VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

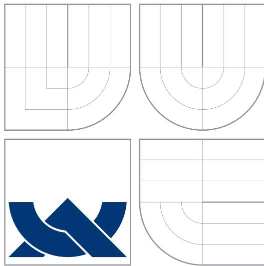
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PROGRAMOVÁ KNIHOVNA PRO PRÁCI S UMĚLÝMI NEURONOVÝMI SÍTĚMI S AKCELERACÍ NA GPU

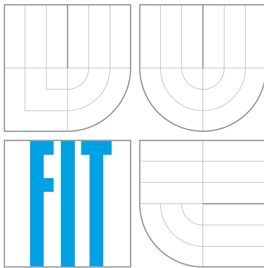
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. ANDREJ TRNKÓCI
AUTHOR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PROGRAMOVÁ KNIHOVNA PRO PRÁCI S UMĚLÝMI NEURONOVÝMI SÍTĚMI S AKCELERACÍ NA GPU

SOFTWARE LIBRARY FOR ARTIFICIAL NEURAL NETWORKS WITH ACCELERATION USING GPU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. ANDREJ TRNKÓCI
AUTHOR

VEDOUCÍ PRÁCE Doc. Ing. FRANTIŠEK V. ZBOŘIL, CSc.
SUPERVISOR

BRNO 2013

Abstrakt

Umelé neuronové siete sú náročné na výpočtovú silu počítaču. Ich urýchlenie môže znamenať otvorenie nových možností výskumu či aplikácie v tejto oblasti. Práve to je cieľom tejto práce. Použitie grafických kariet na učenie neurónových sietí je jeden spôsob ako spomínané urýchlenie dosiahnuť. Táto práca rozoberá teoretické východiská a následne implementáciu softvérovej knižnice pre učenie algoritmom Backpropagation s podporou urýchlenia na grafickej karte.

Abstract

Artificial neural networks are demanding to computational power of a computer. Increasing their learning speed could mean new posibilities for research or aplication of the algorithm. And that is a purpose of this thesis. The usage of graphics processing units for neural networks learning is one way how to achieve above mentioned goals. This thesis is offering a survey of theoretical background and consequently implementation of a software library for neural networks learning with a Backpropagation algorithm with a support of acceleration on graphics processing unit.

Klíčová slova

GPGPU, OpenCL, Backpropagation, umelé neurónové siete.

Keywords

GPGPU, OpenCL, Backpropagation, artificial neural networks.

Citace

Andrej Trnkóci: Programová knihovna pro práci s umělými neuronovými sítěmi s akcelerací na GPU, diplomová práce, Brno, FIT VUT v Brně, 2013

Programová knihovna pro práci s umělými neuronovými sítěmi s akcelerací na GPU

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Doc. Ing. Františka Zbořila CSc.

. . . .
Andrej Trnkóci
22. mája 2013

Poděkování

Na tomto mieste by som chcel poďakovať vedúcemu mojej práce, pánovi Doc. Ing. Františkovi Zbořilovi CSc. za ústretový prístup a cenné rady, ktoré mi poskytol pri riešené mojej práci.

© Andrej Trnkóci, 2013.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Princípy fungovania a architektúry grafických kariet **4**
2.1 Klasická architektúra grafických kariet 4
2.2 Odlišnosti všeobecných a grafických procesorov a ich princípy 6
2.3 Moderná architektúra grafických kariet 6
2.4 nVIDIA série 8 7
2.5 nVIDIA G86M 8
2.6 nVIDIA GTX 280 8
2.7 nVIDIA Fermi 9
2.8 nVIDIA Kepler 11
2.9 Grafické karty ATI/AMD Radeon 13
2.10 ATI/AMD Evergreen 14
2.11 ATI/AMD Cayman 15
2.12 AMD Graphics Core Next (GCN) 16
2.13 Accelered processing unit (APU) 17

3 Nástroje pre programovanie všeobecných výpočtov na grafických kartách, OpenCL **19**
3.1 Model platformy (Platform model) 20
3.2 Model spúšťania 20
3.3 Pamäťový model 21

4 Umelé neurónové siete **23**
4.1 Základné princípy umelých neurónových sietí 23
4.1.1 Topológie neurónových sietí 26
4.1.2 Dopredná sieť so spätným šírením chyby 26

5 Analýza a návrh riešenia **30**
5.1 Predchádzajúce práce na tému paralelného riešenia algoritmu Backpropagation 30
5.2 Analýza algoritmu Backpropagation a možností jeho paralelizácie 31
5.3 Rozdelenie paralelného algoritmu na menšie celky 34
5.4 Spravovanie kontextu a zdrojov grafickej karty 34

6 Implementácia **35**
6.1 Dátová reprezentácia neurónovej siete 35
6.2 Programy vykonávané na grafickej karte. 37

7 Testovanie **40**
7.1 Metodika testovania 40

8 Možnosti rozšírenia navrhnutej knižnice o ďalšie algoritmy **43**

9 Záver **45**

A Obsah CD **48**

B Manual **49**

Kapitola 1

Úvod

V histórii výpočtových systémov sa niekoľko desaťročí zrýchľovali procesory najmä zmenšovaním tranzistorov v čipe a zvyšovaním ich pracovných frekvencií. Okolo roku 2005 však technológia kremíkových čipov začala narážať na fyzikálne limity. Kvôli exponenciálnemu nárastu spotreby energie pri zvyšovaní pracovnej frekvencie existuje hranica, nad ktorou už zvýšený výkon nestojí za zníženie efektivity. Navyše by bolo problematické, či dokonca nemožné odvádzať vznikajúce teplo. Z toho dôvodu je ďalšie zrýchľovanie jedného procesorového jadra čoraz komplikovanejšie a preto narastá s ďalšími generáciami pomalšie ako v minulosti. Preto sa od spomínaného obdobia začali procesory vybavovať viacerými výpočtovými jednotkami (jadrami). Pri pridaní ďalšieho jadra sa zvýši spotreba energie vzhľadom k nárastu výpočtového výkonu lineárne, čo je oveľa priaznivejšia situácia. Na druhej strane je ale potrebný softvér, ktorý dokáže využiť viac procesorových jadier. Ak teda chceme riešiť úlohu, ktorá sa dá rozdeliť na paralelne riešiteľné podúlohy, potom je pre nich výhodnejší procesor s veľkým počtom jednoduchších a pomalších jadier. Presne táto myšlienka sa využíva v grafických akcelerátoroch. Grafické karty sú navrhnuté na rasterizáciu grafických entít, čo je úloha, ktorá je vhodná na paralelné spracovanie. Grafické karty boli v minulosti navrhované len s ohľadom na tento jeden účel. To sa ale zhruba v posledných piatich rokoch zmenilo a súčasné grafické karty od dvoch hlavných výrobcov v oblasti osobných počítačov (nVIDIA a AMD) sú prispôsobené na všeobecné výpočty. Vďaka tomu je možné efektívne využiť ich výpočtovú silu, ktorá obvykle niekoľkonásobne prevyšuje výkon klasických procesorov [20]. Iná úloha vhodná pre paralelné spracovanie je učenie umelých neurónových sietí, ktoré môžeme definovať ako štruktúru zloženú z husto prepojených adaptívnych a jednoduchých výpočtových elementov (nazývaných umelé neuróny alebo uzly [12]. Už z tejto definície vyplýva, že umelé neurónové siete sú vo svojej podstate paralelný systém, kde sa v každom kroku vykonáva množstvo operácií súčasne a preto sa dá očakávať, že táto úloha bude vhodná pre akceleráciu na grafickej karte, čo je jeden z dôvodov, prečo som si vybral túto tému. Druhý dôvod je, že programový výstup tejto práce by mohol byť využitý aj v ďalších študentských prácach a vďaka urýchleniu výpočtov by mohol uľahčiť prácu iným študentom.

Kapitola 2

Princípy fungovania a architektúry grafických kariet

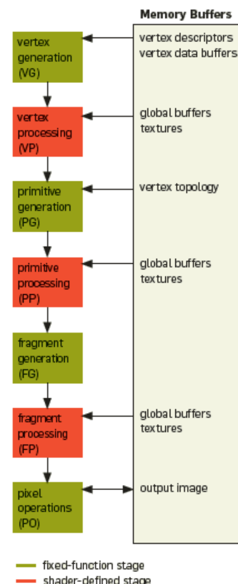
V tejto kapitole sa budem venovať základnému princípu fungovania grafických kariet. Načrtnem v nej základné princípy, aké sa používajú u grafických procesorov a poukážem na odlišnosti medzi klasickými a grafickými procesormi. Ďalej bude nasledovať prehľad existujúcich architektúr grafických kariet. Zameriam sa tu na grafické akcelerátory dvoch popredných výrobcov a to konkrétne od priekopníka na trhu s grafickými akcelerátormi - spoločnosť nVIDIA a od spoločnosti, ktorá nVidiu nasledovala - ATI, ktorá je už dnes súčasťou spoločnosti AMD (Advanced Micro Devices). V časti 2.1 uvediem klasickú architektúru grafických kariet. V nasledujúcej časti 2.2 vysvetlím odlišnosti grafických procesorov a procesorov pre všeobecné použitie. Ďalej v nej budú vysvetlené princípy, ktoré sa pri grafických procesoroch využívajú pre dosiahnutie čo najvyššej aritmetickej priepustnosti. Nasledujúce časti sú venované existujúcim moderným architektúram grafických kariet. Predovšetkým tým, ktoré sú vhodné pre všeobecné výpočty.

2.1 Klasická architektúra grafických kariet

Grafické akcelerátory generujú zobrazenia virtuálnej scény. Tieto scény sú určené geometriou, orientáciou a materiálovými vlastnosťami povrchu objektov, ich pozíciou a charakteristikou zdrojov svetla. Zobrazenie tejto scény je potom definované umiestnením virtuálnej kamery. Princíp rasterizácie takejto scény je dobre vysvetlený na zjednodušenom modeli zreťazenej linky (pipeline) ktorú ilustruje obrázok 2.1. Tá sa skladá z viacerých stupňov, pri ktorých sa striedajú programovateľné stupne s jednotkami s nemennou funkciou. Výstup každého stupňa je vstupom do nasledujúceho stupňa [9].

- vertex generation (VG): načítava vrcholy poskytnuté aplikáciou a posiela ich ďalej v podobe deskriptorov vrcholov. Tie zvyčajne obsahujú pozíciu v scéne (x,y,z) prípadne farbu povrchu a normálový vektor.
- Vertex processing (VP): Jeho správanie je programovateľné. Jeho výstupom je presne jeden deskriptor vrcholu na každý vstupný záznam. Typickou operáciou vykonávanou vo VP je dvojrozmerná projekcia vrcholov na obrazovku.
- Primitive generation (PG): Spája záznamy o vrcholoch do záznamov o primitívach.

- Primitive processing (PP): Druhá programovateľná časť. Spracúva každú grafickú primitívu zvlášť. Jej výstupom môže byť 0 alebo viac primitív.
- Fragment generation (FG): V tejto časti zreťazeného spracovania sa rasterizujú primitívi. Vytvárajú sa ich obrazy v priestore obrazovky zložené z fragmentov. Výstupom je tok záznamov o fragmentoch, kde každý obsahuje svoju pozíciu na obrazovke, vzdialenosť od virtuálnej kamery a tiež výsledky interpolácie parametrov vrcholov zdrojovej primitívi.
- Fragment processing (FP): Je posledná programovateľná časť zreťazenej linky. Využíva sa na určenie farby jednotlivých fragmentov. Farba sa zvyčajne určuje pomocou priradených textúr a polohy a parametrov svetiel.
- Pixel operation (PO): Vytvára výstupný obraz z fragmentov. Pomocou informácií o vzdialenosti od virtuálnej kamery PO zahodí fragmenty, ktoré nie sú viditeľné, lebo ich blokuje iný fragment bližší ku kamere. Prípadne pri čiastočne priesvitných objektoch vypočíta farbu pixelu zo všetkých prekrývajúcich sa fragmentov.



Obrázok 2.1: Klasická grafická pipeline. Prevzaté z [9]

Programovateľné časti zreťazenej linky sa nazývajú "shader"procesory. Vďaka vzájomnej nezávislosti mnohých výpočtov pri spracovaní objektov scény a následnom renderingu je možné spracovávať tieto dáta paralelne. Preto bolo logické urýchľovať zobrazovanie grafiky pomocou veľkého počtu procesorov. Aj vďaka tomu majú dnešné grafické karty mnohonásobne vyššiu maximálnu teoretickú výpočtovú silu ako bežné procesory. Pre predstavu, napríklad grafická karta nVIDIA GeForce 280 so 480 aritmeticko-logickými jednotkami s pracovnou frekvenciou 1,3 GHz je schopná výkonu 933 GFLOPS (Giga Floating point Operations Per Second - miliárd operácií s desatinnými číslami za sekundu). Naproti tomu procesor Intel Core 2 Quad so štyrmi jadrami o frekvencii 3 GHz je schopný dosiahnuť maximálne 96 GFLOPS [9]. Vyšší teoretický výkon bol dôvodom snáh využiť tento potenciál ešte v dobách, keď boli grafické karty stavané len pre rendering priestorových scén. Už pri

prvom pohľade na klasickú zreťazenú linku na obrázku 2.1 je zrejmé, že nešlo o jednoduchú úlohu. Bolo nutné transformovať polia na textúry a pre samotný výpočet použiť tzv. off-screen rendering. Jedná sa o princíp, kedy sa využíva na ukladanie výsledkov off-screen buffer namiesto zobrazovacieho buffera [15].

2.2 Odlišnosti všeobecných a grafických procesorov a ich princípy

Ako už bolo naznačené v úvodnej časti, pokiaľ je to možné, je výhodnejšie vykonávať určitý algoritmus na viacerých procesoroch bežiacich na pomalšej frekvencii, ako na jednom extrémne rýchlom procesore a to hlavne z hľadiska efektivity a spotreby energie. Pri grafických algoritmoch je obvykle potenciál pre paralelizmus značný a preto sa grafické procesory vyvíjali práve týmto spôsobom, čím vznikol rozdiel v koncepcii klasických a grafických procesorov. V tejto časti textu preto uvediem niektoré základné rozdiely na ktoré treba myslieť pri programovaní grafických kariet.

Najzákladnejší rozdiel, už bol naznačený, je ním rozdiel medzi malým počtom veľmi rýchlych procesorových jadier pri klasických procesoroch a veľkom počte (v niektorých prípadoch aj viac ako tisíc) pomalých jadier pri grafických procesoroch.

Z toho sa odvíja ďalší podstatný rozdiel a to typické vzory prístupov do pamäte u procesorov na všeobecné použitie a grafických procesorov. V prvom prípade sa obvykle vykonávajú jednovláknové programy s náhodným prístupom do pamäte. Pri každom prístupe sa väčšinou načítava malé množstvo dát. V tomto prípade je pre výkonnosť aplikácie najdôležitejším parametrom doba odozvy (angl. latency). Naopak, pri vysoko paralelných výpočtoch na grafickej karte sa obvykle jedná o čítanie/zápis pamäťových blokov, kedy sa načíta veľa dát z operačnej pamäte naraz. A preto v tomto prípade je doba odozvy druhoradá a najdôležitejším parametrom je dátová priepustnosť. Pri vysoko paralelných architektúrach je dôležité preniesť dostatočné množstvo dát za určitý čas a to v takom objeme, aby mohli všetky jadrá pracovať súčasne.

Samozrejme aj v prípade grafických procesorov je doba odozvy pamäte nezanedbateľný parameter. Ak by všetky jadrá museli čakať stovky cyklov na dáta z operačnej pamäte, výrazne by to znížilo ich výkonnosť pri riešení danej úlohy. Preto sa grafické procesory snažia skrývať dobu odozvy pamäte (angl. latency hiding). Pre tento účel je u nich umožnené prepnúť vykonávanie inštrukcií z vlákien, ktoré čakajú na prístup do pamäte na vlákna, ktoré sú pripravené na vykonávanie inštrukcií. Tento princíp sa zvykne označovať ako hardware multithreading. Rovnaký princíp sa využíva aj u moderných procesorov spoločnosti Intel, kde každé jadro dokáže prepínať medzi dvomi vláknami. Tento princíp je známy pod názvom hyperthreading. U grafických kariet je ale možné striedať o mnoho viac vlákien, ako v prípade spomínaných procesorov.

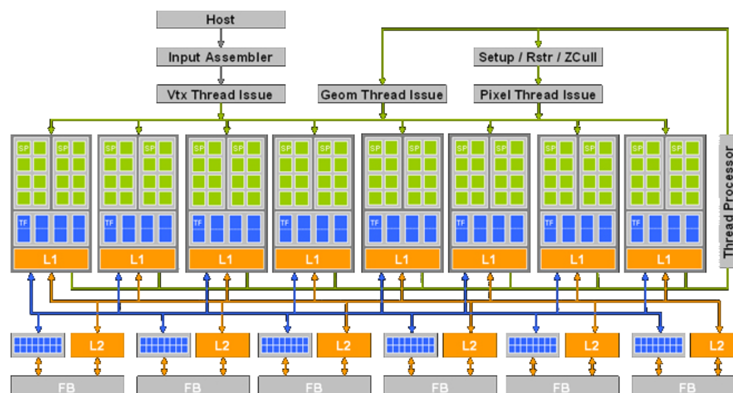
2.3 Moderná architektúra grafických kariet

Veľký pokrok v oblasti GPGPU nastal vydaním grafických kariet nVIDIA série 8 v roku 2006. Išlo o prvú architektúru s unifikovanými shader procesormi, ktoré sa všetky dali využiť ako pri spracovaní vrcholov, tak aj na spracovanie primitív a fragmentov (pixelov). Tieto procesory teda nahradili predchádzajúci pipeline model. Oproti pôvodnému návrhu táto architektúra umožnila lepšie využitie zdrojov, na rozdiel od zreťazeného spracovania, kde shader procesory na niektorej úrovni v zreťazenej linke boli menej vyťažené ako ostatné [4].

