodobi šele po predstavitvi vseh učnih primerov. Zaradi hitrosti izračuna je uteži najbolje najprej sešteti in šele nato seštevek normalizirati in pomnožiti s stopnjo učenja. Pri seštevanju na grafični napravi je potrebno upoštevati še, da vse niti ne morejo hkrati pisati na iste lokacije glavnega pomnilnika, zato ima vsaka nit svoje polje sprememb uteži, ki se naknadno seštejejo.

Samo seštevanje se izkaže, da ni najbolj primerna operacija za izvajanje na grafični napravi, zaradi česar se je ta operacija sprva izvajala na CPE. Ta izvedba se je izkazala za precej slabo na napravah z veliko multiprocesorji. Na napravah z malo multiprocesorji je bila metoda relativno hitra kljub počasnemu podatkovnemu vodilu. Velika razlika se je pojavila pri preiskusu na napravi s 30-timi multiprocesorji, kjer je že prenos na CPE trajal približno toliko časa kot samo računanje.

V takratni implementaciji se je namreč vse posodobitve uteži za vse niti prenašalo na gostitelja. Pri 60-ih blokih in 256-ih niti bi pri velikostih 64 vhodnih, 40 skritih in 26 izhodnih nevronih to pomenilo polja v velikosti:

1) $60 * 256 * (64 + 1) * 40 * 4$ B $\doteq 160$ MB – za uteži iz vhodnega na skriti nivo;

2) $60 * 256 * (40 + 1) * 26 * 4$ B $\doteq 65$ MB – za uteži iz skritega na izhodni nivo;

3) $60 * 256 * 65 * 4$ B $\doteq 4$ MB – za vhodne nivoje;

4) $60 * 256 * 41 * 4$ B $\doteq 3$ MB – za skrite nivoje;

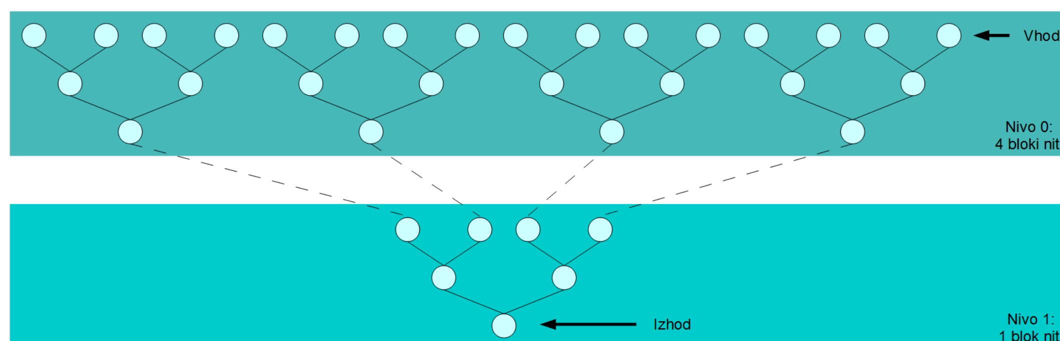
5) $60 * 256 * 26 * 4$ B $\doteq 2$ MB – za izhodne nivoje;

kar je okoli 234 MB shranjenih podatkov v globalnem pomnilniku.

Zaradi relativno počasnega prenosa med napravo in gostiteljem velja pravilo, da če se lahko opravi kakšna operacija, ki ne prinese bistvene pohitritve,

vendar zmanjša število prenosov med CPE in GPE, potem naj se jo raje opravi na GPE.

Vse nadaljnje implementacije so vsebovale različne načine vzporednega seštevanja polj uteži, vendar nobena ni bila tako učinkovita kot modificiran primer 'reduction' iz NVIDIA SDK. Reduction primer je prikaz seštevanja ogromnega polja v eno vrednost in je namenjen za prikaz različnih optimizacij in njihovih pohitritev. V celoti obsega sedem različnih jeder, kjer je zadnje najbolj optimizirano in uporabljeno za seštevanje uteži. Seštevanje deluje po principu drevesne strukture (slika 4.7), kjer vsaka nit sešteje nekaj vrednosti in na koncu vsak blok proizvede eno vrednost. V programski kodi je reduction



Slika 4.7: Drevesna struktura seštevanja.

primer spremenjen tako, da se opisani postopek izvede za vsak element polja uteži. Pri utežeh iz vhodnega na skriti nivo bi pomenilo, da se reduction izvede $65 * 40$-krat pri poljih v velikosti $60 * 256$ elementov.

Končni rezultat seštevanj so vsote istoležnih elementov preko vseh polj, ki se še vedno nahajajo v glavnem pomnilniku. Zaradi hranjenja uteži v bralnem pomnilniku le-tega ni mogoče spremeniti na grafični napravi, zato se na tem mestu izvede prenos sprememb uteži na gostitelja, kjer se uteži normalizira, pomnoži s stopnjo učenja in sešteje. Pri seštevanju je dodan še t. i. momentni člen iz enačbe 3.30 in upoštevano odstranjevanje uteži po enačbi 3.31.

4.5.2 Uporaba bralnih pomnilnikov

Med bralne pomnilnike spadajo vsi pomnilniki:

- pomnilnik konstant,
- pomnilnik tekstur,

- glavni pomnilnik.

Vendar sta le prva dva samo bralna in hkrati predpomnjena pomnilnika. Dostop do konstant je hiter kot dostop do registrov, če vse niti znotraj polovice snopa hkrati dostopajo do iste konstante in se le-ta nahaja v predpomnilniku. V nasprotnem primeru ob zgrešitvi, prenos traja enako kot branje iz glavnega pomnilnika.

Pri večkrat uporabljenih konstantah se zaradi hitrega dostopa do predpomnilnika splača hraniti konstante v tem pomnilniku. Mednje spadajo velikosti posameznih nivojev nevronov in tudi kakšne predizračunane vrednosti. Ker deluje branje konstant po principu razpršenega oddajanja, kjer vse niti hkrati berejo isto konstanto, ni primeren za hranjenje elementov polj. Za ta namen je uporabljen pomnilnik tekstur, v katerem so shranjene uteži sinaptičnih povezav med posameznimi nivoji nevronov. Pomnilnik tekstur je tudi predpomnjen, s čimer se zagotovi hiter dostop v primeru zadetka in zmanjša število branj iz teksturnega pomnilnika.

4.6 Povečevanje zasedenosti jeder

V določenih primerih je zelo pomembno, da pazimo, kako je napisana programska koda. Preveč uporabljenih spremenljivk bi pomenilo preveč dodeljenih registrov za hranjenje teh spremenljivk in posledično zmanjšanje zasedenosti (occupancy), ker vse niti ne bi imele dovolj registrov za svoje normalno delovanje. Pri napravah računske zmogljivosti 1.1 imamo na voljo 8192 registrov znotraj multiprocesorja. Če želimo zagnati multiprocesor z maksimalnim številom aktivnih niti to je 768, potrebujemo vsaj 10 prostih registrov na nit, da bo zasedenost 100 %. Kakršna koli uporaba večjega števila registrov bi pomenila zmanjšanje zasedenosti. Zgornjo mejo uporabljenih registrov na nit sicer lahko nastavimo s parametrom `--maxrregcount=N`, vendar je s tem pogojena hitrost izvajanja.

Število potrebnih registrov lahko zmanjšamo z uporabo deljenega pomnilnika za shranjevanje spremenljivk. Ker do deljenega pomnilnika dostopamo z odmikom, se seveda postavi vprašanje, če je to smiselno izvesti zaradi nekaj dodatnih računskih operacij za dostop do podatkov. Izkaže se, da je na napravi s štirimi multiprocesorji verzija z maksimalno uporabljenimi 14-timi registri na nit in deljenim pomnilnikom za približno 10 odstotkov hitrejša od verzije z 19-timi registri na nit brez deljenega pomnilnika. Pri prvi je bila zasedenost 67 odstotna, medtem ko pri drugi le 33 odstotna. Razlika bi se pojavila pri napravah z računsko zmogljivostjo 1.2 in novejšimi, ki imajo dvakrat več

registrov kot naprave 1.1 in 1.0. V tem primeru bi bila zasedenost okoli 100 odstotna proti 75 odstotni za uporabljenih 19 registrov na nit.

