VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE NEURONOVÝCH SÍTÍ S VYUŽITÍM GPU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. ONDŘEJ ŠIMÍČEK
AUTHOR

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE NEURONOVÝCH SÍTÍ S VYUŽITÍM GPU

THE GPU BASED ACCELERATION OF NEURAL NETWORKS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. ONDŘEJ ŠIMÍČEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. JIŘÍ PETRLÍK
SUPERVISOR

BRNO 2015

Abstrakt

Tato práce se zabývá využitím grafických čipů pro akceleraci neuronových sítí s učením typu backpropagation. Pro řešení tohoto problému byla zvolena technologie OpenCL umožňující využít grafické čipy různých výrobců. Hlavním cílem práce byla akcelerace časově náročného procesu učení sítí a procesu klasifikace. Zrychlení bylo docíleno trénováním velkého množství neuronových sítí současně. Získané zrychlení bylo využito pro nalezení vhodné topologie a nastavení neuronové sítě pro zadanou úlohu pomocí genetického algoritmu.

Abstract

The thesis deals with the acceleration of backpropagation neural networks using graphics chips. To solve this problem it was used the OpenCL technology that allows work with graphics chips from different manufacturers. The main goal was to accelerate the time-consuming learning process and classification process. The acceleration was achieved by training a large amount of neural networks simultaneously. The speed gain was used to find the best settings and topology of neural network for a given task using genetic algorithm.

Klíčová slova

Neuronová síť, trénování, klasifikace, OpenCL, GPGPU, GPU, genetický algoritmus

Keywords

Neural network, learning, classification, OpenCL, GPGPU, GPU, genetic algorithm

Citace

Ondřej Šimíček: Akcelerace neuronových sítí s využitím GPU, diplomová práce, Brno, FIT VUT v Brně, 2015

Akcelerace neuronových sítí s využitím GPU

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Jiřího Petrlíka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

. . . . . .
Ondřej Šimíček
26. května 2015

Poděkování

Rád bych poděkoval svému vedoucímu diplomové práce panu Ing. Jiřímu Petrlíkovi za vedení a hodnotné rady v průběhu tvorby práce a také bych rád poděkoval panu Ing. Jiřímu Jarošovi, Ph.D. za poskytnuté technické rady.

Obsah

Kapitola 1

Úvod

Neuronové sítě jsou inspirované fungováním lidského mozku, který obsahuje přes 100 miliard neuronů. Tyto neurony jsou mezi sebou propojené a umožňují zpracovávat příchozí signály, uchovávat a odesílat informace. Toto chování umožňuje učit se a je inspirací pro mnoho algoritmů umělé inteligence. Algoritmy z oblasti umělé inteligence se v praxi používají v mnoha oborech, například v bankovnictví pro určení schopnosti klienta splácet půjčku, na burze pro předpověď dalšího vývoje cen nebo v průmyslu pro kontrolu kvality výroby.

Neuronové sítě s rostoucím počtem neuronů a vrstev umožňují řešit složitější problémy s větší přesností. Proces učení neuronových sítí je však výpočetně náročný a se zvyšujícím se počtem neuronů a vrstev naráží na omezené výpočetní možnosti procesorů. Tato práce si klade za cíl zrychlit tento proces učení pomocí převodu výpočtů na grafický čip, který na rozdíl od procesoru umožňuje běh stovek vláken simultánně. Pro implementaci algoritmu je použit framework OpenCL umožňující využít nejen většinu dostupných grafických karet na trhu, ale také klasické procesory nebo FPGA pole.

Tato práce je rozdělena do šesti kapitol. Úvodní kapitola obsahuje potřebné teoretické informace o neuronových sítích. Nejprve je popsán biologický neuron, který je inspirací umělých neuronových sítí. Dále pak následuje podrobný popis modelu umělého neuronu a jeho aktivační a bázové funkce. Topologie neuronových sítích s různými druhy propojení a typy učení jsou popsány ve vlastní podkapitole. Neuronové sítě s učením typu backpropagation umožňující zpětné šíření chyb jsou rozebrány v poslední podkapitole. Třetí kapitola vysvětluje použití grafických čipů, jejich topologii a rozdíl oproti běžným procesorům. Čtvrtá kapitola je zaměřena na framework OpenCL. Obsahuje informace o výpočetním modelu tohoto frameworku, o jazyku OpenCL C a o problémech spojených s paralelizací. Další část práce se věnuje návrhu řešení neuronových sítí na grafickém čipu s ohledem na maximální využití výpočetních prostředků. Následující kapitola se věnuje implementaci a použitým optimalizačním krokům. V poslední kapitole se nachází výsledky měření porovnávající rychlost implementace na několika grafických kartách a jednom procesoru.

Kapitola 2

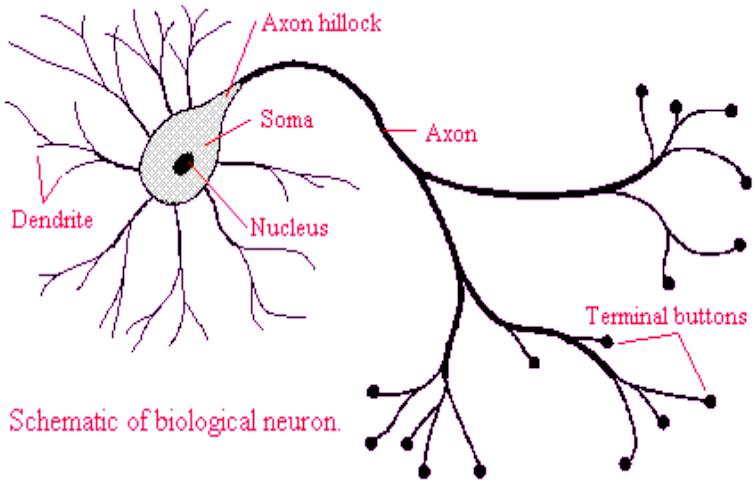
Neuronové sítě

Neuronová síť je výpočetní model inspirovaný fungováním mozku. Tento model je často využíván v algoritmech umělé inteligence. Neuronová síť je tvořena z mnoha vzájemně propojených neuronů, které upravují a přenášejí signály. Síť umožňuje počítači strojové učení a lze pomocí ní řešit řadu úloh. Například úlohy klasifikace, regrese, shlukové analýze, komprese dat a další. Úloha klasifikace si klade za cíl vstupní vektor přiřadit do jedné z několika kategorií. Oproti tomu regrese dokáže vstupu přiřadit spojitou hodnotu, díky tomu lze neuronovými sítěmi aproximovat průběh funkcí. Klasifikace i regrese využívají pro svou činnost učení s učitelem. Typickou úlohou využívající učení bez učitele je shlukování, kde se vstupní vektory shlukují do skupin, tak aby vektory ve skupině byly podobnější než vektory ležící v odlišných skupinách. Tato úloha může mít zajímavé využití při ztrátové kompresi dat kdy se zvolí počet skupin například na 256, kdy vstupní vektory mají mnohem více variací než oněch 256.

Počtem neuronů a strukturou jejich vzájemného propojení definujeme topologii neuronové sítě. V této kapitole je nejprve popsán biologický neuron, z kterého celý systém vychází. Poté je popsán model umělého neuronu včetně druhů bázových a aktivačních funkcí. Dále pak jsou rozebrány jednotlivé topologie neuronových sítí, jejich vrstvy a propojení. Nakonec bude podrobněji popsána neuronová síť s učením typu backpropagation. [8][11]

2.1 Biologický neuron

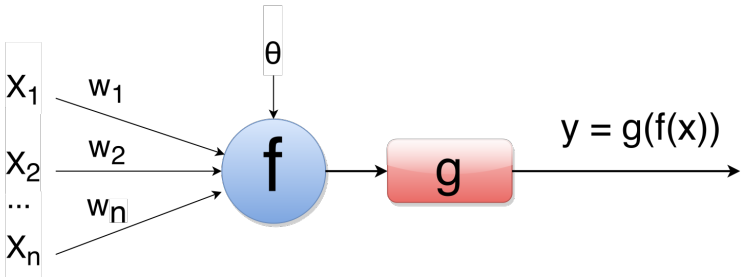
Pochopení funkce mozku bylo možné díky přínosu výzkumníka Santiago Ramón y Cajal, které popsal neurony jako samostatné jednotky, které mezi sebou komunikují pomocí spojů. V dnešní době víme o fungování mozku mnohem více. Nejvýznamnějšími buňkami mozku jsou neurony, které zpracovávají, přenášejí a uchovávají informace. Lidský mozek obsahuje okolo 100 miliard neuronů. Každý neuron má průměrně 7 000 synaptických spojení s dalšími neurony. Stavba neuronu je znázorněna na obrázku 2.1. Skládá se z těla (soma), které obsahuje tisíce vstupů (dendrite) a jeden výstup (axon), který se pak dále dělí na mnoho terminálů. Rozhraní mezi dendritem jednoho a axonem druhé neuronu se nazývá synapse. Elektrický signál je přenášen mezi buňka prostřednictvím synapsí. Síla signálu přijatého neuronem závisí na efektivitě synaptického spojení. Pokud množství vstupních signálů překročí určitou mez, dojde k odeslání signálu ven z neuronu. [8][11]



Obrázek 2.1: Biologický neuron (dendrit, soma, jádro - nucleus, axon, terminály) [5]

2.2 Umělý model neuronu

Umělý neuron je základní prvek umělých neuronových sítí inspirovaný biologickým neuronem. Umělý neuron (dále neuron) je znázorněn na obrázku 2.2. Neuron má n vstupů x, kde každý vstup má váhu w. Vstupy se svými vahami jsou vyhodnoceny pomocí bázové funkce f, jejíž výstup je předán do aktivační funkce g. Bázová funkce neuronu nejčastěji vrací vážený součet vstupů x a vah w. Aktivační funkce tvoří tu část neuronu, která nejvíce ovlivňuje je funkci, blíže bude popsána v další podkapitole. [8][20]



Obrázek 2.2: Struktura umělého neuronu

2.2.1 Bázová funkce

Bázová funkce agreguje vstupní vektor s pomocí vektoru vah do skalární hodnoty. Existují dva základní druhy bázových funkcí - lineární a radiální. Oba tyto typy funkcí jsou popsány níže. [8][22]

Lineární bázová funkce

Lineární bázová funkce plní funkci váženého součtu vstupu. Tato funkce je popsána vzorcem 2.1. Vypočte se sečtením *biasu* θ *(prahovací váhy) s* lineární kombinací (skalárním souči-

nem) vstupního vektoru x a vektoru vah neuronu w. Jedná se o jednoduchou a výpočetně nenáročnou funkci.

$$f(\vec{x}) = \theta + \sum_{i=1}^{n} w_i x_i \tag{2.1}$$

Radiální bázová funkcemi

Radiální bázová funkce vypočítává euklidovskou vzdálenost vstupního vektoru x a vektoru vah w neuronu. Tato funkce je popsána vzorcem 2.2. V některých neuronových sítích se tato bázová funkce používá v první skryté vrstvě. Umožňuje lépe řešit aproximaci funkcí nebo predikci časových řad.

$$f(\vec{x}) = \sqrt{\sum_{i=1}^{n} (w_i - x_i)^2} \tag{2.2}$$

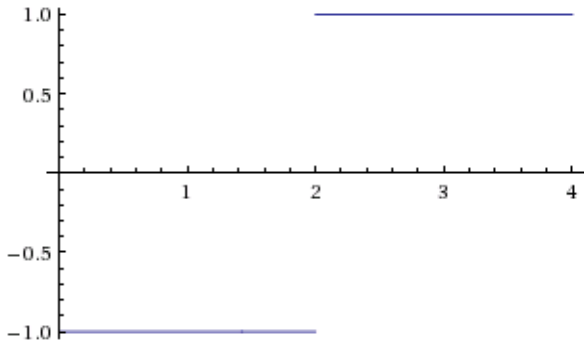
2.2.2 Aktivační funkce

Stěžejním prvkem výpočtů v neuronu je aktivační funkce, která se počítá z výstupu bázové funkce a vytváří výstup neuronu. Zpravidla je tato funkce neklesající a jelikož definuje chování neuronu, vybírá se podle účelu neuronové sítě. Níže jsou uvedené nejzákladnější typy aktivačních funkcí. [8][20]

Skoková aktivační funkce

Rozhoduje na základě porovnání s prahovou hodnotou. Pokud je hodnota báze vyšší než práh T, neuron vygeneruje konstantu a. V opačném případě vygeneruje konstantu b. Nejčastěji se tato funkce používá při klasifikaci dat do tříd. Skoková aktivační funkce je popsána vzorce 2.3, její průběh je znázorněn na obrázku 2.3.

$$y = f(x) = \begin{Bmatrix} a & prox < T \\ b & prox \geqslant T \end{Bmatrix} \tag{2.3}$$

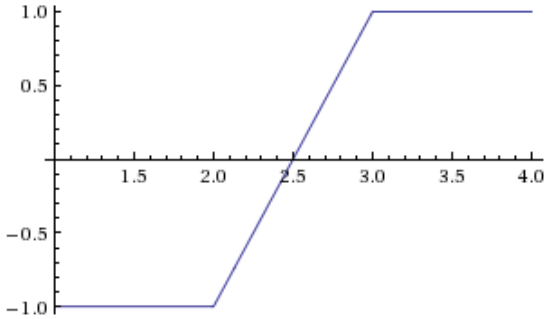


Obrázek 2.3: Skoková aktivační funkce pro a=-1, b=1, T= 2

Po částech lineární aktivační funkce

Průběh funkce je konstantní po práh T_1, dále je lineární až po práh T_2 a dále je zase konstantní. Může být využita jako zjednodušení hyperbolické tangenty. Po částech lineární aktivační funkce je popsána vzorce 2.4, její průběh je znázorněn na obrázku 2.4.

$$y = f(x) = \left\{ \begin{array}{cc} a & prox < T_1 \\ b & proX > T_2 \\ a + \frac{(b-1)(x-T_1)}{T_2-T_1} & proT_1 \leqslant x \leqslant T_2 \end{array} \right\} \tag{2.4}$$

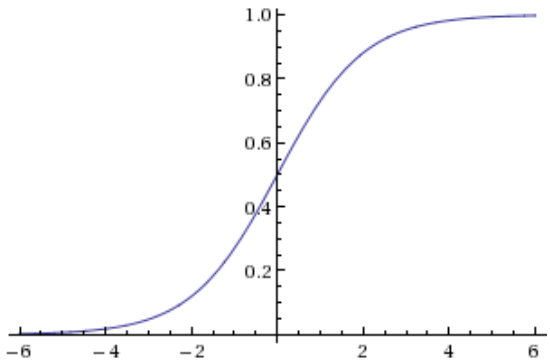


Obrázek 2.4:
Po částech lineární aktivační funkce pro a=-1, b=1, T_1= 2, T_2=3

Sigmoidální aktivační funkce

Sigmoidální aktivační funkce je nejběžněji používanou aktivační funkcí. Tato funkce je definována vzorcem 2.5, kde a určuje hodnotu v mínus nekonečnu, b určuje hodnotu v plus nekonečnu, c určuje posun funkce a λ určuje strmost funkce. Průběh sigmodiální aktivační funkce je znázorněn na obrázku 2.5.

$$y = f(x) = a + \frac{b-a}{1+e^{-\lambda x+c}} \tag{2.5}$$

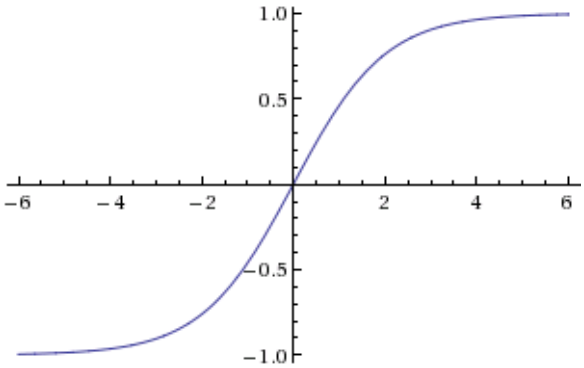


Obrázek 2.5: Sigmoidální aktivační funkce pro a=0, b=1, λ=1, c=0

Aktivační funkce hyperbolické tangenty

Tato aktivační funkce má průběh velmi podobný sigmoidální aktivační funkci. Lze jí získat jejím posunutím a dilatací. Funkce hyperbolické tangenty je definována vzorcem 2.6, kde a určuje hodnotu v mínus nekonečnu, b určuje hodnotu v plus nekonečnu, c určuje posun funkce a λ určuje strmost funkce. Průběh funkce hyperbolické tangenty je znázorněn na obrázku 2.6.

$$y = f(x) = 0.5(a + b + (b - a)\tanh(\lambda x + c)), \tag{2.6}$$



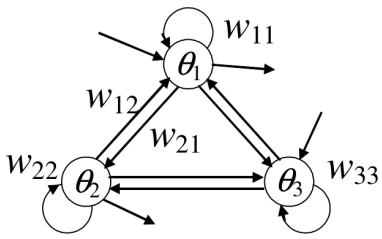
Obrázek 2.6: Aktivační funkce hyperbolické tangenty pro a=-1, b=1, λ=1, c=0

2.3 Topologie

Neuronové sítě lze rozdělit z různých hledisek. Rozdělení podle architektury zkoumá propojení jednotlivých neuronů mezi sebou. Dále sítě můžeme dělit podle způsobu učení nebo použití. [8][20]

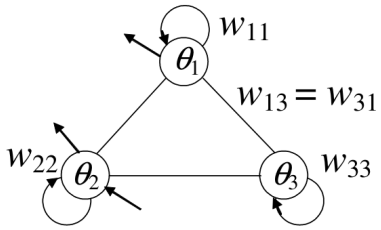
Architektury sítí

- **Plně propojená síť** - má všechny neurony vzájemně propojeny. Tato síť je znázorněna na obrázku 2.7.