Vďaka týmto architektonickým zmenám to bola prvá grafická karta vhodná pre vykonávanie všeobecných výpočtov. Navyše spoločnosť nVIDIA pri vydaní tejto generácie vydala aj nástroje pre programovanie všeobecných výpočtov v mierne pozmenenom jazyku C nazvaný CUDA (Compute Unified Device Architecture)[1].

2.4 nVIDIA série 8

Najvýkonnejšou grafickou kartou tejto série sa stala Geforce 8800 GTS, ktorej architektúru môžeme vidieť na obrázku 2.2. Typ jej jadra má kódové označenie G80. Shader procesory, nazývané aj ako Stream procesory (SP), sú označné zelenými štvorčekmi. Pôvodné shader procesory v predchádzajúcej generácii grafických čipov boli v novej architektúre vlastne zjednotené do procesorov jedného typu. Preto sa často stretávame aj s pomenovaním ünified shaders". Namiesto jednej zreťazenej linky nová architektúra opakovane posiela dáta cez zredukovaný počet stupňov. Vstup je najprv poslaný cez zjednotený procesor, následne je výsledok uložený do registrov a v ďalšej fáze je znova poslaný na vstup jadra pre vykonanie ďalšej operácie. Ďalším rozdielom oproti starším generáciám je použitie skalárnych procesorov namiesto vektorových. Pôvodne boli tieto procesory vektorové, zvyčajne VLIW architektúra so štyrmi výkonnými jednotkami. Mali takú podobu z toho dôvodu, že pri spracovaní priestorovej scény sa často vykonávali operácie nad vektorom o šírke 4 (napríklad farba pixelu udaná v tvare R-G-B-A, alebo matice 4x4 pri výpočte 3D transformácií). Nevýhodou bolo to, že v prípade skalárnych inštrukcií sa využila len štvrtina dostupných výpočtových prostriedkov[4].



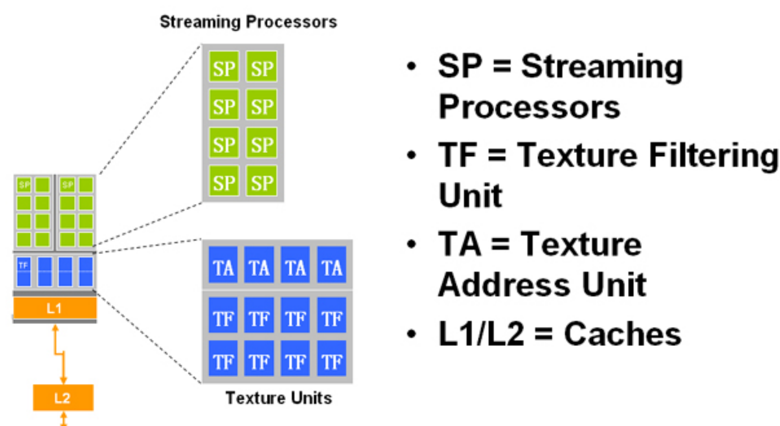
Obrázok 2.2: Grafická karta nVIDIA série 8. Prevzaté z [4]

Shader procesory sú zoskupené do väčších celkov, výpočtových "clusterov"po šestnástich SP. V dokumentoch od nVIDIE [5] sú označené ako "Thread Processing Cluster TPC. Jeden TPC sa delí na dve časti nazývané "Streaming Multiprocessor"(SM). Tieto jednotky zložené z viacerých procesorov sú typické pre moderné grafické karty oboch predných výrobcov.

Jeden TPC v karte GeForce 8800 GTS je zobrazený na obrázku 2.3. V oboch SM sú inštrukcie vykonávané synchrónne. Môže sa jednať o inštrukcie MAD (Multiply Add), MUL (Multiply), alebo prístupu do pamäti. Každý SM teda môžeme označiť za SIMD cluster. Na rozdiel od vlákien v SP, v každom TPC môže byť vykonávaný úplne rozdielny program. Nad

[1] http://www.nvidia.com/object/cuda_home_new.html

jednotkami TPC je plánovač, ktorý priraďuje vlákna jednotlivým SP a vyvažuje vyťaženie jednotlivých procesorov tak, aby sa celkový výkon využil čo najlepšie. Architektúru tejto grafickej karty môžeme v krátkosti označiť za MIMD (Multiple Instruction Multiple Data) architektúru zloženú zo SIMD clusterov [11].



Obrázok 2.3: Streaming multiprocessor grafickej karty nVIDIA série 8. Prevzaté z [4]]

Okrem SP obsahuje TPC aj jednotky pre filtrovanie textúr (TF) a adresovacie jednotky pre textúry (TA). V SM je možné načítavať textúry z pamäti čo znižuje zdržanie SP pri prístupoch do pamäti.

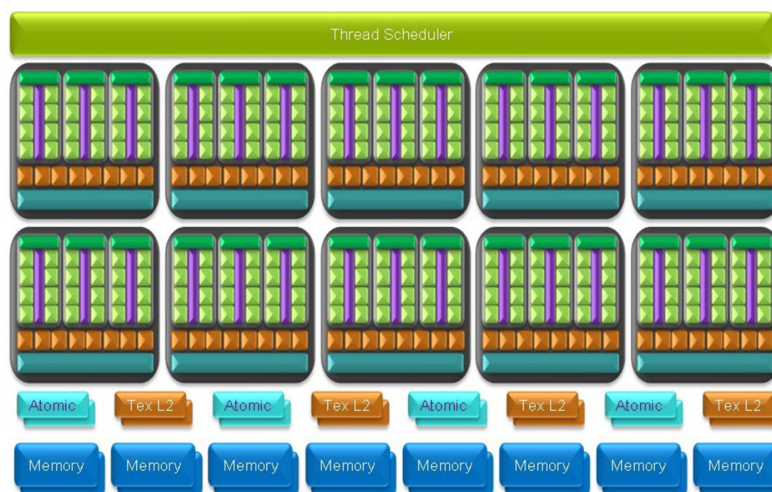
2.5 nVIDIA G86M

Čip s označením G86M vychádza z čipu G80. Odlišuje sa vo výrobnom procese (80nm oproti 90nm pri G80) a použitím pamäte DDR2 namiesto GDDR3. Dôvodom, prečo tento čip tu spomínam je to, že na tejto architektúre je postavená karta v mojom počítači (Quadro NVS 140M), ktorá bude použitá najmä v počiatočných testoch. Je ťažké nájsť špecifické informácie o čipe G86M, ale zo špecifikácií sa dajú vyvodiť určité uzávery. Tento grafický procesor obsahuje len šestnásť SP. Počet textúrovacích jednotiek je tak isto osemkrát menší ako pri grafickej karte 8800 GTS. Dá sa teda predpokladať, že Quadro NVS 140M obsahuje len jeden z ôsmych SM jednotiek. Ako mobilná grafická karta tiež pracuje na nižších frekvenciách. Podľa informácií z internetových stránok nvidia.com majú obe tieto karty "CUDA computing capability"1.1, čo je akási úroveň určujúca funkcie a schopnosti pre GPGPU výpočty jednotlivých grafických kariet. Keďže ide v zásade o rovnakú generáciu ako u čipoch G80, dá sa očakávať, že program optimalizovaný na čipoch G86M by mal fungovať optimálne aj na čipoch G80.

2.6 nVIDIA GTX 280

Druhou generáciou kariet so zjednotenou architektúrou od spoločnosti nVIDIA sa stala séria označená GT200. Vychádza zo základov, ktoré boli položené architektúrou G80.

Architektúra GT200 je zobrazená na obrázku 2.4. Oproti G80 sa zvýšil počet TPC z osem na desať. Okrem toho sa v TPC zvýšil počet SM z dvoch na tri, pričom počet procesorov v SM ostal nemenný. Celkový počet procesorov sa tým zvýšil zo 128 na 240 a teoretický výkon sa tiež zvýšil z 518 na 933 GFLOPS. Táto generácia tiež priniesla podporu dvojitej presnosti desatinných čísel (FP64). V každom SM sa nachádza jedna výpočtová jednotka pre počítanie s dvojitou presnosťou, čo je celkovo tridsať 64 - bitových procesorových jadier pre výpočty s dvojitou presnosťou. Grafická karta GTX 280 je schopná pri dvojitej presnosti dosiahnuť výkon 78 GFLOPS [5].



Obrázok 2.4: Architektúra grafickej karty nVIDIA GTX 280. Prevzaté z [5]

2.7 nVIDIA Fermi

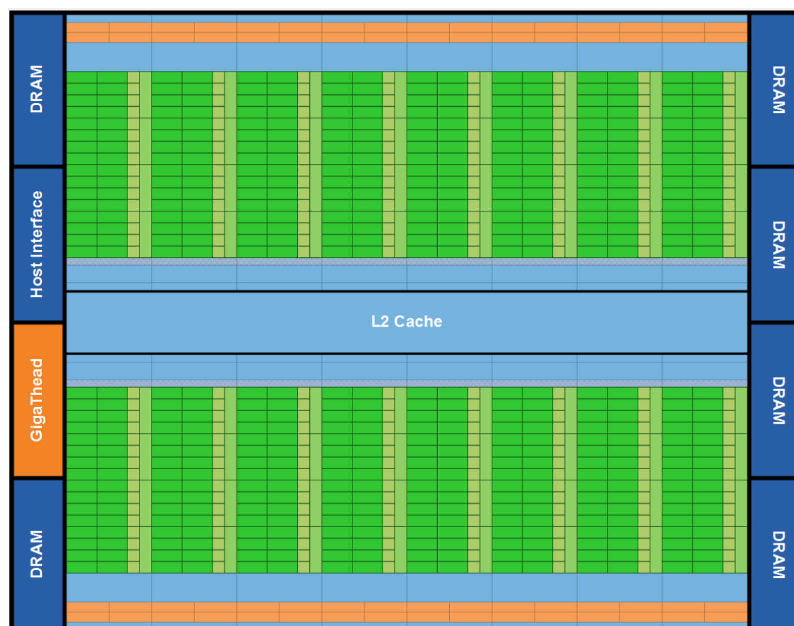
Fermi je označenie kariet série 4xx, ktorých kódové označenie je GF100. V tejto architektúre bol kladený dôraz na zvýšenie schopnosti karty vykonávať všeobecné výpočty. K tomu účelu priniesla táto architektúra nasledujúce vylepšenia [7]:

- Zvýšená výkonnosť výpočtov s desatinnými číslami s dvojitou presnosťou.
- Podpora pamätí s Error Correction Code (ECC).
- Skutočne hierarchické cache pamäte.
- Viac lokálnej pamäte shader procesorov.
- Rýchlejšie prepínanie kontextov
- Rýchlejšie atomické operácie.
- Vyššia presnosť operácií s desatinnými číslami podľa štandardu IEEE 754-2008. Implementuje operáciu "fused multiply add"(FMA), ktorá zvyšuje presnosť inštrukcie MAD tým, že nezaokrúhľuje medzivýsledok.

Pohľad z vysokej úrovne na architektúru grafických procesorov Fermi je na obrázku 2.5. Obsahuje 3 miliardy tranzistorov a 512 CUDA jadier (SP). Každé jadro môže za každý

hodinový cyklus vykonať jednu celočíselnú inštrukciu, alebo jednu inštrukciu s desatinnými číslami. Tieto jadrá sú organizované v šestnástich SM jednotkách po tridsaťdva SP. O prideľovanie úloh jednotlivým SM jednotkám sa stará GigaThread plánovač, ktorý posiela úlohy lokálnym plánovačom na úrovni SM. Architektúra Fermi obsahuje dva plánovače na úrovni jednotiek SM 2.6. Tieto plánovače plánujú vlákna do skupín vlákien po 32 nazývaných "warps". Dva plánovače sú v jednom SM za účelom súčasného spracovania dvoch warpov. Tie môžu byť naraz pridelené ku skupine šestnástich SP, k šestnástim Load/Store jednotkám, alebo ku štyrom jednotkám pre špeciálne funkcie (SFU - Special Function Unit). Vďaka tomuto prístupu je na SM jednotkách dosahovaný takmer vrcholný výkon [7].

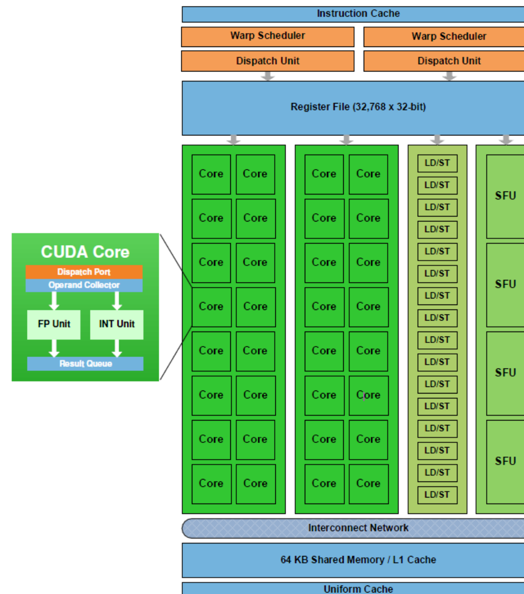
To, že táto architektúra bola navrhnutá z ohľadom na všeobecné výpočty, je vidno aj na jej schopnosti vykonávať operácie s dvojitou presnosťou. Fermi je schopná 256 FMA operácií s dvojitou presnosťou oproti 512 FMA operácií s jednoduchou presnosťou. Pri vyššej presnosti sa teda výpočty spomalia len o polovicu. Naproti tomu pri GT200 to bolo 240 MAD operácií s jednoduchou presnosťou a 30 FMA operácií s dvojitou presnosťou. V tomto prípade je teda spomalenie osemnásobné.



Obrázok 2.5: Architektúra grafickej karty nVidia Fermi.Prevzaté z [7]

Najvýkonnejšia karta tejto série nVIDIA Geforce GTX 480 vraj musela mať z technických problémov niektoré SM jednotky vypnuté. Tieto problémy boli vyriešené pri ďalšej generácii čipov s označením GF110. Najvýkonnejšia grafická karta tejto série bola GTX 580 a dá sa považovať za vynovenú verziu GF100 [2].

[2]http://www.tomshardware.com/reviews/geforce-gtx-580-gf110-geforce-gtx-480,2781.html



Obrázok 2.6: Streaming multiprocessor grafickej karty nVidia Fermi.Prevzaté z [7]

2.8 nVIDIA Kepler

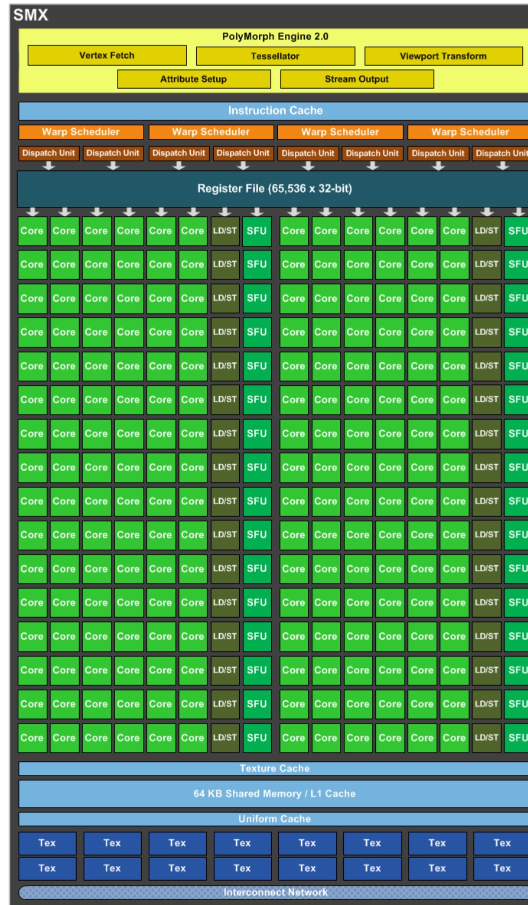
Grafické karty nVIDIA Kepler boli vydané v prvej polovici roku 2012. Oproti predchádzajúcej generácii majú výrazne vyšší počet procesorov - SP. Sú zrejme jednoduchšie a teda zaberajú menšiu plochu čipu. Konkrétne grafická karta GTX 680 obsahuje 1536 Cuda jadier. Grafické jadro je taktované na 1005 MHz, čo vo výsledku umožnilo veľmi vysoký výkon pri počítaní s číslami v jednoduchej presnosti FP32, ktorý sa blíži k teoretickému maximu 3090 GFLOPs. V materiáloch od spoločnosti nVIDIA sa však neuvádza výkon v dvojitej presnosti desatinných čísel FP64. Ani jednotky SM neboli popísané do takej miery, aby sa tento údaj dal odhadnúť. Táto karta v testoch údajne v tomto smere zaostáva za predchádzajúcou sériou. Neoficiálne je výkon pri počítaní s desatinnými číslami v dvojitej presnosti u tejto karty šestnásťkrát nižší, ako pri jednoduchej presnosti [3]. Odzrkadľuje to fakt, že ide o grafickú kartu určenú primárne na hry, kde je dôležitý predovšetkým výkon pri jednoduchej presnosti. Na GPGPU sú určené grafické karty rady Tesla.