Večja zasedenost jedra ne pomeni vedno povišano zmogljivost. Jedra, ki so omejena s pasovno širino, lahko dosežejo povečano zmogljivost zaradi boljšega skrivanja latenc dostopov do pomnilnika. Vendar, če ozko grlo ni pasovna širina, je to najverjetneje hitrost preračunavanja. V takih primerih povečanje zasedenosti jeder ne bo povečalo zmogljivosti. Po drugi strani ima prirejanje programske kode za povečevanje zmogljivosti včasih celo več slabih lastnosti. Lahko pride do dodatnih ukazov, prelivov v lokalni naslovni prostor, vejitve . . . V primeru nevronskih mrež je največja slabost preračunavanje odmikov in razumljivost kode. V programu 6 je prikaz funkcij, kjer je prva veliko bolj razumljiva, vendar porabi več registrov za svoje delovanje. Druga funkcija pa je prikaz kode z manj potrebnimi registri in uporabljenim deljenim pomnilnikom za hranjenje nekaterih spremenljivk.

Pri slednji je sprogramirano tako, da se preračunavanje odmika izvrši čim manjkrat. Zato notranja zanka dostopa neposredno do naslova `j[threadIdx.x]`, medtem ko je za zunanjo zanko, ki se izvede manjkrat, naslov preračunan iz `threadIdx.x+blockDim.x`.

Med možne optimizacije z zmanjševanjem registrov lahko omenimo še pretvorbo for-zank v while-zanke z dodanimi pogoji. V funkciji `optimal` v programu 6 pri trenutni for-zanki potrebujemo polje dolžine števila vseh niti znotraj bloka, ker vsaka niti hrani svoj števec v tem polju. Ker so vrednosti vseh števcev znotraj polja v nekem trenutku enake, je to zelo neizkoriščen prostor, vendar pri dostopanju ni nikakeršnih konfliktov in je zato hitro dostopan. Velikost polja pri while-zanki se da zmanjšati do te mere, da obstaja samo ena spremenljivka kot števec za vse niti. Če imamo obstoječo for-zanko:

```
for(j[threadIdx.x] = 0; j[threadIdx.x]<const; j[threadIdx.x]++) {
  array[lokacija*j[threadIdx.x]]  = j[threadIdx.x];
}
```

se jo da enostavno pretovoriti v:

```
if(threadIdx.x == 0) var[0] = 0; __syncthreads();
while( var[0] < const ) {
  array[lokacija*var[0]]  = var[0];
  if(threadIdx.x == 0) var[0]++; __syncthreads();
}
```

Na ta način imamo spremenljivke za števce v polju `var[idx]`, kjer indeks `idx` pomeni različno spremenljivko. Zaradi razpršenega oddajanja podatkov

Program 6 Prikaz zmanjševanja zasedenih registrov.

```
__device__ void simple(float* fcudInputNeurons, int startD) {
  float  sum = 0.0f;
  int k,j;
  for(k = 0; k<cuHidN; k++) {
    for(j = 0; j<cuInNPlus1; j++) {
      sum += tex2D( dwInputToHiddenTex, k, j) *
        fcudInputNeurons[startD+blockDim.x*j];
    }
  }
}

__device__ void optimal(float* fcudInputNeurons, int startD) {
  float  sum = 0.0f;
  for(k[threadIdx.x+blockDim.x] = 0;
      k[threadIdx.x+blockDim.x]<cuHidN;
      k[threadIdx.x+blockDim.x]++) {
    for(j[threadIdx.x] = 0;
        j[threadIdx.x]<cuInNPlus1;
        j[threadIdx.x]++) {
      sum += tex2D( dwInputToHiddenTex,
                    k[threadIdx.x+blockDim.x],
                    j[threadIdx.x]) *
      fcudInputNeurons[startD+blockDim.x*j[threadIdx.x]];
    }
  }
}
```

v deljenem pomnilniku, ko do njega hkrati dostopajo vse niti, ne bo prišlo do konfliktov pri branju, pri pisanju pa piše le prva nit celotnega bloka. Število uporabljenih registrov s tem ostane pri 14, vendar se malenkost poslabša čas izvajanja iz približno 14.3 s na 15 s pri 100 epohah.

4.7 Primerjava implementacije algoritma za CPE in GPE

Kot že omenjeno sta kodi za CPE in GPE implementaciji zelo podobni. Obe sta napisani v programskem jeziku C z osnovnimi elementi brez uporabljenih kompleksnih stuktur. Različica za CPE tako ne uporablja kakšnih naprednih elementov, kot so povezani seznami ali objekti, temveč samo polja, kot tudi GPE izvedba. S tem se doseže hitro izvajanje in karseda najboljšo primerljivost rezultatov med izvedbama. Programska koda je ločena v več funkcij:

- initalize,
- forwardPass,
- backwardPass,
- trainNetwork,
- pomožne funkcije.

Funkcija `initalize` vsebuje vse potrebne ukaze za vzpostavitev in začetno dodelitev polj. `ForwardPass` je funkcija preračunavanja nevronov v smeri od vhodnih vrednosti proti izhodnimi po enačbi 3.10. Funkcija `backwardPass` pa preračunavanje po enačbi 3.11 v obratni smeri. Jedro programa je učenje nevronske mreže v funkciji `trainNetwork`, kjer se znotraj prve zanke predstavi celotno učno množico in šele nato posodobi uteži s klicem `updateWeights`. Pri posodobitvi uteži se le-te izračuna po enačbi 3.30, kjer se upošteva še odstranjevanje uteži iz enačbe 3.31. Med pomožne funkcije spadajo funkcije kot so izračun sigmoidalne funkcije pri nekem parametru, njenega odvoda ipd.

Programska koda, namenjena izvajanju za GPE, vsebuje podobne funkcije kot CPE različica, s tem da se bistvene funkcije začnejo s predpono `cuda`. Glavni funkciji `backwardPass` in `forwardPass` za GPE sta praktično enaki kot CPE različici, vsa umetnost SIMD načina programiranja pa se skriva v preračunavanju odmikov podatkov. Večja razlika je znotraj funkcij `initalize` in `trainNetwork`. Prva je za več kot dvakrat daljša zaradi potrebnih dodelitev

polj na grafični napravi in gostitelju, druga pa se razlikuje v tem, da znotraj for-zanke ne predstavimo vseh učnih primerov na gostitelju, temeč to naredi klicoča funkcija na napravi.

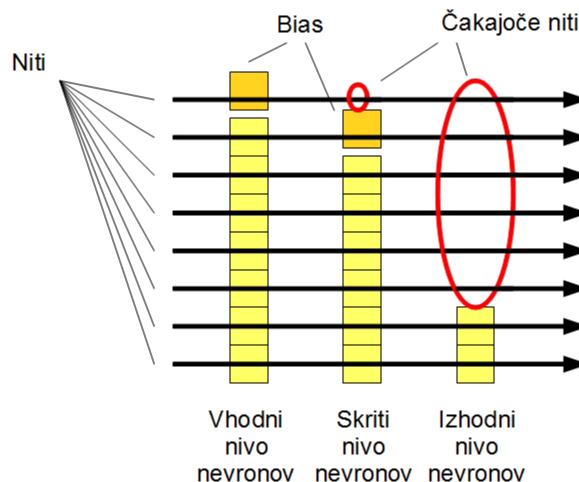
Pri opisani implementaciji je še nekaj prostora za dodatne izboljšave. Ker hranimo vse podatke v glavnem pomnilniku, lahko uporabimo deljenega kot predpomnilnik. S tem se dodatno zmanjša število dostopov. Podobno lahko storimo, če jedra zaženemo z dovolj velikim številom niti tako, da vsaka nit obdeluje le en podatek. V tem primeru nam ni potrebno čistiti polja in prištevati delta uteži že obstoječim vrednostim. Omenjeni izboljšavi prineseta nekaj odstotno dodatno pohitritev, vendar programska koda postane še slabše berljiva. Veliko večjo pohitritev se doseže, če uporabimo drugačen način implementacije, kar je opisano v nadaljevanju.