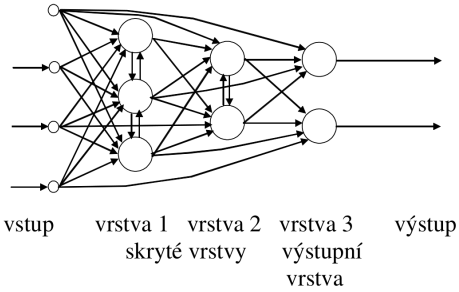
Obrázek 2.7: Plně propojená neuronová síť [20]

- **Plně propojená symetrická síť** - jedná se o variantu*plně propojené* sítě, kde hodnoty váh propojených neuronů jsou stejné v obou směrech $W_{ij} = W_{ji}$. Tato síť je znázorněna na obrázku 2.8.



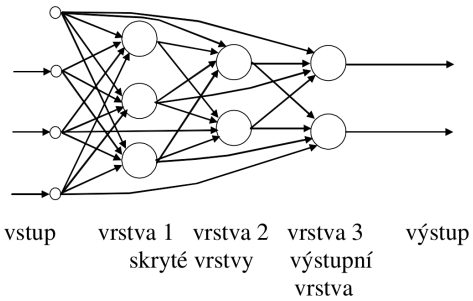
Obrázek 2.8: Plně propojená symetrická neuronová síť [20]

- **Vrstvová síť** - neurony jsou rozděleny do vrstev. Vstupy jsou napojeny na první skrytou vrstvu. Po skrytých vrstvách následuje výstupní vrstva. Neurony jedné vrstvy neovlivňují neurony vrstev předcházejících jako je znázorněno na obrázku 2.9.



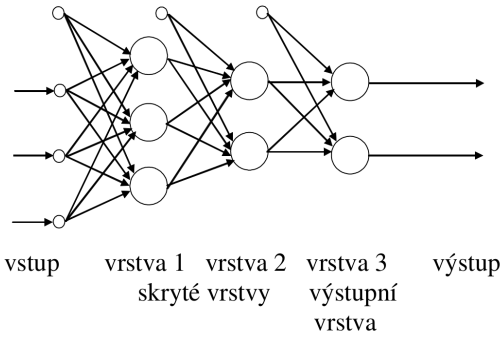
Obrázek 2.9: Vrstvová neuronová síť [20]

- **Acyklická síť** - jedná se o variantu *vrstvové sítě*, kde neexistují spojení mezi neurony stejné vrstvy. Tato síť je znázorněna na obrázku 2.10.



Obrázek 2.10: Acyklická neuronová síť [20]

- **Dopředná síť** - druh *acyklické sítě*, ve které existují propojení jen mezi neurony sousedních vrstev. Tato síť je znázorněna na obrázku 2.11.

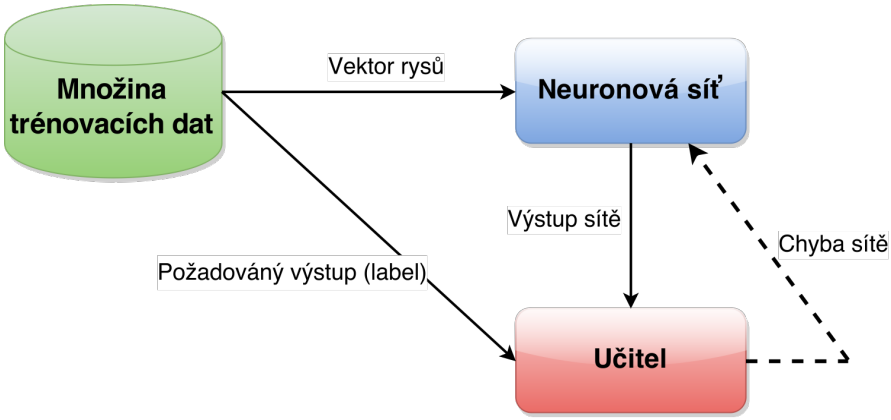


Obrázek 2.11: Dopředná neuronová síť [20]

Učení sítí

Princip učení neuronových sítí je založen na úpravě vah sítí tak, aby síť reagovala na vstupy požadovaným způsobem. Tento proces učení probíhá v adaptační fázi, poté následuje fáze vybavování. V této fázi se již váhy neupravují a probíhá pouze převod vstupních dat na výstup.

- **Učení s učitelem** - množina vstupních dat se náhodně rozdělí na trénovací množinu a testovací množinu. Z množiny trénovacích dat jsou vybírány prvky. Každý prvek obsahuje vektor rysů a hodnotu požadovaného výstupu. Vektor rysů je zpracován neuronovou sítí a výstup sítě je porovnán s požadovanou hodnotou výstupu. Odchylka výstupu od požadovaného výstupu se použije pro úpravu vah neuronů sítě. Tento proces je ilustrován na obrázku 2.12.



Obrázek 2.12: Učení neuronové sítě s učitelem.

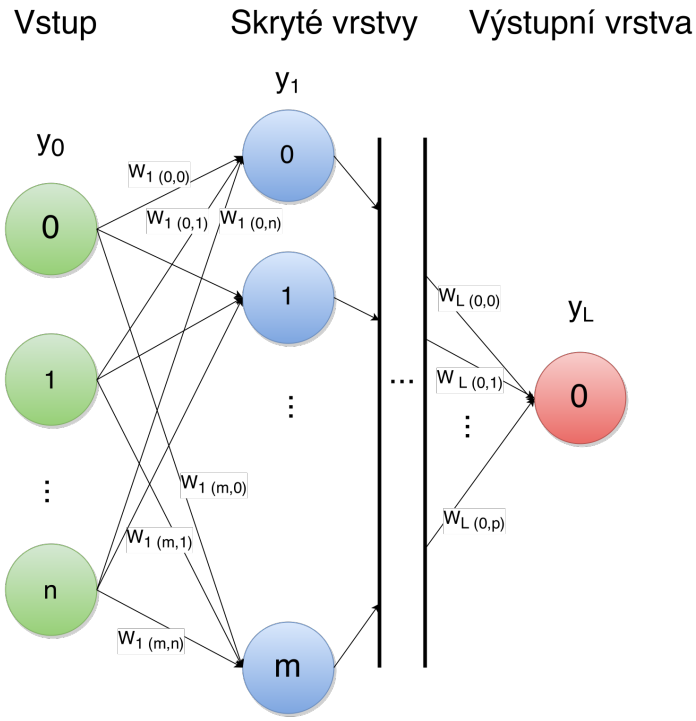
Naučená síť se poté testuje na testovací množině dat. Testovací data jsou předána na vstup sítě. Výstup sítě se porovná s požadovaným výstupem a počítá se celková

chyba neuronové sítě. Může dojít k situaci, kdy neuronová síť má nízkou chybovost při trénování, ale při testování vykazuje velkou chybu. Tento jev se nazývá přeučení a znamená, že síť negeneralizuje problém správným způsobem.

- **Učení bez učitele** - prvky z trénovací množiny neobsahují hodnoty požadovaných výstupu, které by se použilo pro určení správnosti výstupů sítě. Trénovací množina obsahuje pouze data pro vstup sítě. Síť se učí sama na základě souvislostí ve vstupních datech, které třídí do skupin a přizpůsobuje svou topologii podle vlastností vstupům.

2.4 Sítě s učením typu backpropagation

Metoda učení *backpropagation* v překladu znamená učení se zpětnou propagací chyb. Jedná se o nejpoužívanější metodu u neuronových sítí. Je určena pro nerekurentní vrstvovou síť s jednou a více skrytými vrstvami. Backpropagation je metoda učení s učitelem, který síti předkládá trénovací vstupy a výstup porovnává s očekávaným výstupem. Na základě odchylky od očekávaného výstupu jsou pak modifikovány váhy neuronů. Tato modifikace probíhá směrem od výstupní vrstvy, přes skryté vrstvy ke vstupu, proto se tato metoda nazývá *backpropagation*. [11][21]



Obrázek 2.13: Plně propojená neuronová síť s L-1 skrytými vrstvami.

Backpropagation algoritmus

V následujícím odstavci bude popsán algoritmus backpropagation převzatý z [18]. Jak již bylo řečeno, učení typu backpropagation je učení s učitelem a je založeno na zpětném

šířením chyb. Máme množinu trénovacích vektoru X. Nejprve se inicializuje vstup pomocí vektoru x_0

$$\vec{y}_0 = \vec{x}_0, \tag{2.7}$$

dále se data šíří dopředně z první skryté vrstvy l=*1*, přes další vrstvy až k poslední výstupní vrstvě l=L,

$$\vec{y}_l = f(W_l \cdot \vec{y}_{l-1}), \tag{2.8}$$

kde pod f se rozumí aktivační funkce a W_l je matice vah vrstvy l, kde každý řádek matice reprezentuje vektor vah pro každý neuron vrstvy. Vektor y_l obsahuje hodnoty výstupů neuronů vrstvy l. Když výpočet dojde k výstupní vrstvě, začne vypočítávat chybu neuronů výstupní vrstvy

$$\vec{\Delta}_L = (\vec{t} - \vec{y}_L) f'(\vec{y}_L), \tag{2.9}$$

kde t je vektor očekávaného výstupu a f' je derivace aktivační funkce. Poté pokračuje zpětná propagace chyb přes všechny skryté vrstvy,

$$\vec{\Delta}_l = (W_{l+1}^T \vec{\Delta}_{l+1}) f'(\vec{y}_l). \tag{2.10}$$

Nakonec se aktualizují hodnoty vah,

$$W_l \leftarrow W_l + \eta \vec{\Delta}_l y_{l-1}^T, \tag{2.11}$$

kde η je faktor učení ovliňující konvergenci sítě. Faktor učení může být zadán jako libovolná kladná hodnota. Na obrázku 2.13 je znázorněna neuronová síť s *L-1* skrytými vrstvami. U každé vrstvy je vyznačen vektor y obsahující hodnoty výstupů neuronů dané vrstvy a matice vah W, u které jsou v závorkách uvedeny pozice vah. Souřadnice vah v matici mající formát *(x, y)*.

Kapitola 3

Výpočty s využitím grafických karet

Výpočty s využitím grafických karet General-purpose computing on graphics processing units (GPGPU) je koncept umožňující na grafickém čipu, na kterém typicky probíhají grafické výpočty, provádět obecné výpočty dříve určené pro procesor počítače. Procesory počítačů se vyvíjejí mnohem pomaleji než grafické čipy. Typický procesor má dvě až čtyři jádra a jeho výkon se pohybuje pod 100 GFLOPS. Oproti tomu GPU obsahují desítky jader běžících na nižší frekvenci.Tyto jádra jsou od počátku vyvíjené pro paralelní zpracování velkého objemu obrazových dat. Celkově obsahují stovky až tisíce výpočetních prvků a celkový výkon se běžně pohybuje 2 000 GFLOPS což je 20x více. [7]

CUDA (Compute Unified Device Architecture) je hardwarová a softwarová architektura. Byla představena společností NVidia v roce 2006 jako reakce na vzrůstající poptávku po možnosti obecných výpočtů na grafických kartách. CUDA nabízí vývojové prostředí, ve kterém vývojář programuje v rozšířené variantě jazyka C/C++ nebo Fortran. CUDA dává vývojářům velkou volnost pro psaní nejrůznějších algoritmů. Významným limitujícím faktorem této technologie je uzavřenost a nutnost použít zařízení společnosti NVidia.

OpenCL je platforma nabízející jednotné vývojové prostředí pro velké množství více jádrových zařízení. Podobnost s technologií CUDA je u implementace kernelů v jazyce C a také organizace vkláken do skupin, které jsou prováděny v jádrech. Hlavní rozdíl mimo odlišnou terminologii je podpora výrobců karet. Na rozdíl od CUDA jej najdeme u grafických karet více výrobců (AMD, NVidia, Intel). Jelikož OpenCL je určen pro heterogenní platformy, podporuje kromě grafický karet také klasické procesory a FPGA čipy.

Framework OpenCL za dobu své pětileté existence prošel vývojem, na kterém se podílela celá řada společností. V současné době nabízí pro práci bohaté aplikační rozhraní a několik ladících nástrojů. Nedávno byla vydána nová verze 2.0, bohužel chvíli potrvá než se nová zařízení, podporující tuto verzi, rozšíří. Pro řešení této práce byla zvolena verze OpenCL 1.1, která zajistí kompatibilitu s širokou řadou zařízení. Více se technologii OpenCL věnuje čtvrtá kapitola. [7][17]

Vytvořená implementace byla vyvíjena testována na grafické kartě Sapphire HD 7970. Tato karta byla pro práci zvolena, protože poskytuje dostatečný výkon pro řešení náročných výpočtů a zároveň je ekonomicky dostupnější v porovnání s kartami společnosti NVidia. Přehled vlastností karty je uveden v následující podkapitole.

Za účelem lepšího otestování a porovnání výkonu byly použity ještě další grafické karty. Jedná se o karty společnosti NVidia. První kartou je GTX 580, jejíž cena byla stejně vysoká

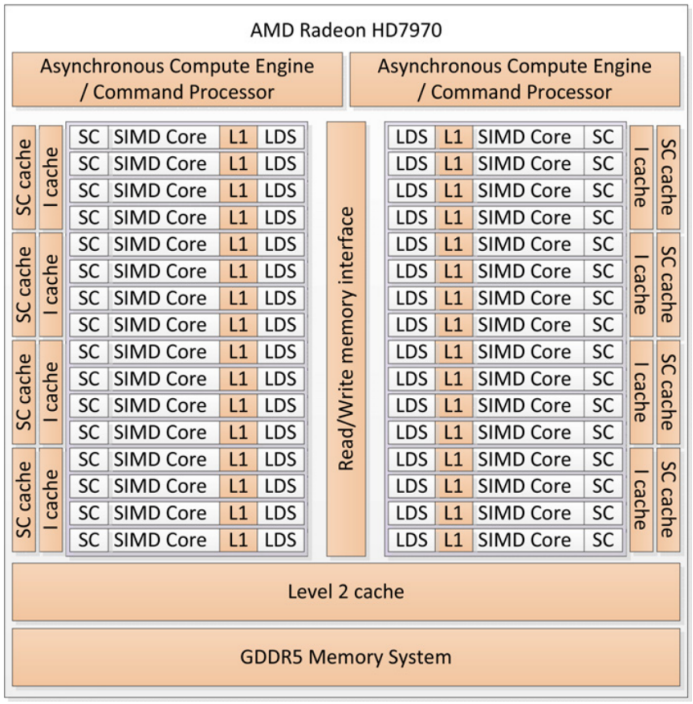
jako cena karty HD 7970, ale přišla na trh o rok dříve. Druhá karta NVidia GTX 980 patří v současné době mezi špičku grafických karet. Shrnutí vlastností všech tří použitých grafických karet je v tabulce 3.1.

Grafická karta	**HD 7970**	**GTX 580**	**GTX 980**
Rok	2011	2010	2014
Frekvence jádra	925 Mhz	772/1544 MHz	1291 MHz
Výpočetní jednotky	**32**	**16**	**16 (64)**
Výkonné prvky	**2048**	**512**	**2048**
Frekvence paměti	5500 Mhz	4000 MHz	7000 MHz
Velikost paměti	3072	1536	4096
Cena v době vydání	500 USD	500 USD	600 USD

Tabulka 3.1: Srovnání parametrů použitých grafických karet.

3.1 AMD Radeon HD7970

Jak již bylo zmíněno, pro vývoj byla použita výkonná grafická karta s čipem Radeon HD7970. Tento čip s označením Tahiti byl představen na konci roku 2011. Grafické znázornění struktury čipu je na obrázku 3.1. Výroba čipu probíhá 28nm technologií, kde je na čip vměstnáno 32 výpočetních jednotek typu GCN 1.0 (Graphics Core Next).



Obrázek 3.1: Architektura AMD HD 7970. Zařízení má celkem 32 jader (core). Každé jádro obsahuje skalární jednotu (SC) a 4 vektorové SIMD jednotky. Každé jádro je také vybaveno vlastní lokální pamětí (LDS) a obsahuje L1 cache.