Blokový diagram zobrazujúci architektúru grafického jadra v karte GTX 680 je na obrázku 2.7. Skladá sa zo štyroch častí ktoré nVIDIA nazýva "Graphics Processing Clusters"(GPC). V každom z nich je jeden SM, ktorý teraz v nVIDII nazvali "next-generation Strieaming Multiprocessors"(SMX). Čip je vybavený štyrmi radičmi pamätí. V karte GTX 680 má každý GPC dve SMX jednotky. S celkovo ôsmimi SMX po 192 SP obsahuje celkovo 1536 SP - Cuda jadier. Tak, ako aj v predchádzajúcej architektúre, je v nad SMX jadrami plánovač nazvaný Giga Thread Engine. SMX jednotky zdieľajú pamäť L2 Cache, ktorá je pripojená k pamäťovým radičom.

[3]http://parallelis.com/kepler-underperform-on-gpgpu-gtx680/

Obrázok 2.7: Grafická karta nVidia Kepler. Prevzaté z [8]

Detailnejší pohľad na SMX Cluster je na obrázku 2.8. Procesory SP v tejto pamäti vykonávajú operácie nad pixelmi, vrcholmi a geometriou a tiež všeobecné výpočty. Jednotky Load/Store (Na obrázku LD/ST) sa starajú o zápis a čítanie z pamäte a textúrovacie jednotky vykonávajú filtrovanie textúr. Naproti tomu jednotky pre špeciálne funkcie (SFU - Special Function Unit) zvládajú transcendentálne inštrukcie a grafickú interpoláciu. Na koniec Polymorph Engine vykonáva načítanie vrcholov z pamäte, teseláciu a iné. SMX obsahuje tiež štyri plánovače, kde každý z nich je schopný vyslať dve inštrukcie na jeden warp v každý hodinový cyklus [8].



Obrázok 2.8: Streaming multiprocessor grafickej karty nVidia Kepler. Prevzaté z [8]

2.9 Grafické karty ATI/AMD Radeon

Prvé grafické karty od spoločnosti AMD, ktoré podporujú OpenCL bola séria HD 4000. Predchádzajúca generácia HD 3000 (RV670) s architektúrou VLIW5 mala tiež určitú softwarovú podporu pre GPGPU. Spoločnosť AMD poskytovala vysoko úrovňový jazyk BROOK+, čo bolo rozšírenie jazyka C/C++ o nové kľúčové slová a syntax. Okrem jazyka BROOKS sa dal použiť aj nízko úrovňový jazyk nazývaný compute abstraction layer (CAL). Navyše v jadre RV670 bola implementovaná podpora pre výpočty s desatinnými číslami o dvojitej presnosti. [14]. Tieto generácie ale pri GPGPU výpočtoch ešte neboli veľmi efektívne. Ďalšia generácia, HD 5000, už bola lepšie navrhnutá pre všeobecné výpočty [15]. Najnovšia séria, architektúra nazývaná "Graphics Core Next"(GCN) posúva ešte ďalej koncept grafickej karty vhodnej na všeobecné výpočty. Aj preto sa o nej veľakrát hovorí ako o Fermi od AMD.

Spoločnosť AMD sa dlho držala architektúry VLIW. Prvá generácia s architektúrou VLIW od AMD bola vydaná už v roku 2007 ako séria HD2000 [20]. Ako už bolo spomenuté v predchádzajúcich častiach, architektúra VLIW bola vhodná najmä na spracovanie grafiky. Aj pri hrách sa zavedením API (Application Programming Interface) DirectX verzie 10 sa začali meniť aritmetické úlohy či zloženie inštrukcií, ktoré vynakladali grafické karty. Okrem grafiky sa navyše začali spracovávať výpočty fyzikálnych simulácií v herných

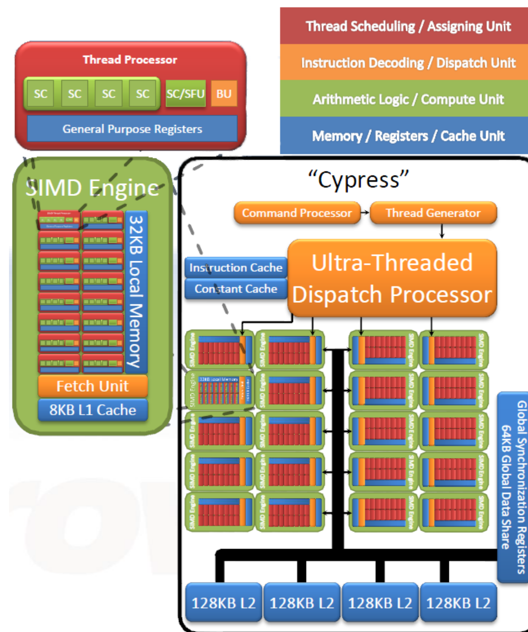
prostrediach. Aby boli pri všeobecných výpočtoch aj pri moderných grafických aplikáciách výpočtové prostriedky lepšie využité, spoločnosť AMD prešla z architektúry VLIW5 na VLIW4 s architektúrou Cayman (HD6970). Najväčšie zmeny a prispôsobenie pre všeobecné výpočty však prišli o ďalšiu generáciu neskôr s architektúrou GCN [20]. V nasledujúcich častiach budú popísané architektúry VLIW5, VLIW4 a GCN.

2.10 ATI/AMD Evergreen

Evergreen (karty Radeon HD 5xxx) je architektúra založená na princípe VLIW5. V ďalšej generácii po AMD Evergreen už AMD prechádzalo na VLIW4, aj keď len pri najvýkonnejšom modeli HD6970. Architektúra VLIW5 sa podobne ako pri architektúrach od spoločnosti nVIDIE rozdeľuje na viacero clusterov, v tomto prípade nazvaných SIMD Engine. Jej bloková štruktúra je na obrázku 2.9. Môžeme ju prirovnať k SM jednotke pri architektúrach od spoločnosti nVIDIE. Úlohy prideľuje jednotlivým clusterom Ultra-Thread Dispatch Processor. Tiež sa tu uplatňuje princíp viacerých úrovní cache pamäte. Okrem pamäte L2 je ešte v každom multiprocesore 32KB cache L1. Architektúra Cypress obsahuje dvadsať týchto multiprocesorov. V každom SIMD multiprocesore sa nachádza šestnásť výpočtových jednotiek nazvaných Thread Processor (TP). Na tejto úrovni sa dostávame k významu názvu tejto architektúry - VLIW5. V každom TP sa totiž nachádza päť procesorov Stream Core (SC), z ktorých posledný je rozšírený o schopnosť vykonávať špeciálne funkcie, ako napríklad transcendentálne operácie. Štyri jednoduchšie procesory dokážu v jednom hodinovom cykle vykonať jednu z nasledujúcich možností [17]:

- 4x 32- bitová operácia FP Fused MUL + ADD
- 2x 64-bitová operácia Dual Precision (DP) násobenie alebo sčítanie
- 1x 64-bitová operácia DP Fused MUL + ADD
- 4x 24-bitová operácia s celými číslami (Integer) násobenie alebo sčítanie

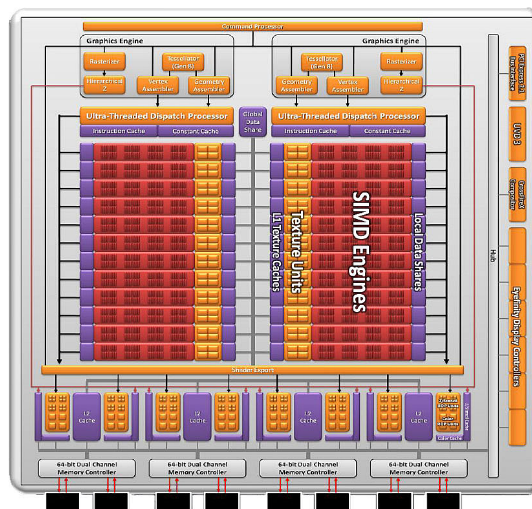
Funkcia týchto procesorov je riadená inštrukciou, kde nachádza kód pre každý procesor. V prípade potreby niektoré jadrá nemusia vykonávať žiadnu inštrukciu, ak sa program nedá rozdeliť na päť navzájom nezávislých inštrukcií. V tom prípade sa samozrejme znižuje výkon, aký tento grafický procesor dosahuje. Práve zlepšenie využitia výpočtových prostriedkov, ako už bolo naznačené, je dôvod prechodu z architektúry VLIW5 na VLIW4.



Obrázok 2.9: Grafická karta AMD/ATI Evergreen. Prevzaté z [17]

2.11 ATI/AMD Cayman

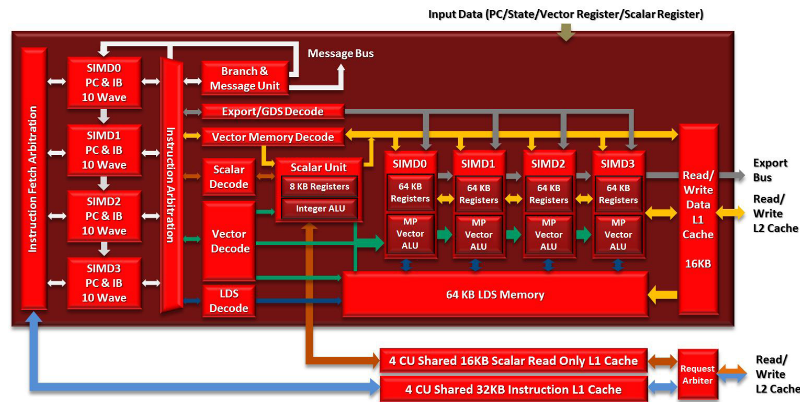
Architektúra Cayman sa nachádzala v grafických čipoch Radeon HD6970, prípadne v nižších modeloch s vypnutou časťou procesorov. Schéma danej architektúry je zobrazená na obrázku 2.10. Zariadenie je rozdelené na dve polovice, kde sa v každej z nich stará o plánovanie a zasielanie inštrukcií jeden plánovač (Ultra-Thread Dispatch Processor, alebo wave scheduler). Celkovo sa na jednom čipe nachádza dvadsaťštyri SIMD Engines, kde každý obsahuje šestnásť VLIW procesorov (SPU). Každý z týchto procesorov obsahuje privátnu cache pamäť úrovne L1, ako aj lokálnu zdieľanú pamäť [20]. Na rozdiel od VLIW5, však každý SPU obsahuje len štyri výpočtové jednotky. Z SP bol teda odstránený piaty procesor určený pre špeciálne funkcie. Transcendentálne operácie teda teraz vykonávajú tri zo štyroch výpočtových jednotiek vo VLIW procesore. Vďaka tomu sa zjednodušili VLIW procesory, na plochu čipu sa ich zmestilo viac. Vo výsledku to znamenalo zvýšenie výkonu na plochu čipu.



Obrázok 2.10: Grafická karta AMD/ATI Cayman. Prevzaté z [20]

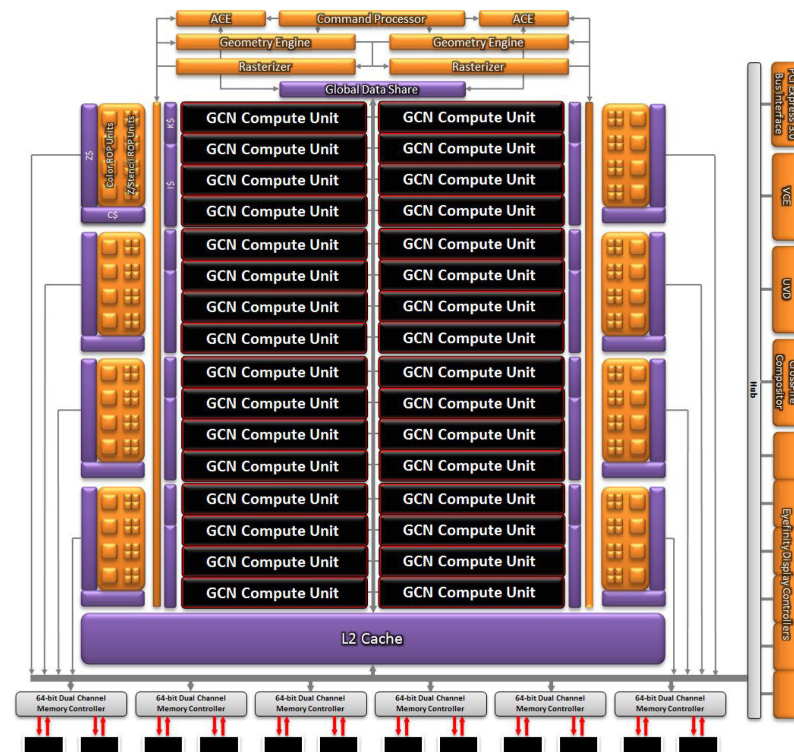
2.12 AMD Graphics Core Next (GCN)

V dobe písania tejto práce ide o najnovšiu architektúru grafického čipu od spoločnosti AMD. Logika čipu je založená na viacerých výpočtových jednotkách, ktorých vnútorná štruktúra je zobrazená na obrázku 2.11. V prípade karty AMD Radeon HD 7970 je ich tridsaťdva a v každom z nich sa nachádzajú maximálne štyri SIMD procesory, pričom každý z nich je schopný vykonávať paralelne šestnásť programových vlákien, ale každý z týchto procesorov je schopný pracovať na inom wavefronte. Výpočtové jednotky - GCN jadrá majú úplne novú inštrukčnú sadu, ktorá je jednoduchšia pre prekladače a vývojárov. Každé GCN jadro má kombináciu zdieľaných a privátnych zdrojov. Zásobníky inštrukcií, registre a vektorové aritmeticko-logické jednotky sú privátne pre všetky štyri SIMD jadrá. Iné zdroje, ako napríklad dátová cache pamäť sú zdieľané kvôli energetickej efektivite. V prípade cache pamäti bol u novej architektúre implementovaný koherenčný protokol, ktorý zdieľa dáta cez L2 cache pamäť, ktorá je výrazne rýchlejšia a efektívnejšia ako grafická pamäť mimo čipu grafického procesora. Okrem týchto vylepšení bola cez hardware a programové ovládače v architektúre GCN implementovaná virtuálna pamäť kompatibilná s virtuálnou pamäťou u procesorov x86, čo uľahčuje prenášanie dát medzi procesorom a grafickou kartou. Tiež je to ďalší krok k bližšej integrácii procesorových a grafických jadier na jednom čipe a k heterogénnej architektúre [13]. Týmto kombinovaným čipom bude venovaná nasledujúca časť textu.



Obrázok 2.11: Grafické jadro GCN. Prevzaté z [13]

Najrýchlejší grafický čip tejto série, AMD Radeon HD 7970 GHz dosahuje maximálny výkon pri dvojitej presnosti viac ako 1 TFLOPS a 4 TFLOPS pri jednoduchej presnosti FP32. Jeho blokový diagram je na obrázku (Obrázok 12).



Obrázok 2.12: Grafická karta AMD/ATI Cayman. Prevzaté z [13]

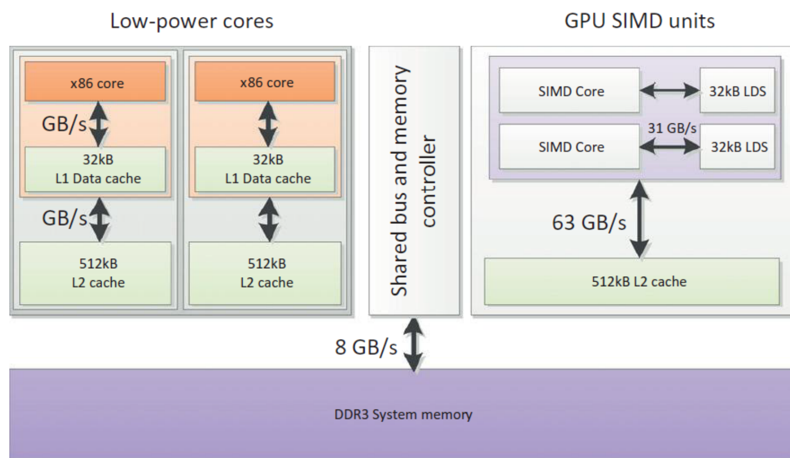
2.13 Accelered processing unit (APU)

Termín uvedený v nadpise používa spoločnosť AMD ako označenie pre sériu svojich čipov obsahujúcich procesorové jadrá, ako aj grafické jadro. Ako APU môžeme označiť aj moderné procesory od spoločnosti Intel generácie Sandy Bridge a tiež najnovšej generácie Ivy

Bridge. Takéto skombinované výpočtové jednotky sú typické už dlhšiu dobu pre procesory v mobilných zariadeniach, kde sa vo veľkej miere uplatňujú integrované obvody označované ako System on Chip (SoC). V nich je na jednom čipe umiestnená logika, ktorá je u iných zariadení umiestnená na viacerých čipoch pospájaných doskou plošného spoja.