4.8 Hitrejša implementacija algoritma

V času pisanja diplomskega dela sem na novo napisal GPE različico, ki v večji meri izrablja deljeni pomnilnik. V tej izvedbi vsak blok obdeluje svoj podatek. Pri tem ostane željena fleksibilnost skaliranja na poljubno število multiprocesorjev, sočasna obdelava več podatkov in dodana je vzporedna obdelava enega primera znotraj bloka. Uporabnik nima več nadzora nad številom niti v bloku, saj so le-te določene kot maksimalno število nevronov+1 na katerem koli nivoju. Vsaka nit zato obdeluje le en nevron. Grafično delovanje bloka prikazuje slika 4.8. V programski kodi je tokrat veliko več vejitvenih stavkov, saj z njimi niti, ki nimajo na voljo nevrona za preračunati, počivajo. Najbolj optimalna nevronska mreža bi vsebovala (n-1) vhodnih, (n-1) skritih in n izhodnih nevronov.

Z obdelovanjem enega učnega primera v bloku potrebujemo v primerjavi s prejšno izvedbo precej manj podatkov na blok. Dinamično je deljeni pomnilnik zato zaseden v velikosti (vhodnih+1 + skritih+1 + izhodnih∗2) nevronov in se zato vsa preračunavanja algoritma shranijo vanj. Dostopi so brez konfliktov bank, izredno majhna poraba deljenega pomnilnika pa nakazuje na možnost še dodatne vzporednosti, kjer bi en blok preračunaval več primerov sočasno. Ta način optimizacije je opisan v nadaljevanju in se izkaže za najhitrejšo izvedbo. S tem se znebimo nepotrebnega večkratnega branja in shranjevanja sprememb uteži. Do globalnega pomnilnika se sicer dostopa le v primeru branja tekstur, konstant in zapisovanja novih uteži.

Dostop do globalnega pomnilnika je tako, kot je opisano v poglavju 4.4.1, tudi tukaj v vezanem načinu. Posebnost pri novejši implementaciji je, da vezan



Slika 4.8: Prikaz izvajanja niti. Nit, ki seka posamezni nivo nevronov, nevron na tem nivoju obdela, sicer pa čaka.

dostop ni odvisen od števila niti, ampak od velikosti polja uteži na posameznem nivoju. Da bi bil dostop že v osnovi v vezanem načinu, bi velikost polja morala biti deljiva s številom 16. V primeru, ko ni, ga moramo razširiti tako, da je. Vsak blok tako dostopa do podatkov, ki se začnejo na indeksu deljivem s 16.

Na sami grafični napravi imamo tako toliko polj sprememb uteži, kolikor je blokov. Izrazito nevzporeden problem "reduction", seštevanja polj v eno polje, je posledično odpravljen in se namesto njega izvede prenos vseh polj sprememb uteži na gostitelja. Tam se spremembe uteži pomnoži s stopnjo učenja, prišteje momentni člen in člen za odpravljanje uteži.

Programska koda nove različice se od stare ločuje po priponi `Adv` pri klicu funkcije. Vsebuje zgolj eno jedro `cudaRunTrainEpochAdv` in uporablja le 8 registrov na nit. To pomeni, da je programska koda precej lažje berljiva, kot prikazuje program 7.

V trenutno najhitrejši različici programa nevronskih mrež za GPE en blok zaporedno obdeluje po 4 primere. V programu 7 se znotraj for-zanke izvedejo 4 izračuni različnih vsot, nato 4 izračuni sigmodialnih funkcij, katerih vrednost se shrani v ločeni del deljenega pomnilnika, in na koncu še 4 prireditve vrednosti 1 za bias. Podobno se izvede še na ostalih nivojih nevronske mreže. Takšna implementacija sedaj porabi 16 registrov na nit in ima precej nižjo zasedenost kot prej. Prednost ta izvedbe je v tem, da zaradi zaporednega pisanja v deljeni pomnilnik niti ne čakajo pri ponovnem branju takoj po pisanju iz iste lokacije. Zaporedno se namreč obdela 4 različne lokacije in nato šele nadaljuje s prvo.

Program 7 Izračun vrednosti skritega nivoja nevronov.

```
if( threadIdx.x < cuHidN ) {
  float sum = 0.0f;
  for(int k=0; k<cuInNPlus1; k++) {
    sum += tex2D(dwInputToHiddenTex, threadIdx.x, k)
           * inputNeurons[k];
  }
    hiddenNeurons[threadIdx.x+cuInNPlus1] = sigmoidFunction(sum);
}else if( threadIdx.x == cuHidN ) { // bias
  hiddenNeurons[threadIdx.x+cuInNPlus1] = 1.0f;
}
__syncthreads();
```

Ker blok zaporedno obdeluje več primerov, je deljeni pomnilnik sedaj zaseden v velikosti (vhodnih+1 + skritih+1 + izhodnih$*$2)$*$4 nevronov. Na hitrosti se tako še vedno pridobi pri dostopanju do globalnega pomnilnika. Pred zapisom sprememb uteži v globalni pomnilnik se le-te sešteje in nato naredi en dostop namesto štirih.

Poglavje 5

Rezultati poskusov

Meritve so bile opravljene na dveh ločenih sistemih. Prvi je vseboval dvojederni procesor AMD Athlon^{TM} 64 X2 4000+, 2 GB pomnilnika tipa DDR2, matično ploščo Asus M2N-E SLI in grafično napravo Gainward GeForce 9500 GT z 1 GB pomnilnika. Drugi sistem je precej novejši in je vseboval štirijederni Intelov Core^{TM}2 Quad Q9300 procesor, 4 GB DDR2 pomnilnika, matično ploščo Asus P5KC in grafično napravo Zotac Geforce GTX 280 z 1 GB pomnilnika. Oba sistema vsebujeta 64-bitno različico Windows 7 in vsi rezultati merjeni na CPE so merjeni le na enem jedru CPE.

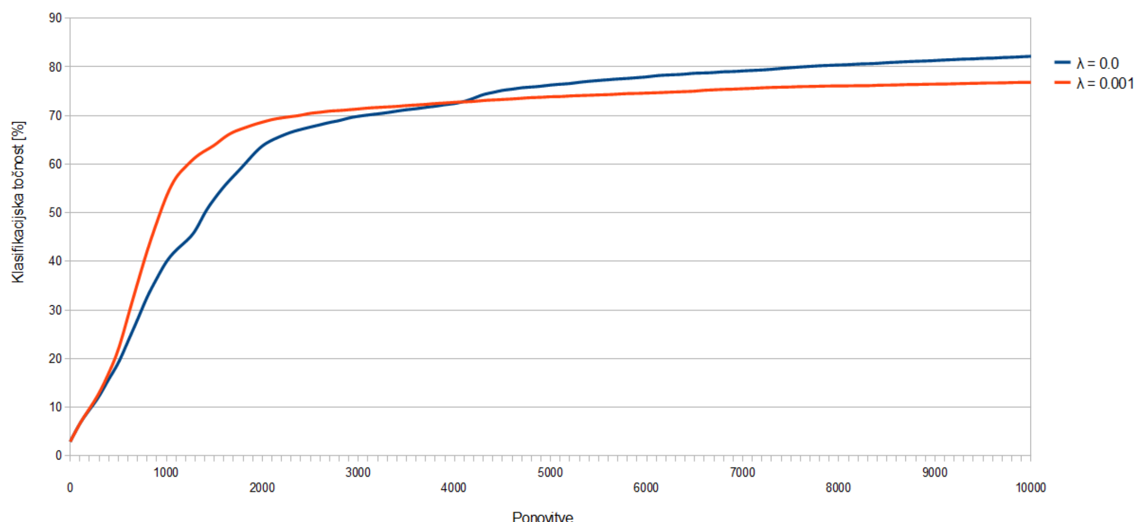
5.1 Lastnosti in uspešnost nevronske mreže

Med pisanjem diplomskega dela so nastale tri različice nevronskih mrež. Prva je napisana za CPE in nima nobenih strogih omejitev za izvajanje. CPE izvedba se uporablja kot referenčna nevronska mreža, s katero primerjamo rezultate ostalih mrež. Druga in tretja sta napisani za GPE in sta poimenovani z GPEv1 in GPEv2 (v nadaljevanju v1 oz. v2).

V1 različica za svoje "učinkovito" delovanje potrebuje veliko neodvisnih učnih primerov. Omejitev pri v1 je, da je število učnih primerov deljivih s produktom števila niti in številom blokov. Zaradi velike zasedenosti pomnilnika naprave je pomembno še, da pazimo na število dodeljenih blokov in niti ter jih imamo hkrati dovolj za učinkovito skrivanje latenc. Za razliko od naprednejše različice lahko uporabnik tukaj sam določa število potrebnih niti in blokov, saj je število niti pri izvedbi v2 določeno z mrežo.