Výpočet na kartě obstarává 2048 výkonných prvků rovnoměrně rozložených do 32 výpočetních jednotek. Na každou výpočetní jednotu vychází 64 výkonných prvků. Frekvence jádra je nastavena na 925MHz a frekvence připojené 3GB GDDR5 paměti je 1375Mhz. Tato frekvence paměti nabízí, se šířkou sběrnice 384 bitů, teoretickou propustnost až 264GB/s. Teoretický výkon karty je 3788 GFLOPS. Nutno zmínit, že tento výkon je silně teoretický, protože vychází z použití pouze mad instrukcí (mul a add), kde se provede násobení a sečtení v jednom kroku. Hlavní předností této karty je velmi příznivý poměr ceny a poskytnutého výkonu.

Kapitola 4

OpenCL

OpenCL (Open Compute Language) je framework pro programování paralelních výpočtů na různých hardwarových platformách. Umožňuje přenositelnost mezi zařízeními různých výrobců bez výrazné ztráty výkonu. Na jeho vývoji pracuje průmyslové konsorcium Khronos, do kterého patří mnoho velkých společností jako je Apple, AMD, Intel, IBM a další. Khronos vytváří otevřené standarty, mezi které patří například OpenGL standart pro tvorbu počítačové grafiky. OpenCL definuje abstraktní hardwarové zařízení a k němu softwarové rozhraní. Podpora CPU s SSE, grafické procesory NVidia GeForece a ATI Radeon, DSP, mobilní čipy, FPGA a další. Programování se skládá ze dvou částí. Programování hosta, který je prováděn na procesoru. Dále pak z programování kernelů, které běží na zařízeních podporujících OpenCL. Kompilace probíhá ve dvou fázích. V první fázi je zkompilován program hosta. V druhé fázi během provádějí programu hosta je dynamicky zkompilován kód kernelu pro zařízení. Verze systému OpenCL je definována verzí platformy, verzemi zařízení a verzí jazyka OpenCL C. Verze platformy označuje jaká verze runtime OpenCL je podporována, která definuje API pro hosta. Verze zařízení je nejvyšší verze OpenCL s kterou umí zařízení pracovat avšak není vyšší než verze platformy. Verze jazyka OpenCL C udává s jakou verzí může vývojář pracovat. Tento jazyk je navržen tak aby byl zpětně kompatibilní. Výchozí verzí jazyka je ta nejvyšší dostupná avšak nesmí být vyšší než je verze platformy. Architektura OpenCL lze popsat těmito modely. Platformním modelem, paměťovým modelem, vykonávacím modelem a programovacím modelem. Podrobněji jsou tyto modely popsány v dalších podkapitolách. [7][17]

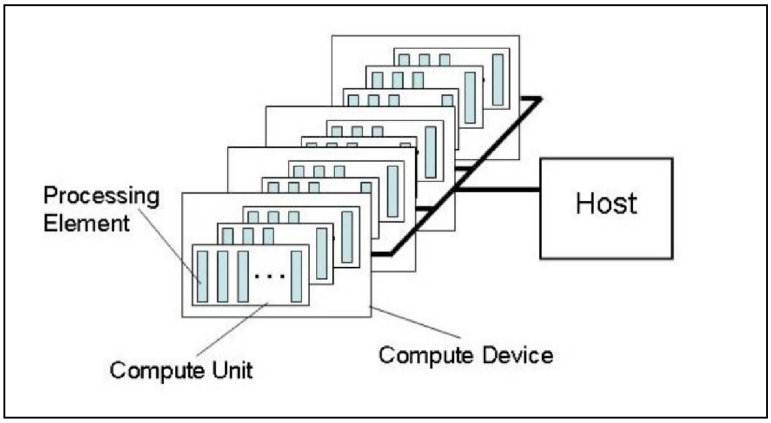
4.1 Platformní model

Platformní model OpenCL definuje jeden procesor (host), který vše řídí jako je znázorněno na obrázku 4.1 níže. K hostu je připojeno několik zařízení (compute device), které obsahují jednu nebo více vypočteních jednotek. Každá výpočetní jednotka (compute unit) je dále rozdělena do jednoho nebo více výkonných prvků (processing element). OpenCL aplikace běží na hostitelském systému, odkud odesílá příkazy směrem k zařízením, ve kterých výkonné prvky provedou výpočty. Výkonné prvky v rámci jedné výpočetní jednotky se chovají jako SIMD (Single instruction, multiple data). Všechny výkonné prvky provádí synchronně v každý okamžik stejnou instrukci. [7][6]

Platformní model rozděluje systém do těchto částí:

Host je ve většině případu CPU počítače, které řídí činnost systému a tvoří propojení mezi zařízeními.

Zařízení (compute device) je fyzické zařízení, které podporuje technologii OpenCL. Například grafická karta, procesor, FPGA zařízení a další.
Výpočetní jednotka (compute unit), někdy také nazývaná stream multiprocesor, obsahuje vlastní skalární procesor, lokální paměť, sady registrů a další jednotky. Výpočetní jednotky obsahují plánovač, který řídí činnost *výkonných prvků*.
Výkonný prvek (processing element) je virtuální skalární procesor, který slouží k výpočtu jednoduchých úloh. Výkonné prvky jsou uspořádány do, v kterých potom dohromady provádí jednoduché vektorové instrukce jako jsou sčítání, násobení, bitové posuvy a podobné.



Obrázek 4.1: Platformní model - host s několika připojenými zařízeními, které obsahují jednu nebo více výpočetních jednotek s řadou výkonných prvků. [6]

4.2 Exekuční model

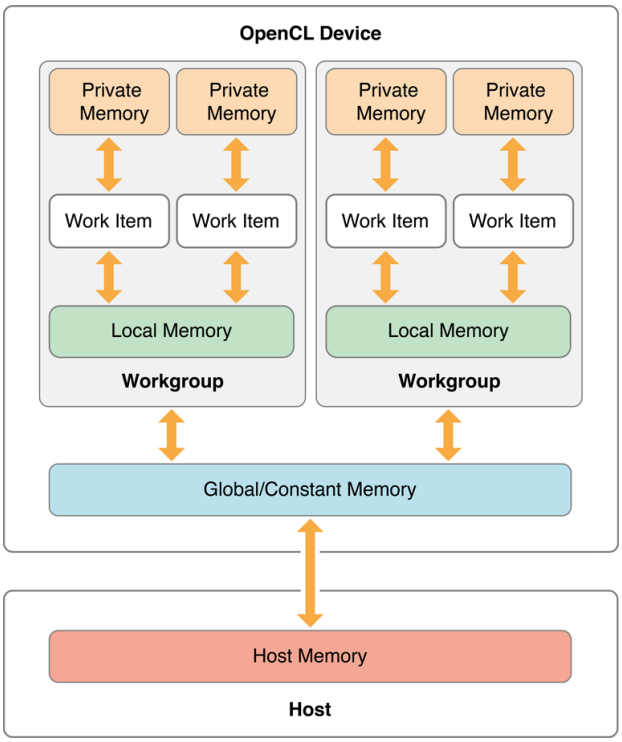
Program OpenCL se provádí na dvou úrovních. První úroveň je kód hosta, který běží na CPU a druhou je kód zařízení napsaný v OpenCL C. Programy spuštěné na zařízení se nazývají kernely. Tyto kernely jsou řazené do fronty, kde jsou vykonávány in-order, nebo out-of-order. Pokud je kernel do fronty zařazen jako in-order, je vykonán jakmile je předchozí kernel dokončen. To znamená, že se nemusí využít všechny výkonné prvky zařízení (processing element). Pokud je kernel zařazen jako out-of-order může část využít část výkonných prvků a další kernel využije druhou část. Out-of-order kernely se spouští ihned jakmile jsou volné zdroje zařízení. Instance kernelu spuštěná na výkonném prvku zařízení se nazývá pracovních položka (work item). Zařízení umožňuje běh více pracovních položek, které jsou rozděleny do pracovních skupin Pracovní položky pracovní položky ve stejné pracovní skupině mohou spolu komunikovat. Kontext v OpenCL se skládá ze zaregistrovaných zařízení. Díky tomu, že jsou zařízení v jednom kontextu mohou sdílet data. K více-jádrovému procesoru se chová jako k jednomu zařízení a kernely se provádí na všech jádrech. Spojené grafické karety se také tváří jako jedno zařízení. V rámci kontextu se každému zařízení přiřadí fronta příkazu (command queue). Pomocí těchto front se přiřazují úkoly každému zařízení.

Wavefront je aktuálně prováděna skupina pracovních položek na výpočetní jednotce. V terminologii CUDA je ekvivalentní termín *warp*. Pokud je velikost pracovní skupiny větší

než počet výkonných prvků ve výpočetní jednotce, pak plánovač výpočetní jednotky rozdělí pracovní položky pracovní skupiny do sad, které provádí cyklickým způsobem. Stejně tak, pokud je pracovních skupin více než je výpočetních jednotek, bude se více pracovních skupin provádět na jedné výpočetní jednotce cyklickým způsobem. Pojem *wavefront* je těžko přeložitelný, proto se bude dále v práci používat nepřeložený. [7][6]

4.3 Paměťový model

Paměť v OpenCL je rozdělena do dvou částí. **Paměť hosta** je viditelná pouze pro hosta, který definuje práci s ní. Data mezi pamětí hosta a pamětmi zařízení se přesouvají pomocí OpenCL API nebo pomocí rozhraní virtuální paměti. **Paměť zařízení** je přímo přístupná kernelů prováděných na zařízení. Dělí do několika skupin zobrazených na obrázku 4.2. Oprávnění pro zápis a čtení jsou znázorněny v tabulce 4.1.[7][6]



Obrázek 4.2: Konceptuální OpenCL architektura rozdělení paměti v zařízení. [6]

- **Soukromá paměť** (private memory) je unikátní prostor pro každou pracovní položku. Jsou zde uloženy lokální skalární proměnné.
- **Lokální paměť** (local memory) určena pro všechny pracovní položky v rámci jedné pracovní skupiny. Pracovní položky si přes ní mohou předávat data mezi sebou.
- **Globální paměť** (global memory) je sdílena mezi všemi pracovními položkami (work item) v rámci všech pracovních skupin (workgroup) v rámci jednoho zařízení. Obsahuje mimo jiné i oblast, v které jsou alokovaná statická data.

	Globální	Glob. konstanty	Lokální	Soukromá
Host	dynamická alokace čtení / zápis	dynamická alokace čtení / zápis	dynamická alokace nemá přístup	nealokuje nemá přístup
Kernel	nealokuje čtení / zápis	statická alokace čtení	statická alokace čtení / zápis	statická alokace čtení / zápis

Tabulka 4.1: Skupiny pamětí zařízení - alokace a přístupy k nim.

Konflikty v lokální paměti

Sdílená lokální paměť pracovních položek je v rámci jedné výpočetní jednotky fyzicky uložena na jednom paměťovém čipu. Velikost paměťového čipu pro soukromou paměť každé výpočetní jednoty se v současné době pohybuje okolo 64KB. Tato paměť je rozdělena do 16 někdy 32 banků, takovým způsobem, že nultá položka paměti (její velikost je 32/64 bitů) patří do nultého banku, 15. položka patří do 15. banku a 16. položka patří zase do nultého banku. Toto rozdělení lokální paměti je ilustrováno na obrázku 4.3.

Obrázek 4.3: Rozdělení lokální paměti do banků.

Maximální efektivita nastává, když pracovní položky pracují s položkami paměti, které patří do různých banků. Pokud dvě pracovní položky přistupují k různým položkám paměti v rámci jednoho banku, nastává konflikt. Druhá pracovní položka musí čekat než se vybaví požadavek první pracovní položky. Pokud dvě pracovní položky čtou ze stejné položky paměti, konflikt nenastává a hodnota se vyčte současně pro obě pracovní položky.

4.4 Programovací model

OpenCL podporuje datově paralelní i úkolově paralelní programovací model. Oba tyto modely lze kombinovat do docílit hybridního modelu. Pro rozdělení problému mezi několik výpočetních prostředků existují dvě standardní metody. První je metoda *Rozděl a panuj* (Divide and conquer) iterativně rozděluje problém na menší části do té doby, než je lze napasovat na výpočetní prostředky. Druhá metoda *Rozsyp sesbírej* (Scatter-gather) rozloží vstup na podmnožiny, které rozvrhne mezi paralelní výpočetní prostředky. Výsledky zase spojí dohromady. Tato metoda má velmi blízká datově paralelnímu programování. [7][6]

Datově paralelní programovací model

Tento model je zaměřen na paralelní provádění stejného kódu nad mnoha daty. Podle počtu čítačů instrukcí se dělí na SPMD a SIMD. SIMD (Single Instruction, Multiple Data) je systém výpočtu, u kterého simultánně provádí jedna operace na všech výkonných prvcích

vektorového procesoru, nad odlišnými daty. Oproti tomu SPMD (single program, multiple data) systém prování simultánně jeden program, na více procesorech, nad odlišnými daty. V OpenCL datově paralelní programování znamená spuštění mnoha pracovních položek (work-items) se stejným kódem, který zpracovávají společný balík dat. Není nutné aby bylo dodrženo mapování mezi pracovní položkou a prvkem paměti 1:1, různé pracovní položky mohou pracovat se stejným prvkem paměti.

Úkolově paralelní programovací model

Tento model je postaven na rozdělení problému do několika úloh, které budou prováděny paralelně. V OpenCL lze na jednom zařízení paralelně provádět více různých kernelů. Tyto kernely mohou spolupracovat a předávat si výstupy mezi sebou. Pro synchronizaci takových kernelů se používají OpenCL události.

4.5 Jazyk OpenCL C

Tento jazyk vychází ze standardu C99, ale oproti němu nepodporuje standardní hlavičkové soubory, neobsahuje ukazatele na funkce, nepodporuje rekurzi, pole s proměnnou délkou a pole bitů. Přidává funkce pro zjištění informací o pracovních položkách a o pracovních skupinách (work groups), dále pak přidává vektory a synchronizaci. Jazyk OpenCL c je rozšířen o kvalifikátor adresového prostoru (global, local), optimalizovaný přístup k obrázkům a vestavěné funkce. [6]

Ukázka zdrojového kódu 4.1 kernelu pro výpočet druhých mocnin ze seznamu čísel. Vstupem je pole desetinných čísel *input*. Každá pracovní položka přečte jednu hodnotu ze vstupního pole, provede umocnění a zapíše výsledek na odpovídající místo ve výstupním poli *output*. Pomocí vestavěné funkce *get_global_id* se zjistí pozice pracovní položky v rámci jednorozměrné pracovní skupiny.

Zdrojový kód 4.1: Kernel pro výpočet druhých mocnin seznamu čísel.

```
__kernel void square(__global float* input,
                     __global float* output) {
    int index = get_global_id(0);
    output[index] = input[index] * input[index];
}
```

Před spuštění kernelu, podvádějícím druhou mocninu 4.1, je zapotřebí provést několik kroků souvisejících s přípravou zařízení, kódu a paměti. Tento kód hosta 4.2 je psán v jazyce C++, který s OpenCL rozhraní komunikuje pomocí knihovny OpenCL C++ wrapper. Pro snadnější práci s rozhraním byla implementována pomocná třída *OpenclHelper*, která usnadňuje výběr zařízení a kompilaci kódu.

Nejprve je vybráno zařízení s indexem 0 na platformě 0. S tímto zařízením je vytvořen OpenCL kontext a fronta příkazů. Následuje načtení souboru s kódem kernelu a jeho následná kompilace. V dalším kroku se alokuje paměť pro vstupní a výstupní čísla. Následuje naplnění vstupních čísel. Ze seznamů čísel se vytvoří OpenCL buffery, které se odešlou do zařízení pomocí fronty příkazů. Před spuštěním samotného kernelu se mu nastaví velikost pracovní skupiny a následuje jeho zařazení do fronty příkazů. Po dokončení zpracování se načtou výsledky umocnění do výsledného seznamu čísel.