Výhodou týchto kombinovaných procesorov je možnosť zdieľania niektorých zdrojov, ako napríklad operačnej pamäte. To umožňuje zásadne znížiť čas na kopírovanie zadania výpočtovej úlohy do grafickej karty. Tým je aj umožnená užšia integrácia algoritmu medzi GPU a CPU jadrami, čo pri diskrétnej grafickej karte nie je možné kvôli zdržaniu na zbernici PCI Express. Na druhej strane to prináša nevýhodu v tom smere, že grafické jadro musí využívať rovnakú pamäť, ktorá je prispôsobená skôr na rýchlu odozvu, ako na veľkú priepustnosť, čo môže pri určitých algoritmoch spôsobiť zníženú výkonnosť, ktoré potrebujú často pristupovať do pamäti. Ale algoritmy, ktoré nemôžu fungovať efektívne na grafických kartách kvôli tomu, že grafické karty nie sú optimálne na vykonávanie sériového kódu, môžu lepšie fungovať na zariadeniach typu APU, ak sa výpočet vhodne rozdelí medzi CPU a GPU.

Ako príklad pre APU môžeme uviesť procesor AMD E350 "Zacate"2.13. Ide o procesor pre mobilné zariadenia, ako sú lacné notebooky a tablety. Na jeho čipe sú dve procesorové jadrá a grafický procesor, ktorý sa skladá z dvoch SIMD jadier, kde každé z nich je schopné vykonať naraz osem VLIW inštrukcií zložených z piatich elementárnych inštrukcií. Jeho grafická karta je teda zrejme založená na architektúre AMD VLIW5.



Obrázok 2.13: AMD E350 Zacate. Prevzaté z [10]

Kapitola 3

Nástroje pre programovanie všeobecných výpočtov na grafických kartách, OpenCL

Dnes sa na využitie grafických kariet na všeobecné výpočty nemusí používať žiaden trik v podobe kódovania úlohy do grafických entít. Ako už bolo vyššie spomenuté, grafické karty sa dnes už aj navrhujú s ohľadom na ich využiteľnosť v širokej škále úloh. Zároveň s nimi sa vyvíjajú aj nástroje, ktoré umožňujú širokej skupine vývojárov a programátorov tieto procesory využiť. V tejto kapitole stručne popíšem nástroje pre programovanie všeobecných výpočtov na grafických kartách. Dôraz bude kladený hlavne na štandard OpenCL, z toho dôvodu, že ten bude použitý pri mojej práci.

Najznámejšie z týchto programových nástrojov sú CUDA Toolkit od spoločnosti nVIDIA, otvorený štandard OpenCL[10] a nástroj vyvíjaný spoločnosťou Microsoft nazývaný Direct Compute. CUDA Toolkit podporuje iba grafické karty od spoločnosti nVIDIA. DirectCompute však umožňuje použitie grafických kariet od rôznych výrobcov. Na druhej strane je však použiteľný len na operačnom systéme Microsoft Windows a nie je prenositeľný medzi operačnými systémami.

OpenCL je otvorený štandard pre heterogénny programovací systém, ktorý spravuje neziskové technologické konzorciom Khoros Group. Vďaka otvorenosti spomínaného štandardu môže každý výrobca výpočtových systémov vytvoriť svoju vlastnú implementáciu OpenCL. Vďaka tomu tiež OpenCL podporuje veľké množstvo zariadení. To je hlavný rozdiel oproti nástroju CUDA Tookit, čo je uzavretý nástroj a jeho využitie je viazané len na grafické karty od spoločnosti nVIDIA. OpenCL okrem grafických kariet od rôznych výrobcov umožňuje tvoriť program aj na viacjadrové procesory a dokonca aj na niektoré zariadenia typu FPGA. Vďaka tomu je OpenCL, čo sa týka prenositeľnosti veľmi dobré riešenie. Na druhej strane jeho nevýhodou oproti CUDA Toolkitu je, že sa jedná o mladý štandard. Pre účely tejto práce je ale prenositeľnosť veľkou výhodou, lebo vďaka nej bude môcť výsledok tejto práce použiť širšia skupina vývojárov. Práve preto bude táto kapitola venovaná hlavne OpenCL. Inak sú ale princípy oboch nástrojov v podstate rovnaké a mnohokrát sú odlišnosti len v terminológii.

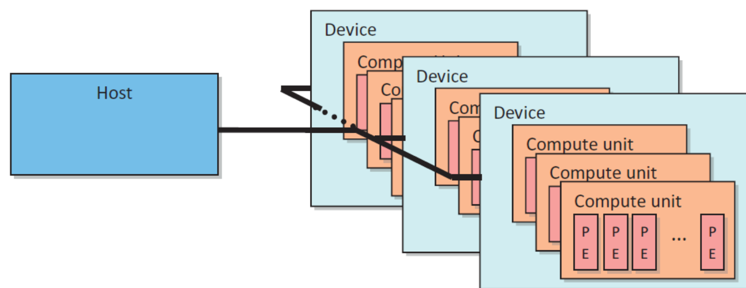
Ako už bolo spomenuté vyššie, štandard OpenCL umožňuje programovať širokú škálu procesorov a iných akcelerátorov. OpenCL API (Application Programming Interface) sa vyznačuje nielen obecnosťou, ale aj adaptabilitou umožňujúcou dosiahnuť vysoký výkon na rôznych platformách.

V špecifikácii OpenCL je definovaná akási abstraktná reprezentácia použitého výpočtového zariadenia. Táto špecifikácia je definovaná na štyroch úrovniach, sú to: model platformy, model spúšťania, pamäťový model a model programovania.

3.1 Model platformy (Platform model)

Model platformy špecifikuje výpočtový systém ako procesor koordinujúci vykonávanie programu nazývaný ako "host"(hostiteľské zariadenie) a jedno alebo viac procesorov schopných vykonávať OpenCL C kód - zariadenia označované ako "device"(OpenCL zariadenie). OpenCL C je obmedzená verzia jazyka v štandarde C99 s rozširujúcimi kľúčovými slovami vhodnými pre tvorbu dátovo-paralelného kódu. Funkcie definované v tomto jazyku pre vykonávanie na strane OpenCL zariadení sa nazývajú kerneli (kernels).

Model platformy je znázornený na obrázku 3.1. Vykonávanie programu riadi jedno hosťujúce zariadenie ako napríklad procesor architektúry x86. K tomuto procesoru je pripojených niekoľko OpenCL zariadení. Ako typický príklad možno uviesť grafický procesor prípadne iný akcelerátor schopný vykonávať kód definovaný jazykom OpenCL C.



Obrázok 3.1: Model platformy OpenCL. Prevzaté z [10]

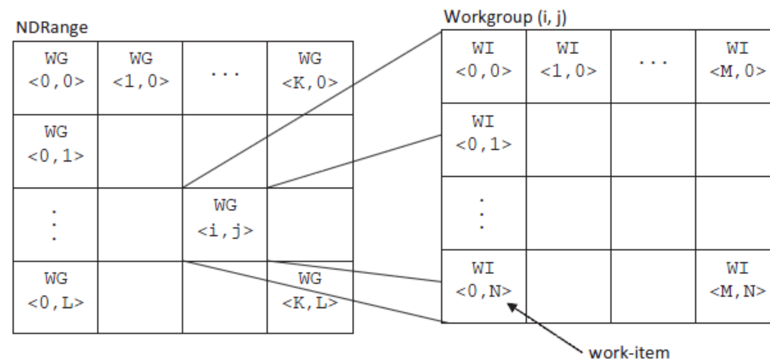
Na platformu sa môžeme pozerať ako na implementáciu OpenCL API od špecifického výrobcu hardvéru. Ak je zvolená platforma jednej spoločnosti, nemôžeme využiť zariadenie od inej spoločnosti. Platforma tiež definuje mapovanie abstraktnej architektúry na fyzický hardvér. Ako príklad môžeme uviezť mapovanie abstraktnej architektúry a grafickú kartu AMD Radeon 6970. V abstraktnej architektúre patrí pod OpenCL zariadenie (Device). Táto karta obsahuje dvadsaťštyri SIMD jadier. Tieto sú modelované ako výpočtové jednotky (Compute unit). Každé SIMD jadro obsahuje šestnásť VLIW procesorov, ktoré prislúchajú k jednotke spracovávania (Processing Element).

3.2 Model spúšťania

Model spúšťania definuje ako je konfigurované prostredie OpenCL na strane hostiteľského zariadenia (host) a ako sú kerneli vykonávané na strane OpenCL zariadení. To zahŕňa vytvorenie kontextu OpenCL na hosťujúcom zariadení, poskytnutie mechanizmu pre interakciu medzi hosťujúcim a OpenCL zariadením a definovanie modelu súbežného spracovania použitého pri vykonávaní kernelu na OpenCL zariadení.

Ako veľa iných nástrojov pre programovanie paralelných výpočtov , aj OpenCL je syntakticky podobný s jazykom C. Prináša navyše kľúčové slová, ktoré rozširujú tento programovací jazyk. Medzi najdôležitejšie patria funkcie pre identifikáciu jednotlivých progra-

mových vlákien pomocou jednorozmerných až trojrozmerných indexov. Indexovanie jednotlivých procesov je znázornené na obrázku (obrázok 19). Na najvyššej úrovni máme "NDRange"(n-dimensional range, n-rozmerný rozsah), čo je vlastne jedno-, dvoj- alebo trojrozmerný indexový priestor vlákien . Pri spúšťaní kernelu sa špecifikuje jeho veľkosť v jednom, dvoch alebo troch rozmeroch a tým sa aj určí maximálny počet bežiacich procesov v aplikácii, ktorý sa zvyčajne mapuje na rozmery vstupov, alebo výstupov. Tieto procesy potom môžeme rozdeliť do menších skupín nazývaných "work group", na ktoré sa ďalej budem odvolávať ako na skupinu procesov. Rozdelenie sa určí nastavením veľkosti týchto skupín procesov vo všetkých používaných rozmeroch. OpenCL vyžaduje, aby bol indexový priestor rovnomerne rozdeliteľný na skupiny procesov o daných rozmeroch. Vlákna v jednej skupine procesov majú medzi sebou špeciálny vzťah. Môžu sa vzájomne synchronizovať špeciálnym príkazom "barrierä navyše majú všetky prístup do jednej zdieľanej pamäti. U grafických kariet je táto vlastnosť dosiahnutá tým, že sú všetky vlákna v jednej skupine procesov spúšťané na jednom SM. Pre optimálne vyťaženie zariadenia a škálovateľnosť algoritmu je dôležité nastaviť NDRange vhodne. Veľakrát ale nie je jednoduché určiť najvhodnejšie parametre a je nutné ich zistiť testovaním.



Obrázok 3.2: OpenCL NDRange. Prevzaté z [10]

V modeli spúšťania sa ďalej definuje spôsob komunikácie medzi hosťujúcim zariadením a akcelerátorom, prostredie pre spustenie (vytvorenie kontextu, fronty príkazov, udalosti pre reprezentáciu závislostí, spôsob alokovania pamäte na OpenCL zariadení) a podobne [10].

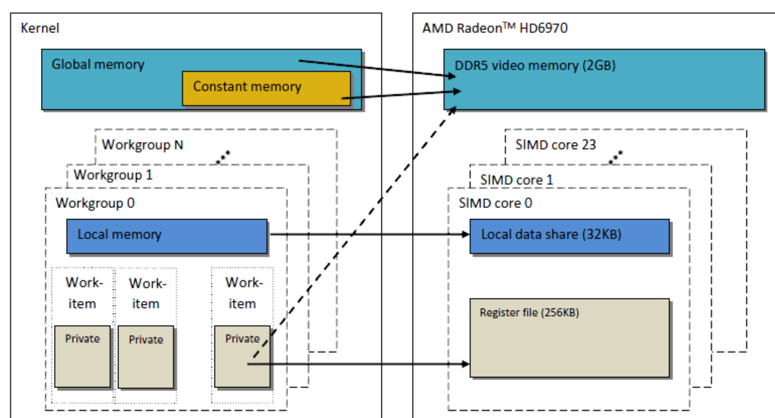
3.3 Pamäťový model

Vo výpočtových zariadeniach sa ich pamäťový podsystém obecne odlišuje. Preto je pre prenositeľnosť programov medzi rozličnými zariadeniami nutné, aby bol v OpenCL definovaný pamäťový model. Ten poskytuje programátorovi rôzne druhy pamäte, s ktorými môže pracovať. Je potom na výrobcovi hardvéru, aby pri svojej implementácii štandardu OpenCL definoval mapovanie z abstraktného modelu do fyzickej pamäte.

V pamäťovom modeli OpenCL sú definované nasledujúce typy pamäte: Globálna pamäť je viditeľná pre všetky výpočtové jednotky a tiež je to pamäť využívaná na komunikáciu s hosťujúcim zariadením, lebo hosťujúce zariadenie môže pri zápise a čítaní pristupovať len do tejto pamäti. Pamäť konštánt je navrhnutá pre prípad, kedy na každý element pristupuje simultánne viac vlákien. Lokálna pamäť má pre každý multiprocesor (SM) unikátny

adresový priestor. Do tejto pamäte majú prístup len tie vlákna, ktoré sú v tej istej skupine procesov podľa definície NDRange pri spustení kernelu. Vďaka obmedzeniu prístupu môže byť prístup do nej mnohonásobne rýchlejší, ako v prípade globálnej pamäti. Privátna pamäť je unikátna pre individuálny proces (work-item). Využíva sa pre uloženie lokálnych premenných a parametrov kernelu, ktoré nie sú ukazovateľom.

Pamäťový model OpenCL a príklad jeho mapovanie na grafickú kartu AMD Radeon HD6970 je na obrázku (Obrázok 20).



Obrázok 3.3: Mapovanie pamäťového modelu OpenCL na fyzický hardware. Prevzaté z [16]

Kapitola 4

Umelé neurónové siete

Obsahom tejto kapitoli budú základné princípy umeleých neurónových sieti. Hlavne dopredná neurónová sieť a algoritmus backpropagation. Tento algoritmus bude rozobratý podrobnejšie a to z toho dôvodu, že súčasťou tejto práce bude jeho implementácia.

Umelá neurónová sieť (UNS) je abstraktný počítačový model ľudského mozgu. Odhaduje sa, že ľudský mozog obsahuje $10^{1}1$ malých jednotiek nazývaných neuróny, ktoré sú prepojené $10^{1}5$ spojeniami. Aj keď o funkcii mozgu veľa vecí nevieme, považuje sa za zdroj inteligencie, ktorá zahŕňa vnímanie, poznávanie a učenie sa živých bytostí [18]. Myšlienkou UNS nie je presne kopírovať činnosť mozgu, ale využiť to, čo je známe o fungovaní biologických sietí pre riešenie komplexných problémov [12].

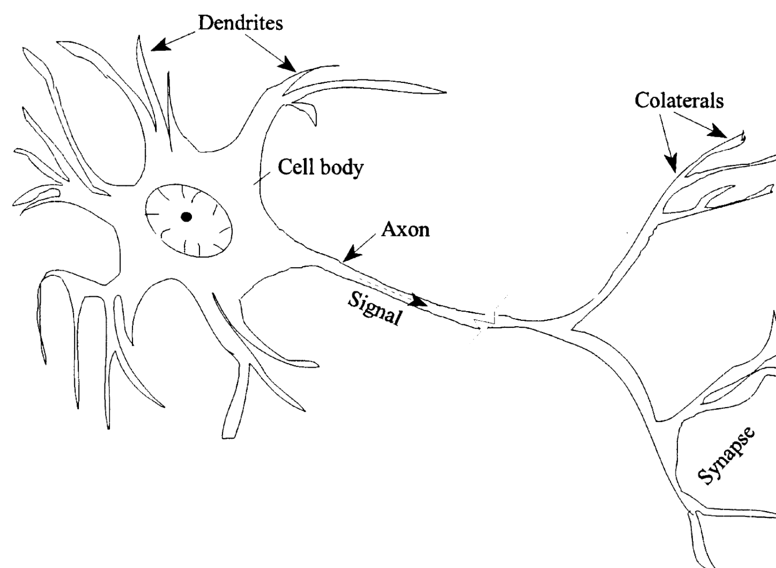
Podobne ako pri ľudskom mozgu, je UNS zložená z umelých neurónov a spojení medzi nimi. Aj keď pri umelých neurónových sieťach počítame zvyčajne s oveľa menej neurónmi, aj tak pomáhajú riešiť problémy umelej inteligencie, pri ktorých sa stretávame s neurčitosťou a chybami vo vstupných dátach.

V tejto kapitole bude najprv načrtnutý základný princíp umelých neurónových sietí. Potom budú predstavené najpoužívanejšie typy neurónových sietí s dôrazom na tie, ktoré budú implementované v programovej časti tejto práce. Na záver bude uvedený základný algoritmus pre učenie vybraných typov sietí.