Pri v2 izvedbi so omejitve malenkost drugačne. Ta različica ne potrebuje veliko število neodvisnih primerov za "učinkovito" izvedbo, temveč bolj kompleksno nevronsko mrežo. Jedro se izvede najhitreje, ko je število nevronov

na vseh nivojih približno enako in v območju od 128 do 512. 512 je pri trenutni implementaciji tudi dodatna omejitev, saj je to največje število niti na blok pri napravah računske zmogljivosti 1.1. Zadnjo omejitev se da razširiti na večrazsežno polje niti.



Slika 5.1: Točnost nevronske mreže pri večjem številu ponovitev in dodanem odstranjevanju uteži.

Kot že omenjeno v poglavju 4.2, originalna podatkovna zbirka vsebuje 20000 primerov, 16 atributov in 26 razredov, na katerih je avtor zbirke Slate et al [34] z uporabo t. i. “Holland-style” adaptivnega klasifikatorja na 16000 učnih primerih dosegel okoli 80.8 % točnost na neodvisnih testnih podatkih. Boljšo klasifikacijsko točnost 85.8 % sta na enako številčnih učnih in 4000 testnih podatkih dobila Philip in Joseph [31] z uporabo manj intenzivnega Bayesovega klasifikacijskega algoritma, z uporabo RBFN mrež pa so Daqi et al [7] dosegali 91.85 % točnost. Enega izmed najboljših rezultatov na tej podatkovni zbriki je dosegel Fogarty [9] pri 95.7 % klasifikacijski točnosti z uporabo manjše spremembe algoritma k-najbližjih sosedov, 96.18 % točnost pa sta dosegla Erkmen in Yildirim [8] z uporabo verjetnostnih nevronskih mrež.

Moja osnovna nevronska mreža za problem razpoznavanja črk je imela 64 vhodnih, 63 skritih in 26 izhodnih nevronov. Nevroni vsakega nivoja so povezani z vsakim nevronom na naslednjem nivoju. Uteži se posodabljajo z vzvratnim razširjanjem napake. Učenje je potekalo na 15360 učnih primerih, 60 blokih, 64 nitih in pri 10000 epohah z uporabo paketnega učenja. Z uporabo opisane mreže sem pri CPE izvedbi online učenja dosegal čez 93 %

klasifikacijsko točnost na učni in testni množici, pri uporabi paketnega učenja pa okoli 82 % točnost na učni množici in 79 % točnost na testni množici, ki je nevronska mreža predhodno še ni videla. Za doseganje boljše točnosti bi bilo potrebno bolje nastaviti učne parametre, za primer paketnega učenja pa je tudi znano, da počasneje konvergira. Pri parametru λ=0.001 so rezultati malenkost drugačni. Nevronska mreža se hitreje nauči različne primere, a manj natančno. Algoritem je tu dosegel 76.8 % točnost na učni množici, na testni množici pa okoli 75 %. Dosežena točnost pri različnih epohah je za oba primera prikazana na sliki 5.1. Potrebno je še omeniti, da namen diplomskega dela ni bilo doseganje čim večje klasifikacijske točnosti, temveč čim bolj uspešna implementacija nevronske mreže na grafični napravi. Slednje sem dosegel z združevanjem podatkovne vzporednosti in vzporednosti preračunavanja znotraj enega primera.

Kot je že omenjeno je hitrost nevronske mreže delno odvisna tudi od števila vhodnih in izhodnih atributov. Zaradi konstantega števila teh atributov pri nekem realnem problemu, bi za določitev "optimalnega" števila vhodnih in izhodnih atributov in s tem posledično "optimalne" hitrosti delovanja potrebovali učno množico s primernim številom vhodnih in izhodnih atributov. Iskanje takšnih množic realnih problemov bi bila prezahtevna naloga, poleg tega pa namen diplomske naloge ni doseganje čim večje uspešnosti, temveč čim večjo pohitritev, zato sem se odločil za naključno ustvarjeno učno množico, katere namen je zgolj poiskati optimalno število atributov za najhitrejše delovanje.

5.2 Merjenje časa

Merjenje časa je realizirano s klicem števcev implementiranih v pomožnih programih CUDA SDK. Števec ustvarimo pred začetkom prvega klica funkcije neke implementacije (npr. CPU, GPEv1 ali GPEv2) in ustavimo po zadnjem klicu. Med samim merjenjem je število izpisovanj na zaslon minimalno, izračun posameznih točnosti pa se izvede po končanem merjenju.

Pri merjenju časa posamezne izvedbe merjenje vključuje vzpostavitev vseh potrebnih polj, začetno dodelitev vrednosti tem poljem, učenje nevronske mreže, posodobitev uteži in čiščenje vseh dodeljenih polj razen polja uteži, ki je uporabljeno za primerjavo z ostalimi izvedbami. Pri GPE različicah se meri še čas prenosa med gostiteljem in napravo ter čiščenje polj na napravi. Primer osnovnih klicev v2 izvedbe je podan v programu 8.

Program 8 Primer klicev funkcij za delovanje nevronske mreže.

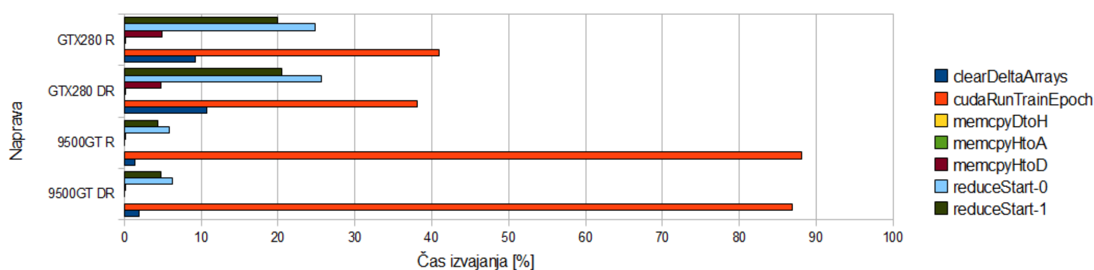
```
unsigned int timerGPU = 0;
CUT_SAFE_CALL( cutCreateTimer( &timerGPU));
CUT_SAFE_CALL( cutStartTimer( timerGPU));
cudaInitNeuralNetworkAdv(parametri nevronske mreže);
cudaTrainNetworkAdv(število epoh);
cudaCleanArraysAdv();
cudaThreadExit();
CUT_SAFE_CALL( cutStopTimer( timerGPU));
printWeightsGPU();
printf("GPU čas izvajanja: %lf ms", cutGetTimerValue(timerGPU));
```

5.3 Časovna porazdelitev izvajanja jeder

Učinkovitost implementacije posameznega jedra lahko na enostaven način preverimo, če CUDA program zaženemo v orodju Nvidia CUDA Visual Profiler. To orodje ob izvajanju programa meri več različnih parametrov, mednje spada število vezanih načinov dostopa, vejitve, število uporabljenih niti, število dodeljenih blokov, velikost uporabljenega deljenega pomnilnika, čas izvajanja posameznega jedra ipd.

Če čase izvajanja posameznih jeder prikažemo grafično (sliki 5.2 in 5.3), lahko ob primerjavi istih jeder na različnih napravah opazimo dobre in slabe vzporedne implementacije jeder. Kot je omenjeno že v poglavju 4.8, je seštevanje polj uteži izrazito nevzporeden problem, kar se jasno vidi na sliki 5.2. Pri grafih želimo, da se glavno jedro, to je `cudaRunTrainEpoch`, izvaja kar največ procesorskega časa, ostala jedra pa čim manj. Na počasnejši napravi se tako 87 % celotnega časa izvaja glavno jedro, večino preostalega časa pa se sešteva polja uteži. Veliko bolj je problem seštevanja očiten na napravi z več multiprocesorji. V tem primeru se glavno jedro izvaja le še okoli 40 % celotnega časa, zaradi več blokov pa se dlje tudi sešteva polja uteži.