Zdrojový kód 4.2: Kód hosta spouštějící kernel pro výpočet druhých mocnin.

```
int main(int argc, char **argv) {
   cl_int status;
   // zarizeni 0 na platforme 0
    Device device = OpenclHelper::get_device(0, 0);
    VECTOR_CLASS<Device> devices;
    devices.push_back(device);
    // vytvoreni kontextu nad jednim zarizenim
    Context ctx(devices, NULL, NULL, NULL, &status);
    CommandQueue queue(ctx, device, NULL, &status);

    char *knl_text = OpenclHelper::read_file("square.cl");
    std::vector<std::string> codes;
    codes.push_back(knl_text);
    // kompilace kodu
    Program program = OpenclHelper::build_program(ctx, codes);
    Kernel knl(program, "square", &status);
    // alokace pameti
    int len = 100;
    float *numbers_in = (float *)malloc(len * sizeof(float));
    float *numbers_out = (float *)malloc(len * sizeof(float));
    for (int index = 0; index < len; index++) {
      numbers_in[index] = index / 3.f;
    }
    Buffer buffer_in(ctx, CL_MEM_READ_ONLY, len * sizeof(float),
                    (void *)numbers_in, &status);
    Buffer buffer_out(ctx, CL_MEM_WRITE_ONLY, len * sizeof(float),
                    (void *)numbers_out, &status);
    // nastaveni parametru kernelu
    knl.setArg(0, buffer_in);
    knl.setArg(1, buffer_out);
    // spusteni kernelu
    queue.enqueueNDRangeKernel(knl,
                               NullRange,
                               NDRange(len, 1, 1),
                               NDRange(len, 1, 1),
                               NULL, NULL);
   // cteni vysledku
   queue->enqueueReadBuffer(buffer_out, CL_TRUE, 0,
                            len * sizeof(float), numbers_out);

   free(numbers_in);
   free(numbers_out);
}
```

Ukázka zdrojového kódu kernel 4.3 pro převod obrázku do odstínů šedé. Pracovní položky si obraz rozdělí mezi sebou na menší části a každá zpracuje svoji část. Vstupem kernelu je celý barevný obrázek *input*. Každá pracovní položka si spočítá souřadnice a velikost výřezu, s kterým bude pracovat. Velikost obrazu zjistí pomocí funkce get_image*e_dim*. Výřez spočítá pomocí funkce get_global_size*(0|1)* vracející rozměr pracovní skupiny v osách x a y. Funkce *get_global_id(0|1)* vrací informace o poloze pracovní položky v rámci pracovní skupiny. Získaný výřez se prochází bod po bodu. Hodnota bodu se načítá pomocí

read_imageui vracející čtveřici typu *uchar*. Tato čtveřice obsahuje informace o barevných kanálech a alfa kanále. Z barevných kanálu se spočítá intenzita, která se zapíše do výstupního obrazu *output*.

Zdrojový kód 4.3: Kernel pro převod obrázku do odstínu šedi.

```
__kernel void to_grayscale(__read_only image2d_t input,
                           __write_only image2d_t output) {

    const int2 img_dim = get_image_dim(input);
    // nastaveni vyrezu
    int fromX = get_global_id(0) * (img_dim.x / get_global_size(0));
    int toX = fromX + img_dim.x / get_global_size(0);
    int fromY = get_global_id(1) * (img_dim.y / get_global_size(1));
    int toY = fromY + img_dim.y / get_global_size(1);

    // uprava podle hranic
    if (toX > img_dim.x) {
        toX = img_dim.x;
    }
    if (toY > img_dim.y) {
        toY = img_dim.y;
    }

    for (int y = fromY; y < toY; y++) {
        for (int x = fromX; x < toX; x++) {
            const int2 pos = {x, y};
            uint4 pixel = read_imageui(input, sampler, pos);
            // vypocet odstinu sede
            uint gray = (float)0.299 * pixel.x + (float)0.587 * pixel.y +
        (float)0.114 * pixel.z;
            write_imageui(output, pos, (uint4)(gray, gray, gray, gray));

        }
    }
}
```

Kapitola 5

Návrh

Kapitola návrhu se nejprve věnuje analýze efektivního využití GPU zdrojů pro implementaci algoritmu Backpropagation. V první podkapitole jsou popsány jednotlivé přístupy paralelizace neuronových sítí a jejich použitelnost pro akceleraci pomocí moderních GPU. V druhé podkapitole je podrobněji vysvětlen zvolený návrh implementace neuronových sítí s využitím technologie OpenCL.

5.1 Využití zdrojů GPU

Před návrhem je důležité se zamyslet nad způsobem jakým algoritmus Backpropagation funguje a jaké má omezení z pohledu paralelismu. Tento algoritmus se skládá ze tří částí. V první části se vypočítají výstupy neuronů na základě vstupního vektoru. V druhé části se zpětným směrem vypočtou chyby a v poslední části se upraví váhy neuronů na základě velikosti chyb neuronů. Při výpočtu výstupů je nutné provádět výpočty po vrstvách, protože výstup aktuální vrstvy závisí na výstupech předešlé vrstvy. Neurony ve stejné vrstvě se navzájem neovlivňují a jejich výstup může být spočítán paralelně. Podobný problém je i u výpočtu chyb jen s tím rozdílem, že u něj chyba neuronů aktuální vrstvy závisí na chybách následující vrstvy. Úprava vah všech neuronů může probíhat paralelně, protože není vzájemně závislá.

Z výše uvedeného vyplývá, že lze paralelně provádět pouze neurony ve stejné vrstvě. V naivní implementaci bychom neurony rozdělili mezi výpočetní jednotky a prováděli výpočty. To by vyžadovalo synchronizaci, která mezi pracovními skupinami běžícími na výpočetních jednotkách nejsou.

Jedinou možností, jak provádět trénování neuronové sítě na všech výpočetních jednotkách, je pro každý výpočet vrstvy spustit kernel, provést výpočet a následně předat řízení hostu, který následně opět zavolá kernel provádějící další vrstvu. Tímto postupem se synchronizují výpočty mezi vrstvami. Na tomto principu pracuje mnoho implementací, které se snaží trénovat neuronovou síť opakovaným volání kernelu. Vrstvy jsou pak zpracovány pomocí maticových a vektorových operací za použití paralelní redukce. Z principu tato implementace vyžaduje velké množství neuronů ve skrytých vrstvách aby zkrácení času výpočtu převážilo vzniklé latence při přístupu do globální paměti. Hlavní nevýhodou tohoto přístupu je časová režie způsobená častým přepínáním mezi hostitelem a zařízením. Síť se třemi skrytými vrstvami výstupní vrstvou se pro výpočet výstupu jednoho vstupního vektoru, přepne čtyřikrát. Čtyřikrát také přepne při zpětné propagaci chyb a jednou pro aktualizaci vah. Celkově se přepne devět krát při zpracování jednoho vstupu. Vstupů může

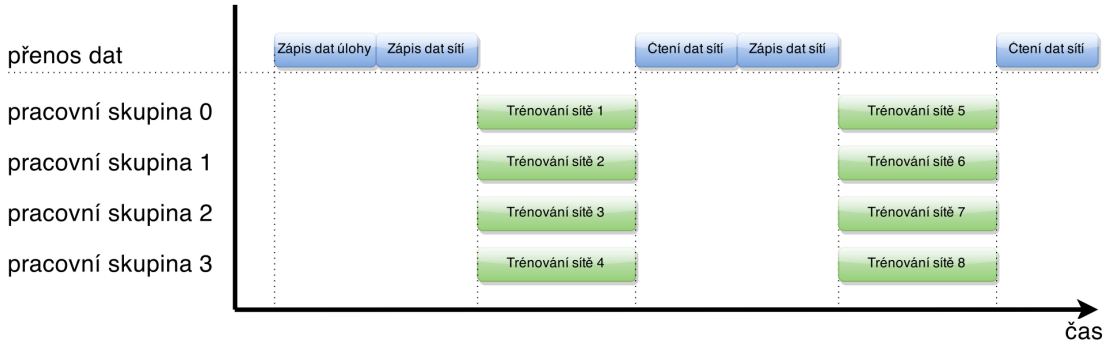
být řádově desetitisíce a toto trénování probíhá řádově desítky kol. Z toho vyplývá, že během trénování neuronové sítě může dojít k milionům přepnutí mezi hostem a zařízení. Jedno takové přepnutí trvá okolo 100 µs, což ve výsledku znamená stovky sekund strávených pouze přepínáním.

Mnou navržené řešení je provádět jednu neuronovou síť pouze v jedné pracovní skupině na jedné výpočetní jednotce. Všechny pracovní prvky v jedné pracovní skupině lze synchronizovat, proto není nutné přepínání mezi hostem a zařízením. Zrychlení se docílí pomocí běhu více neuronových sítí současně. Dokonce může běžet více neuronových sítí než je počet výpočetních jednotek. Toto pomůže se zakrytím latence při přístupu do globální paměti. Výpočet neuronů v jedné vrstvě je navržen tak, že každý neuron je prováděn jednou pracovní položkou. Podrobnější popis návrhu je v následující podkapitole.

5.2 Návrh algoritmu

Hlavním cílem návrhu je maximalizace využití výpočetních prostředků grafického čipu. V předchozí kapitole je uveden a odůvodněn směr návrhu. Grafická karta bude provádět běh několika neuronových sítí současně. Každá neuronová síť poběží ve vlastní pracovní skupině. Díky tomu běh jedné neuronové sítě bude trvat déle, protože nebude využívat celý potenciál grafického čipu, ale při běhu více neuronových sítí současně se dosáhne zrychlení.

Pro běh více neuronových sítí se provede pouze jedno spuštění kernelu. Na obrázku 5.1 je ukázáno trénování osmi neuronových sítí ve čtyřech pracovních skupinách. Nejprve se provede zápis společných trénovacích dat. Poté se přenesou nastavení neuronových sítí. Pracovní prvky dané pracovní skupiny si načtou nastavené konkrétní neuronové sítě a začnou provádět trénování nad společnými trénovacími daty. Po dokončení výpočtu se předá řízení zpět hostu a načtou se výsledky výpočtu. Následuje přenos nastavení dalších neuronových sítí. Trénovací data zůstávají pořád stejná a není je nutné znovu přenášet.

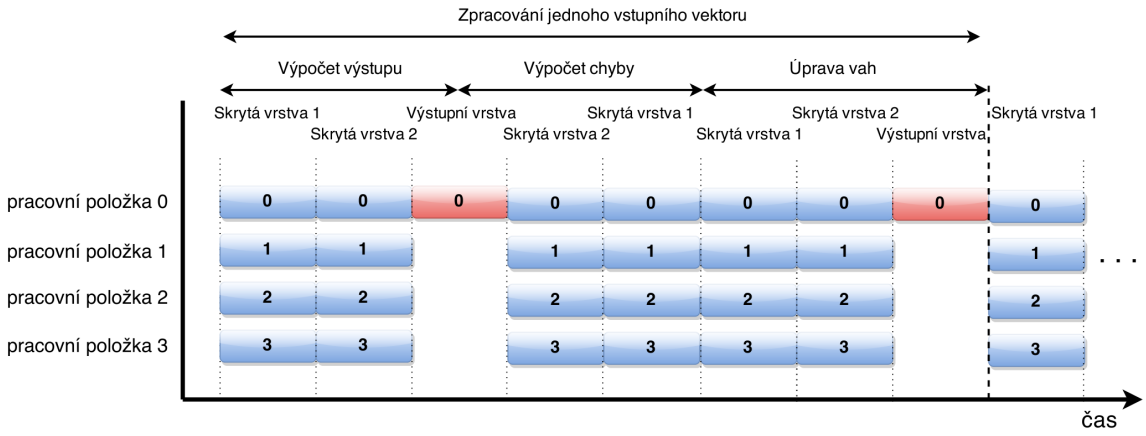


Obrázek 5.1: Paralelní provádění čtyř neuronových sítích současně.

Vstupním parametrem kernelu jsou trénovací data ve formě matice obsahující na každém řádku vstupní vektor neuronové sítě a matice s očekávanými výstupy. Dále je předán seznam nastavení sítí s popisem topologie pro všechny neuronové sítě, které mají být spuštěny. Každá pracovní skupina si podle svého ID vybere odpovídající nastavení sítě. Ke každému nastavení sítě je připojen seznam matic vah neuronů, kde každá matice popisuje váhy neuronů jedné vrstvy. Jelikož běh sítě probíhá v rámci jedné pracovní skupiny je možné maximálně využít lokální paměti, která je sdílená mezi pracovními položkami

v rámci pracovní skupiny. V lokální paměti budou uloženy všechny hodnoty neuronů kromě vah, protože množství vah může násobně přesáhnout velikost lokální paměti.

Pracovní jednotky pracovní skupiny provádí celý běh neuronové sítě. Tento běh je ilustrován na obrázku 5.2 na neuronové síti se dvěma skrytými vrstvami po čtyřech neuronech a jedním neuronem ve výstupní vrstvě. Kernel běží ve smyčce a provádí trénování pro každý vstupní vektor. Výpočty výstupů jednotlivých vrstev neuronové sítě se provádí sekvenčně, protože neurony každé vrstvy předávají výsledky neuronům vrstvy následující. Pro výpočet této neuronové sítě budou stačit čtyři pracovní položky. Ty nejdříve paralelně vypočtou výstup první skryté vrstvy. Poté přejdou do druhé skryté vrstvy a také provedou paralelní výpočet výstupů. Následuje výpočet ve výstupní vrstvě, po kterém se výstup porovná s požadovaným výstupem. Výsledkem porovnání je chyba neuronů výstupní vrstvy. Následuje procházení vrstvami zpět a paralelní výpočet chyb neuronů ve vrstvě na základě chyb neuronů vrstvy následující. Zjištěné chyby neuronů se poté aplikují na váhy. Váhy se mění od první vrstvy až k poslední výstupní vrstvě. Po úpravě všech vah je jeden trénovací cyklu u konce, následuje načtení nového vstupního vektoru a celý proces se opakuje dokud nedojdou vstupní vektory.



Obrázek 5.2: Jeden cyklus trénování neuronové sítě, která obsahuje dvě skryté vrstvy po čtyřech neuronech. Cyklus se skládá z výpočtu výstupů neuronů, zpětné propagace chyb a úpravy vah neuronů.

Kapitola 6

Implementace

Tato kapitola se věnuje navržené implementaci neuronových sítí. Samotná implementace se dělí na dvě části. První část se zabývá implementací v jazyce C umožňující efektivní běh na CPU. Druhá část obsahuje popis OpenCL implementace, která je vhodná pro běh na GPU. Dvě rozdílné implementace byly vytvořeny z těchto důvodů. U kódu napsaném v jazyce C lze lépe odladit a odstranit chyby. Dalším důvodem byla možnost porovnat výstupy s implementací v OpenCL. Implementaci v jazyce C je také možné využít pro porovnání rychlosti s implementací pro grafickou kartu. Pokud bychom porovnávali běh OpenCL implementace na GPU a CPU, pak by byl procesor znevýhodněn, protože se jedná o další technologickou vrstvu přinášející režii.

OpenCL řešení se skládá z kódu hosta a kódu zařízení. Kód hosta byl implementován v jazyce C++ a s OpenCL API komunikuje přes OpenCL C++ wrapper, což je rozšíření mapující standardní OpenCL rozhraní psané v jazyce C na třídy a objekty.

6.1 Implementace v jazyce C

V rámci práce byla vytvořena implementace neuronových sítí v programovacím jazyce C. Neuronová síť je v ní reprezentována pomocí hierarchie struktur obsahující informace o topologii, stavu sítě a dále pak podmínkách ukončení. Nejdůležitější částí struktur jsou ukazatelé na tři pole. Tyto pole slouží k uložení vah, hodnot a odchylek neuronů. Všechny tyto data mají maticový charakter, ale pro rychlejší přístup jsou uchována lineárně v poli takovým způsobem, aby při výpočtu mohly být sekvenčně čteny.

Algoritmus se skládá z cyklického provádění dvou fází. První fáze je trénování, kdy se na základě trénovacích dat upravují váhy neuronů. V druhé fázi probíhá testování natrénované sítě pomocí testovacích dat. Výstupem testování je kvadratická chyba klasifikace neuronovou sítí vypočtena podle následujícího vzorce 6.1

$$err = \frac{o_{max} - o_{min}}{N * P} * \sum_{p=1}^{P} \sum_{n=1}^{N} (o_{pn} - t_{pn})^2 \tag{6.1}$$

kde o_{max} a o_{min} je maximální a minimální hodnota výstupu, N je velikost výstupního vektoru, P je počet trénovacích vektorů, o jsou výstupy sítí a t jsou očekávané výstupy. Trénování a testování se cyklicky opakuje, řádově probíhají desítky opakování. Výsledkem běhu neuronové sítě je nejmenší kvadratická chyba, která byla dosažena a pořadí cyklu, kterém byla dosažena. S pomocí výsledné kvadratické chyby můžeme porovnat různé neuronové sítě mezi sebou.

Trénování neuronové sítě probíhá na základě zpracování sady vstupních vektorů z trénovacích dat. Pro každý vstupní vektor se nejprve postupně vypočtou výstupy všech neuronů. Hodnoty výstupních neuronů se porovnají očekávaným výstupem a vypočte se chyba výstupních neuronů. Tato chyba je od výstupní vrstvy šířena zpět, pomocí vah neuronů, až po první skrytou vrstvu. Poté, co je známa chyba pro každý neuron, se upraví váhy neuronů. Jakmile je zpracován vstupní vektor začne zpracování dalšího vstupního vektoru.

Testování sítě je výpočetně méně náročné. Skládá se z načtení vstupního vektoru z testovacích dat a jeho následného průchodu sítí. Očekávaný výstup klasifikace je pouze jeden aktivní výstupní neuron. Toto je implementováno jako výběr nejvíce vybuzeného neuronu z výstupní vrstvy.