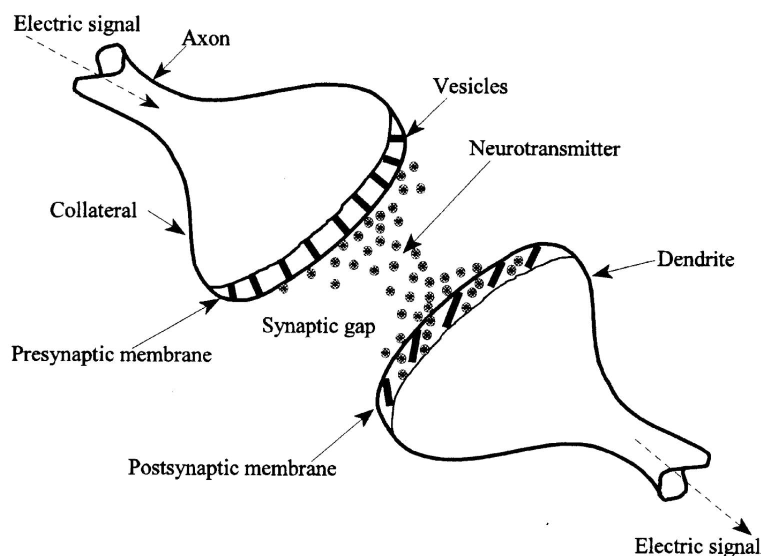
4.1 Základné princípy umelých neurónových sietí

Ako už bolo spomenuté v úvode tejto kapitoly, UNS je abstraktný model inšpirovaný ľudským mozgom. Inšpiráciou pre základný stavebný prvok UNS je biologický neurón, ktorého zjednodušený nákres je zobrazený na obrázku 4.1. Biologický neurón obsahuje tri hlavné funkčné časti a to dendrity, bunkové telo a axón. Dendrity prijímajú signály z ostatných neurónov a posielajú ich do tela bunky. Axón, ktorý sa rozvetvuje do súbežných výbežkov nesie signál z tela bunky do synapsii (mikroskopická medzera) a do dendritov susedných neurónov. Schématické znázornenie prenosu signálu v synapsijách je uvedené na obrázku 4.2. Impulz vo forme elektrického signálu putuje smerom k pre-synaptickej membráne. Pri príchode signálu k membráne je z dutín (vesicles) vylúčená chemikália nazývaná neurotranzmiter a to v množstve úmernému k intenzite prichádzajúceho signálu. Neurotranzmiter sa rozptýli v synaptickej medzere k post-synamtickej membráne a nakoniec do dendritov susediaceho neurónu , čím ho prinúti (podľa prahu prijímajúceho neurónu) generovať nový elektrický signál. Intenzita signálu, ktorá prejde cez prijímajúci neurón závisí od intenzity signálu každého neurónu, ktorý je k nemu pripojený, od sily ich synaptického spojenia a pra-

hu prijímajúceho neurónu. Vstupné signály môžu pomáhať (excitovať), alebo potlačovať spúšťanie generovania signálu daného neurónu [12].



Obrázok 4.1: Biologický neurón. Prevzaté z [12]



Obrázok 4.2: Synaptická medzera. Prevzaté z [12]

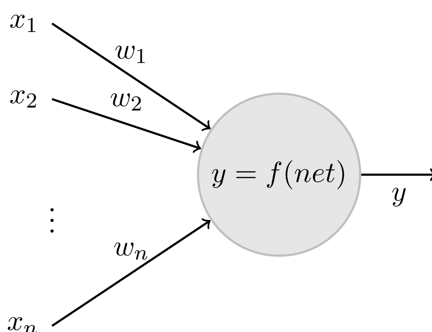
Pri UNS biologický neurón nahrádzame jeho abstrakciou, ktorá je zobrazená na obrázku 4.3. Tento umelý neurón je základným stavebným kameňom pre UNS. Má niekoľko vstupov z predchádzajúcich neurónov $(x_1, x_2, ...x_n)$, ktoré sú pripojené k neurónu cez pripojenia s prislúchajúcimi váhami $(w_1, w_2....w_n)$. Pri simulácii priechodu signálu sa najprv vynásobia výstupné hodnoty predchádzajúcich neurónov $(x_1, x_2, ...x_n)$ s ich odpovedajúcimi váhami $(w_1, w_2....w_n)$. Celkový vstup do neurónu je spočítaný ako súčet všetkých vážených

vstupných hodnôt. V terminológii neurónových sietí sa táto hodnota väčšinou označuje ako net. Funkcia, ktorú neurónová sieť používa na spočítanie hodnoty net sa zvyčajne označuje ako bázová funkcia (f) [1]. Bázová funkcia opísaná v predchádzajúcich riadkoch vyjadrená vzorcom ako

$$net = \sum_{i=1}^{n} w_i x_i \tag{4.1}$$

sa nazýva lineárna bázová funkcia. Okrem lineárnej bázovej funkcii sa zvykne často používať aj radiálna bázová funkcia vyjadrená nasledujúcim vzorcom.

$$net = ||\vec{x} - \vec{w}|| = \sqrt{\sum_{i=1}^{n} (x_i - w_i)^2} \tag{4.2}$$



Obrázok 4.3: Schéma umelého neurónu

Hodnota net je vstupom takzvanej aktivačnej alebo prenosovej funkcie, ktorú môžeme označiť ako $g(net)$. Každý umelý neurón má svoju aktivačnú funkciu. Obecne sa dá výpočet intenzity výstupného signálu zo vstupov vyjadriť vzťahom $y = g(f(\vec{x}))$. Na záver je výsledok aktivačnej funkcie distribuovaný do vstupov ostatných neurónov cez vážené spojenia. Model neurónu na obrázku 4.3 s lineárnou bázovou funkciou sa nazýva Perceptrón. Jeho výstup je daný vzťahom:

$$y = f(net) = f(\vec{w} \cdot \vec{x}) = f\left(\sum_{j=1}^{n+1} w_j x_j\right) = f\left(\sum_{j=1}^{n+1} w_j x_j - \theta\right) \tag{4.3}$$

Pri tejto notácii predpokladáme, že má umelý neurón $n + 1$ vstupov. Posledný vstup je vždy rovný -1 a $w_{n+1} = \theta$, čo je hodnota prahu excitácie (angl. hreshold) [2].

Keď sú váhy nastavené v učiacej fáze UNS by mala byť schopná mapovať vstupné vzory na správne výstupné vektory.

Výstup neurónu ovplyvňuje aktivačná funkcia. Aktivačná funkcia hlavne obmedzuje rozsah výstupnej hodnoty neurónu. Zobrazené aktivačné funkcie umožňujú aby výstup neurónu nadobúdal hodnoty 0,1 v prípade funkcie typu prah, a hodnoty v rozsahu $< 0, 1 >$ alebo $< -1, 1 >$ v prípade ostatných. Obecne sa však aj tento rozsah môže u jednotlivých implementácií líšiť.

4.1.1 Topológie neurónových sietí

Existuje veľa druhov umelých neurónových sietí a tiež množstvo faktorov, podľa ktorých sa dajú klasifikovať. Klasifikácia UNS môže byť založená na [12]

- funkcii, pre ktorú bola UNS navrhnutá
- stupeň prepojenosti neurónov (plne/čiastočne)
- smer toku informácií v sieti (rekurentná / nerekurentná)
- typ učiaceho algoritmu, ktorý reprezentuje sada systematických rovníc
- učiace pravidlo
- typ riadenia učenia (supervised/unsupervised)

4.1.2 Dopredná sieť so spätným šírením chyby

Za najviac prakticky používaný typ UNS sa považuje dopredná neurónová sieť so spätným šírením chyby (angl. Backpropagation model). Aj keď nebol v tomto smere vykonaný prieskum, predpokladá sa že viac ako 90 percent všetkých komerčných a priemyselných aplikácií neurónových sietí využíva práve model so spätným šírením chyby alebo model z neho odvodený [18]. Ďalšou prednosťou tohto modelu popri jeho popularite je flexibilita a adaptabilita v širokom spektre riešených problémov [12].

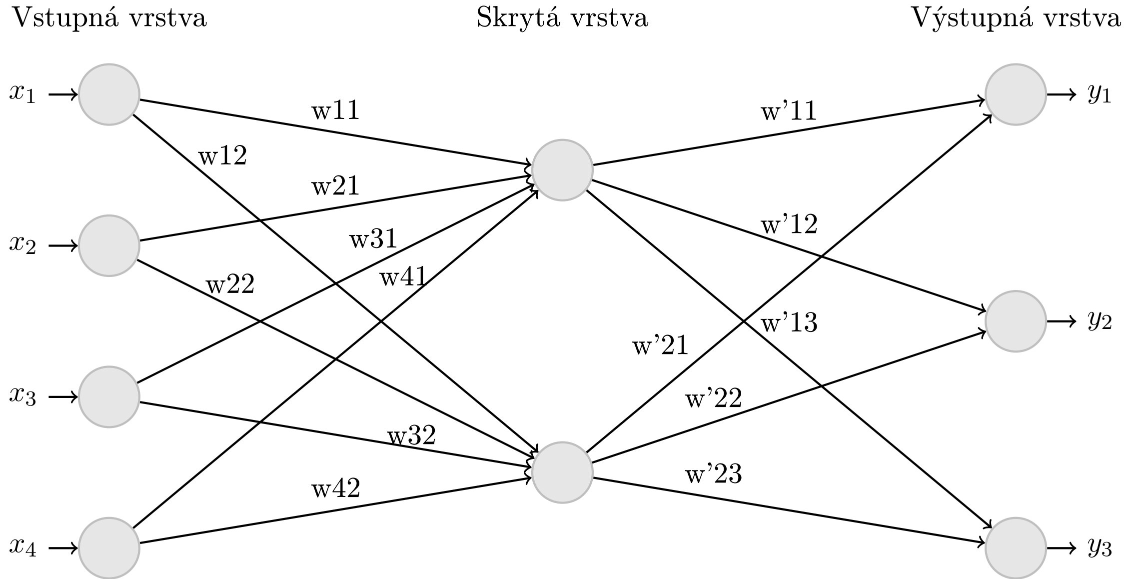
Architektúra (vzor spojení medzi neurónmi [18] doprednej neurónovej siete so spätným šírením chyby je na obrázku 4.4. Jedná sa o vrstvenú architektúru, pretože jej neuróny sú rozdelené vo viacerých vrstvách. Tento typ siete sa nazýva dopredná z toho dôvodu, že sa v nich nachádzajú len dopredné spojenia, ktoré prenášajú signál od vstupnej vrstvy smerom k výstupu. Každý neurón jednej vrstvy vysiela signál na každý neurón nasledujúcej vrstvy. Naopak spojenia prenášajúce signál v rámci vrstvy prípadne alebo do predchádzajúcej vrstvy tu neexistujú. V tomto prípade má neurónová sieť tri vrstvy, ktoré sa nazývajú vstupná, skrytá a výstupná vrstva. Obecne pri tomto type UNS môžeme mať jednu vstupnú, jednu výstupnú a ľubovoľný počet skrytých vrstiev. Tri alebo viac skrytých vrstiev sa ale používa len zriedka. Medzi autormi zaoberajúcimi sa neurónovými sieťami ale nie je zjednotený názor či považovať vstupnú vrstvu medzi vrstvy tejto siete, alebo nie a to preto, že neuróny vo vstupnej vrstve nič nepočítajú, len preposielajú signál z prislúchajúceho vstupu všetkým neurónom ďalšej vrstvy. Preto je lepšie počítať len počet skrytých vrstiev [18].

Počet neurónov vo vstupnej a výstupnej vrstve záleží od počtu vstupov a výstupov, teda od konkrétneho problému a spôsobu jeho riešenia. Počet neurónov v skrytej vrstve sa väčšinou zisťuje experimentálne systémom pokus - omyl.

Algoritmus spätného šírenia chýb

V tejto sekcii uvediem princíp fungovania algoritmu spätného šírenia chýb pre potreby jeho implementácie. Podrobnejšie je tento princíp vysvetlený v mnohých zdrojoch[18][1][2].

Učenie alebo trénovanie umelej neurónovej siete je proces, v ktorom sa nastavujú váhy jej spojení tak, aby naučená sieť bola schopná mapovať vstupné vzory na správne odpovedajúce výstupy. Pre učenie dopredných neurónových sietí sa používa algoritmus spätného šírenia chýb (angl. Backpropagation). Algoritmus pre učenie neurónovej siete potrebuje



Obrázok 4.4: Dopredná sieť so spätným šírením chyby

tzv. tréningovú množinu. Prvky tejto množiny sú dvojice vstupov a správnych výstupov. Bez nich algoritmus spätného šírenia chýb nie je schopný túto neurónovú sieť učiť a teda sa jedná o metódu učenia s učiteľom (angl. supervised learning).

Prvým krokom učenia je inicializácia váh. Typicky sa váhy inicializujú náhodne v dopredu určenom rozsahu, ktorý závisí od špecifickej aplikácie. Niekedy sa váhy inicializujú konštantami, čo má výhodu v opakovateľnosti výsledku učenia.

Algoritmus spätného šírenia sa snaží minimalizovať chybovú funkciu pomocou úpravy váh. Chybovú funkciu definujeme ako:

$$E_p = \frac{1}{2} \sum_{k=1}^{K} (d_{pk} - y_{pk})^2 \tag{4.4}$$

Teda je to súčet štvorcov chýb na všetkých výstupných neurónoch. Algoritmus spätného šírenia chyby [18] obsahuje dva hlavné cykly. Vonkajší cyklus, ktorý sa opakuje až kým je neurónová sieť schopná mapovať správne všetky vzory a vnútorný cyklus, v ktorom sa opakujú nasledujúce tri kroky až kým je výstupný vektor $\vec{y}$ rovný alebo dostatočne blízky požadovanému vektoru $\vec{d}$ pri požadovanom vstupnom vektore $\vec{x}$.

- Krok 1: Polož vektor $\vec{x}$ na vstup siete.
- Krok 2: Vypočítaj výstupný vektor $\vec{y}$ postupným prechodom cez sieť od vstupnej vrstvy cez skryté vrstvy až po výstupnú vrstvu.
- Krok 3: Spätné šírenie korekcií chýb. Porovnaj $\vec{y}$ s $\vec{d}$. Ak sú $\vec{y}$ a $\vec{d}$ dostatočne podobné, choď na začiatok vonkajšieho cyklu. Inak šír spätne korekcie chyby cez neurónovú sieť pre nastavenie váh, aby bolo $\vec{y}$ bližšie k $\vec{d}$ a potom sa vráť na začiatok vnútorného cyklu.

Prvé dva kroky sú jednoduché a stačí nám postupne vyhodnocovať výstupy neurónov aplikovaním bázovej funkcie a následne aktivačnej funkcie na ich vstupy. Celá podstata

algoritmu spätného šírenia chýb je ukrytá až v poslednom bode. Ako už bolo spomenuté ide o úpravu váh tak, aby sa minimalizovala chybová funkcia E_p. Na to sa využíva inkrementálna metóda negatívneho gradientu, nazývaná aj metóda najprudšieho spádu, alebo najstrmšieho zostupu (angl. steepest descent). Pri tejto inkrementálnej metóde sa každá váha upraví tak, aby sme sa posunuli v priestore funkcie E_p smerom, ktorým jej hodnota klesá najviac. Za tým účelom upravíme každú váhu neurónu vo výstupnej vrstve podľa vzťahu:

$$\Delta w_{jk} = -\alpha \frac{\partial E_p}{\partial w_{jk}} = -\alpha \frac{\partial E_p}{\partial (net_k)} \frac{\partial (net_k)}{\partial w_{jk}} = \alpha \delta_{ok} x_j \tag{4.5}$$

Kde α je rýchlosť učenia a záporná derivácia $-\partial E_p / \partial (net_k) = \delta_{ok}$ je zovšeobecnený chybový signál produkovaný k-tym výstupným neurónom. Pre δ_{ok} platí:

$$\delta_{ok} = -\frac{\partial E_p}{\partial (net_k)} = -\frac{\partial E_p}{\partial y_{pk}} \frac{\partial y_{pk}}{\partial (net_k)} = (d_{pk} - y_{pk}) f'_k \tag{4.6}$$

Kde f'_k je derivácia aktivačnej funkcie neurónu podľa net_k. Pravidlo pre zmenu j-tej váhy k-teho výstupného neurónu je potom:

$$\Delta w_{jk} = \alpha (d_{pk} - y_{pk}) f'_k x_j \tag{4.7}$$

Vzťah pre úpravu skrytých vrstiev dostaneme veľmi podobne:

$$\Delta v_{ij} = -\alpha \frac{\partial E_p}{\partial v_{ij}} = -\alpha \frac{\partial E_p}{\partial (net_j)} \frac{\partial (net_j)}{\partial v_{ij}} = \alpha \delta_{hj} x_j \tag{4.8}$$

Záporná derivácia $-E_p / \partial (net_j) = \delta_{hj}$ je zovšeobecnený chybový signál produkovaný j-tým skrytým neurónom.

$$\delta_{hj} = -\frac{\partial E_p}{\partial (net_j)} = -\frac{\partial E_p}{\partial y_j} \frac{\partial y_j}{\partial (net_j)} = -\frac{\partial E_p}{\partial y_j} f'_j \tag{4.9}$$

f_j je derivácia výstupnej funkcie skrytého neurónu podľa net_j.

$$\begin{aligned} \frac{\partial E_p}{\partial y_j} &= -\sum_{k=1}^{K} (d_{pk} - y_{pk}) \frac{\partial \{f(net_k)\}}{\partial y_j} = -\sum_{k=1}^{K} (d_{pk} - y_{pk}) \frac{\partial f(net_k)}{\partial net_k} \frac{\partial net_k}{\partial y_j} = \\ &\quad -\sum_{k-y}^{K} (d_{pk} - y_{pk}) f'_k w_{jk} \end{aligned} \tag{4.10}$$

A teda pre chybový signál platí:

$$\delta_{hj} = \left(\sum_{k=1}^{K} \delta_{ok} w_{jk} \right) f'_j \tag{4.11}$$

Váhy v skrytej vrstve upravíme podľa nasledujúceho vzťahu:

$$\Delta v_{ij} = \alpha \left(\sum_{k=1}^{K} \delta_{ok} w_{jk} \right) f'_j x_i \tag{4.12}$$

Tým máme vzťahy pre úpravu váh výstupnej aj skrytej vrstvy. Ak máme viac skrytých vrstiev, používame rovnaké rovnice a vždy dosadíme do rovnice pre chybový signál výsledný chybový signál z predchádzajúcej vrstvy.