Uspešnost vzporedne implementacije med posameznimi jedri lahko primerjamo tudi s teoretično pohitritvijo med različno hitrimi napravami. Hitrost vsake naprave izračunamo iz števila CUDA jeder, ki jih pomnožimo z uro procesorja in številom operacij v plavajoči vejici, ki jih naprava izračuna v eni urini periodi. Tako dobimo flope na sekundo za računano napravo. Naprava 9500GT ima zgornjo mejo 134.4 GFlopov/s in GTX280 933 GFlopov/s. Teoretično je torej naprava GTX280 skoraj 7-krat hitrejša od 9500GT. V primeru,



Slika 5.2: Grafični prikaz deleža časa izvajanja različnih jeder na napravah 9500GT in GTX280 za različico v1 pri 100 epohah. Izvedba DR je implementacija razpoznavanja črk, R pa predstavlja skoraj optimalno nevronsko mrežo.

Jedro	**Čas izvajanja [ms]**		**Pohitritev**
	9500GT	**GTX280**	
cudaRunTrainEpoch	17611.900	1840.090	9.57
reduceStart-0	1273.980	1239.950	1.03
reduceStart-1	968.159	992.868	0.98
clearDeltaArrays	395.321	520.930	0.76

Tabela 5.1: Tabela pohitritev posameznih jeder različice v1 za problem razpoznavanja črk.

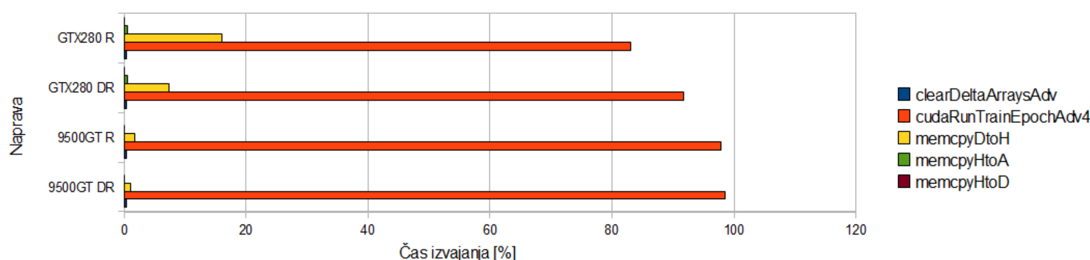
da je realna pohitritev jedra blizu maksimalni teoretični pohitritvi, je vzporednost implementacije dobra.

Izračuni pohitritev med jedri za različico v1 so podani v tabeli 5.1. Pri meritvah podanih v tej tabeli je bilo na napravi GTX280 uporabljenih 60 blokov in 256 niti, pri 9500GT pa le 8 blokov in 192 niti pri 100 epohah. Število niti je v teh primerih različno zaradi omejitve pri v1 izvedbi. Jedro `cudaRunTrainEpoch` je na napravi GTX280 zaradi novejše arhitekture bolj zasedeno in pohitritev je posledično tudi večja od teoretične. K temu malenkost prispeva še število niti znotraj bloka, s čimer se bolj uspešno skriva latence dostopov do pomnilnika. Ker je bilo na napravah dodeljenih ravno dvakrat več blokov, kot je multiprocesorjev, `reduceStart` jedri ne dosegata nobene pohitritve. Omenjena primerjava zaradi različnih arhitektur naprav ni najbolj natančna in priporočljiva za primerjavo hitrosti, vendar nam vseeno da nek približek, ki ga lahko pričakujemo. Običajno je ta približek različen od faktorja pohitritve prenosov med napravami, kar je zelo pomembno pri določanju, ali je program omejen s hitrostjo prenosa podatkov, ali s hitrostjo preračunavanja

Jedro	**Čas izvajanja [ms]**		**Pohitritev**
	9500GT	**GTX280**	
cudaRunTrainEpochAdv DR	4624.290	982.816	4.71
cudaRunTrainEpochAdv R	22173.300	2453.060	9.04

Tabela 5.2: Tabela pohitritev posameznih jeder različice v2. Izvedba DR je implementacija razpoznavanja črk, R pa predstavlja skoraj optimalno nevronsko mrežo.

ukazov.



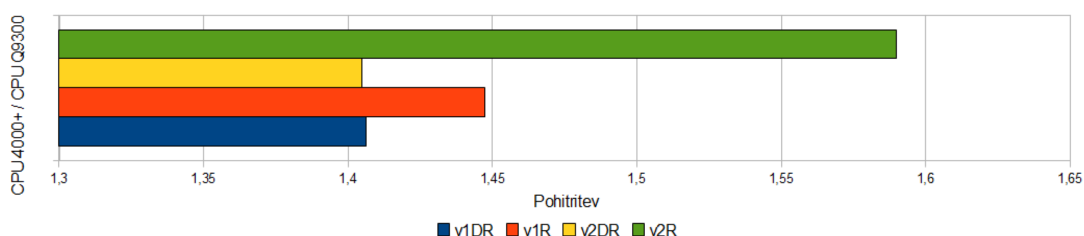
Slika 5.3: Grafični prikaz deleža časa izvajanja različnih jeder na napravah 9500GT in GTX280 za različico v2 pri 100 epohah. Izvedba DR je implementacija razpoznavanja črk, R pa predstavlja skoraj optimalno nevronsko mrežo.

Pri boljši različici v2 se glavno jedro na počasnejši napravi izvaja približno 98 % celotnega časa, na napravi s 30 multiprocesorji pa približno 91 % časa (slika 5.3). Preostanek časa je porabljen za prenos podatkov iz naprave na gostitelja. Ker seštevanje polj uteži ni bilo izvedeno na napravi, je prenos polj uteži na gostitelja začel zavzemati vedno večji delež celotnega izvajanja. To je malenkost bolj opazno na hitrejši napravi pri mreži 120-ih nevronov na vseh nivojih (izvedba R), saj izvajanje glavnega jedra pade na 83 % celotnega časa. Izvedba DR je vsebovala 64 vhodnih, 63 skritih in 26 izhodnih nevronov. Obe izvedbi in napravi sta imeli dodeljenih 60 blokov pri 100 epohah. Izračuna pohitritve za obe izvedbi R in DR sta podana v tabeli 5.2.

Iz opisanega sledi, da so vsa jedra dobro vzporedno implementirana, razen problema seštevanja. Problem se pojavi še pri prenosu podatkov hitrejše izvedbe v2. Če dodelimo preveč blokov na napravo in je mreža velika, se izkaže, da prenos podatkov začenja zavzemati vedno večji delež celotnega izvajanja jeder.

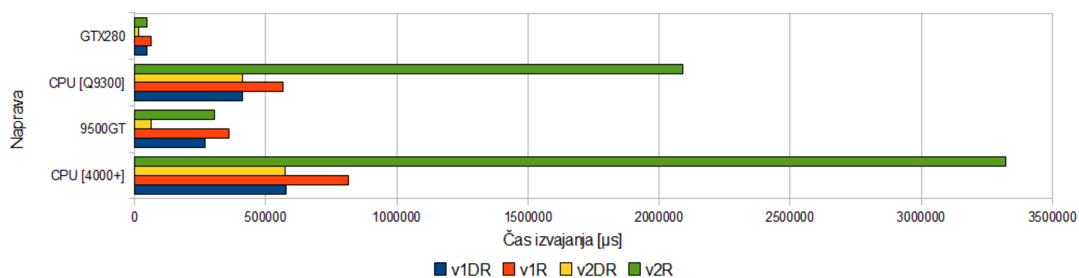
5.4 Pohitritev GPE v primerjavi s CPE

Podrobna primerjava izvajanja hitrosti na grafični napravi in CPE zaradi različnih arhitektur naprav ni možna, možno pa je primerjati v kolikšnem času grafična naprava oz. CPE vrneta primerljive rezultate.



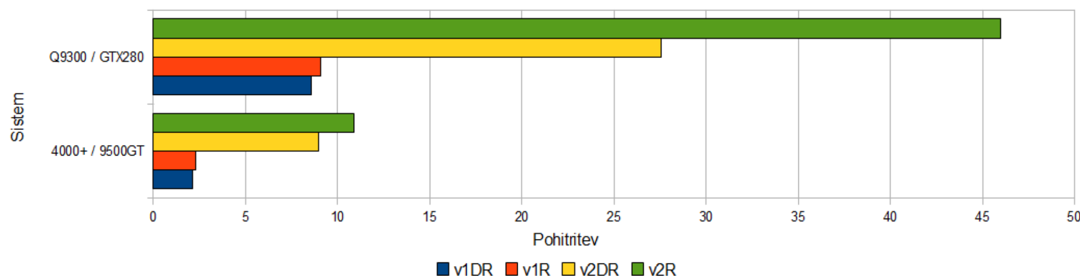
Slika 5.4: Pohitritev med CPE Q9300 in CPE 4000+.