6.2 OpenCL implementace

Exekuční model OpenCL se skládá z hostitelského kódu běžícím na procesoru a z kódu zařízení, který běží na grafické kartě. Hostitelský kód je psán v jazyce C++ za použití OpenCL C++ wrapperu. Jeho úkolem je nejprve načíst a připravit trénovací a testovací data, která budou zpracována neuronovými sítěmi. Druhá věc, o kterou se stará, je příprava sady neuronových sítí, které budou provedeny na grafické kartě. Připravená data a neuronové sítě odesílá ke zpracování. Kód zařízení provede trénování a testování neuronových sítí nad dodanými daty.

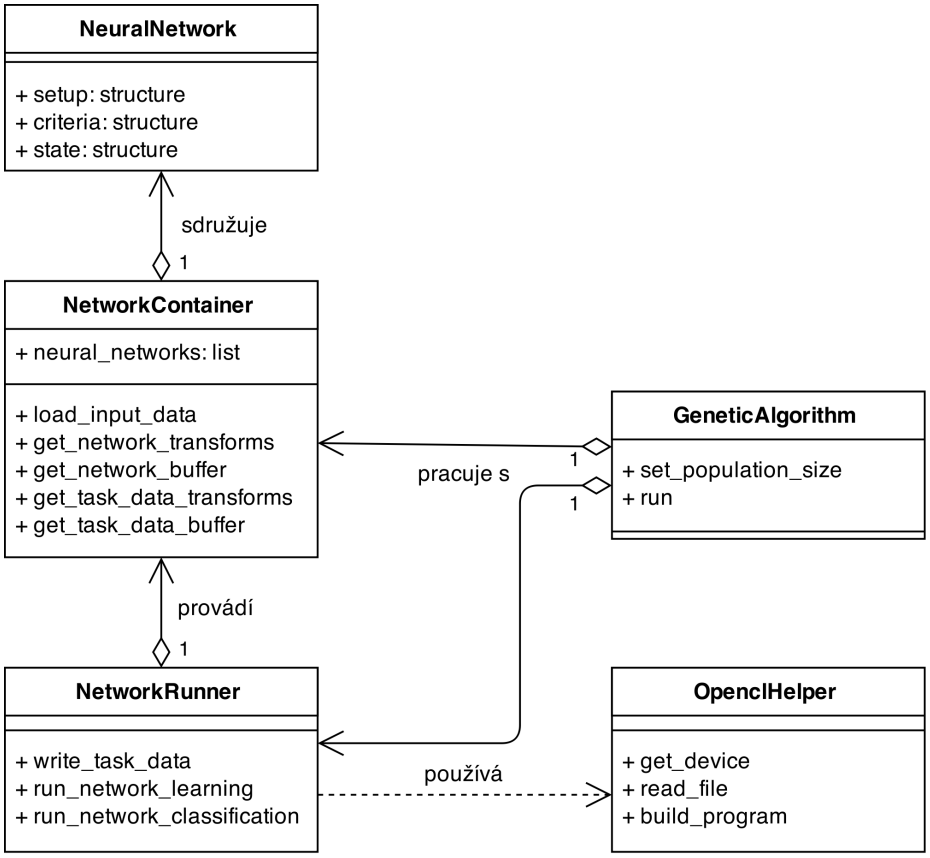
6.2.1 Hostitelská část

Diagram tříd na obrázku 6.1 zobrazuje objektový návrh implementace. Základními entitami jsou objekty třídy *NeuralNetwork*, které popisuje konfiguraci neuronové sítě. Před spuštěním jsou neuronové sítě sdruženy do kontejneru, který je implementován třídou *NetworkContainer*. Tento kontejner načítá data úlohy, alokuje souvislou paměť pro všechny neuronové sítě a vytváří transformační struktury. O kompilaci kernelu, posílání a čtení dat do OpenCL zařízení a o spuštění kernelu se stará třída *NeuralRunner*. Tato třída využívá C++ wrapper pro práci s OpenCL rozhraním. Pro zjednodušení práce s OpenCL byla vytvořená pomocná třída *OpenclHelper*, která umožňuje snadný výběr OpenCL zařízení, čtení souboru s kernelem a jeho kompilaci.

Nad rámec zadání práce byla vytvořena implementace genetického algoritmu ve třídě *GeneticAlgorithm*, která vytváří sadu různých neuronových sítí, kterou následně nechá trénovat. Natrénované sítě vyhodnotí a na základě nich vytvoří novou sadu sítí, kterou opětovně nechá provést. Podrobněji se této implementaci věnuje podkapitola 6.4 Genetický algoritmus.

6.2.2 Výměna dat

Protože při spouštění více neuronových sítí současně se provádí pouze jedna inicializace kernelu je potřeba dodat data sítím společně. Je potřeba předat informace o nastavení všech sítí, alokovat prostor pro váhy, hodnoty a chyby. Také je nutné předat trénovací a testovací data. OpenCL umožňuje kernelu předat pouze 8 parametrů. Z toho důvodu je nutné sdružit data všech sítí dohromady. OpenCL podporuje jednoduché datové typy, struktury a pole struktur jako parametr. Jediná podmínka je, aby předávaná hodnota zabírala v paměti souvislý prostor. Proto nelze předávat struktury, které obsahují ukazatele.



Obrázek 6.1: Diagram tříd popisující OpenCL impelentaci hostitelské části.

Implementace v jazyce C má neuronovou síť reprezentovanou strukturou s ukazateli do paměti, kde jsou uloženy váhy, hodnoty a chyby neuronů. V OpenCL nelze takovouto strukturu předat jako parametr kernelu. Řešením toho problému je alokace prostoru pro váhy, hodnoty a chyby do jednoho souvislého kusu paměti. Předávání dat hodnot a chyb sítí mezi kernelem a hostem je důležité pouze z hlediska ladění, pro samotný výpočet není důležitý. Proto v následujícím popisu budeme předpokládat, že se mezi hostem a kernelem přenáší pouze váhy neuronů. Aby bylo možné v kernelu zrekonstruovat stejnou strukturu jako je v C implementaci, je potřeba vytvořit transformační strukturu, která obsahuje nastavení sítě a offset do alokované paměti pro váhy. Stejný princip je použit pro přenesení trénovacích a testovacích dat úlohy. Data úlohy se však přenáší jen jednou a sdílí se mezi neuronovými sítěmi. Dále bude blíže popsáno pět parametrů, které se kernelů předávají.

Transformační struktury sítí

Pole transformačních struktur neuronových sítí. Počet struktur odpovídá počtu neuronových sítí. Pro každou síť je definováno nastavení a offset pro váhy do bufferu sítí. Každá pracovní skupina podle svého ID vybere právě jednu transformační strukturu.

Buffer sítí

Rozsáhlý buffer obsahující váhy všech sítí. Váhy jsou řazeny postupně takovým způsobem, aby první síť měla uloženy všechny váhy na začátku bufferu a další sítě je mají postupně uloženy až za ní.

Transformační struktura úkolu

Tato struktura je společná pro všechny sítě. Obsahuje informace o rozložení trénovacích a testovacích dat v bufferu úlohy. Nachází se v ní offsety pro vstupní a výstupní trénovací vektory, vstupní a výstupní testovací vektory a počty záznamů.

Buffer úlohy

Buffer obsahující data pro trénování a testování sítí. Jsou v něm vstupní a výstupní vektory pro obě fáze výpočtu. Tento buffer může být značně velký a je nutné pamatovat na omezení dané zařízením. Každé zařízení definuje maximální velikost objektu v paměti. Z důvodu dlouhé doby kopírování dat mezi pamětí hosta a pamětí zařízení, jsou data úlohy přenesena do zařízení pouze jednou a poté jsou používána při každém dalším volání kernelu.

Počet neuronových sítí

Udává kolik neuronových sítí je potřeba natrénovat. Jedna neuronová síť odpovídá jedné pracovní skupině a je potřeba kontrolovat jestli nebylo vytvořeno více pracovních skupin než je počet sítí.

6.2.3 Stavba kernelu

Všechny neuronové sítě běží ve stejném kernelu. Identifikace konkrétní neuronové sítě probíhá na základě ID pracovní skupiny. Kernel se skládá z několika základních částí. První částí je import dat. Jedná se o proces deserializace předaných transformačních struktur a bufferů. Výsledkem deserializace je struktura reprezentující neuronovou síť a struktura reprezentující řešený úkol. Tyto struktury mají podobu blízkou implementaci v jazyce C. V další části se provádí trénování nad dodanými trénovacími vektory. Po trénování následuje testování nad testovacími vektory. Výsledkem testování je kvadratická chyba klasifikace. Fáze trénování a testování se provádí několikrát, aby bylo možné vyhodnotit nejlepší dosaženou kvadratickou chybu pro neuronovou síť. V poslední části probíhá export dat zpět do transformačních struktur a bufferu. Podrobnějšímu popisu těchto částí se budou věnovat následující odstavce.

Počet neuronů v jedné skryté vrstvě nebo ve výstupní vrstvě je omezen na maximální velikost pracovní skupiny. Toto je dáno návrhem, kdy každý neuron ve vrstvě je zpracován právě jednou pracovní položkou. Maximální velikost pracovní skupiny se v současné době pohybuje okolo hodnoty 256, u historických karet může být hodnota pouze 128.

Import a export dat

Po spuštění kernelu proběhne identifikace pracovní skupiny a na základě ní se vybere transformační struktura sítě. Transformační struktura sítě slouží k sestavení struktury reprezentující síť. Tato transformační struktura obsahuje hodnoty nastavení jako jsou faktor učení, maximální počet cyklů, počet vrstev a hodnoty popisující aktuální stav jako jsou aktuální

cyklus, aktuální kvadratická chyba, index trénovacího nebo testovacího vektoru. Kromě hodnot sítě obsahuje také důležitý offset do společného bufferu obsahující váhy. Vedle struktury popisující síť existuje ještě struktura popisující řešenou klasifikační úlohu. I pro tuto strukturu existuje transformační struktura úlohy. Tato struktura obsahuje hodnoty jako jsou celkový počet trénovacích a testovacích vektorů. Mimo tyto hodnoty obsahuje také 4 offsety do bufferu s úlohou, který obsahuje vstupní trénovací data, očekávané výstupy trénovacích dat, vstupní testovací data a očekávané výstupy testovacích dat.

Na začátku běhu kernelu se provede import dat. Během tohoto importu se naplní struktura reprezentující síť a struktura reprezentující řešenou úlohu. Struktura reprezentující síť obsahuje kromě jednoduchý hodnot ukazatel do globální paměti, kde jsou uloženy váhy, a ukazatele do lokální paměti, kde jsou uloženy hodnoty a chyby neuronů. Důvod proč nejsou váhy uloženy v rychlé lokální paměti je malá velikost lokální paměti. Vah sítě může být v řádu desítek nebo stovek tisíc, do lokální paměti je vejde jen pár tisíc hodnot.

Export dat je opačný proces, kdy se ze struktury popisující síť naplní data do transformační struktury. Data vah sítě jsou uložena v globální paměti, proto není nutné žádným způsobem data upravovat.

Trénovací cyklus

Trénování neuronové sítě je výpočetně velmi náročný proces. Tento proces je založeno na opakování trénovacího cyklu nad trénovacími daty. Trénovací cyklus se skládá ze tří fází, které se musí provést sekvenčně za sebou, protože každá fáze využívá s výsledky té předešlé. Mezi tyto fáze patří dopředný výpočet výstupů neuronů, zpětný výpočet chyby neuronů a dopředná úprava vah neuronů.

V první fázi se pro každou vrstvu vypočítají výstupy neuronů. Výstupy neuronů se vypočítají pomocí bázové a aktivační funkce. Tyto výstupy závisí na vahách neuronů a na výstupech neuronů předchozí vrstvy. Jelikož výstup vrstvy závisí na výstupu předchozí vrstvy, nelze výstupy neuronů různých vrstev počítat paralelně. Paralelně se počítají pouze výstupy neuronů stejné vrstvy.

Jakmile jsou vypočítány výstupy výstupních neuronů, vypočte se jejich chyba na základě odchylky od očekávaného výstupu. Poté se postupuje vrstvami zpět a vypočítávají se chyby všech neuronů ve vrstvě na základě vah a chyb neuronů v následující vrstvě. Proto nelze paralelně vypočítat chyby neuronů v různých vrstvách, ale pouze ve stejné vrstvě.

V poslední fázi se upravují váhy neuronů na základě velikosti chyby neuronu a hodnoty výstupu odpovídajícího neuronu. Tady lze provádět úpravy vah v libovolné pořadí. Avšak s ohledem na uspořádání dat v paměti a zarovnání dat v paměti se tato úprava vah provádí po vrstvách. Vrstvami se postupuje od první skryté vrstvy k výstupní vrstvě. Váhy všech neuronů v jedné vrstvě se upravují současně.

Testovací cyklus

Oproti trénovacímu cyklu je testovací cyklus mnohem méně výpočetně náročný. Cyklus se skládá ze dvou fází. Mezi tyto fáze patří dopředný výpočet výstupů neuronů a vyhodnocení hodnot výstupních neuronů.

Výpočet výstupů neuronů je řešen stejně jako u trénování sítě. Výstupy neuronů se vypočítávají postupně přes vrstvy až k výstupní vrstvě. Výstupy neuronů ve stejné vrstvě se provádí paralelně.

Vyhodnocení hodnot výstupních neuronů se provádí dle zadané klasifikační úlohy. Výsledek klasifikace je určen nejvíce vybuzeným neuronem. Vyhledání nejvíce vybuzeného

neuronu provádí sekvenčně jeden pracovní prvek. Paralelizace za použití redukce zde nemá opodstatnění, většinou výstupních neuronů je do deseti a případná režie by byla příliš vysoká.

Klasifikace

Klasifikace dat neuronovou sítí probíhá v samostatném kernelu. Tento proces se liší od procesu trénování a testování. Hlavním rozdílem je práce pouze s jednou konfigurací neuronové sítě. Všechny pracovní skupiny provádí klasifikaci pomocí stejné neuronové sítě nad společnými daty úlohy. Pracovní skupiny si mezi sebou rovnoměrně rozdělí data klasifikační úlohy, které poté společně zpracují.

Průběh výpočtu kernelu je následující. Na začátku se importují datové struktury stejným způsobem jako u trénování. Všechny pracovní skupiny mají stejnou strukturu reprezentující neuronovou síť. Na základě ID pracovní skupiny si každá pracovní skupina určí část dat klasifikační úlohy, kterou bude řešit. Každý vstupní vektor je klasifikován pomocí neuronové sítě. Nejprve jsou vypočteny výstupy neuronů stejným způsobem jako u testování neuronové sítě. Poté se z výstupních neuronů vybere nejvíce vybuzený neuron, který reprezentuje třídu. Tímto postupem vznikne výstupní vektor, který se uloží do globální paměti. Výsledkem klasifikace je seznam výstupních vektorů, které jsou řešením zadané klasifikační úlohy.

6.3 Optimalizace

V rámci práce bylo potřeba optimalizovat řadu procesů. Ty nejvýznamnější optimalizační kroky jsou popsány v následujících sekcích.

6.3.1 Přerušování běhu kernelu

Ovladače grafické karty sledují chování karty. Při běhu kernelu nastává stav, kdy grafická karta provádí výpočty. Pokud výpočty trvají příliš dlouho (řády sekund), ovladače grafické karty výpočet násilně přeruší.

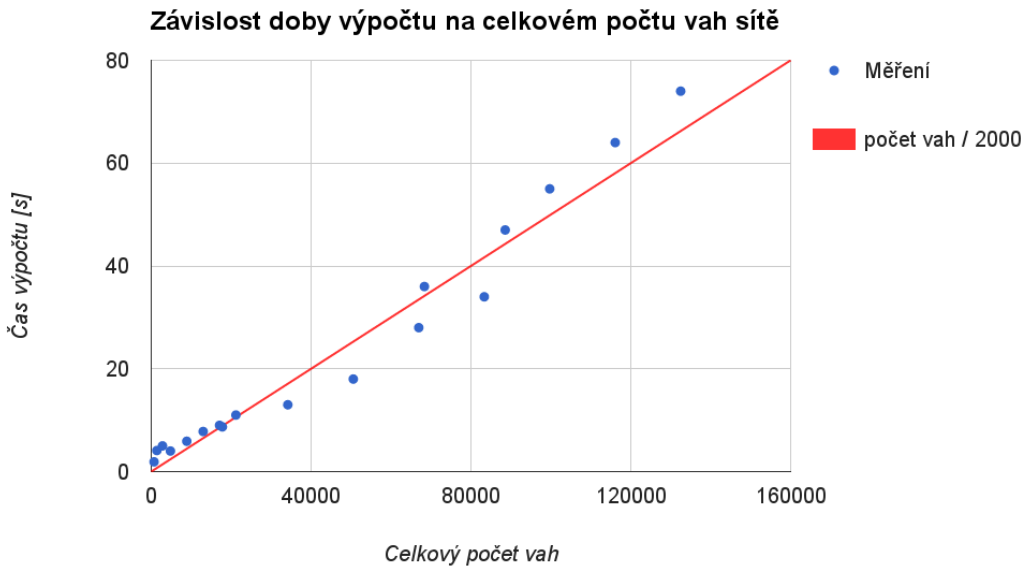
Trénování a testování dat probíhá v jednom běhu kernelu. U malých sítí tento výpočet trvá pouhé desítky milisekund. Oproti tomu velké sítě se stovkami neuronů v mnoha vrstvách se provádí řádově minuty. V kernelu byl implementován přerušovač, který po zpracování určitého množství vstupních vektorů přeruší výpočet, data uložena v lokální paměti nakopíruje do globální paměti a provede ukončení kernelu. Host po každém skončení provádění kernelu kontroluje jestli nebyl přerušen. Pokud byl kernel přerušen, vyvolá se znovu jeho zpracování. Kernel si při startu nakopíruje data z globální paměti do lokální, poté zjistí kde byl před přerušení a od toho místa pokračuje dál. Množství přerušení během trénování je závislé na topologii sítě a na počtu vstupních vektorů. U sítí s jednoduchou topologií a malým počtem vstupních vektorů nemusí dojít k žádnému přerušení.