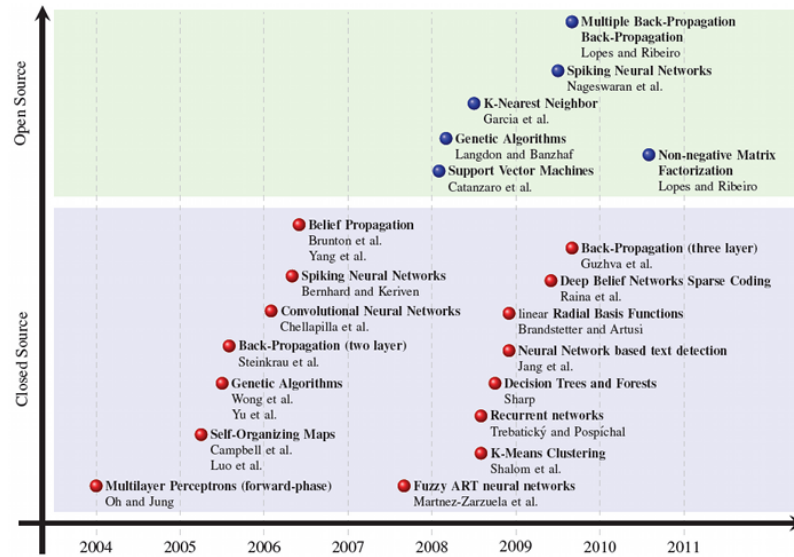
Kapitola 5

Analýza a návrh riešenia

V tejto kapitole sa budem zaoberať predchádzajúcimi prácami na tému učenie neurónových sietí. Druhá časť bude venovaná analýze návrhu programovej knižnice pre učenie neurónovej siete s podporou akcelerácie učenia na grafickej karte.

5.1 Predchádzajúce práce na tému paralelného riešenia algoritmu Backpropagation

Myšlienka riešiť učenie neurónovej siete paralelným algoritmom z ďaleka nie je nová. Nakoniec samotný princíp fungovania neurónových sietí je vo svojej podstate paralelný. Napríklad už v roku 1993 robili Faramarz Valafar a Okan K. Ersoy výskum v oblasti paralelnej implementácii tohto algoritmu na paralelnom výpočtovom zariadení typu SIMD MASPAR MP-1 [25]. Vo svojej práci autori delia paralelizmus pri učení doprednej neurónovej siete na paralelizmus na úrovni architektúry a na dátový paralelizmus. Na úrovni architektúry sa výpočty dajú vykonávať na jednotlivých neurónoch naraz, keďže medzi nimi nie sú spojenia a ich výstupy sú vzájomne nezávislé. Dátový paralelizmus vysvetľujú tak, že je možné počítať zmeny váh na viacerých dátach s rôznymi tréningovými vzormi. Výsledné zmeny váh sa potom môžu spriemerovať a tieto priemerné hodnoty sa dajú použiť na úpravu pôvodných váh. Implementácia vyššie spomínaných autorov nazvaná SIMD-BP využívala na urýchlenie oba druhy paralelizmu. Keďže problémy v oblasti strojového učenia sú väčšinou výpočtovo náročné, sú pre ne obvykle vhodné paralelné riešenia. Napriek tomu ešte existuje len zopár programových knižníc riešiacich tieto problémy na GPU (Graphics Processing Unit) a pri tom väčšina z nich nemá voľne dostupné zdrojové kódy. To bola motivácia pre vznik programovej knižnice GPUMLib predstavenú v [16]. GPUMLib využíva architektúru CUDA, ktorá dokáže fungovať na grafických kartách od spoločnosti nVIDIA. Autori tvrdia, že ich riešenie bolo schopné učiť umelú neurónovú sieť algoritmom backpropagation, prípadene multiple backpropagation, až 175 krát rýchlejšie ako riešenie na procesore s použitím grafickej karty nVIDIA GTX 280. Okrem týchto algoritmov však spomínaná knižnica implementuje mnoho iných algoritmov pre učenie neurónových sietí. V práci autori tiež uviedli prehľadný zoznam implementácií rôznych učiacich algoritmov s podporov grafickej karty 3.3.



Obrázok 5.1: Prehľad existujúcich implementácií knižníc pre strojové učenie s podporou grafickej karty. Prevzaté z [16]

Ďalšia implementácia algoritmu, ktorá sa volá GNeuron je popísaná v [21]. Ide o mierne staršiu publikáciu, na ktorej je zaujímavé to, že bola testovaná na grafickej karte z generácie ešte pred uvedením prvej grafickej karty s unifikovanými "shader"procesormi. Konkrétne sa jednalo o nVIDIA GeForce 6150 Go. Namiesto CUDA či OpenCL tu bola použitá knižnica Microsoft Research Accelerator GPGPU library. Prezentované výsledky ukazovali, že pri sieťach s malým počtom neurónov bol výpočet na procesore rýchlejší, ale ako narastala veľkosť siete, začala sa prejavovať výhoda paralelného spracovania. Článok [3] popisuje implementáciu učiaceho algoritmu na architektúre CUDA. V článku sú tiež načrtnuté niektoré problémy s aritmetikou desatinných čísel pri učení veľkých sietí . Posledná publikácia, ktorú tu uvediem je venovaná implementácii algoritmu backpropagation pomocou OpenCL(citácia 24). Práca obsahuje zaujímavé porovnanie, nielen verzii na procesor a OpenCL implementácii na grafickej karte, ale aj OpenCL program bežiaci na šesťjadrovom procesore od spoločnosti AMD. Výsledky tam ukazujú, že klasická sériová implementácia je pri malých sieťach (približne menej ako 200 neurónov v skrytých vrstvách) rýchlejšia ako riešenie na grafickej karte.

5.2 Analýza algoritmu Backpropagation a možností jeho paralelizácie

Základný princíp algoritmu Backpropagation je uvedený v predchádzajúcej kapitole. V tejto časti textu sa na neho pozrieme skôr z pohľadu programátora a bude navrhnutá jeho implementácia.

Učenie pomocou algoritmu Backpropagation môžeme rozdeliť na dve základné fázy a to: 1. dopredné šírenie signálu (feedforward) 2. spätné šírenie chýb (error backpropagation)

Prvá fáza učenia opakuje rovnaké výpočty nad všetkými vrstvami od vstupnej vrstvy po výstupnú, pričom vstupná vrstva je vynechaná a za jej výstupy považujeme vstupy siete. Výpočet na ostatných vrstvách je vyjadrený vzorcom 4.5. Tento krok pri m neurónoch v predchádzajúcej vrstve a n neurónov vo vrstve. pre ktorú počítame výstup je vlastne

násobenie vektora s maticou, kde násobíme stĺpcový vektor o m prvkoch obsahujúci vstupy z predchádzajúcej siete s maticou o rozmeroch m x n, v ktorej na pozícii $[k, l]$ je váha medzi k-tým neurónom predchádzajúcej vrstvy a l-tým neurónom počítanej vrstvy. Náročnosť tohto kroku výpočtu na počet inštrukcií je $mn + (m - 1)n = (2m - 1)n$ flop kde mn je počet inštrukcií s pohyblivou desatinnou čiarkou pri násobení každej váhy s príslušným vstupom a (2m-1)n je počet inštrukcií na redukciu všetkých spočítaných m-tíc s operáciou sčítania. Pri tomto kroku je potrebné, aby sa uchovali výstupy z každej vrstvy, lebo budú potrebné pri spätnom šírení chýb.

Druhá fáza je odlišná v závislosti od toho, či ju počítame na skrytej vrstve, alebo výstupnej. V obidvoch prípadoch ale potrebujeme najprv spočítať chybový signál δ.

Vo výstupnej vrstve ho spočítame podľa vzorca 4.6. V ňom derivácia aktivačnej funkcie f'_k máme zvyčajne vyjadrenú ako funkciu výstupu o_k. Pri aktivačnej funkcii typu sigmoida so strmosťou (angl. steepness) λ sa jej derivácia spočíta z výstupu ako $\lambda o_k(1 - o_k)$. Teda pri počte výstupných neurónov o nám tento výpočet zaberie $5o$ inštrukcií.

V skrytej vrstve učiace signály spočítame podľa vzorca 4.9. V tomto prípade sa jedná znova o násobenie vektora s maticou a to konkrétne vektora chybových signálov z predchádzajúcej vrstvy a transponovanou maticou váh medzi vrstvou, pre ktorú počítame učiace signály s predchádzajúcou vrstvou. Náročnosť tohto kroku je teda rovnaká ako v prípade jedného kroku dopredného šírenia signálu. Skutočnosť, že v tomto prípade sa v redukčnej fáze sčítavajú výsledky zo stĺpcov matice namiesto riadkov značne komplikuje efektívny prístup do pamäte grafickej karty pri tomto kroku výpočtu. Vysvetlenie bude podané v ďalšej časti.

Po spočítaní chybových signálov je potrebné vypočítať zmeny váh podľa vzorca 4.12 . Tu musíme vynásobiť každú váhu vstupom z predchádzajúcej vrstvy, ktorý prechádza spojením, ktoré reprezentuje daná váha a s konštantou určujúcou rýchlosť učenia. Celkovo teda v prípade m neurónov v predchádzajúcej vrstve a n neurónov vo vrstve, nad ktorou počítame krok spätného šírenia chyby je potrebné v tejto fáze vykonať $3mn$ flop.

Po vykonaní prvej (feedforward) a následne druhej fáze (feedbackward) postupne pre všetky vrstvy máme pripravené diferencie všetkých váh. Posledným krokom teda bude pripočítať ich k pôvodným váham. Pre každú maticu váh o rozmeroch m X n je potrebné vykonať mn operácií. Tento krok je v podstate totožný so sčítavaním vektorov, kedy vektory utvoríme uložením jednotlivých riadkov matice váh a matice diferencií váh za seba. Potom môžeme algoritmus ukončiť, alebo pokračovať ďalšou iteráciou.

Z predchádzajúceho textu vyplýva, že v jednotlivých fázach je potrebné vykonať rádovo podobný počet inštrukcií. Pri sériovom algoritme sa bude časová náročnosť optimálneho algoritmu odvíjať od počtu inštrukcií, ktoré treba vykonať. Pri paralelnom výpočte môže byť časová zložitosť nižšia pokiaľ máme k dispozícii dostatočný počet procesorov na daný algoritmus. V prípade násobenia vektora a matice je pri roznásobovaní prvkov z vektora s prvkami z matice možné teoreticky dosiahnuť konštantnú časovú zložitosť (násobenie každej z dvojíc možno vykonať súčasne). Následná redukcia výsledkov za uvedených podmienok môže dosahovať logaritmickú časovú zložitosť. Posledný krok - úprava váh môžeme vykonať teoreticky za ideálnych podmienok tiež v konštantnom čase, lebo nie sú medzi jednotlivými operáciami dátové závislosti. V skutočnosti ale budeme mať vždy obmedzený počet procesorov a navyše, ako uvidíme v ďalšej časti textu, je veľmi problematické využiť všetky potrebné procesory na grafickej karte kvôli komplikovanej synchronizácii či zdieľaní medzivýsledkov medzi procesormi, ktoré nie sú na tom istom SM a tiež kvôli obmedzeniam v spôsobe prístupu do jej globálnej pamäte, alebo pre maximálnu priepustnosť zbernice.

Vo výsledku teda môžeme povedať, že pri algoritme Backpropagation sú jednotlivé kroky

výpočtu rádovo podobne náročné a preto nemôžeme určiť jeden krok, ktorý by bol výrazne zložitejší ako ostatné kroky. Preto sa javí ako najlepšie riešiť všetky hore spomínané kroky na grafickej karte a pre procesor prenechať úlohu riadenia vykonávania a synchronizácie jednotlivých úloh.

Pri návrhu paralelného algoritmu musíme okrem všeobecných znalostí o paralelných algoritmoch uplatniť aj informácie o princípoch a procesoch dejúcich sa na grafických kartách. Keďže v dispozícii mám grafickú kartu od výrobcu spoločnosti nVIDIA, primárne je paralelná časť kódu orientovaná na ich hardvér. Veľmi užitočné informácie o tom, čomu sa programátor musí vyvarovať aby výsledný paralelný kód fungoval optimálne na grafických kartách spoločnosti nVIDIA môžeme nájsť v [6].

Aby boli jednotlivé jadrá optimálne vyťažené, musí im byť priradené dostatočné množstvo vlákien. Dôvodom je už spomínaný princíp hardwarového prepínania kontextu. Ak totiž SM nemá dostatok úloh, potom v prípade, kedy vlákna čakajú napríklad na prístup do pamäte, stratíme množstvo hodinových cyklov, kedy sa nevykonávajú žiadne výpočty. Naopak ak je dostatok čakajúcich úloh jednoducho sa prepne kontext a pokračuje sa s minimálnou, alebo dokonca žiadnou stratou času ďalej. Z toho vyplýva, že je vhodné, aby počet procesov na grafickej karte bol niekoľkonásobne vyšší ako počet jej procesorov.

Zastavenie vykonávania kernelu u časti procesorov môže spôsobiť aj vetvenie programu a následná divergencia vlákien. V jednom SM totiž môžu súčasne bežať len tie vlákna, ktoré vykonávajú tú istú inštrukciu. To znamená, že v extrémnom prípade sa môže stať, že sa na grafickej karte nevykonajú inštrukcie tridsiatich dvoch vlákien ale len jedného.

Pre optimálne využitie prenosovej rýchlosti pri čítaní z globálnej pamäte je potrebné pristupovať do nej "zarovnane"(angl. coalesced). Globálnu pamäť je potrebné vidieť ako jeden celok pozostávajúci so zarovnaných segmentov dlhých 16 prípadne 32 bitových slov. Zarovnaný prístup sa vzťahuje na jeden "pol-warp"teda šestnásť procesov. Aby bol prístup do pamäte zarovnaný, musia všetky procesy pristúpiť do pamäte, tak ako to zobrazuje obrázok (obrázok 22). Tu platí, že nemusí každý proces čítať z pamäte, ale všetky ostatné musia pristúpiť do správnej bunky pamäte. V opačnom prípade bude načítanie fungovať sériovo a teda postupne pre každý proces, výsledkom čoho je zníženie prenosovej rýchlosti približne osemkrát. Kvôli presnosti treba dodať, že toto platí len pre grafické karty spoločnosti nVIDIA s verziou "computing capability"1.0 alebo 1.2. V prípade novších grafických kariet spomalenie spôsobené nezarovnaným prístupom nie je až tak výrazné.

Ešte pomalší prenos dát ako pri globálnej pamäti je prenos dát medzi hosťujúcim zariadením a OpenCL zariadením cez zbernicu PCI Express. Preto je potrebné minimalizovať túto komunikáciu. Je to ďalší dôvod na to vykonávať všetky kroky algoritmu backpropagation na grafickej karte. Inak by zrejme bolo potrebné prenášať medzivýsledky cez túto zbernicu.

Kvôli vyššie uvedenému spôsobu prístupu globálnej do pamäte je problematické vykonávať výpočet chybových signálov pre skryté vrstvy. Problém nastáva v prípade, ak chceme umožniť, aby sa pre každý neurón skrytej vrstvy počítal chybový signál v rôznej OpenCL pracovnej skupine, potom by programové vlákna v každej skupine museli pristupovať do pamäte k informáciám o váhach nezarovnane. Nepomohlo by ani zoradiť váhy na grafickej karte tak, aby prístupy do pamäte v tejto fáze výpočtu boli zarovnané, pretože by to spôsobilo rovnaký problém s nezarovnaným prístupom pri doprednom šírení chyby. Druhé riešenie, ktoré sa zdanlivo ponúka je do jednej pracovnej skupiny rozdeliť procesy, ktoré pristupujú do pamäťových buniek v jednom bloku. Tento princíp by zas spôsobil problém pri redukcii výsledkov, pretože by procesy boli rozdelené v rôznych pracovných skupinách a v tom prípade nemôžu si posielať medzivýsledky cez zdieľanú pamäť. Okrem

toho nemožno použiť ani obvyklé metódy na ich synchronizáciu.