Celoten čas izvajanja je delno odvisen tudi od hitrosti CPE, uporabljene pri merjenju izvajanja. Zato je podan še graf (slika 5.4), kolikokrat zmogljivejša CPE hitreje izvede CPE različico nevronske mreže od počasnejše. Na grafih, prikazanih v tem poglavju, so meritve za obe različici (v1 in v2) ter pri različnih primerih. Realen problem razpoznavanja črk različice v1 je tako označen z "v1DR", optimalna velikost nevronske mreže različice v1 pa z "v1R". Podobno velja tudi za različico v2.



Slika 5.5: Celoten čas izvajanja posamezne različice nevronske mreže na različnih napravah pri 1000 epohah.

V primeru problema razpoznavanja črk je bilo pri različici v1 uporabljenih 64 vhodnih, 63 skritih in 26 izhodnih nevronov, 60 blokov in 256 niti na obeh napravah. Pohitritev v1 različice je zaradi precejšne neodvisnosti od velikosti same mreže dokaj konstantna in je zato podobna tudi v primeru mreže s 120 nevroni na vseh nivojih. Na počasnejšem sistemu se faktor pohitritve giblje okoli 2.2 in na hitrejšem okoli 9 (slika 5.6).



Slika 5.6: Pohitritev GPE v primerjavi s CPE.

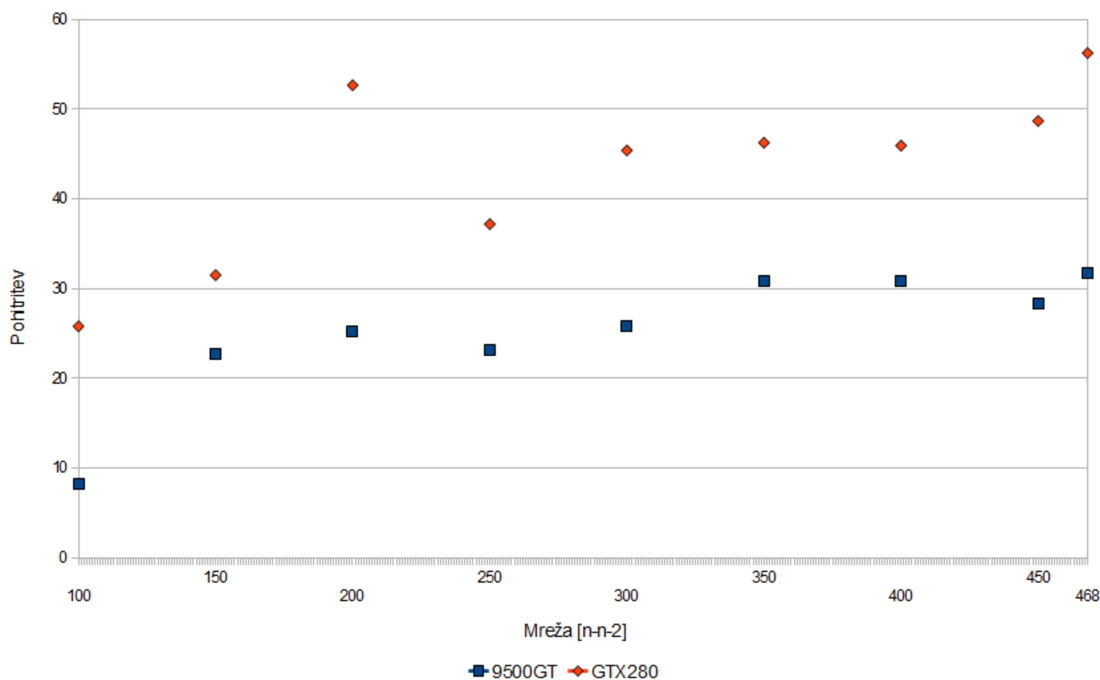
Celoten čas izvajanja za vse naprave in vse različice nevronskih mrež je podan na sliki 5.5. Na grafu lahko razberemo, da se najdlje preračunavajo nevronske mreže na obeh CPE napravah, nato na grafični napravi 9500GT in najhitreje se preračuna na napravi GTX280. Med izvedbama "v1DR" in "v2DR" je opaziti, da se na CPE izvajata enako časa, saj sta mreži enako veliki. Večja razlika se pojavi v GPE izvedbi, kjer je "v2DR" opazno hitrejša od "v1DR". Na grafu najbolje izstopa čas izvajanje izvedbe "v2R". V tem primeru je bila uporabljena "teoretično" ena izmed najboljših možnih mrež s 120 nevroni na vseh nivojih in je zato posledično preračunavanje daljše.

Zanimivejši so faktorji pohitritev med GPE in CPE za različico v2 (slika 5.6). Pri problemu razpoznavanja črk sem na počasnejšem sistemu dosegel skoraj 9-kratno pohitritev in na hitrejšem 27.58-kratno. V primeru različice "v2R" je bil faktor pohitritve na počasnejšem sistemu 10.87 in na hitrejšem skoraj 46.

Pri bolj realnih nevronskih mrežah, ki imajo n-vhodnih in skritih nevronov in 2 izhodna nevrona, bi pričakovali, da ne bi najbolje delovale, saj na zadnjem nivoju le 2 niti delujeta in ostale čakajo. Izkazalo pa se je, da pri nevronski mreži s 468 nevroni[1] na vhodnem in skritem nivoju pri hitrejši napravi dosežemo več kot 56-kratno pohitritev in blizu 38-kratni na počasnejši napravi. Razlog za takšno pohitritev je najverjetneje v velikem številu niti dodeljenih na blok. Pri meritvah, prikazanih na sliki 5.7, je opaziti, da imamo dva lokalna maksimuma. Prvi se nahaja na območju med 200 in 250 nevroni, zadnji pa je pri 468 nevronih. Do tega pride, ker je bilo dodeljenih ravno dvakrat več blokov, kot je imela naprava multiprocesorjev, in je na začetku ostalo še dovolj prostega deljenega pomnilnika, da je multiprocesor lahko hkrati obdeloval po dva bloka, nato pa le še enega.

Ker je hitrost izvajanja različice v2 pogojena s toplogijo nevronske mreže,

[1] To je v trenutni implementaciji največja oz. skoraj največja možna mreža zaradi velikosti deljenega pomnilnika. Večje bi bile možne, če bi blok obdeloval manj kot 4 primere sočasno.



Slika 5.7: Pohitritve pri različnih številih nevronov.

se pojavi vprašanje, kakšna bi bila pohitritev v najslabšem primeru. Ena izmed najslabših možnih nevronskih mrež bi imela 2 vhodna, 511 skritih in 2 izhodna nevrona. Izkaže se, da je v tem primeru na počasnejšem sistemu izvajanje še vedno malenkost hitrejše kot na CPE. Pohitritev je tukaj 1.279-kratna, na hitrejšem sistemu z napravo GTX280 pa je pohitritev okoli 6.289-kratna.

5.5 Najpomembnejše optimizacije

V primeru moje implementacije nevronskih mrež se izkaže, da je bila izbira drugačnega pristopa vzporednosti ključen pogoj za boljšo pohitritev. Izbira vezanega načina dostopa do pomnilnika je za hitro delovanje skoraj obvezna, vendar to nič ne pomaga, če imamo dostopov veliko in je program omejen s širino vodila namesto s hitrostjo preračunavanja. Zelo pomembna optimizacija je tudi uporaba deljenega pomnilnika, s katerim lahko zmanjšamo število dostopov do globalnega pomnilnika. Pri uporabi deljenega pomnilnika je potrebno paziti, da ne prihaja do konfliktov bank, saj je takrat dostop najhitrejši.

Pri SIMD arhitekturi procesorjev, kjer več procesorjev v nekem trenutku

izvaja isti ukaz, je pomembno, da znotraj njih ne prihaja do vejitev. V primeru nevronskih mrež je pri boljši implementaciji vejitev kar nekaj, vendar prihaja le znotraj enega snopa do dejanske vejitve, druge niti pa izvajajo enake ukaze. Nekaj odstotno pohitritev se da doseči še z zmanjševanjem prevelikega števila uporabljenih registrov in občasno z večjo zasedenostjo multiprocesorjev, s katero lažje skrivamo latence dostopa do glavnega pomnilnika.