Doba zpracování jednoho vstupního vektoru neuronovou sítí silně závisí na topologii sítě a na frekvenci grafické karty. Další faktory ovlivňující dobu zpracování jsou těžko předvídatelné. Mezi ně patří množství *wavefront* a topologiích ostatních neuronových sítí. Rozdíl ve frekvencích grafických karet je nízký, proto se pro výpočet hranice přerušení používá pouze informace o topologii. Nejlépe náročnost výpočtu určité topologie popisuje celkové množství vah neuronů. Empiricky byl vytvořen vzorec pro získání hranice přerušení na základě celkového množství vah neuronů.

Měřením rychlosti, opakovaného zpracování 450 vstupních vektorů v 20-ti kolech, byla nalezena lineární závislost mezi délkou výpočtu a počtem vah popsaná vzorcem 6.2. Měření jsou zachycena na obrázku 6.2 spolu s nalezenou aproximací. Tato měření byla prováděna při maximálním vytížení. Při nižším zatížení karty bude rychlost výpočtu vyšší, což bude mít za následek provádění přerušení v kratších časových intervale, ale režie s tím spojená je zanedbatelná. Vzorec pro výpočet hranice přerušení 6.3 vychází z celkového počtu zpracovaných vektorů $450 * 20 = 9000$, které by mělo zařízení zpracovat během jedné sekundy, a z odhadnuté doby výpočtu. Různé velikosti vstupních a výstupních vektorů se na vzorci projeví různým počtem vah neuronové sítě.

$$doba\_behu(pocet\_vah) = \frac{pocet\_vah}{2000} \tag{6.2}$$

$$limit\_preruseni(pocet\_vah) = \frac{9000}{\frac{pocet\_vah}{2000} + 1} + 1 \tag{6.3}$$



Obrázek 6.2: Závislost délky výpočtu celkovém počtu vah neuronové sítě.

6.3.2 Zarovnání dat v globální paměti

Data z globální paměti se načítají po zarovnaných částech paměti. V současné době je typická velikost načítaného bloku 64 nebo 128 bajtů. Tyto bloky se ukládají do cache L1 paměti, která má podobu tabulky. Velikost řádku odpovídá velikosti získané bloku. Tato koncepce paměti má za následek načítání 64 bajtů bloku paměti při čtení jedné *float* hodnoty, která má pouze 4 bajty. Při načítání další hodnoty, která se nachází ve stejném bloku jako hodnota předešlá, se provede čtení pouze z cache paměti.

S rozdělením paměti do bloků je potřeba počítat při procházení hodnot v cyklu. Například pokud chceme načíst 16 hodnot, které leží ve stejném bloku, stačí nám jedna operace fetch do globální paměti. V případě, že těchto 16 hodnot leží v oblasti přes 2 bloky bude zapotřebí dvou operací fetch.

Při implementaci neuronových sítí bylo co nejvíce dat uloženo v rychlé lokální paměti. Výjimku tvoří pole s hodnotami vah. Už středně velká neuronová síť se sto neurony může obsahovat více hodnot vah než je možné v lokální paměti uložit. Nejen, že váhy neuronů bylo nutné uložit do pomalé globální paměti, ale přistupuje se k nim velmi často, proto tvoří nejpomalejší místo implementace. Z tohoto důvodu bylo nutné optimalizovat způsob uložení dat.

Zarovnání vah v paměti je ilustrováno na obrázku 6.3. Každý neuron má pro své vstupy sadu vah. Všechny své váhy neuron používá při výpočtu výstupu, pak při výpočtu odchylky a při následné jejich úpravě. Tyto váhy jsou čteny paralelně pracovními prvky tak, že každý pracovní prvek načte první váhu neuronu a pracuje s ní. Cílem optimalizace bylo zarovnaní dat vah takovým způsobem, aby sada dat s první váhou všech neuronů začínala na začátku bloku a vyplnila násobek velikosti bloku. Totéž platí pro každou další sadu vah neuronů. Velikost paměti uchovávající váhy se kvůli přidanému zarovnání průměrně zvětšil o 9%, avšak následné **zrychlení bylo o 15%**.

Obrázek 6.3: Na obrázku je ilustrováno zarovnání globální paměti s 11 neurony, kdy každý neuron má 4 váhy. Šedě je vyznačena paměť využita pro zarovnání. Paměťová buňky mají velikost 4 bajty a jsou uspořádány do bloků po 16.

6.3.3 Sčítání paralelní redukcí

Váhy jsou v paměti uložena takových způsobem, aby výpočet výstupů neuronů ve vrstvě mohlo být vektorizováno. Takové uložení vypadá tak, že první váhy neuronů jsou uloženy za sebou, druhé a další váhy za nimi. Díky tomu je možné načíst blok paměti s prvními váhami a vektorově je zpracovat. Tento způsob uložení není ideální pro zpětný výpočet chyby neuronů. Tento výpočet probíhá po vrstvách tak, že pro každý neuron vrstvy se vypočte vážený součet chyb neuronů následující vrstvy pomocí odpovídajících vah. Velikost chyby n-tého neuronu vrstvy l se vypočte podle vzorce 6.4.

$$\Delta_n^l = \sum_{i=1}^{|l+1|} \Delta_i^{l+1} \cdot W_{i,n}^{l+1} \tag{6.4}$$

Pokud by se chyby neuronů ve vrstvě l vypočítávaly paralelně, docházelo by k načítání nového bloku z paměti pro každý neuron. Protože první neuron pracuje s prvními váhami

neuronů následující vrstvy, druhý neuron s druhými váhami a tak dále. Dokonce by docházelo k načtení bloku paměti pro každou váhu, protože cache nedokáže pojmout tak velké množství dat.

Tento problém je optimalizován paralelní výpočtem chyby jednoho neuronu. Všechny pracovní položky v jeden okamžik zpracovávají výpočet chyby právě jednoho neuronu n. Pracovní položky podle vzorce 6.5 nejprve provedou paralelně výpočet součinu váhy a chyb neuronů následující vrstvy a výsledek uloží do pomocného pole součinů alokovaného v rychlé lokální paměti.

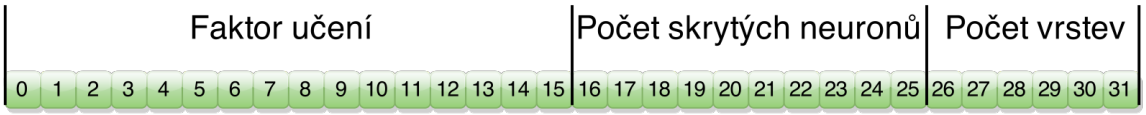
$$pole\_soucinu = \vec{\Delta}^{l+1} \cdot W_n^{l+1} \tag{6.5}$$

Vzniklé pole součinů se sečte pomocí paralelní redukce. Během redukce se v každém kole provede součet odpovídajících prvků mezi první polovinou pole a druhou polovinou pole. Výsledek se poté uloží do první poloviny pole. V dalším kole se postupuje stejně, jen se pracuje pouze s první polovinou pole, která se zase dělí. Výsledek součtu udává velikost chyby neuronu. Tento proces výpočtu chyby se zopakuje pro všechny neurony ve vrstvě. Časová složitost výpočtu je logaritmická, což v porovnání s původním výpočtem, který měl složitost lineární, přináší výrazné zrychlení. Měřením bylo zjištěno **zrychlení 10%**.

6.4 Genetický algoritmus

Nad rámec zadání práce byl implementován jeden z evolučních algoritmů. Celá implementace neuronových sítí směřovala k možnosti trénování velkého množství sítí současně. Tento přístup umožňuje vyzkoušet velké množství různých neuronových sítí. Tyto neuronové sítě se mohou lišit faktorem učení, počtem neuronů ve skryté vrstvě a počtem skrytých vrstev. Počet možných nastavení sítí je velké množství a nelze je vyzkoušet všechny. Z tohoto důvodu byl vybrán genetický algoritmus pro rychlé nalezení výhodného nastavení neuronové sítě pro zadaný úkol.

Základním stavebním kamenem genetického algoritmu je chromozom. Tento chromozom reprezentuje nastavení jedné neuronové sítě. Nastavení je binárně zakódováno v 32 bitovém čísle promocí Grayova kódu. Podrobnější popis struktury chromozomu je na obrázku 6.4. Chromozomy dohromady tvoří populaci, v které se mezi sebou kříží nejúspěšnější jedinci. Úspěšnost jedince je odvozena od kvadratické chyby, kterou dosahuje jím popisovaná neuronová síť.



Obrázek 6.4: Struktura chromozomu.

Chromozom ilustrovaný na obrázku 6.4 má podobu 32 bitového čísla, v kterém je na prvních 16 bitech zakódován faktor učení sítě. Jedná se o zakódované desetinné číslo nabývající hodnotu z intervalu 0 až 8,8192 s rozlišením 0,00012. Počet neuronů ve skryté vrstvě za uložen v následujících 10 bitech a může nabývat hodnot 1 až 1024. Poslední údaj o počtu skrytých vrstev je uložen na zbývajících 6 bitech a nabývá hodnot z rozsahu 0 až 63.

Popis výpočtu

Při inicializaci algoritmu se nejprve provede vytvoření populace chromozomů s náhodnými hodnotami. Na základě těchto hodnot se připraví sada neuronových sítí pro běh na grafické kartě. Po běhu všech sítí na grafické kartě se pro každý chromozom vyhodnotí úspěšnost nastavení podle kvadratické chyby sítě. Následně proběhne selekce, kde se ruletovým způsobem vyberou páry chromozomů a poté se provede jejich reprodukce. Tato reprodukce se skládá z nahodilého křížení chromozomů a mutace, čímž dojde ke změně nastavení sítě, kterou chromozom reprezentuje. Míra mutace byla nastavena na 5 procent. Průběh evoluce byl rozšířen o elitismus, při kterém jsou dva nejlepší jedinci předání do další generace aniž by u nich došlo ke křížení nebo mutací. Právě tato metoda zaručuje, zachování nejlepšího jedince v populaci. Po provedení evoluce nastane znovu fáze otestování nových neuronových sítí na grafické kartě. Tímto způsobem jsou provedeny desítky vhodné a nalezne se výhodného nastavení neuronové sítě pro zadaný problém.

6.5 Ovládání programu

Projekt je koncipován jako knihovna, kterou lze zakomponovat do většího celku. Pro jednodušší použití byl vytvořen program, v kterém lze pomocí parametrů nastavit trénování a klasifikaci neuronových sítí nebo měření výkonu zařízení. Při řešení úlohy klasifikace program využívá genetického algoritmu, který vybere vhodnou topologii a nastavení neuronové sítě pro zadanou úlohu. Poté se natrénuje a připraví síť podle nalezeného nastavení. Nakonec se provede klasifikace pro zadané data a výsledek se zapíše do výstupního souboru. Parametry pro nastavení OpenCL jsou následující:

-i
Výpis informací o všech OpenCL zařízeních, které jsou dostupné. Obsahují informace o platformě, počtu výpočetních jednotek, frekvenci jádra, velikost sdílené paměti a další informace. Tyto informace mohou být vodítkem pro vhodné nastavení výpočtu.

-p
OpenCL platforma, na které se bude provádět výpočet. Jedná se o celočíselnou hodnotu udávající index platformy. Vhodnou hodnotu lze zjistit z OpenCL výpisu pomocí přepínače **-i** .Výchozí hodnota je 0.

-d
OpenCL zařízení, na kterém se budou provádět výpočty. Jedná se o celočíselnou hodnotu udávající index zařízení v rámci platformy. Vhodnou hodnotu lze zjistit z OpenCL výpisu pomocí přepínače **-i**. Výchozí hodnota indexu zařízení je 0.

-c
Cesta k souboru obsahující zadání klasifikační úlohy. Formát souboru je následující. První tři čísla popisují velikost vstupního vektoru, výstupního vektoru a počet vektorů ke klasifikaci. Následují vektory určené ke klasifikaci.

-o
Cesta v výstupnímu souboru klasifikace. Každý řádek odpovídá jednomu výstupnímu vektoru klasifikace.

Pro úpravu nastavení genetického algoritmu je možné zadat tyto parametry:

-w
Počet neuronových sítí, které se provádějí na zařízení v jeden okamžik. Prováděním se rozumí proces trénování neuronových sítí i proces klasifikace s jejich pomocí. Na základě této hodnoty je odvozena velikost populace genetického algoritmu. Výchozí počet neuronových sítí je 256.

-g
Počet generací, které se provedou v rámci genetického algoritmu. Díky většímu počtu generací se může nelézt lepší konfigurace neuronové sítě. Doba výpočtu je lineárně závislá na počtu generací. Výchozí počet generací je 10.

-m
Minimální počet skrytých vrstev generovaných neuronových sítí. Genetický algoritmus generuje neuronové sítě s počtem skrytých vrstev v rozsahu 0 až 63. Výchozí chování neomezuje minimální počet skrytých vrstev.

-x
Maximální počet skrytých vrstev generovaných neuronových sítí. Počet skrytých vrstev neuronových sítí, které jsou generované s pomocí genetického algoritmu, může být v rozsahu 0 až 63. Omezení maximálního počtu skrytých vrstev přináší výrazné zrychlení výpočtu. Doba výpočtu je lineárně závislá na počtu skrytých vrstev neuronových sítí. Ve výchozím nastavení je maximální počet skrytých vrstev 4.

-t
Cesta k souboru, který obsahuje data úlohy pro trénování neuronových sítí. Formát souboru je následující. První čtyři čísla v souboru popisují velikost vstupního vektoru, velikost výstupního vektoru, počet trénovacích vektorů a počet testovacích vektorů. Na každém řádku postupně následují trénovací vektory a po nich testovací vektory. Trénovací i testovací vektor se skládá ze vstupního vektoru a výstupního vektoru.

Neuronovým sítím lze nastavit tyto parametry:

-n
Maximální počet neuronů v jedné skryté vrstvě. U testování výkonu udává přesný počet neuronů v každé skryté vrstvě. Tato hodnota silně ovlivňuje celkový počet vah a tady i dobu výpočtu. S rostoucím počtem neuronů ve vrstvách se kvadraticky prodlužuje doba výpočtu. Výchozí hodnota je 256.

-e
Počet kol trénování neuronové sítě. Po dokončení každého kola se provede testování a uložení kvadratické chyby. Se zvyšujícím se počtem kol trénování se zpřesňuje klasifikace. Je třeba být opatrný, protože pro příliš vysoký počet kol dojde k přetrénování a další výpočet nic nepřináší.

Program umožňuje otestovat výkon zařízení. Pomocí výše zmíněných parametrů lze zadat počet **-w** počet sítí, **-n** počet neuronů a **-e** počet kol trénování. Pro testování je potřeba zadat navíc tyto parametry:

-b
Přepínač zapínající testování výkonu zadaného zařízení. Měří čas provádění výpočtu, který lze porovnat s jinými zařízeními. Jelikož je výpočet optimalizován pro GPU zařízení, je vhodné porovnávat výsledek testu pouze s GPU zařízeními. Porovnání s CPU zařízením tímto způsobem není vhodné.

-l
Počet skrytých vrstev testovaných neuronových sítí. Ve výchozím nastavení je počet skrytých vrstev 3.

6.6 Testování

V rámci projektu byla implementována sada integračních testů ověřující správnou funkčnost OpenCL implementace. Integrační testy porovnávají výstup procesu trénování a klasifikace OpenCL implementace s implementací v jazyce C. Tolerovaný rozdíl výsledků je 0.01%. Ten může nastat při odlišném provedení operací s desetinnými na procesoru a grafické kartě.

Testy se zaměřují na porovnání výsledku trénování a klasifikace neuronových sítí při různém nastavení faktoru učení, různých počtech skrytých vrstev a různém počtu neuronů. Testy také kontrolují hromadné provádění kontejneru neuronových sítí na grafické kartě, kde mají sítě odlišné nastavení.

Kapitola 7

Výsledky

V této kapitole jsou popsány měření rychlosti vytvořené implementace trénování neuronových sítí a klasifikace pomocí nich. Cílů měření bylo několik. Prvním cílem bylo porovnání rychlosti implementace v jazyce C běžící na procesoru i7 920 s OpenCL implementací běžící na třech různých grafických kartách. Při měření byla použita jedna AMD grafická karta HD 7970 a dvě grafické karty NVidia GTX 580 a GTX980. Porovnání parametrů grafických karet je v tabulce 3.1.