5.3 Rozdelenie paralelného algoritmu na menšie celky

Pred navrhnutím paralelnej implementácie je potrebné rozhodnúť, či je lepšie algoritmus backpropagation naprogramovať do jedného kernelu, alebo ho rozdeliť do viacerých. Na jednej strane je pravda, že s prepnutím vykonávaného programu na grafickej karte súvisí aj určitá réžia. Vo výsledku sa teda algoritmus spomalí. Na druhej strane ale hosťujúce zariadenie poskytuje synchronizačný mechanizmu nielen na úrovni jednej OpenCL pracovnej skupiny, ale umožňuje synchronizovať celý výpočet. Na synchronizovanie vlákien v odlišných pracovných skupinách síce môžeme použiť atomické inštrukcie na globálnej pamäti, tie ale zo sebou prinášajú aj nevýhody a je dobrým zvykom vyhnúť sa takejto koordinácie [1]. Nevýhodou je, že pre synchronizáciu musíme používať pomalú globálnu pamäť a použitím tohto princípu môžeme zaviesť do systému "deadlock". Okrem toho atomické inštrukcie tiež limitujú prenositeľnosť daného riešenia, lebo sa jedná o rozšírenie (angl. extension) OpenCL, ktoré je podporované len niektorými zariadeniami. V prípade jedného kernelu by sme sa synchronizácii medzi OpenCL pracovnými skupinami nevyhli, aj keď medzi neurónmi v jednej vrstve nie sú žiadne závislosti, medzi vrstvami je to úplne naopak. Pred počítaním výstupu určitého neurónu musíme mať spočítané výstupy všetkých neurónov v predchádzajúcich vrstvách. Navyše je pravdepodobné, že ak by sme chceli použiť jeden kernel, pravdepodobne by sme museli použiť viac inštrukcií pre riadenie toku programu, čo by mohlo zvýšiť divergenciu warpov. Rozdelenie programu na viac kernelov tiež umožní ich jednoduchšie ladenie a testovanie po častiach. Keďže paralelné aplikácie sa ťažko ladia ide o veľkú výhodu. Z uvedených dôvodov preto kernel rozdelím na menšie funkcie, ktoré budú volané v správnom slede za sebou.

5.4 Spravovanie kontextu a zdrojov grafickej karty

Pred spustením kernelu na grafickej karte je potrebné vykonať nutnú inicializáciu. V prvom kroku sa musia získať dostupné platformy na systéme. Z nich je potrebné vybrať jednu a vybrať zariadenie pod danou platformou. V tomto kroku obmedzíme výber na zariadenia typu GPU. Po zvolení zariadenia musíme vytvoriť OpenCL kontext. Následne musíme vytvoriť frontu príkazov nad kontextom a preložiť program kernelov a vytvoriť ich objekty. Táto časť inicializácie výpočtu na grafickej karte nie je závislá od konkrétnej neurónovej siete a jej parametrov. Pokiaľ teda nebudeme meniť zariadenie pre výpočet, potom počas behu programu stačí vykonať tieto kroky len raz a potom ich držať v pamäti, aj keď v niektorých prípadoch by to mohlo byť vhodné ich z pamäti odstrániť. V ďalšej úrovni inicializácie musia byť vytvorené a inicializované pamäťové štruktúry na grafickej karte. V tomto kroku musia byť známe parametre neurónovej siete. Konkrétne musíme poznať počet vrstiev a množstvo neurónov v každej z nich. Od toho sa totiž odvíja koľko blokov pamäte a v akej veľkosti budeme potrebovať. Z toho tiež vyplýva, že v prípade ak sa menia spomenuté parametre, potom sa stane abstraktný pamäťový model neurónovej siete na grafickej karte neplatný. V tom prípade je potrebné uvoľniť alokovanú pamäť a v prípade novej požiadavky na učenie na grafickej karte vytvoriť nové štruktúry. Podobné pravidlá platia aj pre kópiu neurónovej siete v grafickej karte. Tie dáta sa ale stanú neplatné aj v prípade, ak sa spustí učenie tejto neurónovej siete na procesorovej časti.

[1] http://www.codeproject.com/Articles/143395/Part-3-Work-Groups-and-Synchronization

Kapitola 6

Implementácia

V tejto kapitole opíšem postup implementácie programovej knižnice pre prácu s neuronovými sieťam. Uvediem tu svoje úvahy nad problematikou a tiež zdôvodnenia svojich riešení.

Pre implementáciu tohto programu som sa rozhodol použiť jazyk C++ spolu s OpenCL. Dôvody prečo je OpenCL vhodným nástrojom pre túto prácu už bolo naznačené v predchádzajúcich kapitolách. Aj keď je v súčasnosti ťažké predpovedať budúcnosť OpenCL a do akej miery sa bude používať o niekoľko rokov, fakt že sa jedná o otvorený štandard, ktorý prijalo množstvo významných spoločností je veľkou výhodou. Na jednej strane je dobré, ak sa nebude použitie výslednej softwarovej knižnice viazať na jeden operačný systém alebo na zariadenia od jedného výrobcu. Aj pre mňa môžu byť nadobudnuté skúsenosti o to cennejšie v budúcnosti.

Na riešenie som sa rozhodol použiť operačný systém Linux ale nepoužíval som žiadne nástroje a knižnice, ktoré by neumožňovali, aby bolo moje riešenie prenesené aj na Windows prípadne iný operačný systém. V prvom kroku riešenia som sa rozhodol najprv implementovať učiaci algoritmus fungujúci len na procesore. To umožní využitie výsledného programu aj na počítačoch ktoré neobsahujú žiadne zariadenie podporujúce OpenCL.

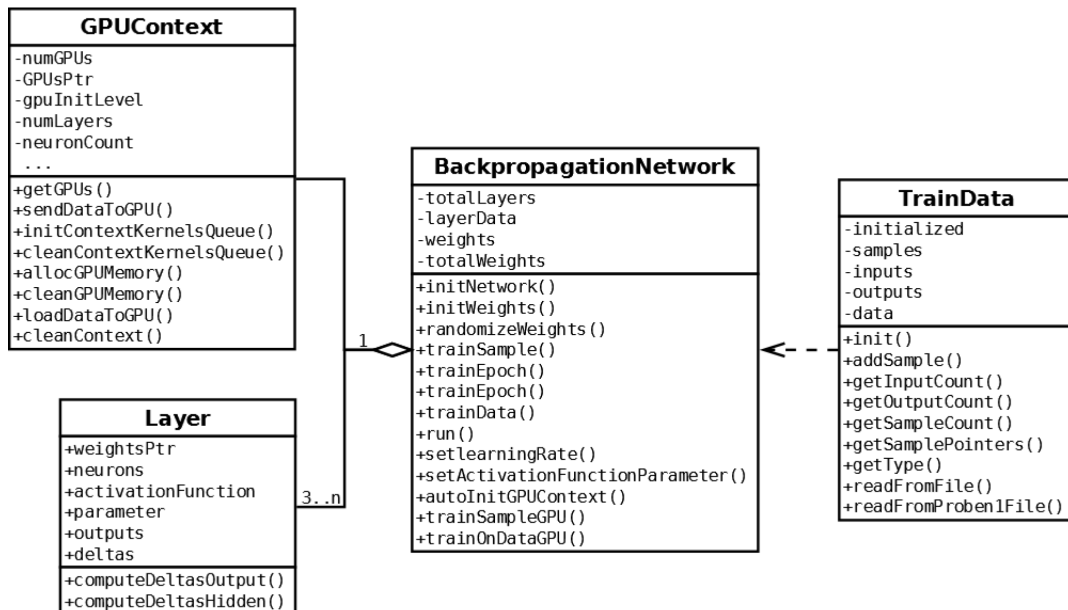
6.1 Dátová reprezentácia neurónovej siete

Pre reprezentáciu umelej neurónovej siete učenej pomocou algoritmu Backpropagation je potrebné uchovávať nasledujúce dáta:

- počet vrstiev (vo svojej implementácii budem považovať aj vstupnú vrstvu za skutočnú vrstvu)
- maticu váh pre každú skrytú vrstvu a pre výstupnú vrstvu
- pre každú vrstvu ďalej musíme uchovať počet neurónov v danej vrstve a tip aktivačnej funkcie

Pri implementácii bol využitý objektový návrh. Neformálny diagram tried medzi ktorými bola rozdelená funkcionalita softvérovej knižnice je na obrázku 6.1. Tento diagram neobsahuje úplný zoznam členov znázornených tried, ale len tie, ktoré sú podstatné pre ukázanie účelu jednotlivých tried v navrhnutom objektovom modeli. Prvou úlohou pri tvorení tejto softvérovej knižnici bolo vytvoriť abstraktný model umelej neurónovej siete. Ten sa nachádza v triede *BackpropagationNetwork*. Okrem modelu neurónovej siete obsahuje aj

metódy pre prácu nad danou neurónovou sieťou a poskytuje rozhranie pre vonkajšie aplikácie. V tejto triede architektúru neurónovej siete reprezentujú nasledujúce dátové elementy:



Obrázok 6.1: Objektový model aplikácii

Pole objektov typu *Layer* , kde každý z týchto objektov uchováva informácie o jednej vrstve umelej neurónovej siete. Celočíselnú premennú *totalLayers* , v ktorej je informácia o celkovom počte vrstiev neurónovej siete. Ukazovateľ na pole s desatinnými číslami weights reprezentujúcimi váhy neurónovej siete. Celočíselná premenná *totalWeights* s informáciou o celkovom počte váh. Zvyšné informácie o neurónovej sieti sú v objektoch triedy *Layer*. Konkrétne sa jedná o členov: celočíselná premenná neurons, ktorá uchováva informáciu o počte neurónov v konkrétnej vrstve. Premenná *activationFunction* určujúca typ aktivačnej funkcie. Premenná typu float pod názvom parameter určujúca konštantný parameter aktivačnej funkcie ako napríklad strmosť (angl. steepness) pri aktivačnej funkcii typu sigmoida. V triede Layer sa nachádzajú navyše vektory výstupov a chybových signálov konkrétnej vrstvy (outputs a deltas). Tieto informácie sú spočítané pri doprednom šírení signálu ale využijú sa aj pri spätnom šírení chyby a preto je vhodné ich ukladať. Trieda Layer obsahuje tiež metódy pre výpočet chybových signálov.

Tým je popis tried reprezentujúcich umelú neurónovú sieť so spätným šírením chýb úplný. Pre jej učenie je ale vhodné mať v programovej knižnici pripravené nástroje pre prácu s tréningovými vzormi. Pre tento účel sa v popisovanej knižnici nachádza trieda TrainData.

Pre správne používanie triedy TrainData musí byť objekt tejto triedy zinicializovaný pre rovnakú šírku vstupných a výstupných vzorov, ako je počet neurónov vo vstupnej a výstupnej vrstve neurónovej siete, pre ktorú tieto tréningové dáta pripravujeme. Táto trieda umožňuje okrem pridávania a vracania tréningových vzorov aj ich načítanie z textového súboru. V súčasnosti podporuje dva typy súborov. Prvým je jednoduchý textový súbor, ktorý obsahuje na prvom riadku tri čísla vyjadrujúce po rade počet tréningových vzorov, počet vstupov a počet výstupov. Pod nimi sú potom striedavo riadky so vstupným a výstupným vektorom všetkých vzorov postupne. Príklad súboru s dátami pre učenie neurónovej siete

```
4 2 1
1 1
0
1 0
1
0 1
1
0 0
0
```

na vykonávanie funkcie exkluzívnej disjunkcie (XOR) je zobrazený nižšie . Druhým formátom súboru je formát používaný v databáze pre testovanie neurónových sietí PROBEN1 [22].

Myšlienka spravovania kontextu už bola vysvetlená v predchádzajúcej kapitole. A práve na ten účel slúži trieda GPUContext. Táto trieda uchováva informácie o inicializácii OpenCL zariadenia a stará sa o ukazovatele do pamäte grafickej karty. Stupeň inicializácie OpenCL zariadenia je v nej reprezentovaný aj vnútorným stavom, ktorý sa môže meniť pri volaní metód tejto triedy. Stavy, v akej sa môže nachádzať sú uvedené v tabuľke 6.1.

Stav triedy GPUContext:	Význam:
GPU_INIT_LEVEL_DEVICE_NOT_SELECTED	základný stav, žiadne zariadenie nie je vybrané
GPU_INIT_LEVEL_GPU_SELECTED	je vybraná grafická karta a trieda GPUContext uchováva jej ID
GPU_INIT_LEVEL_CONTEXT_KERNELS_QUEUE	je vytvorený kontext, pripravené kerneli aj príkazová fronta
GPU_INIT_LEVEL_GPU_MEMORY_ALLOCATED	na grafickej karte je alokovaná pamäť pre neurónovú sieť, ale ešte nie je skopírovaná
GPU_INIT_LEVEL_LOADED	dáta potrebné pre učenie neurónovej siete sú uložené na grafickej karte a všetko je pripravené na zahájenie výpočtov

Obrázok 6.2: Stavy triedy GPUContext

6.2 Programy vykonávané na grafickej karte.

Predchádzajúca časť bola venovaná predovšetkým objektovému návrhu a dátovým štruktúram programu. V tejto časti bude naopak stručne vysvetlené ako funguje paralelný algoritmus. V predchádzajúcej kapitole boli tiež vysvetlené dôvody, prečo sa javí program rozdelený na viacero kernelov ako lepšie riešenie. Takýto prístup bol nakoniec aj zvolený a algoritmus backpropagation bol rozdelený do nasledujúcich podprogramov:

- Feedforward - z vektora vstupov a matice váh spočíta výstupy vrstvy, nad ktorou je spustený. Tento program vykonáva násobenie matíc ale navyše vykoná nad výsledkami aktivačnú funkciu.

- BackpropagateOutput - spočíta chybové signáli na výstupnej vrstve a následne spočíta diferencie váh spojení medzi poslednou skrytou vrstvou a výstupnou vrstvou. Výstupom budú teda dva vektory v pamäti. Jeden s chybovými signálmi a jeden obsahujúci za sebou nasledujúce riadky matice diferencií váh.

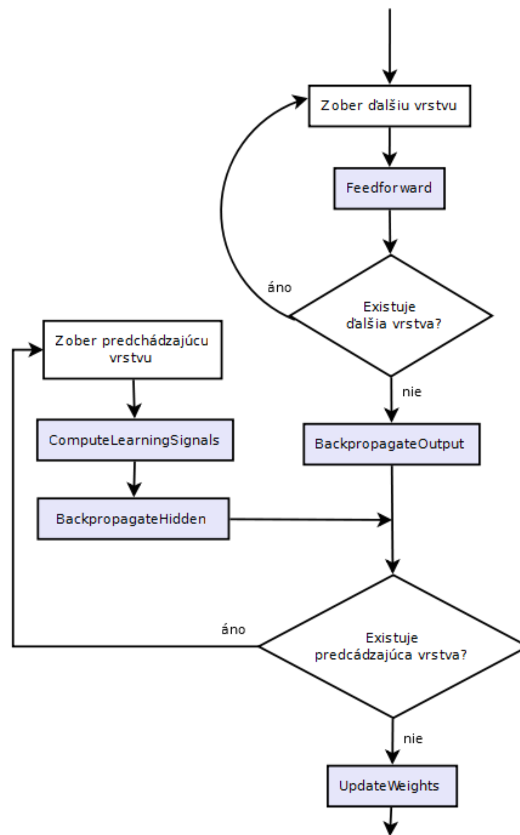
- ComputeLearningSignals - spočíta chybové signály pre zadanú skrytú vrstvu. Jedná sa vo svojej podstate o ďalšie násobenie vektoru a matice s tým rozdielom, že z pohľadu tejto funkcie sú váhy rozmiestnené v pamäti inak. Váhy, ktoré sa majú zúčastniť tej istej redukcie totiž nie sú za sebou. Tento problém bol popísaný v predchádzajúcej kapitole.

- BackpropagateHidden - spočíta chybové signály na skrytej vrstve. Jej rozdiel oproti funkcii

- BackpropagateOutput spočíva v tom, že vo výstupnej vrstve je počítanie chybových signálov triviálne a BackpropagateOutput si ich spočíta sám. BackpropagateHidden ale využíva chybové signály z funkcie ComputeLearningSignals. Kerneli ComputeLearningSignals a BackpropagateHidden teda spoločne počítajú na skrytých vrstvách to, čo kernel BackpropagateOutput na výstupnej vrstve a ich odlišnosť vychádza z faktu, že na skrytých vrstvách je výpočet chybových signálov komplikovanejší ako na vrstve výstupnej.

- UpdateWeights - vykoná posledný krok a tým je úprava váh. Tento kernel vlastne paralelne pripočíta všetky diferencie váh k pôvodným váham.

Vývojový diagram zobrazujúci algoritmus, ktorý spúšťa kerneli na grafickej karte je na obrázku 6.3. Výsledkom je jedna iterácia nad jedným učiacim vzorom. Celý postup sa opakuje pre všetky učiace vzory a v každej iterácii sa navyše počíta celková chyba nad tréningovým vzorom. Táto operácia sa vykonáva na procesorovej časti.



Obrázok 6.3: Objektový model aplikácii

Kapitola 7

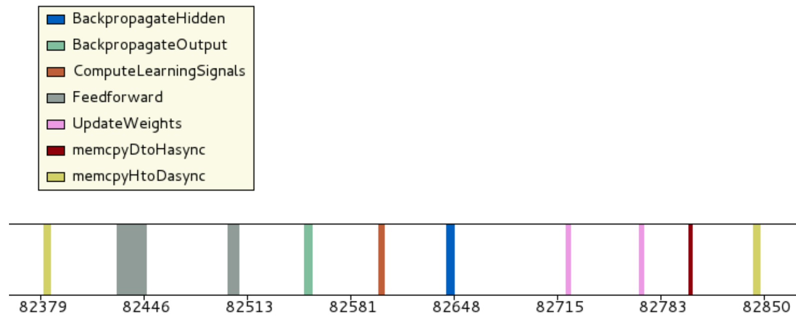
Testovanie

V tejto kapitole budú prezentované výsledky testov zvoleného riešenia a porovnanie medzi výkonom grafickej časti a procesorovej. V prvej časti tejto kapitoly bude popis použitých testovacích databáz a spôsob merania jednotlivých parametrov. Nasledujúca časť bude venovaná výsledkom jednotlivých testov. Na záver posledná časť tejto kapitoly bude venovaná krátkemu zhodnoteniu výsledkov.