Poglavje 6

Sklepne ugotovitve

V diplomski nalogi je predstavljena CUDA arhitektura in C za CUDA programski jezik, v katerem sta sprogramirani dve različici algoritma vzvratnega razširjanja napake. Ob primerjavi s primerljivo CPE izvedbo smo ugotovili, da se obe različici nekajkrat hitreje izvedeta na grafični napravi. Izkazalo se je, da je pohitritev posamezne implementacije zelo odvisna od uporabljenih optimizacij in izbire algoritma.

Samo programiranje GPE izvedbe je na začetku precej dolgotrajneše, kot za CPE, saj moramo pri pravem zaporedju ukazov upoštevati še način dostopanja do podatkov, kar je bistvo optimizacij. S poznavanjem CUDA arhitekture in načinov, ki pospešijo izvajanje, programiranje postane mnogo hitrejše, a še vedno ne tako hitro, kot za CPE. V prihodnosti lahko pričakujemo, da bodo prevajalniki omogočali boljšo abstrakcijo in poenostavili programiranje. Tako ne bo več potrebno natančno poznavanje arhitekture naprave in tudi omejitve programov, ki se najpogosteje nanašajo na lastnosti naprave, ne bodo več tako stroge.

Opisani izvedbi vzporednosti predstavljeni v poglavju 4 seveda nista edina načina vzporednega preračunavanja in bi bilo potrebno preučiti še alternativne algoritme, ki bolje izkoriščajo lokalnost izračunov, izrabljajo številčnost niti in uporabljajo čim manj dostopov do globalnega pomnilnika.

Sklenem lahko, da CUDA arhitektura ob nekoliko večji težavnosti programiranja omogoča precej hitrejše splošnonamensko preračunavanje podatkov na grafičnih napravah.

Dodatek A

Namestitev orodja CUDA

Navodila za namestitev programskega orodja CUDA so napisana za operacijski sistem Windows 7 Professional in za različico 10.6 operacijskega sistema Mac OS X ter so namenjena za pripravo delovanja obstoječe programske kode nevronskih mrež.

A.1 Windows 7 Professional

Za namestitev v operacijskem sistemu Windows je bil uporabljen 64 biten Windows 7 Professional, 190.38 različica CUDA 64 bitnih gonilnikov, 64 bitni CUDA Toolkit 2.3, 64 bitni CUDA SDK 2.3 ter Visual Studio 2008 Professional Edition (VS). Za pomoč pri ustvarjanju novih projektov je bil dodan še čarovnik za CUDA 64 bitne Windowse.

Programi so bili nameščeni v opisanem vrstnem redu in z njimi ni bilo nikakeršnih problemov. Opozoriti je potrebno, da je bila v prvem poiskusu uporabljena različica VS Express Edition, vendar le-ta ne vsebuje 64 bitnih prevajalnikov in ostalih sorodnih orodij. Popolna različica je profesionalno orodje VS 2008, ki vsebuje vse potrebno, vendar privzeta nastavitev tudi ne namesti 64 bitnih prevajalnikov. Priporočeno je izbrati popolno namestitev oz. poiskati 64 bitna orodja za C++ programski jezik in jih označiti za namestitev.

Pri izdelavi projekta znotraj VS-ja uporabimo izvorno kodo projekta nevronskih mrež. Nov projekt poimenovan `nevronskeMreze` ustvarimo kot nov CUDA64 projekt, nato izberemo konzolsko aplikacijo in označimo "precompiled headers" ter končamo z izbiro. Pri odprtem projektu nam čarovnik generira začetno stanje CUDA64 projektov z dodano datoteko sample.cu in ReadMe.txt, ki ju lahko izbrišemo.

Pod direktorij `VS2008\Projects\nevronskeMreze\nevronskeMreze` je sedaj potrebno prekopirati izvorno kodo nevronskih mrež s končnicami *.cu, *.cuh, *.h. Znotraj VS-ja moramo prekopirane datoteke uvoziti kot obstoječe elemente: pod "Source Files" uvozimo: cudaNeuralNetwork.cu, dataLoader.cu, main.cu, neuralNetwork.cu, reduction.cu, reduction_kernel.cu, pod "Header Files" pa vse s končnico *.h. Kot zadnje, kar je potrebno izbrati, je Release način prevajanja in x64 tip arhitekture.

Program bi se sedaj že moral uspešno prevesti, vendar za zagon manjkajo še datoteke cudart.dll, cutil64.dll in data.txt, ki jih prekopiramo pod direktorij `VS2008\Projects\nevronskeMreze\x64\Release`. Prva se nahaja pod nameščenim `CUDA\bin` direktorijem, druga v CUDA SDK direktoriju pod `C\bin\win64\Release` in zadnja v izvorni kodi nevronskih mrež.

A.2 Mac OS X 10.6

Za namestitev v operacijskem sistemu Mac OS X različice 10.6 so na voljo le 32 bitni paketi. Uporabljeni so bili CUDA gonilniki 2.3.1a, CUDA Toolkit 2.3a, CUDA SDK 2.3a in gcc različica 4.0.1. Zaradi hrošča v inštalaciji, ki se občasno pojavi, je priporočljivo najprej namestiti Toolkit in nato gonilnike.

Takoj po namestitvi so se pojavili določeni problemi, ker se po resetiranem računalniku jedro CUDA ni naložilo. Postopek, kako odpraviti težavo, opisuje David Jenkins na svojem blogu [16]:

- Najprej odstranimo vse obstoječe CUDA programe z ukazi:
  1. sudo rm -r /usr/local/cuda
  2. sudo rm -r /System/Library/Extensions/CUDA.kext
  3. sudo rm -r /System/Library/StartupItems/CUDA
- Resetiramo računalnik in ga zaženemo v 32 bitnem načinu (to storimo z držanjem tipki 3 in 2 ob zagonu).
- Namestimo opisane pakete v omenjenem vrstnem redu.
- Popravimo dovoljenja:
  1. sudo chmod g-w /System/Library/StartupItems/CUDA/*
  2. sudo chmod g-w /System/Library/StartupItems/CUDA/
  3. sudo chmod -R 775 /usr/local/cuda/lib/

- Kot zadnje še spremenimo simbolične linke iz verzije 4.2 na 4.0:
    1. cd /usr/bin
    2. sudo rm gcc g++
    3. sudo ln -s gcc-4.0 gcc
    4. sudo ln -s g++-4.0 g++

Po uspešni namestitvi programskih paketov moramo posodobiti še spremenljivki PATH in DYLD_LIBRARY_PATH z ukazom export:

```
export PATH=/usr/local/cuda/bin:$PATH
export DYLD_LIBRARY_PATH=/usr/local/cuda/lib:$DYLD_LIBRARY_PATH
```

Za stalen učinek vrednosti teh spremenljivk je priporočeno to shraniti v datoteko .bash_profile.

CUDA SDK najprej prevedemo z ukazom make v direktoriju /Developer/GPU Computing/C. V poddirektorij src (C/src/nevronskeMreze) prekopiramo celotno izvorno kodo nevronskih mrež in se premaknemo v ta direktorij. Program prevajamo z ukazom "make clean2; make" in ga zaženemo z relativno potjo: ../../bin/darwin/release/nevronskeMreze. Z relativno potjo je potrebno zagnati program zato, da baze podatkov ni potrebno prekopirati v direktorij, kjer se nahaja izvršljiva koda.

Dodatek B

Opis podatkovne zbirke

Podatkovna zbirka vsebuje atributni opis popačenih črk angleške abecede. Iz 20-ih pisav je vsaka črka naključno popačena, tako da podatkovno zbirko sestavlja 20000 unikatnih primerov. Vsak primer je nadalje opisan s 16 atributi, kjer je vsak razširjen na območje med 0 in 15 celoštevilčnih vrednosti.