Použitý procesor Intel i7 920 má 4 jádra s frekvencí 2,66 GHz. Tento procesor obsahuje technologii Hyper-threading, která umožňuje efektivní běh 8 procesů současně. Měření rychlosti výpočtů na tomto procesoru probíhalo při spuštění 8 procesů, kdy se v jeden okamžik trénovalo 8 neuronových sítí současně. Kód implementace v jazyce C byl kompilován pomocí kompilátoru *icpc*, který umožňuje dobře optimalizovat kód pro použitý procesor společnosti Intel. Výsledný program je tři krát rychlejší než kdyby byl kompilován pomocí standardního g++ kompilátoru.

Měření výkonu implementace neuronových sítí bylo prováděno nad datovou kolekcí *cancer* projektu PROBEN [16]. Tato kolekce nabízí pro trénování 700 vstupních vektorů. Pro měření výkonu klasifikace byla vytvořena datová sada se 100 000 vstupními vektory. Vstupní vektory této kolekce obsahují 9 hodnot a výstupní vektory obsahují puze dvě hodnoty. Následující podkapitoly se detailněji věnují měření výkonu trénování a klasifikace s pomocí neuronových sítí.

7.1 Měření výkonu trénování

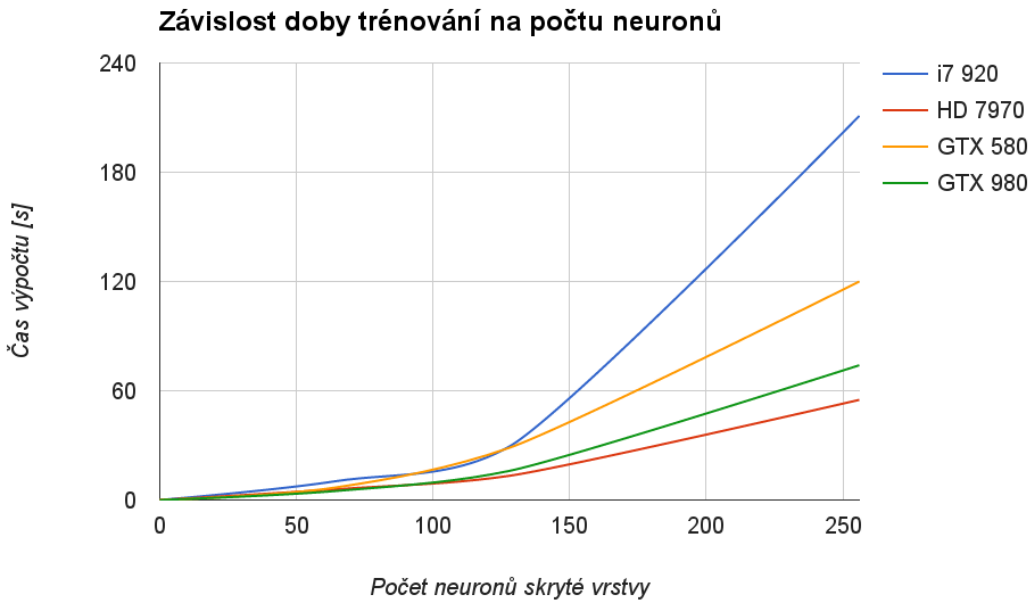
Tato podkapitola se zaměřuje na měření výkonu trénování neuronových sítí. Zkoumá dobu provádění výpočtu v závislosti na mnoha faktorech. Hlavními faktory, pro které bylo provedeno měření, jsou počty neuronů ve skrytých vrstvách, počet skrytých vrstev a počty pracovních skupin.

Měření rychlosti trénování neuronových sítí bylo prováděno nad datovou kolekcí *cancer* projektu PROBEN [16]. Tato kolekce obsahuje se 350 trénovacích vektorů a 350 testovacích vektorů. Trénování a testování každé sítě bylo opakováno během 20 kol.

7.1.1 Závislost na počtu neuronů

Měření závislosti doby trénování neuronových sítí na počtu neuronů ve skrytých vrstvách je zachyceno na obrázku 7.1. Toto měření ukazuje nelineární časovou složitost, která je

důsledkem fungování neuronových sítí s učením typu backpropagation. Při výpočtu výstupu každého neuronu vrstvy provádíme sumu součinů všech neuronů předchozí vrstvy a příslušných vah. Pokud počet neuronů zdvojnásobíme, počet násobení se zečtyřnásobí. Trénování na všech zařízení mělo stejný cíl - natrénovat 256 neuronových sítí se třemi skrytými vrstvami. Měření času trénování bylo provedeno pro 64, 128 a 256 neuronů v každé skryté vrstvě. Z měření, kromě nelineární závislost doby trénování na počtu neuronů, také vyplývá, že u malých sítí s počtem skrytých neuronů 64 a 128 je zrychlení pouze dvojnásobné oproti běžnému procesoru. U větších sítí s 256 skrytými neurony je zrychlení oproti běžnému procesoru až čtyřnásobné.



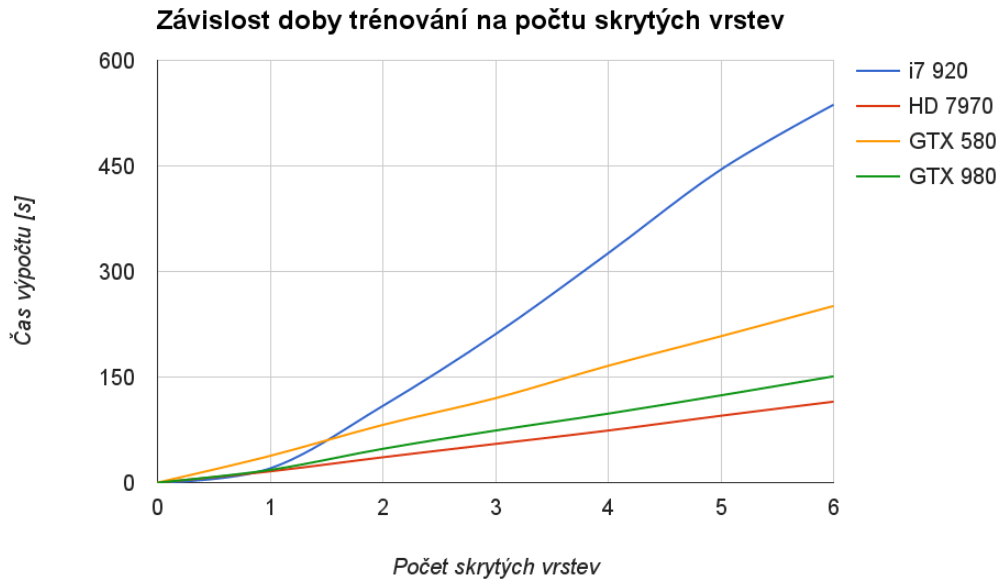
Obrázek 7.1: Závislost doby trénování na počtu neuronů ve skrytých vrstvách neuronových sítí. Měřeno na neuronových sítích se třemi skrytými vrstvami.

U karty GTX 980 lze pří nastavení 256 neuronů pozorovat zpomalení oproti kartě HD 7970, přitom tato karta má podle technické specifikace vyšší výkon. Toto zpomalení souvisí s velikostí lokální pamětí. Karta GTX 980 má 16 výpočetních jednotek s 98 kB lokální paměti. Dohromady tedy 1568 kB lokální paměti. Oproti tomu karta HD 7970 má 32 výpočetních jednotek po 64 kB lokální paměti a tedy celkově 2048 kB paměti. Použitá neuronová síť se 128 neurony ve vrstvě potřebuje celkově okolo 4,2 kB lokální paměti. Po zvýšení počtu neuronů na 256 síť potřebuje okolo 7,2 kB lokální paměti. U karty HD 7970 z hlediska velikosti lokální paměti může běžet na jedné výpočetní jednotce maximálně $64/7,2 = 8$ neuronových sítí současně. Všechny výpočetní jednotky umožňují dohromady běh $32 * 8 = 256$ sítím současně. Oproti tomu u karty GTX 980 může běžet na jedné výpočetní jednotce maximálně $98/7,2 = 13$ neuronových sítí současně. Všechny výpočetní jednotky, kterých je méně, umožňují běh $16 * 13 = 208$ sítí současně. To znamená, že se na kartě provede nejprve trénování prvních 208 sítí a poté trénování zbylých 48 sítí.

7.1.2 Závislost na počtu vrstev

Časová závislost provádění neuronových sítí na počtu skrytých vrstev je lineární. Tato lineární závislost je zachycena na obrázku 7.2. Lineární časová závislost je dána tím, že dobu provádění jedné skryté vrstvy můžeme brát jako konstantu a celková doba výpočtu je jen násobení této konstanty.

Výpočty na všech zařízení měly stejný cíl - provést trénování 256 neuronových sítí, jejichž každá skrytá vrstva obsahuje 256 neuronů. Měření probíhalo pro 1 až 6 skrytých vrstev sítí. Z měření lze pozorovat, že pro sítě s více než jednou skrytou vrstvou je zrychlení až čtyřnásobné.



Obrázek 7.2: Závislost doby trénování na počtu skrytých vrstev neuronových sítí. Měřeno pro 256 neuronů v každé skryté vrstvě.

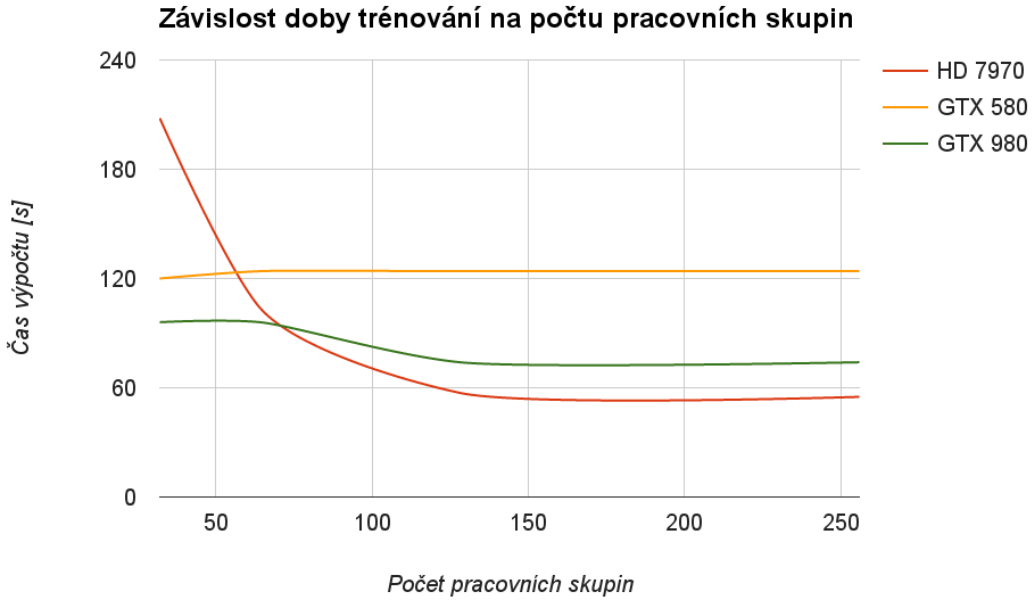
7.1.3 Závislost na počtu pracovních skupin

Měření v této části se zaměřuje na pozorování vlivu počtu pracovních skupin na celkovou dobu trénování sítí. Naměřené hodnoty jsou znázorněny na obrázku 7.3.

Toto měření probíhalo pouze na grafických kartách, protože se jedná o nastavení OpenCL. Všechny karty měly za úkol provést trénování 256 neuronových sítí se třemi skrytými vrstvami po 256 neuronech. Jedna pracovní skupina provádí trénování jedné neuronové sítě. Při trénování bylo použito 32, 64, 128 a 256 pracovních skupin. Čím méně bylo puštěno pracovních skupin, tím pro plánovač v grafické kartě bylo složitější skrýt latence do globální paměti. Trénování 256 neuronových sítí, za pomocí 256 pracovních skupin, bylo možné provést během jednoho běhu. Oproti tomu trénování pomocí 32 pracovních skupin muselo být voláno osmkrát aby se provedlo trénování všech 256 sítí.

Z měření vyplývá, že u grafických karet společnosti NVidia má počet pracovních skupin malý vliv na rychlost výpočtu. Oproti tomu u grafické karty AMD, při použití pouze 32

pracovních skupin, trvá výpočet výrazně déle než při použití 128 pracovních skupin. Toto je dáno tím, že na grafické kartě HD 7970 je celkem 32 výpočetních jednotek. Pokud se na každé jednotce trénuje právě jedna síť, plánovač nemá možnost, při čekání na čtení a zápis do globální paměti, přepnout na trénování jiné sítě. U použitých NVidia karet je počet výpočetních jednotek 16, proto se latence daří lépe zakrýt.



Obrázek 7.3: Závislost doby trénování na počtu pracovních skupin. Měřeno na neuronových sítích se třemi skrytými vrstvami po 256 neuronech.

7.2 Měření výkonu klasifikace

Tato podkapitola se zaměřuje na analýzu výkonu klasifikace neuronovými sítěmi. Oproti trénování je klasifikace výrazně jednodušší výpočetní proces. Měření rychlosti bylo prováděno nad datovou kolekcí *cancer* projektu PROBEN [16]. Pro zátěžové testy byla použita datová sada se 100 000 vstupními vektory, pro které se pomocí klasifikace neuronovými sítěmi vytvořila sada výstupních vektorů.

Klasifikaci předcházelo trénování neuronové sítě, které probíhalo 20 kol. Pro provádění klasifikace se vybrala nejlépe natrénovaná varianta sítě. Měřený čas klasifikace zahrnuje pouze dobu samotné klasifikace bez trénování sítí.

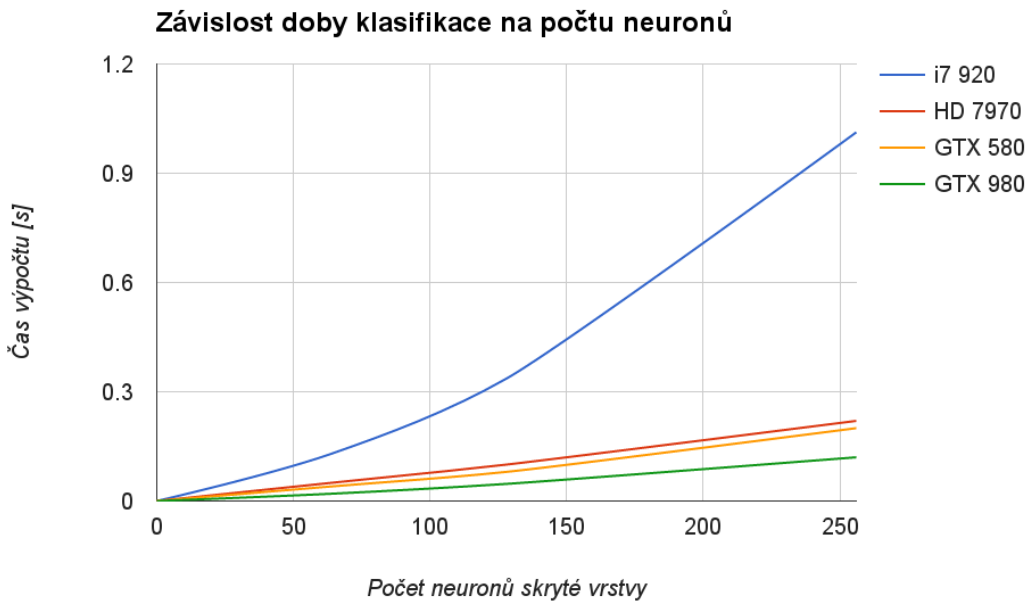
V další části této podkapitoly budou blíže rozebrány jednotlivé druhy měření. Mezi zkoumaní měření patří analýza závislosti doby výpočtu na počtu neuronů ve skrytých vrstvách, na počtu skrytých vrstev a na počtu pracovních skupin.

7.2.1 Závislost na počtu neuronů

Měření závislosti doby provádění klasifikace na počtu neuronů ve skrytých vrstvách je zachyceno na obrázku 7.4. K měření byla použita neuronová síť s třemi skrytými vrstvami a klasifikace byla prováděna 128 pracovními skupinami, protože při tomto počtu pracovních skupin je rychlost klasifikace nejvyšší. Počty neuronů ve skrytých vrstvách sítí byly 64, 128 a 256.

Podobně, jako u stejného měření při trénování neuronové sítě, je závislost doby výpočtu na počtu neuronů nelineární. Díky velmi krátké době výpočtu je nelineárnost velmi málo zřetelná. Z měření je patrné, že karty NVidia provádí klasifikaci výrazně rychleji, než karta AMD. Nejrychlejší grafická karta GTX 980 prováděla klasifikaci devět krát rychleji než procesor i7. Tato karta dokáže u zadané úlohy zpracovat necelý milión vstupních vektorů za sekundu.