7.1 Metodika testovania

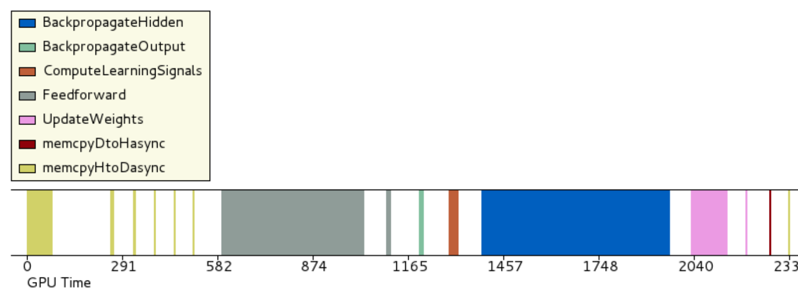
Analyzovanie výkonu kernelov fungujúcich na grafickom jadre je možné vykonať pomocou nástroja od spoločnosti nVIDIA pod názvom Compute Visual Profiler. Umožňuje získavať rôzne štatistické údaje získané za behu a tiež ich prezentáciu prostredníctvom grafov a tabuliek. Na testovanie riešenia zadaného problému boli použité testovacie vzory z databázy PROBEN1 [22]. Ide o súbor dát k viacerým reálnym úlohám, ktoré boli riešené pomocou metód strojového učenia. Filozofiou tejto testovacej sady je, aby sa ušetril čas potrebný pri zbieraní dát potrebných na testovanie výsledkov vedeckých prác. Ďalšou myšlienkou je, aby výsledky testovania jednotlivých prác na tému strojového učenia boli porovnateľné a aby sa predišlo nejasnostiam ktoré vznikajú v prípade horšie dokumentovaných testovacích databáz ktoré nie sú zverejnené. Databáza PROBEN1 bola použitá v mnohých prácach ako aj v [3].

Na testovanie bol použitý počítač s procesorom Core 2 Duo T8100 na frekvencii 2.1 GHz. Použitá grafická karta je NVIDIA Quadro NVS 140M. Prvé testy boli spustené s jednoduchým učením funkcie XOR. Jednalo sa teda o neurónovú sieť s dvomi vstupnými neurónmi, s jedným výstupným a s dvadsatimi skrytými neurónmi. Na obrázku 7.1 je zobrazený priebeh jednej iterácie učenia. Vidno na ňom ako dlho boli jednotlivé kerneli spustené. Môžeme usúdiť, že medzi ukončením jedného kernelu a spustením ďalšieho je s pravidla doba nečinnosti.



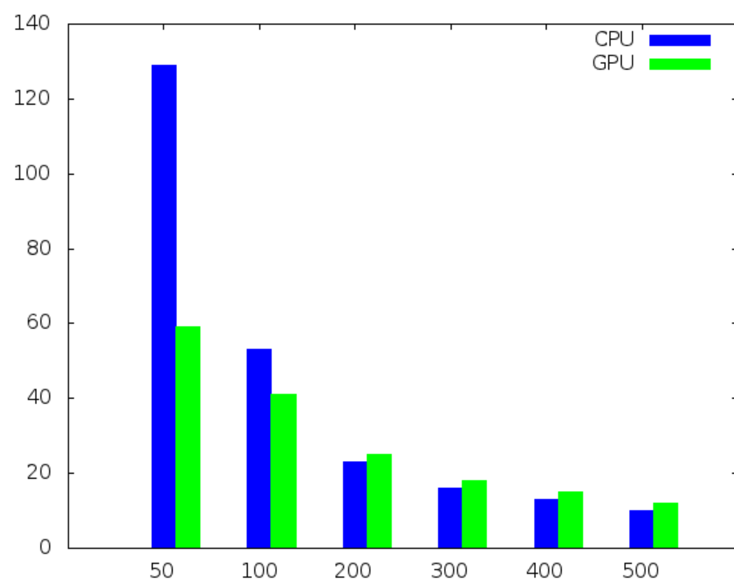
Obrázok 7.1: Kerneli počítajúce sieť s malým počtom váh.

Po spustení programu nad tréningovou množinou Gene z databázy PROBEN1 so 400 skrytými neurónmi vyzerá diagram hustejšie a zdá sa, že sa využíva väčšie percento času na výpočty kernelov. Výsledky môžeme vidieť na obrázku 7.2.



Obrázok 7.2: Kernely počítajúce sieť s veľkým počtom váh.

Testovanie programu bolo zamerané na výkonnosť. Pre porovnanie rýchlosti výkonu algoritmu pracujúceho na procesor s algoritmom počítajúcim na grafickej karte bola použitá tréningová množina Gene z databázy PROBEN1. Zvolená databáza obsahuje vzory so stodvadsiatimi vstupmi a tromi výstupmi. Takýchto vzorov obsahuje 3175. Testy boli vykonané na umelej neurónovej sieti s tromi vrstvami, pri ktorých počet neurónov v skrytej vrstve boli menené, aby sa vyskúšal vplyv počtu neurónov na výkon. Postupne boli spustené testy s 50, 100, 200, 300, 400 a 500 neurónmi v skrytej vrstve. Učenie sa ukončilo pokiaľ bola prekročená hranica šesťdesiatich sekúnd. Po skončení každej epochy sa ukladal čas od začiatku učenia, výsledná chyba a číslo ukončenej epochy. Výsledky znázorňuje graf na obrázku 7.3. Z uvedeného obrázku môžeme vidieť, že v prípade menšej siete, ktorá obsahovala menej ako 200 neurónov je výpočet na procesore rýchlejší. V prípade 200 neurónov a viac ale bola rýchlejšia grafická karta. Podobné výsledky boli publikované už v minulosti, ako napríklad v prácach, ktoré boli spomenuté v predchádzajúcich kapitolách tejto diplomovej práce [21][19].



Obrázok 7.3: Počet epoch za minút pri rôznom počte skrytých neurónov.

Kapitola 8

Možnosti rozšírenia navrhnutej knižnice o ďalšie algoritmy

Vytvorená knižnica podporuje doprednú neurónovú sieť s neobmedzeným počtom vrstiev (obmedzenie je dané len pamäťou) a algoritmus spätného šírenia chyby. Zhotovené riešenie tiež umožňuje zvoliť ľubovoľný počet neurónov v každej vrstve. Napriek tomu ale existujú aj iné možnosti konfigurácie, ktoré by prospeli lepšou prispôsobivosťou na riešenú úlohu. Môžeme si zobrať príklad z iných knižníc ako napríklad FANN [1]. Knižnica FANN umožňuje definovať aj siete s riedkym (sparse) prepojením, kedy sa pri inicializácii vytvoria spojenia medzi jednotlivými neurónmi len s určenou pravdepodobnosťou. Tieto princípy umožňujú zjednodušiť či urýchliť učenie, pri ktorom to môže, ale nemusí zhoršiť schopnosť siete naučiť sa správne mapovať dané vzory. Na druhej strane v prípade učenia na grafickej karte by to mohlo priniesť viac komplikácii ako prínosu. Bolo by totiž nutné uchovávať s váhami ešte dodatočné informácie, ktoré by umožňovali určiť, ktoré váhy v matici váh sú aktívne. Prípadne by bolo potrebné uchovávať informácie o tom, medzi ktorými neurónmi prenáša signál konkrétna váha. Tieto komplikácie by mohli znížiť efektivitu učiaceho algoritmu.

Inou možnosťou rozšírenia by bolo inkrementálne učenie, pri ktorom sa začína s prázdnou sieťou a postupne sa pridávajú spojenia. Aj v tomto prípade by ale riešenie na grafickej karte komplikovali dôvody uvedené vyššie. Jednoduchou úpravou by mohlo byť tiež pridať do učenia tzv. momentum , ktoré môže prispieť k tomu, aby sa sieť blížila k správnemu stavu váh rýchlejšie.

Veľmi užitočným vylepšením by mohol byť algoritmus RPROP [23]. Ide o úpravu pôvodného algoritmu, ktorý sa snaží znížiť negatívny dopad veľkých parciálnych derivácií na krok v priestore definovanom vektorom váh. Preto pre určenie diferencie váhy nepoužíva spočítanú hodnotu tak, ako pri klasickom algoritme Backpropagation, ale iba jej znamienko. Veľkosť zmeny váhy je určená len podľa parametru špecifického pre váhu, takzvaná obnovovacia hodnota (angl. update-value), ktorý sa upravuje v každom kroku podľa daných pravidiel.

Niektoré úlohy riešené strojovým učením majú tú vlastnosť, že správny výstup siete nezávisí len od súčasných vstupov, ale aj od vstupov minulých. Ako príklad možno uviesť moju predchádzajúcu bakalársku prácu [24]. Pre tento typ úloh sa hodia rekurzívne neurónové siete a preto by ich pridanie do vytvorenej knižnice bolo veľkým prínosom. V rekurzívnych neurónových sieťach môžu existovať spojenia aj medzi neurónmi v tej istej vrstve. Tieto spojenia umožňujú reprezentovať určitý stav vo vnútri sieti vďaka čomu sú

[1] Fast Artificial Neural Network Library http://leenissen.dk/fann/wp/

schopné odpovedať správne na naučené sekvencie vstupov. Pre učenie sa používa algoritmus "Backpropagation through time", ktorý funguje na princípe rozbaľovania cyklov v sieti v čase. Paralelné riešenie na grafickej karte by ale bolo kvôli spojeniam na úrovni jednej vrstvy komplikovanejšie.

V prípade vyššie uvedených rozšírení by bolo možné využiť pôvodnú triedu BackpropagationNetwork a v nej, buď pridať nové metódy s rozširujúcimi algoritmami, alebo prípadne aj vykonať potrebné úpravy v dátach uchovávajúcich neurónovú sieť. Väčšina z nich by ale komplikovala výpočet na grafickej karte a pravdepodobne by sa znížil celkový počet inštrukcií, ktoré by sa dali vykonať paralelne.

Existuje mnoho neurónových sietí odlišných topológií a iným spôsobom učenia ako pri tých, ktoré sa učia algoritmom backpropagation alebo algoritmom od neho odvodeným. Pre ich implementácii by ale kód v triede BackpropagationNetwork nebol použiteľný pre toto riešenie. V takom prípade by bolo potrebné pravdepodobne implementovať novú triedu. Ostatné triedy v kóde by ale mohli byť užitočné aj pre takéto odlišné riešenia. V prípade rozširovania tejto programovej knižnice by podľa môjho názoru bolo najvhodnejšie zamerať sa na príbuzné typy neurónových sietí.

Kapitola 9

Záver

V mojej práci som sa venoval možnostiam akcelerácie umelých neurnových sietí pomocou grafického procesoru. Na úvod som urobil prierez problematikou súčasných grafických procesorov GPU, preštudoval som si princípy využitia GPU na obecné výpočty. Bližšie som pochopil učiace algoritmy pre učenie neurónových sietí, predovšetkým algoritmus Backpropagation. Výsledkom mojej práce je softvérová knižnica, ktorá je použiteľná na riešenie úloh s využitím princípov strojového učenia. Akceleráciou na grafickej karte sa mi podarilo dosiahnuť zrýchlenie učenia pri dostatočne veľkých sieťach. Zrýchlenie nebolo také výrazné ako pri niektorých prácach, ktoré som tu citoval, ale moje výsledky potvrdili niektoré závery vyplývajúce z predchádzajúcich prác. Grafická karta totiž potrebuje čo najviac paralelizmu. Výsledky by bolo možné v budúcej práci vylepšiť hlbšou optimalizáciou programu alebo využitím aj paralelizmu na úrovni tréningových dát. Umelé neurónové siete sú isto príťažlivá téma. Človek sa pri ich študovaní môže dozvedieť aj niečo o svojej podstate a o princípoch myslenia. Algoritmy založené na biologických neurónových sietiach sa uplatňujú, nielen pre automatizáciu a riešenie netriviálnych úloh, ale aj na simuláciu svojho vzoru a možno nám raz odhalia niektorú zo záhad fungovania ľudského mozgu. Možno aj preto ma práca s neurónovými sieťami tak zaujala.

Literatúra

[1] *Biologický a umělý neuron, neuronové sítě.Perceptron, Adaline, Madaline a BP (Back Propagation).*
URL `<https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/SFC-IT/lectures/10sfc_2.pd,>`

[2] *Umelé neurónové siete.*
URL `<http://ii.fmph.uniba.sk/~benus/books/UNS_revised.pdf>`

[3] Chris Oei, A. J., Gerald Friedland: *Parallel Training of a Multi-Layer Perceptron on a GPU.* International computer science institute, 2009.

[4] Corparation, N.: *NVIDIA GeForce 8800 GPU Architecture Overview,World's First Unified DirectX 10 GPU Delivering Unparalleled Performance and Image Quality.* NVIDIA Corparation, 2006.

[5] Corparation, N.: *NVIDIA GeForce GTX 200 GPU Architectural Overview,Second-Generation Unified GPU Architecture for Visual Computing.* 2008.

[6] Corparation, N.: *NVIDIA OpenCL Best Practices Guide.* NVIDIA Corparation, 2009.

[7] Corparation, N.: *NVIDIA's Next Generation CUDATM Compute Architecture:Fermi.* NVIDIA Corparation, 2009.

[8] Corparation, N.: *NVIDIA GeForce GTX 680,The fastest, most efficient GPU ever built.* NVIDIA Corparation, 2012.

[9] Fatahalian, K.; Houston, M.: A closer look at GPUs. *Commun. ACM*, ročník 51, č. 10, 2008: s. 50–57.
URL `<http://doi.acm.org/10.1145/1400181.1400197>`

[10] Gaster, B. R.; Howes, L. W.; Kaeli, D. R.; aj.: *Heterogeneous Computing with OpenCL.* Morgan Kaufmann, 2012, ISBN 978-0-12-387766-6.
URL `<http://www.elsevier.com/permissions>`

[11] for consumer Graphics, R.: *NVIDIA G80: Architecture and GPU Analysis.* http://www.beyond3d.com/content/reviews/1/.

[12] I.A.Basheer, M.: *Artificial neural nerworks: fundamentals,computing,design,and application.* Elsevier Science B,V., 2000.
URL
`<http://www.sciencedirect.com/science/article/pii/S0167701200002013#,>`

[13] Inc., A. M. D.: *AMD GRAPHICS CORES NEXT (GCN) ARCHITECTURE*. Microway,Inc, 2012.

[14] Jang, B.; Do, S.; Pien, H. H.; aj.: Architecture-aware optimization targeting multithreaded stream computing. In *GPGPU, ACM International Conference Proceeding Series*, ročník 383, editace D. R. Kaeli; M. Leeser, ACM, 2009, ISBN 978-1-60558-517-8, s. 62–70.
URL `<http://doi.acm.org/10.1145/1513895.1513903>`

[15] Kobrtek, J.: *Využití gragického procesoru jako akcelerátoru - technologie OPENCL*. Vysoké učení technické v Brně, Fakulta informačních technologií, Ústav počítačové grafiky a multimédií, 2010.

[16] Lopes, N.; Ribeiro, B.: GPUMLib: An Efficient Open-Source GPU Machine learning Library. 2011.

[17] Michael Fried GPGPU Business Unit Manager Microway, I.: *GPGPU Architecture Comparison of ATI and NVIDIA GPUs*. Microway,Inc, 2010.

[18] Munakata, T.: *Fundamentals of the new artificial intelligence - Neural, Evolutionary, Fuzzy and More*. Springer, 2008, ISBN 978-1-84628-839-5.

[19] Nenad Krpan, D. J.: Parallel Neural Network Training with OpenCL.

[20] Nielsen, A. S.; Engsig-Karup, A. P.; Dammann, B.: Parallel Programming using OpenCL on Modern Architectures. 2012.

[21] Prabhu, R. D.: GNeuron: Parallel Neural Networks with GPU.

[22] Prechelt, L.: Proben1 - A Set of Neural Network Benchmark Problems and Benchmarking Rules. 1994.

[23] Riedmiller, M.: Rprop - Description and Implementation Details. 1994.

[24] Trnkóci, A.: *Harmonizace melodie*. Vysoké učení technické v Brně, Fakulta informačních technologií, Ústav počítačové grafiky a multimédií, 2010.

[25] Valafar, F.; Ersoy, O. K.: A Parallel Implementation of Backpropagation Neural Network on MasPar MP-1. 1993.
URL `<http://docs.lib.purdue.edu/ecetr>`

Dodatok A

Obsah CD

- Sources - zdrojové kódy.
- Dokumentácia - programová dokumentácia.
- Text - text práce vo forme zdrojových súborovu programu LaTeX a vo formáte pdf.

Dodatok B

Manual

Požiadavky pre úspešný preklad:

- Grafické ovládače s podporou OpenCL (knižnica libOpenCL).
- Hlavičkové súbory OpenCL. Je možné ich získať zo stránky http://www.khronos.org/registry/cl/.
- Gnu prekladač jazyka C++, prípadne kompatibilný.

Rýchly preklad a vyskúšanie:

```
$make
$./test
```

Súbor test.cpp obsahuje krátky príklad konfigurácie a spustenia knižnice. Podrobnejšia dokumentácia sa nachádza na CD v priečinku Dokumentácia.