Opisi atributov:

1. velika tiskana črka angleške abecede (26 vrednosti od A do Z);
2. vodoravna lega najmanjšega okvira, ki obsega piksle črke, dobljena s štetjem pikslov od levega roba slike;
3. navpična lega najmanjšega okvira, dobljena s štetjem pikslov od dna slike;
4. širina okvira v pikslih;
5. višina okvira v pikslih;
6. število pikslov črke v okviru;
7. povprečje vodoravnih leg pikslov na črki v okviru glede na center okvira (v primeru črke L je slika levo-utežena);
8. povprečje navpičnih leg pikslov na črki v okviru glede na center okvira;
9. povprečje kvadratične vrednosti vodoravnih razdalj pikslov, merjenih v točki 6;
10. povprečje kvadratične vrednosti navpičnih razdalj pikslov, merjenih v točki 7;

11. povprečna vrednost produkta vodoravnih in navpičnih razdalj piklsov črke v okviru, merjenih v točkah 6 in 7;
12. povprečna vrednost kvadratične vodoravne razdalje, pomnožene z navpično razdaljo za vsak piksel na črki;
13. povprečna vrednost kvadratične navpične razdalje, pomnožene z vodoravno razdaljo za vsak piskel na črki;
14. povprečno število robov, dobljenih pri sistematičnem pregledu iz leve proti desni, preko vseh vrstic znotraj okvira;
15. vsota navpičnih leg robov, dobljenih pri predhodnem atributu (pri črki Y bo tako vrednost večja, saj je več robov pri vrhu črke kot pri dnu);
16. povprečno število robov, štetih od dna do vrha okvira;
17. vsota vodoravnih leg robov, dobljenih pri predhodnem atributu.

Vsi atributi razen prvega so celoštevilski in obsegajo vrednosti od 0 do 15. Porazdelitev posameznih črk je podana v naslednji tabeli:

789	A	766	B	736	C	805	D	768	E	775	F	773	G
734	H	755	I	747	J	739	K	761	L	792	M	783	N
753	O	803	P	783	Q	758	R	748	S	796	T	813	U
764	V	752	W	787	X	786	Y	734	Z				

Slike

1.1 Fermi multiprocesor . . . 7
1.2 Zgradba arhitekture Larrabee . . . 9
1.3 Zgradba jedra Larrabee . . . 10
1.4 Arhitektura OpenCL naprav . . . 12

2.1 Prikaz izvajanja blokov . . . 16
2.2 Shematični prikaz množice SIMT multiprocesorjev . . . 18
2.3 Mreža blokov niti . . . 20
2.4 Izvajanje programske kode . . . 23
2.5 Shema prevajanja programske kode . . . 24

3.1 Biološki nevron . . . 30
3.2 Aktivacija nevrona . . . 31
3.3 Model nevrona . . . 32
3.4 Prikaz aktivacijskih funkcij . . . 33
3.5 Dvorazredni klasifikacijski problem . . . 38

4.1 Graf rasti zmogljivosti . . . 47
4.2 Zasedenost CPE in GPE z enotami . . . 48
4.3 Grafa hitrosti preračunavanja in prepustnosti vodila . . . 49
4.4 Pomnilnik in niti v polovici snopa . . . 51
4.5 Vezan dostop do podatkov . . . 51
4.6 Dostop do podatkov pri AoS in SoA . . . 53
4.7 Drevesna struktura seštevanja . . . 57
4.8 Prikaz izvajanja niti pri napredni različici . . . 63

5.1 Uspešnost učenja realnega problema . . . 66
5.2 Grafični prikaz deleža časa izvajanja za v1 . . . 69
5.3 Grafični prikaz deleža časa izvajanja za v2 . . . 70
5.4 Pohitritev med CPE Q9300 in CPE 4000+ . . . 71

5.5 Graf časov izvajanja posamezne različice nevronske mreže 71
5.6 Pohitritev CPE v primerjavi z GPE 72
5.7 Pohitritve pri različnih številih nevronov 73

Tabele

1.1 Tabela sopomenk za CUDA in OpenGL 11

2.1 Tabela lastnosti pomnilnikov 21

5.1 Tabela pohitritev jeder za različico GPEv1 69
5.2 Tabela pohitritev jeder za različico GPEv2 70

Literatura

[1] AMD Stream SDK, dostopno na:
http://developer.amd.com/gpu/ATIStreamSDK/pages/Documentation.aspx

[2] Arbib M. A., *The Handbook of Brain Theory and Neural Networks*, second edition, 2003, str. 3-22.

[3] Asanovic K. et al., *The Landscape of Parallel Computing Research: A View from Berkeley*, dec. 2006.

[4] Bebis G., Georgiopoulos M, Kasparis T., *Coupling weight elimination with genetic algorithms to reduce network size and preserve generalization*, jan. 1997, str. 170-171.

[5] *A Neural Network on GPU*, dostopno na: http://www.codeproject.com/KB/graphics/GPUNN.aspx

[6] CUDA ZONE, dostopno na:
http://www.nvidia.com/object/cuda_home.html

[7] Daqi G., Chengyin L., Changwu L. *Modular Adaptive RBF-Type Neural Networks for Letter Recognition*, Proceedings of the Internal Joint Conference on, Vol. 1, str. 571-576, 2003.

[8] Erkman B., Yildirim T. *Statistical Neural Network Based Classifiers for Letter Recognition*, Vol. 345, str. 1081-1086, 2006.

[9] Fogarty T. *First Nearest Neighbor Classification on Frey and Slate's Letter Recognition Problem*, Vol. 9, str 387-388, 1992.

[10] Frey P. W., Slate D. J., *Letter Recognition Using Holland-Style Adaptive Classifiers*, Boston, 1991.

[11] *GPU Technology Conference*, San Jose, California, sept. 2009.

[12] Haykin, S., *Neural Networks: A Comprehensive Foundation*, druga izdaja, New Jersey: Prentice-Hall, 1999.

[13] Hennessy John L., Patterson David A., *Computer Architecture: A Quantitative Approach*, 4. izdaja, Morgan Kaufmann, San Francisco, 2007.

[14] Intel Pine Trail, dostopno na: http://www.intel.com/pressroom/archive/reference/Pineview_Moblin_disclosure.pdf

[15] islovar, dostopno na: http://www.islovar.org

[16] Jenkins D., dostopno na: http://daviddjenkins.blogspot.com/2009/09/snow-leopard-and-cuda.html

[17] Khoros OpenCL Working Group, *The OpenCL Specification*, Version 1.0, 2009.

[18] Kononenko I., *Strojno učenje*, 2. izdaja, Fakulteta za računalništvo in informatiko, Ljubljana, 2005.

[19] Kukar M., Kononenko I., Silvester T., *Machine learning in prognosis of femoral neck fracture recovery*, Ljubljana, Slovenia, str. 9 in 10.

[20] Lopes N., *Multiple Back-Propagation*, dostopno na: http://dit.ipg.pt/MBP/

[21] Microsoft: DirectX SDK, dostopno na: http://msdn.microsoft.com/en-us/directx/aa937788.aspx

[22] Microsoft: Gamefest 2008 - Microsoft game technology conference, dostopno na: http://msdn.microsoft.com/sl-si/directx/bb896684%28en-us%29.aspx

[23] Monitor, dostopno na: http://www.monitor.si/clanek/linux-in-varnost/

[24] *Neural Networks on the GPU*, dostopno na: http://leenissen.dk/fann/html_latest/files2/gpu-txt.html

[25] Nvidia: *Getting Started: Nvidia CUDA Development Tools 2.3, Installation and Verification on Mac OS X*, jul. 2009.

[26] Nvidia: *NVIDIA CUDA^{TM} Programming Guide*, Version 2.3.1, 2009.

[27] Nvidia: *NVIDIA's Next Generation CUDA^{TM}Compute Architecture: Fermi^{TM}*, V1.1, 2009.

[28] Nvidia: *Optimization: NVIDIA CUDA C Programming Best Practices Guide, CUDA Toolkit 2.3*, jul. 2009.

[29] Nvidia: *The CUDA Compiler Driver NVCC*, jul. 2009.

[30] Nvidia timeline, dostopno na: http://www.nvidia.com/page/corporate_timeline.html

[31] Philip N. S., Joseph K. B. *Disorted English Alphabet Identification: An application of Difference Boosting Algorithm*, CoRR, 2000.

[32] Sluga D., Vodopivec T., Lotrič U., "Primerjava zmogljivosti računanja na gruči računalnikov in na grafični procesni enoti," v zborniku *Zbornik osemnajste mednarodne Elektrotehniške in računalniške konference ERK 2009*, Portorož, Slovenija, sept. 2009, str. 15-18.

[33] Seiler L. et al. *A Many-Core x86 Architecture for Visual Computing*, dostopno na: http://software.intel.com/en-us/articles/larrabee/

[34] Slate D. J., *Letter Recognition Data Set*, dostopno na: http://archive.ics.uci.edu/ml/datasets/Letter+Recognition