Karty při klasifikaci podávají odlišný výkon oproti trénování. Při trénování byla karta HD 7970 výrazně rychlejší oproti ostatním kartám. Při klasifikaci je zase nejpomalejší z karet. Tento rozdíl je způsoben několika faktory. Prvním z nich je lepší vektorizovatelnost procesu trénování oproti procesu klasifikace. Proces klasifikace obsahuje větší množství operací prováděných na skalární jednotce. Architektura karet NVidia se od AMD v mnohém liší. NVidia má vyšší frekvencí jádra a vyšším počtem skalárních jednotek na jedné výpočetní jednotce. Tyto vlastnosti karet se projeví právě při procesu klasifikace.

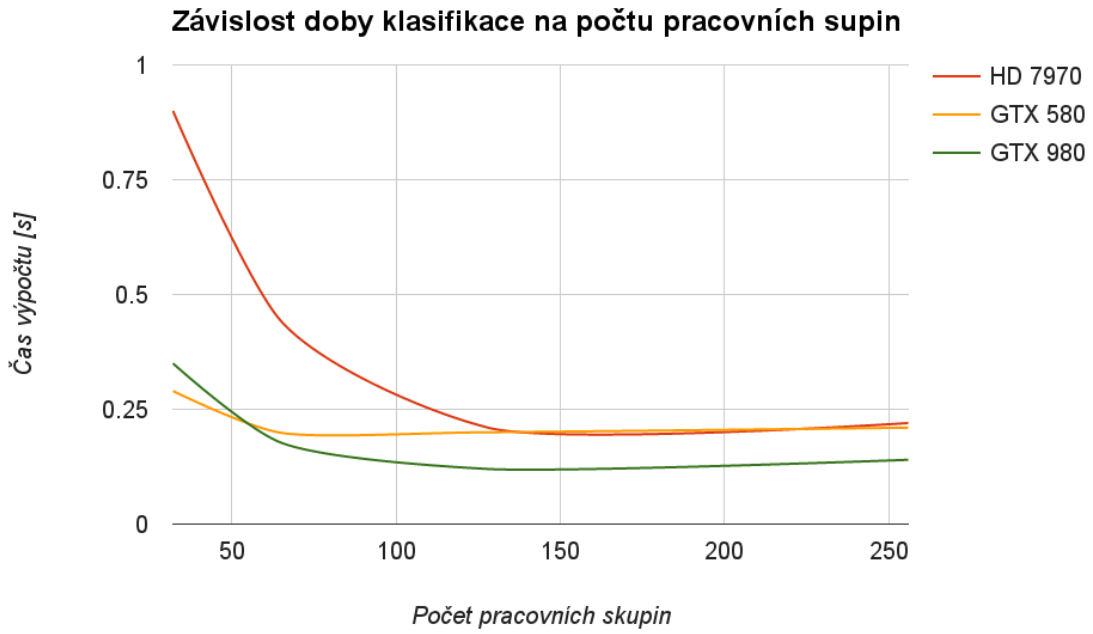


Obrázek 7.4: Závislost doby klasifikace na počtu neuronů ve skrytých vrstvách. Měřeno na neuronových sítích se třemi skrytými vrstvami.

7.2.2 Závislost na počtu vrstev

Měření zachycené na obrázku 7.5 popisuje závislost času provádění klasifikace na počtu skrytých vrstev. Stejně jako u trénování je doba klasifikace lineárně závislá na počtu skrytých vrstev neuronové sítě.

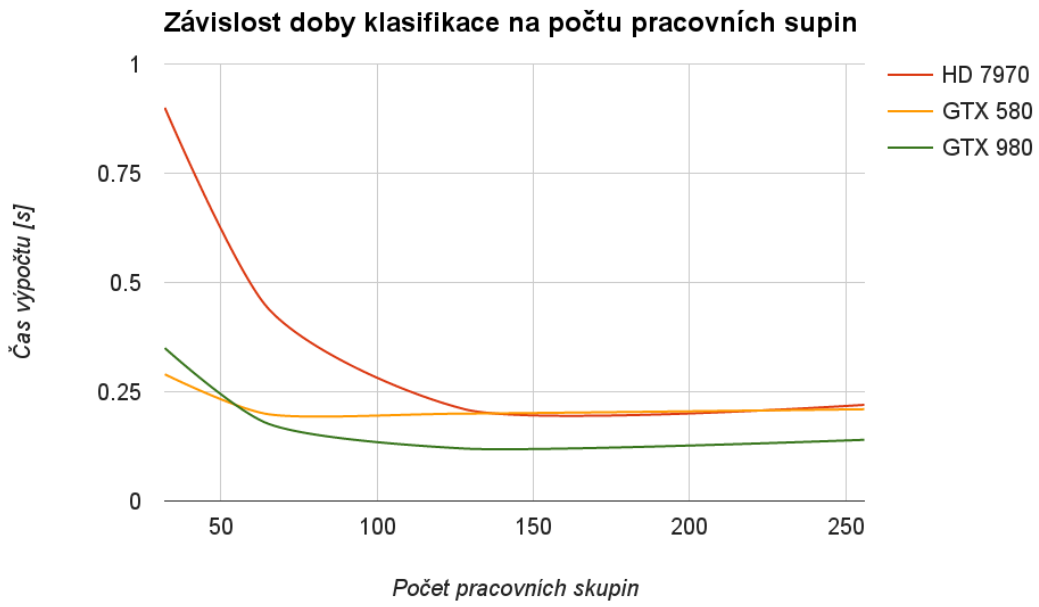
Měření probíhalo na neuronových sítích s 256 neurony v každé skryté vrstvě. Počet skrytých vrstev byl volen z rozsahu 1 až 6. Stejně jak tomu bylo při měření trénování, nalezená závislost je lineární. Výkon grafických karet GTX 580 a HD 7970 je při klasifikaci téměř totožný. Doba výpočtů na kartě GTX 980 je poloviční oproti ostatním kartám.



Obrázek 7.5: Závislost doby klasifikace na počtu skrytých vrstev. Měřeno pro 256 neuronů ve skryté vrstvě.

7.2.3 Závislost na počtu pracovních skupin

Výsledky měření závislosti času provádění klasifikace na počtu pracovních skupin, které jsou zachycené na obrázku 7.6, se mírně liší od výsledku měřeného při trénování. Měření probíhalo na neuronových sítí se třemi skrytými vrstvami po 256 neuronech. Počty pracovních skupin byly 32, 64, 128 a 256. Hlavní rozdíl oproti trénování je u 32 pracovních skupin, kde i karty NVidia podávají nižší výkon. Toto je způsobeno tím, že klasifikace obsahuje méně aritmetických operací a více čtecích operací z globální paměti. Proto je vhodné zvolit větší počet pracovních skupin aby se zakryly latence při práci s globální pamětí.



Obrázek 7.6: Závislost doby klasifikace na počtu pracovních skupin. Měřeno na neuronových sítích se třemi skrytými vrstvami po 256 neuronech.

7.3 Experimenty

Implementace byla procházena pomocí ladícího nástroje CodeXL [4]. Tento nástroj je určen pro analýzu kernelů běžících na kartách společnosti AMD. Nabízí statickou a dynamickou analýzu kódu kernelů. U některých karet nabízí možnost krokového ladění kódu.

Na základě měření promocí programu CodeXL bylo potvrzeno nulové množství konfliktů banků lokální paměti u neuronových sítí s počtem neuronů ve vrstvě větším než 64. To je dáno tím, že konflikty mohou nastat pouze v rámci jedné výpočetní jednotky. Síť s počtem neuronů ve vrstvě větším než 64 zabere ve *wavefront* všechny čtyři SIMD jednotky. Data neuronů jsou v lokální paměti uložena v pořadí v jakém sousedí neurony ve vrstvě. Z toho vyplývá vlastnost, kdy při paralelním čtení hodnot pro 64 neuronů se čte právě 64 sousedících hodnot z lokální paměti. Nutno poznamenat, že 64 hodnot se přes 32 paměťových banků načte během 2 cyklů, ale toto není považováno za konflikt.

Kapitola 8

Závěr

V rámci diplomové práce jsem se seznámil s teorií umělých neuronových sítí s podrobnějším zaměřením na neuronové sítě s učením typu backpropagation. Dále jsem nastudoval principy provádění výpočtů na grafických kartách s využitím technologii OpenCL.

Práce obsahuje návrh akcelerace neuronových sítí na grafickém čipu s pomocí technologie OpenCL. Tento návrh je orientován na paralelní běh více neuronových sítí, kde každá neuronová síť běží na jiné výpočetní jednotce. Díky tomu je možné maximálně využít výpočetní zdroje zařízení. Toto řešení umožňuje běh několika stejných neuronových sítí s různými parametry. U každé neuronové sítě může být nastaven jiný faktor učení, počet kol trénování nebo dokonce odlišná topologie sítě. Topologie sítí může být tvořena velkým množstvím skrytých vrstev a proměnným počtem neuronů.

Práce byla testována a odladěna na datových sadách PROBEN [16]. Tyto sady obsahují celou řadu klasifikačních úloh. Mezi tyto úlohy patří například předpovězení výskytu rakoviny nebo detekce přechodu intron/exon v krátké DNA sekvenci.

Výsledkem práce je OpenCL implementace trénování neuronových sítí. Akcelerace je docíleno trénováním několika různých neuronových sítí současně. Měřením výkonu bylo u grafických karet zjištěno téměř čtyřnásobné zrychlení oproti běhu na klasickém procesoru.

Součástí práce je i OpenCL implementace klasifikace pomocí neuronových sítí. Akcelerace klasifikace je docíleno rozdělením vstupních dat mezi výpočetní jednoty grafické karty. Naměřené zrychlení u grafických karet bylo až pětinásobné oproti běžnému procesoru. Výsledná rychlost klasifikace pro datovou sadu *cancer* byla milion vstupních vektorů za sekundu.

Nad rámec zadání práce byl implementován genetický algoritmus, který umožňuje nalézt výhodného nastavení a topologii neuronové sítě pro zadanou úlohu. Tento genetický algoritmus využívá akcelerované trénování neuronových sítí, kdy provede trénování stovek neuronových sítí s různým nastavením a topologií. Po provedení trénování několika generací se vybere neuronová síť s nejlepší přesností a s touto neuronovou sítí se provede akcelerovaná klasifikace dat úlohy.

Výpočty na GPU se neomezují pouze na technologii CUDA a grafické karty společnosti NVidia. Technologie OpenCL nabízí možnost využití karet různých výrobců. Při testování implementace byla použita karta AMD, která prováděla trénování neuronových sítí výrazně rychleji, než stejně drahá karta společnosti NVidia. Díky technologii OpenCL jsou výpočty na GPU cenově dostupnější.

Literatura

[1] AMD, Inc.: Introduction to OpenCL Programming. In: *AMD Developer Central* [online]. 2010 [cit. 2015-05-19], dostupné z: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/01/Introduction_to_OpenCL_Programming-Training_Guide-201005.pdf.

[2] AMD, Inc.: AMD Graphic Core Next architecture. In: *AMD whitepapers* [online]. 2012 [cit. 2015-05-19], dostupné z: https://www.amd.com/Documents/GCN_Architecture_whitepaper.pdf.

[3] AMD, Inc.: AMD Accelerated Parallel Processing: OpenCL Programming Guide. In: *AMD Developer Central* [online]. 2013 [cit. 2015-05-19], dostupné z: http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf.

[4] AMD, Inc.: AMD CodeXL Quick Start Guide. In: *AMD Developer Central* [online]. 2014 [cit. 2015-05-19], dostupné z: http://developer.amd.com/wordpress/media/2013/12/CodeXL_Quick_Start_Guide.pdf.

[5] FRASER, N.: The Biological Neuron. In: *Neil's Neural Nets* [online]. 1997 [cit. 2015-05-19], dostupné z: http://vv.carleton.ca/ neil/neural/neuron-a.html.

[6] GASTER, B. R.: The OpenCL C++ Wrapper API. In: *The Khronos Group Inc.* [online]. 2010 [cit. 2015-05-19], dostupné z: https://www.khronos.org/registry/cl/specs/opencl-cplusplus-1.1.pdf.

[7] GASTER, B. R. a kol.: *Heterogeneous computing with OpenCL.* Waltham, MA: Morgan Kaufmann, druhé vydání, 2013, ISBN 0-12-405894-1.

[8] HAYKINS, S.: *Neural Networks and Learning Machines.* Upper Saddle River, N.J.: Prentice Hall, třetí vydání, 2008, ISBN 01-312-9376-1.

[9] KYOUNG-SU, O., KEECHUL, J.: GPU implementation of neural networks. In: *School of Media, College of Information Science, Soongsil University* [online]. Soongsil University, 2004 [cit. 2015-05-19], dostupné z: http://www.researchgate.net/publication/222114533_GPU_implementation_of_neural_networks.

[10] MANTOR, M, HUSTON, M: AMD Graphic Core Next. In: *AMD Developer Central* [online]. 2011 [cit. 2015-05-19], dostupné z: http://developer.amd.com/wordpress/media/2013/06/2620_final.pdf.

[11] MEHROTRA, K., MOHAN, C. K., RANKA, S.: *Elements of artificial neural networks.* Cambridge, MA.: MIT Press, první vydání, 1996, ISBN 0-262-13328-8.

[12] MUNSHI, A.: The OpenCL Specification. In: *The Khronos Group Inc.* [online]. 2011 [cit. 2015-05-19], dostupné z: https://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

[13] NVIDIA Corporation: Fermi Architecture. In: *NVIDIA whitepapers* [online]. 2009 [cit. 2015-05-19], dostupné z: http://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf.

[14] NVIDIA Corporation: NVIDIA OpenCL Best Practices Guide Version 1.0. In: *NVIDIA Optimization* [online]. 2009 [cit. 2015-05-19], dostupné z: http://www.nvidia.com/content/cudazone/cudabrowser/downloads/papers/nvidia_opencl_bestpracticesguide.pdf.

[15] NVIDIA Corporation: Nvidia cuda zone. In: *NVIDIA developer site* [online]. 2015 [cit. 2015-05-19], dostupné z: https://developer.nvidia.com/cuda-zone.

[16] PRECHELT, L.: Proben1 - A set of neural network benchmark problems and benchmarking rules. In: *Fakultät für Informatik, Universität Karlsruhe* [online]. 1994 [cit. 2015-05-19], dostupné z: http://page.mi.fu-berlin.de/prechelt/Biblio/1994-21.pdf.

[17] SCARPINO, M.: *OpenCL in action: how to accelerate graphics and computation.* Shelter Island: Manning, první vydání, 2012, ISBN 978-1-617290-17-6.

[18] SIERRA-CANTO, X., MADERA-RAMIREZ, F., UC-CETINA, V.: Parallel Training of a Back-Propagation Neural Network Using CUDA. In: *Ninth International Conference on Machine Learning and Applications* [online]. IEEE Computer Society, 2010 [cit. 2015-05-19], dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5708849.

[19] VRTANOSKI, J., STOJANOVSKI, T. D.: Pattern recognition with OpenCL heterogeneous platform. In: *20th Telecommunications Forum (TELFOR)* [online]. IEEE Computer Society, 2012 [cit. 2015-05-19], dostupné z: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6419306.

[20] ZBOŘIL, F. V.: *Biologický a umělý neuron, umělé neuronové sítě. Perceptron a Adaline.* Přednáška předmětu Soft computing. Fakulta informačních technologií, Vysoké učení technické v Brně, 2014.

[21] ZBOŘIL, F. V.: *Neuronové sítě Madaline a BP (Back Propagation). Neuronové sítě s proměnnou topologií.* Přednáška předmětu Soft computing. Fakulta informačních technologií, Vysoké učení technické v Brně, 2014.

[22] ZBOŘIL, F. V.: *Neuronové sítě RBF a RCE. Topologicky organizované neuronové sítě, soutěživé učení, Kohonenovy neuronové sítě/mapy.* Přednáška předmětu Soft computing. Fakulta informačních technologií, Vysoké učení technické v Brně, 2014.

Příloha A

Seznam použitých zkratek

Zkratka	Význam
ALU	Arithmetic Logic Unit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DSP	Digital Signal Processor
FPGA	Field-Programmable Gate Array
GFLOPS	Giga floating-point operations per second
GPGPU	General-Purpose computing on Graphics Processing Units
GPU	Graphics Processor Unit
OpenCL	Open Computing Language
SIMD	Single Intruction, Multiple Data
SPMD	Single Process, Multiple Data
SSE	Streaming SIMD Extensions

Příloha B

Obsah DVD

dp_xsimic02.pdf - PDF verze technické zprávy diplomové práce.

tex/ - zdrojové soubory technické zprávy diplomové práce pro LaTeX.

src/ - zdrojové kódy implementace v jazyce C, implementace OpenCL, PROBEN datovou testovací sadu *cancer*, sadu testů.

src/README.md - textový soubor ve (formátu MD) obsahující informace pro kompilaci a spuštění aplikace.